Institute of Architecture of Application Systems

Transparent Execution of Data Transformations in Data-Aware Service Choreographies

Michael Hahn, Uwe Breitenbücher, Frank Leymann, Vladimir Yussupov

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{hahnml, breitenbuecher, leymann, yussupov}@iaas.uni-stuttgart.de

---

BibTeX:

```
@InProceedings{Hahn2018_TraDEDataTransformationExecution,
  author    = {Hahn, Michael and Breitenb{\"u}cher, Uwe and Leymann, Frank and
               Yussupov, Vladimir},
  title     = {{Transparent Execution of Data Transformations in Data-Aware
                Service Choreographies}},
  booktitle = {On the Move to Meaningful Internet Systems.
               OTM 2018 Conferences},
  publisher = {Springer International Publishing AG},
  address   = {Cham},
  series    = {Lecture Notes in Computer Science},
  volume    = {11230},
  pages     = {117--137},
  year      = {2018},
  isbn      = {978-3-030-02671-4},
  doi       = {10.1007/978-3-030-02671-4_7}
}
```

© 2018 Springer International Publishing AG.
The original publication is available at https://doi.org/10.1007/978-3-030-02671-4_7 and on Springer Link: https://link.springer.com/.

University of Stuttgart
Germany

Transparent Execution of Data Transformations in Data-Aware Service Choreographies

Michael Hahn, Uwe Breitenbücher,
Frank Leymann, and Vladimir Yussupov

Institute of Architecture of Application Systems (IAAS)
University of Stuttgart, Germany
{firstname.lastname}@iaas.uni-stuttgart.de

Abstract. Due to recent advances in data science, IoT, and Big Data, the importance of data is steadily increasing in the domain of business process management. Service choreographies provide means to model complex conversations between collaborating parties from a global viewpoint. However, the involved parties often rely on their own data formats. To still enable the interaction between them within choreographies, the underlying business data has to be transformed between the different data formats. The state-of-the-art in modeling such data transformations as additional tasks in choreography models is error-prone, time consuming and pollutes the models with functionality that is not relevant from a business perspective but technically required. As a first step to tackle these issues, we introduced in previous works a data transformation modeling extension for defining data transformations on the level of choreography models independent of their control flow as well as concrete technologies or tools. However, this modeling extension is not executable yet. Therefore, this paper presents an approach and a supporting integration middleware which enable to provide and execute data transformation implementations based on various technologies or tools in a generic and technology-independent manner to realize an end-to-end support for modeling and execution of data transformations in service choreographies.

Keywords: Data-aware Choreographies · Data Transformation · TraDE

1 Introduction

With recent advances in data science the importance of data is increasing also in the domain of Business Process Management (BPM) [14, 17]. The concept of Service-Oriented Architectures (SOA), i. e., composing units of functionality as services over the network, has found application in many research areas and application domains besides BPM [4, 24], e. g., in Cloud Computing, the Internet of Things, or eScience. The composition of services can be realized in either an orchestration or choreography-based manner. *Service orchestrations*, or *processes/workflows*, are defined from the viewpoint of one party that acts as a central coordinator [13]. *Service choreographies* are defined from a global

viewpoint with focus on the collaboration between multiple interacting parties, i. e., services, and their conversations without relying on a central coordinator [6]. Services taking part in such collaborations are represented as *participants* of a choreography and their conversations are defined through message exchanges.

Participants rely on their own internal data models on which their business logic is defined. Consequently, differences between the data formats of different participants have to be resolved to enable their interaction and definition of conversations between them. Therefore, *data transformations* have to be introduced to translate the underlying data to the different formats each participant requires. Such data transformations have to be defined within the participants of a choreography model, i. e., as part of their control flow, e. g., by adding corresponding transformation tasks that provide the required transformation logic. This approach is inflexible, time consuming and also pollutes the participants' control flow of choreography models with data transformation functionality that is not relevant from the perspective of individual participants but technically required to realize the conversations between them within a service choreography.

In Hahn et al. [10], we presented a first step to tackle these issues by introducing a modeling extension for defining data transformations in choreography models independently of participants control flow directly between the defined choreography data based on our concepts for *Transparent Data Exchange (TraDE)* [8]. However, execution support for the data transformations specified within a choreography model is missing, i. e., the actual implementations of defined data transformations need to be completely manually integrated into the execution environment to enable their automated execution during choreography run time. Although, this significantly eased the modeling of service choreographies with participants relying on different data formats, concepts for the automated integration and execution of modeled data transformations decoupled from participants' control flow are required to avoid error-prone and cumbersome manual integration steps which require significant technical expertise for integrating the required transformation software to a choreography execution environment.

In this paper, we tackle these issues by introducing concepts and a generic, technology-independent integration middleware which enable to provide, integrate and invoke data transformation implementations in an easy and automated manner to realize an end-to-end support for modeling and execution of data transformations in service choreographies. The contributions of this paper can be summarized as follows: (i) concepts for the specification and packaging of data transformation implementations, (ii) an architecture of a supporting integration middleware which enables the automatic integration of packaged transformation implementations into a choreography execution environment, (iii) concepts for the execution of data transformations in service choreographies decoupled from participants control flow in an automated and transparent manner based on our TraDE concepts, and (iv) the prototypical implementation of an integrated ecosystem for data-aware service choreographies with data transformation support.

The rest of this paper is structured as follows. Section 2 presents the problem statement of this work and introduces previous works on TraDE and our

modeling extension for data transformations. In Sect. 3, the *TraDE Data Transformation* (TDT) approach is introduced. Section 4 presents how modeled data transformations can be executed in an automated and transparent manner, i. e., decoupled from the choreography control flow. The prototypical implementation of an integrated TraDE ecosystem is outlined in Sect. 5. Section 6 presents a case study from the eScience domain as an example for applying the TDT approach to an existing choreography model. Finally, the paper discusses related work (Sect. 7) and concludes with our findings and future work (Sect. 8).

