Universität Stuttgart

Transparent Data Exchange in Service Choreographies: An eScience Perspective

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik sowie dem Stuttgarter Zentrum für Simulationswissenschaft (SC SimTech) an der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Michael Hahn

aus Ostfildern

Hauptberichter:	Prof. Dr. Dr. h. c. Frank Leymann
Mitberichter:	Prof. Dr. Dimka Karastoyanova apl. Prof. Dr.-Ing. Jörg Fehr

Tag der mündlichen Prüfung: 21.02.2023

Institut für Architektur von Anwendungssystemen
der Universität Stuttgart

2023

Contents

Abstract

This work is motivated by the increasing importance and business value of data in the fields of business process management, scientific workflows as a field in eScience, and Internet of Things, all of which profiting from the recent advances in data science and Big data. Although service choreographies provide means to specify complex conversations between multiple interacting parties from a global perspective and in a technology-agnostic manner, they do not fully reflect the current paradigm shift towards data-awareness at the moment. Therefore, the focus of this work is on tackling respective shortcomings. These include the missing modeling support for data flow across participant boundaries or specifying a choreography data model as a contract on the business data relevant to realize the collaboration and all interacting parties agree on.

Towards this goal, we introduce a choreography management life cycle that assigns data its