2 Problem Statement and Background

Commonly, participants in service choreographies rely on their own, custom data formats. To enable the modeling of conversations among them, each participant must understand all involved data formats. This can be achieved by translating the data to the target formats required by individual participants. To define such format translations on the level of service choreographies, *data transformations* have to be specified as parts of the participants' control flow. One solution is to use a standardized choreography modeling language like BPMN [16], which allows modeling data transformations as explicit tasks. However, this approach is error-prone, time consuming and requires considerable amount of efforts. Firstly, modelers have to provide transformation implementations required by the underlying modeling language or execution environment, e. g., using XQuery or XSLT for XML-related data transformations. This requires an extensive level of expertise in transformation languages, technologies and underlying data modeling languages and formats. Moreover, such transformations, when modeled as tasks in possibly multiple participants, pollute the control flow of participants with data transformation functionality that is not relevant from a participants perspective but technically required to realize the communication between participants of a choreography. In addition, since the underlying transformation implementations become a part of the resulting choreography models, they are spread across multiple different models which hinders their reuse and makes it hard to maintain them. This is especially problematic when data formats of choreography participants change over time, as underlying choreography models and all affected transformation tasks have to be adapted to support these new data formats. While providing transformation implementations as services eases the reuse process, modelers must be able to wrap their transformation implementations as services to invoke them in choreography models.

To provide the background for this work, we first compare the state-of-the-art approach for data transformation modeling in choreographies and the TraDE modeling extensions introduced in our previous works [8, 10]. Figure 1 depicts an example choreography modeled using these two approaches. Both choreographies are illustrated as Business Process Management Notation (BPMN) [16] collaboration models. The conversations among participants are defined using BPMN message intermediate events and message flows. Choreography data is modeled

via BPMN data objects on the level of choreography participants and exchanged as part of messages through specified message flows, e. g., *mx1* in Fig. 1.

2.1 State-of-the-Art in Modeling Data Transformations

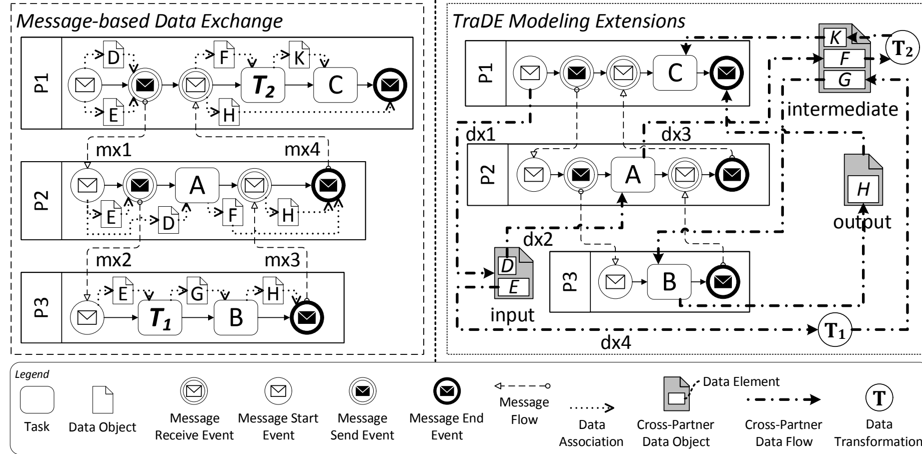

Fig. 1. Comparison of two modeling approaches, based on Hahn et al. [10].

The left model in Fig. 1 demonstrates the state-of-the-art approach for data transformation modeling in choreographies. We call this standard way of modeling and exchanging data in choreographies *message-based data exchange* [8]. Whenever participant *P1* receives a request, modeled as BPMN message start event, the contained data are extracted from the message and stored in data objects *D* and *E*. These data is then wrapped and sent to participant *P2* in a message exchanged via message flow *mx1*. In a similar way, participant *P3* receives these data via message flow *mx2*. At participant *P3*, data object *E* has to be transformed and stored in data object *G*, which is used as an input for task *B*. Therefore, participant *P3* defines transformation task T_1 that executes required data transformation logic. The result of task *B* is stored in data object *H* and sent back to participant *P2* through message flow *mx3*. Similarly, transformation task T_2 is required for obtaining data object *K* which is used as input by task *C*. Upon completion of all tasks, a message to the initial requester with data object *H* is sent as the final result of the choreography execution.

This *message-based data exchange* approach has significant drawbacks [8, 10]. First, the same data objects must be specified at each participant that uses the data, e. g., data objects *E* and *H* have to be specified in all participants. Moreover, the data flow within a participant can be seamlessly modeled through BPMN data associations, but data exchange across participants has to be modeled through a combination of message flows and related control flow modeling elements, e. g.,

BPMN *send* and *receive* tasks, and *message throw* or *message catch* events. Consequently, data cannot be exchanged across participants without introducing additional control flow constructs at the sender and receiver participants.

In addition, required data transformations have to be modeled manually through respective tasks and data associations on the level of choreography participants. For example, participant *P1* defines transformation task T_2 for translating the data produced by participant *P2* (data element F). Such tasks pollute the control flow of participants of choreography models with transformation functionality by mixing business and technical aspects together. Moreover, the underlying transformation implementations must be supported by the choreography modeling language and execution environment, i. e., as the task's implementation or as an invokable service. This approach lacks automation support and depends on the capabilities of the selected modeling language or execution environment. Furthermore, it still requires a significant amount of expertise on transformation languages, technologies and underlying data modeling languages and formats.

2.2 TraDE Data Transformation Modeling Extension

To improve and simplify the modeling of data transformations in service choreographies, we presented concepts for the specification of data as well as its exchange and transformation in service choreographies decoupled and independent from participants control flow [8, 10]. The choreography model depicted on the right of Fig. 1 applies our TraDE concepts and modeling extensions, namely *cross-partner data objects*, *cross-partner data flows*, and *data transformations*, to substitute message-based data exchange and explicit definitions of transformation tasks. Choreography data can be modeled in a participant-independent manner using cross-partner data objects, e. g., *input* in Fig. 1, and the reading and writing of the cross-partner data objects from tasks and events is specified through cross-partner data flows, e. g., *dx1* or *dx3*. Explicitly modeled transformation tasks T_1 and T_2 can be substituted by TraDE data transformations linked to the data objects E and G (for T_1) as well as F and K (for T_2) through cross-partner data flows. These cross-partner data flows allow exchanging the data across participants independently of message flows and, therefore, decouple the exchange of data from the exchange of messages. Moreover, the transformation of data can be specified directly on the data itself, i. e., between cross-partner data objects, instead of introducing transformation tasks on the level of participants.

Each *cross-partner data object* has a unique identifier and contains one or more *data elements*. For example, cross-partner data object *input* in Fig. 1 contains data elements D and E. A data element has a name and contains a reference to a definition of its structure, e. g., a XML Schema Definition [20]. A *TraDE data transformation* (DT) allows to specify a reference to the software that provides the related data transformation logic, e. g., a web service, script, or executable, referred to as *DT Implementation* in the following. The inputs and outputs of a data transformation can be specified by adding cross-partner data flows between a data transformation and one or more cross-partner data objects. If a data transformation requires or produces several inputs or outputs, modelers are able

to map the connected cross-partner data objects to respective inputs and outputs of the underlying DT Implementation through specifying a set of *Input/Output Mappings*. Furthermore, a TraDE data transformation allows to specify a set of *Input Parameters* which enables modelers to define input values for a DT Implementation that are not provided through cross-partner data objects. For example, to provide constant values, e. g., for the configuration or initialization of the underlying DT Implementation. In addition, an optional *Trigger Condition* and *Activation Mode* can be specified for each data transformation. A trigger condition allows to specify a boolean expression which is evaluated before the referenced DT Implementation is executed. The activation mode defines when a data transformation should be conducted: *on-read* or *on-write*.

The binding of modeled data transformations to concrete logic is not required during choreography modeling and can be deferred to choreography deployment. The main idea is to enable a separation of concerns, i. e., participants' business logic and choreography transformation logic is separated from each other, which introduces more flexibility since required transformations can be modeled within choreographies in an abstract manner and their actual binding to concrete DT Implementations can be done at a later point in time using the specified data transformations within a choreography model and their properties as a blueprint to identify or provide required DT Implementations. This allows modelers to focus on the modeling of participants and their conversations without taking care of how the differences of their data models can be solved.

As outlined in Hahn et al. [10], by supporting the definition of data transformations independent of choreography participants' control flow directly between cross-partner data objects, the main challenge is on how to provide, integrate and invoke the underlying data transformation implementations for modeled data transformations within service choreographies during choreography execution.

3 The TraDE Data Transformation Approach

In the following, the *TraDE Data Transformation* (TDT) approach is presented as the main contribution of this work. It combines our TraDE modeling extensions, an extended version of the TraDE Middleware [9], and introduces the new *Data Transformation* (DT) *Integration Middleware* to provide automatic integration and execution support for data transformations in service choreographies.

The presented TraDE data transformation modeling extension allows defining data transformations independent of choreography participants (see Sect. 2.2). However, to support the execution of respective data transformations, referenced transformation implementations (DT Implementation) need to be integrated into the choreography execution environment. In addition, data transformations are often implemented in different programming languages or restricted to certain execution environments. To tackle such heterogeneity, in Sect. 3.1 we first present concepts for the technology-agnostic specification and packaging of DT Implementations in so-called *DT Bundles*. The goal is to abstract away any concrete technologies or tools while automating tedious integration processes to avoid

manual wrapping of software. This allows modelers to easily create and provide their data transformation implementations as DT Bundles.

These DT Bundles can then be published to the DT Integration Middleware to make the contained DT Implementations available for use within choreographies. The architecture of the DT Integration Middleware and how it supports the fully automated provisioning and execution of DT Bundles is presented in Sect. 3.2. Finally, in Sect. 4, an integrated TraDE ecosystem is presented and the transparent execution of defined data transformations is described in more detail.

3.1 Specification and Packaging of DT Implementations

Based on the findings of related work discussed in Sect. 7 and lack of available and suitable standards, we introduce our own conceptual model for an easy and technology-agnostic specification and packaging of data transformation implementations that fully satisfies our requirements. The resulting model can be extended and adapted to support various required use cases and functionalities.

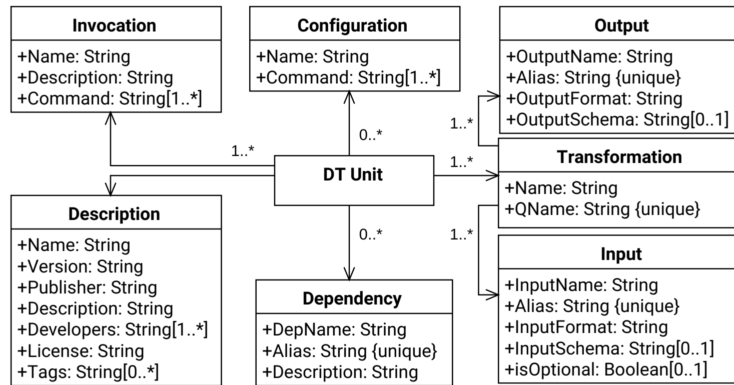

Fig. 2. A conceptual model for specifying DT Units.

The UML class diagram in Figure 2 illustrates our proposed conceptual model for the specification of *data transformation units* (DT Units). In contrast to the already introduced DT Bundle, a DT Unit provides a specification of one or more DT Implementations, e. g., their inputs and outputs, required tools or execution environments. A DT Bundle represents a concrete materialization of a DT Unit by providing also the required concrete resources, e. g., data transformation implementations in form of executables or scripts, configuration files or installation scripts to setup required tools or frameworks for executing respective transformations. Since DT Implementations can vary in different dimensions, a black-box approach is employed for their specification and execution, i. e., considering DT Implementations as atomic reusable entities which have to remain immutable. Apart from general information such as name, version, or

publisher specified as *Description* entity, a DT Unit has several more important characteristics. First, a DT Unit supports one or more transformations, e. g., transforming textual data into several different image formats. This definition is provided in *Transformation* entities representing DT Implementations. A Transformation has a name and an unique fully-qualified name (QName) which can be used, e. g., for searching transformations at the middleware or to reference them as transformation implementations in TraDE data transformations within choreography models. Each transformation is described by one or more inputs and outputs specified through *Input* and *Output* entities. Both entities have a name and must be uniquely identifiable by an alias, which can be used for referencing, e. g., to specify the invocation of a transformation of a DT Unit. Possible types of inputs or outputs can be messages, data streams, files, parameters, or data from databases. The inputs and outputs of a transformation might have a specific format and can therefore provide a schema file describing their data format.

DT Units can have a particular set of dependencies that have to be satisfied to execute their transformations. For example, a DT Unit can depend on certain software, libraries, (configuration) files or even operating system (OS) environment variables. Therefore, a proper specification of dependencies is needed. Such dependencies, represented as *Dependency* entities in Fig. 2, have to be provided or installed in some way, e. g., using an OS-level package manager's command or using a set of materialized files and related installation commands. Moreover, the execution of preparation steps or other logic before invoking a DT Unit can be a prerequisite for running a transformation. Hence, a specification of required configurations in form of *Configuration* entities can be provided. Finally, every DT Unit must specify how to invoke its provided transformations represented through *Invocation* entities. For example, their transformations can be invoked by sending a request to an API or executing a command via the command line interface (CLI) of an OS. Such invocation command might need to reference other model's entities, e. g., inputs, by inserting their defined aliases.

To package described DT Units with their transformation implementations and related files as DT Bundles based on the introduced conceptual model, a standardized packaging format is required. Introducing a predefined structure for packaging and storing DT Units is beneficial as no additional knowledge is needed to process the DT Bundles later. A packaged DT Unit, i. e., a DT Bundle, consists therefore of the following distinct parts: (i) *unit* part contains all DT Unit-related files, e. g., DT Implementation artifacts such as scripts or executables, (ii) *dependencies* part groups all required dependencies, (iii) *schemas* part contains optional schema files which define the structure of transformation inputs and outputs, (iv) *DT Unit specification* is an instance of the conceptual model for a concrete DT Unit materialized as a file, e. g., as JSON file.

3.2 Architecture of the DT Integration Middleware

Figure 3 presents the architecture of the DT Integration Middleware which allows modelers to publish DT Bundles to make their data transformation implementations available for use within choreographies. To allow a broad variety of

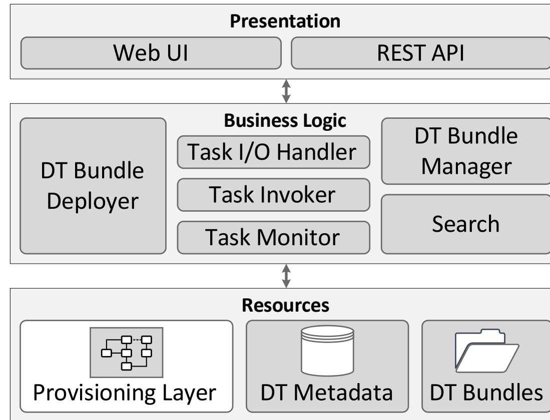

Fig. 3. Architecture of the DT Integration Middleware.

potential implementations, the architecture is defined in a generic and technology-independent manner. Our focus is on the description of the logical building blocks and functionality which can be then combined with or implemented through well-established middleware solutions, e. g., such as Enterprise Service Buses (ESB) [5] know for their integration and transformation capabilities in the context of SOA.

In the following, the architecture will be presented in a top-down manner followed by a more detailed description of its business logic components. The *Presentation* layer enables the communication of external clients with the middleware, e. g., through a *Web UI* or *REST API*. The *Business Logic* layer provides the core functionality of the middleware. Its components are responsible for the publishing, provisioning, and the execution of transformation implementations provided by DT Bundles in a task-based manner. The *Resources* layer provides and integrates actual technologies for storing and provisioning of DT Bundles to enable the execution of their contained transformation implementations. This comprises the storage of the actual files of a DT Bundle (see Sect. 3.1) in the file system (*DT Bundles* in Fig. 3). The related metadata of all managed DT Units and DT Bundles is persisted in a database (*DT Metadata* in Fig. 3) to simplify access and provide query support on the metadata. To support the provisioning of published DT Bundles as a prerequisite for their execution, the middleware relies on a *Provisioning Layer*, e. g., Docker or OpenTOSCA [3], to provide the specified run time environment and dependencies of a DT Bundle.

To make a published DT Bundle and its provided DT Implementations invokable, it has to be prepared for provisioning first. Since the specification of a DT Unit might contain references to remote resources, e. g., files or software dependencies, these references need to be materialized to preserve the state and behavior of the DT Bundle within the middleware, as referenced files can change over time leading to different bundle versions. For example, if a certain dependency is specified, exactly the specified version has to be present for a

DT Bundle within the middleware. If materialization happens, the DT Unit specification has to reflect the changes affecting materialized references. Finally, a published DT Bundle needs to be stored, e. g., using a database, a file system, or a combination of both. The *DT Bundle Manager* shown in Fig. 3 provides the functionality for reference materialization, transforming DT Unit specifications of DT Bundles into provisioning-ready specifications and manages the storage of the resulting refined bundles and their metadata within the Resources layer. Such provisioning-ready specification can be provided in form of, e. g., a Dockerfile or a TOSCA [15] topology. The *Search* component allows to search and identify suitable transformations of available DT Bundles based on user requests utilizing the metadata provided through the available DT Unit specifications. Search can employ various techniques from trivial unique name search to composition of multiple transformations together to produce a desired output from the provided input. The *DT Bundle Deployer* is responsible for deploying DT Bundles to the supported provisioning layer. Therefore, it uses the provisioning-ready specifications generated by the DT Bundle Manager and deploys them to the selected Provisioning Layer. The choice of provisioning technology is not restricted by the architecture, however, the middleware relies on a default provisioning specification leaving the possibility to generate other specification types up to pluggable components and the user's choice. For example, a Dockerfile can be generated if Docker is the default provisioning specification type. As potentially multiple provisioning layers can be used together, the DT Bundle Deployer has to update the metadata of a DT Bundle, e. g., stored in a database, to reflect its deployment status at the Provision Layer, i. e., if it is available for executing its contained transformation implementations.

Another important part of the middleware is the task-based execution of transformations, i. e., the execution of a transformation implementation of a DT Bundle. Therefore, a new *transformation task* can be issued by sending a request to the REST API of the middleware. Such a request contains a reference to a DT Bundle, the fully-qualified name of a Transformation of the specified DT Bundle as well as required information about retrieving input and placing output data according to the DT Unit specification of the DT Bundle (see Sect. 3.1). The *Task I/O Handler* is responsible for preparing the specified inputs as a prerequisite to invoke a transformation as well as processing the resulting outputs. Therefore, inputs can be received in a pull or push-based manner. In the former case, inputs are provided as references within transformation task requests and need to be downloaded and prepared for the invocation. In the latter case, inputs are contained in the request itself. The actual invocation and execution of the specified transformation is managed by the *Task Invoker*. Therefore, it uses the prepared inputs to invoke the transformation based on the DT Unit specification (*Invocation* in Sect. 3.1). During the execution of the transformation, the *Task Monitor* component allows to monitor the state of the execution by sending corresponding requests to the REST API of the middleware. As soon as the transformation is completed, the *Task I/O Handler* component is responsible to process the resulting outputs and pass them back to the requester.

Since the DT Integration Middleware has to support various types of inputs and outputs (e. g., files, messages, or data streams), invocation mechanisms (e. g., CLI or HTTP), and monitoring concepts for different data transformation types, the middleware supports integration of several implementations of these three components in a pluggable manner. To automate the execution of data transformations in choreographies, the middleware is integrated with the TraDE Middleware as discussed in the following.

4 Transparent Execution of Data Transformations

Applying the TraDE Data Transformation (TDT) approach allows to specify and provide data transformation implementations as DT Bundles and execute them with the help of the DT Integration Middleware in a generic task-based manner. What is still missing is how DT Bundles can be used within choreographies to define data transformations and how the execution of a DT Bundle's transformation can be triggered during choreography execution. Therefore, Sect. 4.1 presents the integrated TraDE ecosystem and Sect. 4.2 outlines the execution of DT Bundle transformations through the TraDE Middleware in a transparent manner, i. e., independent of the choreography control flow during choreography run time.

4.1 Integrated TraDE Ecosystem

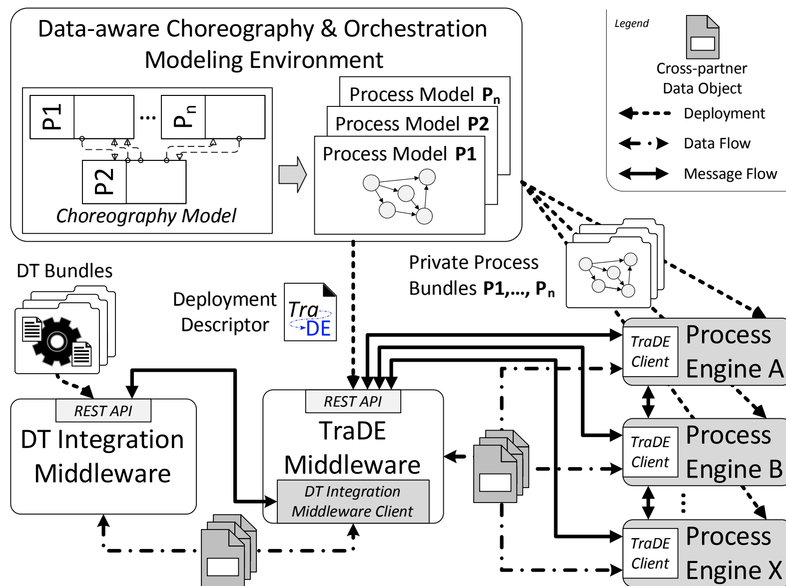

Fig. 4. Integrated system architecture and deployment artifacts of the TraDE ecosystem.

Figure 4 depicts the TraDE ecosystem which provides an end-to-end support for data transformations in service choreographies. Before we describe the execution of data transformations within the ecosystem, first the ecosystem itself is introduced and the resulting deployment artifacts of a data-aware choreography as well as their deployment to the components of the ecosystem are described.

As described in Sect. 3.1, data transformations are provided in form of *DT Bundles* which comprise one or more data transformation implementations and a DT Unit specification. They are published to the *DT Integration Middleware* to make them available to the overall TraDE ecosystem. As outlined in Sect. 2.2, the introduced TraDE data transformation modeling extension [10] allows to specify a reference to the software that provides the underlying data transformation implementation. By following the TDT approach, such references to transformation software, i. e., transformation implementations can now be provided and integrated to choreography models by referencing corresponding DT Bundles with their fully-qualified name (QName) (see Sect. 3.1). This allows the TraDE Middleware to trigger a new transformation task at the DT Integration Middleware as means to conduct a modeled transformation by executing the referenced transformation implementation of a corresponding DT Bundle.

The *Data-aware Choreography & Orchestration Modeling Environment* enables modelers to specify data-aware service choreographies by modeling *cross-partner data objects*, *cross-partner data flows* and *data transformations*. Based on the fact that a lot of choreography modeling languages do not produce executable models, we follow the established approach of transforming choreography models into a collection of executable private process models which collectively implement the overall choreography [7] as shown in Fig. 4. The resulting private process models can then be manually refined by adding corresponding internal logic for each participant. The private process models are packaged together with related files, e. g., process engine deployment descriptors, or interface and schema definitions, as *Private Process Bundles* for their deployment on *Process Engines* as depicted in Fig. 4. Furthermore, all cross-partner data objects and their dependencies defined in a choreography model are exported to a *TraDE Deployment Descriptor* file. This file is uploaded to the *TraDE Middleware* where it is compiled into related internal representations to provide and expose all cross-partner data objects as resources through the middleware's REST API [9] as well as support the triggering of data transformations. Thus, the TraDE Deployment Descriptor contains information about all specified data transformations of a choreography. As introduced in Sect. 2.2, this comprises the reference to a data transformation implementation, Input/Output Mappings, Input Parameters, and an optional Trigger Condition and Activation Mode. The reference to a transformation implementation is specified by adding the fully-qualified name (QName) of a DT Bundle's transformation (see Sect. 3.1). This separation of concerns, i. e., participants' business logic is specified in private process models and transformation logic is specified/referenced in TraDE Deployment Descriptor, allows more flexibility since transformations can be provided and specified as DT Bundles independent of the choreography/private process models and therefore also be easily changed

without affecting the private process models. This allows modelers to focus on the modeling of choreography participants and their conversations without taking care of how the differences of their data models can be solved. The actual binding of concrete transformation logic, i. e., transformation implementations being part of a DT Bundle, can be delayed to choreography deployment since this binding information is provided as part of the TraDE Deployment Descriptor and does not require any changes on the level of the private process models of a choreography.

The TraDE Middleware is integrated with the DT Integration Middleware as well as the respective process engines through clients. Therefore, the process engines are extended with a *TraDE Client* to communicate with the TraDE Middleware through its REST API. The TraDE Middleware contains a *DT Integration Middleware Client* to trigger the task-based execution of data transformations by sending *transformation task* requests to the DT Integration Middleware's REST API. As shown in Fig. 4, this enables the TraDE Middleware to act as a data hub between the private process models implementing choreography participants and referenced DT Bundles, i. e., defined data transformations. In the following, the execution of modeled data transformations within the ecosystem is described.

4.2 Automatic Triggering of Data Transformations

As outlined above, the TraDE Middleware comes with its own internal, choreography language independent metamodel as presented in Hahn et al. [9]. The middleware extracts all defined cross-partner data objects and data elements from the TraDE Deployment Descriptor and translates them to respective *CrossPartnerDataObject* and *DataElement* entities according to its metamodel. To represent the actual data of running choreography instances, i. e., collection of instances of the private process models implementing the choreography model, the metamodel defines further entities. For each choreography instance, *CrossPartnerDataObjectInstance* and *DataElementInstance* entities are created at the middleware with associated *CorrelationProperty* entities which enable to uniquely identify the choreography instance the data object and data element instances belong. The actual data of a choreography instance is represented through *DataValue* entities which are referenced by *DataElementInstance* entities. The TraDE Middleware provides an event model for each entity type, i. e., a life cycle with states and transitions. This allows firing an event whenever an entity changes its state, e. g., a DataValue is *initialized*. Based on these event models, the TraDE Middleware supports an event-based mechanism to trigger actions on respective events. This concept is used to transparently execute data transformations specified in choreography models by triggering the invocation of referenced transformations provided as DT Bundles at the DT Integration Middleware and handling the underlying data exchange.

For example, data transformation T_1 in Fig. 1 will be triggered as soon as data element E of cross-partner data object *input* is initialized or whenever it is modified. The invocation of the respective DT Bundle's transformation itself is straightforward. All required information is sent to the DT Integration Middleware within a request that triggers the task-based execution of a DT

Bundle's transformation. This requires the resolution of DataValue entities for a specific choreography instance that hold the input data of the transformation. For the example depicted in Fig. 1, this means that the middleware has to first identify the DataElementInstance entity of data element E related to the running choreography instance. Next, the DataValue entity associated to the resulting data element instance can be resolved to get the actual data to transform. Instead of passing the input data within the invocation request to the DT Integration Middleware, a URL pointing to the respective resource exposing the required DataValue entity at the TraDE Middleware's REST API is added for each transformation input. The same applies for transformation outputs. Instead of retrieving output data in a response message, a DataValue resource URL is added to the invocation request for each transformation output. When the transformation is completed, the DT Integration Middleware uploads all results to the TraDE Middleware by pushing them to DataValue resources specified through URLs in the invocation request, making the data available for further use.

By default, a transformation is triggered whenever data is written to a DataValue associated to one of its input cross-partner data objects. The *trigger condition* and *activation mode* allow influencing the underlying behavior of the TraDE Middleware. In *on-write* activation mode, the TraDE Middleware first waits until all input data for a data transformation is available, i. e., the DataValue entities associated to cross-partner data objects specified as inputs are successfully initialized, and then invokes the DT Bundle's transformation. Furthermore, whenever one or more of the specified transformation inputs are modified, the TraDE Middleware invokes the DT Bundle's transformation again. Based on that, the specified outputs of a data transformation are always up-to-date. In *on-read* activation mode, the TraDE Middleware triggers a data transformation whenever one of its output cross-partner data objects or data elements are read. For example, data transformation T_1 in Fig. 1 will be triggered whenever data element G of cross-partner data object *intermediate* is read. Since reads and writes of cross-partner data objects are decoupled from choreography execution, the TraDE Middleware has to wait until all the required input cross-partner data objects are available before triggering the invocation of a DT Bundle's transformation. Further fine tuning of the data transformation triggering behavior at the TraDE Middleware is possible through the specification of a *trigger condition*. It allows to define a boolean expression that is evaluated by the TraDE Middleware to check if a data transformation should be triggered or not. For example, this can be used to trigger a transformation only if the value of an input cross-partner data object is within certain margins.

5 Prototype

To prove the technical feasibility of our approach, we describe the prototypical implementation of the TraDE ecosystem and its components in the following. For the modeling of data-aware choreographies with data transformations, the choreography modeling language BPEL4Chor [7] is used and extended. The

underlying Data-aware Choreography & Orchestration Modeling Environment is built on existing tools, i. e., *Chor Designer* [21] and an extended version of the Eclipse *BPEL Designer*. As Process Engine an extended version of the open source BPEL engine Apache *Orchestration Director Engine* (ODE) is used. To enable the execution of cross-partner data flows, the implementation of Apache ODE is extended and integrated with the TraDE Middleware to enable the reading and writing of cross-partner data objects [9].

The TraDE Middleware itself is implemented as a Java-based web application which exposes its functionality through a REST API which is specified using Swagger and implemented with the *Jersey* RESTful Web Services framework. The TraDE internal representations and the actual data processed within the choreographies, can be persisted using MongoDB or the local file system. To support the event-based triggering of DT Bundles within the context of this work, the TraDE Middleware is extended with corresponding functionality implemented using Apache Camel to send requests to the REST API of the DT Integration Middleware. The TraDE Middleware open source code is available on GitHub[1].

The DT Integration Middleware is a web application implemented in Python Flask with its functionality exposed via a REST API. Swagger is used for the specification of the REST API. For storing DT Bundles a combination of MongoDB and a file system is used. The former stores metadata derived from the provided DT Unit specifications, whereas the latter is used for storing the files of DT Bundles. The prototype supports *file-based* DT Implementations which rely on files and parameters as input and output types and can be invoked through CLI commands. To provision DT Bundles, Docker is used as provisioning layer integrated to the middleware through a Docker SDK for Python. More complex DT Implementations and DT Bundles requiring other invocation mechanisms as well as input and output types will be supported using TOSCA [15] in future. The DT Integration Middleware open source code is available on GitHub[2].

6 Case Study

As an example for applying the TDT approach, a case study from the eScience domain is presented in the following. Therefore, the naïve choreography model is presented where data transformations are defined by adding respective transformation tasks. Finally, the TDT approach is applied to the model as an example for easier modeling of data transformations and enabling their provisioning and execution in a technology-independent and transparent manner.

6.1 Naïve Modeling of Data Transformations

Figure 5 shows an excerpt of the choreography model of a Kinetic Monte Carlo (KMC) simulation using the custom-made simulation software *Ostwald ripening*

[1] TraDE: https://github.com/traDE4chor/trade-core/releases/tag/v1.1.0
[2] DT: https://github.com/traDE4chor/hdtapps-prototype/releases/tag/v1.0.0

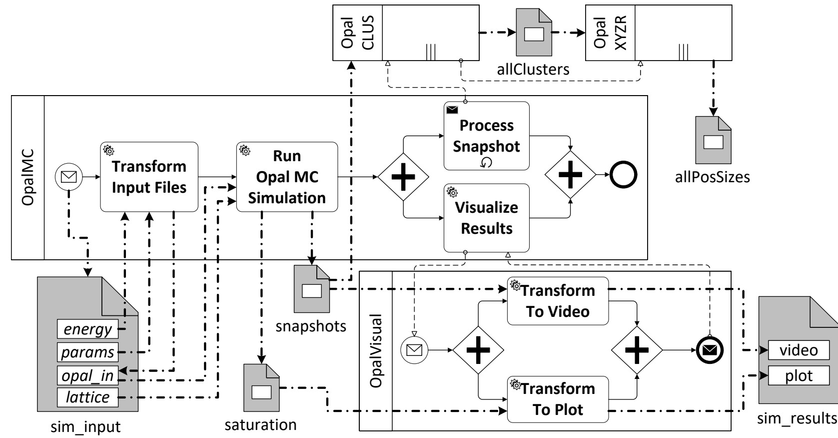

Fig. 5. Choreography conducting a thermal aging simulation from eScience [8].

of Precipitates on an Atomic Lattice (OPAL) [2]. OPAL simulates the formation of copper precipitates, i. e., the development of atom clusters, within a lattice due to thermal aging. The whole simulation consists of four major building blocks which are reflected as participants of the data-aware choreography depicted in Fig. 5. Following our TraDE concepts, all data relevant for the choreography model is specified independently of any participant and in a shared and reusable manner through cross-partner data objects (DO) and data elements (DE).

Whenever the *OpalMC* participant receives a new request, a new instance of the KMC simulation is created. The initial request contains a set of inputs (*sim_input* DO), e. g., parameters such as the number of simulation snapshots to take (*params* DE), an initial energy configuration (*energy* DE), and a lattice (*lattice* DE). First, the *Transform Input Files* service task executes a data transformation to combine input parameters and energy configuration into the input format of the KMC simulation. The transformation result is stored in the *opal_in* DE. Next, the *Run Opal MC Simulation* service task invokes a service which conducts the KMC simulation based on the provided data. According to the specified number of snapshots in *opal_in* DE, the service saves the current state of the atom lattice at a particular point in time as a snapshot and replies all snapshots together (*snapshots* DO) as well as saturation data (*saturation* DO).

Based on the number of snapshots, the *Process Snapshots* send task is conducted multiple times to invoke the analysis of each snapshot individually at the *OpalCLUS* and *OpalXYZR* participants. First, all clusters within each snapshot are identified and stored in the *allClusters* DO through the *OpalCLUS* participant. This cluster information is then processed by the *OpalXYZR* participant to identify the position and size of each cluster. The respective results are stored in the *allPosSize* DO. Since the internal logic of the OpalCLUS and OpalXYZR participants do not provide further insights, they are depicted as a black box.

The *Visualize Results* service task triggers the visualization of snapshot and saturation data at the *OpalVisual* participant. The *Transform To Video* and *Transform To Plot* service tasks invoke related transformation services using the *snapshots* and *saturation* DO as input. The collection of snapshots is transformed into a video of animated 3D scatter plots and the saturation data is transformed to a 2D plot of the saturation function of the precipitation process as a final result which are stored in the respective data elements of the *sim_results* DO.

According to Sect. 2, the choreography model contains three transformation tasks that are not a relevant part of the simulation but technically required for pre-processing (sim_input DO) and post-processing (sim_results) of simulation data. Moreover, the simulation specific transformation implementations have to be manually wrapped as services to enable their integration and invocation in the choreography. This requires expertise and additional effort since the transformation implementations are provided as shell script (*Transform Input Files*) or Python (*Transform To Video*) and Gnuplot scripts (*Transform To Plot*).

6.2 Applying the TraDE Data Transformation Approach

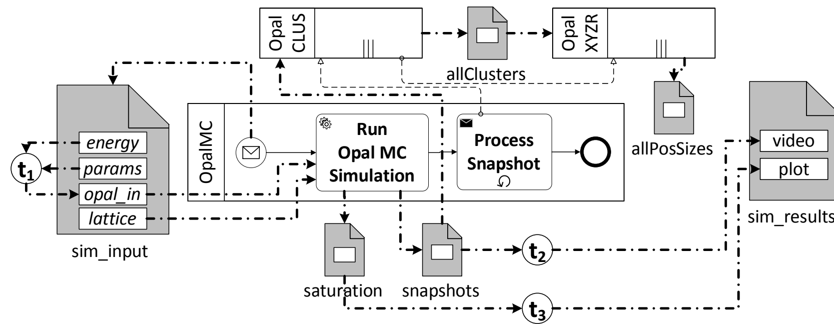

Fig. 6. Choreography model with TraDE data transformations applied.

Figure 6 depicts the OPAL choreography model with the introduced TDT approach applied. The definition of data transformations between cross-partner data objects allows to substitute the explicitly modeled transformations tasks: *Transform Input Files*, *Transform To Video* and *Transform To Plot*. Instead, TraDE data transformations t_1, t_2, t_3 are defined to transform the choreography data as required. This contributes to our goal of specifying data and its transformations independent of any participants in service choreographies directly between cross-partner data objects. Furthermore, the transformation implementations have no longer to be provided as services to enable their integration and invocation through tasks (e. g., *Transform Input Files*). Modelers are now able to specify and integrate their transformation implementations without manual wrapping effort, in form of DT Bundles to enable their transparent execution

within the TraDE ecosystem. With the help of the TDT approach, the original scripts can be automatically wrapped and integrated into the TraDE ecosystem to enable their invocation. Only respective DT Unit specifications have to be created for each of the scripts to enable their packaging together with related files as DT Bundles and finally publish them to the DT Integration Middleware. To get an idea how a concrete DT Bundle looks like, a corresponding example for transformation t_2 (*Transform To Video*) is available on GitHub[3].

7 Related Work

Since the focus of this work is on how to specify, integrate and execute heterogeneous data transformation implementations in an easy and automated manner in the context of data-aware service choreographies, this section presents related work regarding application reuse and wrapping techniques.

Zdun [23] introduces an approach for legacy application migration to the web. He describes a method with the following four steps: (i) providing an API using either wrapping or redevelopment approaches, (ii) implementation of a component responsible for mapping of requests to the legacy API, (iii) as well as implementation of a component responsible for response generation, and (iv) the integration of these components into a web server. Furthermore, a reference architecture supporting the introduced concepts and issues is presented.

Sneed et al. [18] introduce white-box wrapper generation approaches for wrapping functions in legacy applications based on XML descriptions. The presented tool supports, e.g., the transformation of PL/I and COBOL functions' into WSDL interfaces. Additionally, the transformation generates modules responsible for mediation of the input and output data between legacy and WSDL interfaces.

Afanasiev et al. [1] present a cloud platform called MathCloud which allows reusing scientific applications by exposing them as RESTful web services having a uniform interface for a task-based execution. Requests contain the task description, inputs specification and resulting output is returned when the task is completed. Sukhoroslov et al. [19] introduce Everest, a PaaS platform for reusing scientific applications based on the MathCloud platform [1]. The authors further improve the ideas of providing a uniform interface for task-based execution of applications.

Juhnke et al. [12] present the Legacy Code Description Language framework which allows wrapping legacy code. An extensible legacy code specification model is used as a basis for executable wrappers generation. The model stores the information necessary for wrapping binary and source code legacy applications.

Wettinger et al. [22] present an APIfication approach which allows generating API implementations for executable programs. The underlying assumption is that an executable is provided along with metadata describing its dependencies, inputs, outputs, and other required information. Additionally, the authors introduce *any2api* as a generic and extensible framework for reusing executable software.

[3] Opal snapshot-to-video transformation DT Bundle: https://github.com/traDE4chor/hdtapps-prototype/tree/master/samples/opalVideo

Hosny et al. [11] introduce AlgoRun, a container template based on Docker suitable for wrapping CLI-based scientific algorithms and exposing them via a REST interface to simplify the reuse of scientific algorithms. Therefore, the algorithm has to be described using a predefined format. Moreover, a Dockerfile has to be created which wraps the algorithm's source code.

While some of the works are used as a basis for the TDT approach, none of them fit completely our needs. Since our focus is on data transformation software, data-related aspects and capabilities of such application reuse and wrapping techniques are of major relevance. The idea to create specifications for legacy applications similar to ones introduced by Juhnke et al. [12] and Hosny et al. [11] is used as an inspiration. However, we wanted to avoid any tight-coupling with a particular type of infrastructure or provisioning technology, since various provisioning specifications can be generated based on our introduced technology-agnostic DT Bundle specification. Our goal is to provide generic concepts and a supporting middleware for the specification, packaging, provisioning, and invocation of data transformation implementations as DT Bundles to enable their use within service choreographies.

8 Conclusion and Outlook

Data transformations are required on the level of choreographies to mediate between the different data formats of their collaborating participants. To support the execution of data transformations independent of choreography participants' control flow, the TDT approach is introduced to provide and invoke the underlying data transformation implementations for modeled data transformations within service choreographies. The main goal of the approach is to avoid the potentially tedious integration or even complete manual wrapping of required transformation software for its use within choreographies. Therefore, concepts for the specification and packaging of transformation implementations as DT Bundles and a supporting DT Integration Middleware enabling their execution are introduced. The resulting integrated TraDE ecosystem enables a seamless and transparent execution of data transformations within choreographies. Finally, we presented a prototypical implementation and a case study where we applied the TDT approach to an existing choreography model from the domain of eScience to show its feasibility.

In future, we plan to extend transformation capabilities for both modeling and execution of choreographies, e. g., by supporting the specification of trigger conditions for fine-grained control of triggering data transformations. Furthermore, an evaluation of the overall TraDE ecosystem is planned to compare and identify its behavior based on different scenarios, e. g., measure performance variations regarding the number of choreography participants and parallel reading/writing as well as transformation of shared cross-partner data objects.

Acknowledgments This research was supported by the projects SmartOrchestra (01MD16001F) and SePiA.Pro (01MD16013F).

References

1. Afanasiev, A., et al.: MathCloud: publication and reuse of scientific applications as RESTful web services. In: PaCT (2013)
2. Binkele, P., Schmauder, S.: An atomistic Monte Carlo simulation of precipitation in a binary system. Zeitschrift für Metallkunde (2003)
3. Binz, T., et al.: OpenTOSCA - A Runtime for TOSCA-based Cloud Applications. In: ICSOC (2013)
4. Bouguettaya, A., et al.: A Service Computing Manifesto: The Next 10 Years. Communications of the ACM (2017). https://doi.org/10.1145/2983528
5. Chappell, D.: Enterprise Service Bus. O'Reilly Media, Inc. (2004)
6. Decker, G., et al.: An Introduction to Service Choreographies. Information Technology (2008)
7. Decker, G., et al.: Interacting services: from specification to execution. Data & Knowledge Engineering (2009)
8. Hahn, M., et al.: Modeling and Execution of Data-Aware Choreographies: An Overview. Computer Science - Research and Development (2017)
9. Hahn, M., et al.: TraDE - A Transparent Data Exchange Middleware for Service Choreographies. In: OTM Conferences (2017)
10. Hahn, M., et al.: Modeling Data Transformations in Data-Aware Service Choreographies. In: EDOC (2018)
11. Hosny, A., et al.: AlgoRun: a Docker-based packaging system for platform-agnostic implemented algorithms. Bioinformatics (2016)
12. Juhnke, E., et al.: LCDL: an extensible framework for wrapping legacy code. In: iiWAS (2009)
13. Leymann, F., Roller, D.: Production Workflow - Concepts and Techniques. PTR Prentice Hall (2000)
14. Meyer, S., et al.: Towards Modeling Real-world Aware Business Processes. In: WoT (2011)
15. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0 (2013)
16. OMG: Business Process Model And Notation (BPMN) Version 2.0 (Jan 2011)
17. Schmidt, R., et al.: Big Data as Strategic Enabler - Insights from Central European Enterprises. In: Business Information Systems (2014)
18. Sneed, H.M.: Integrating legacy software into a service oriented architecture. In: Software Maintenance and Reengineering (2006)
19. Sukhoroslov, O., Afanasiev, A.: Everest: A Cloud Platform for Computational Web Services. In: CLOSER (2014)
20. W3C: XML Schema Definition Language (XSD) 1.1 Part 1: Structures (2012)
21. Weiß, A., et al.: Modeling Choreographies using the BPEL4Chor Designer. Technical Report 2013/03, University of Stuttgart (2013)
22. Wettinger, J., et al.: Streamlining APIfication by Generating APIs for Diverse Executables Using Any2API. In: CLOSER (2015)
23. Zdun, U.: Reengineering to the web: A reference architecture. In: Software Maintenance and Reengineering (2002)
24. Zimmermann, O.: Microservices tenets. Computer Science - Research and Development (2016)

All links were last followed on August 30, 2018.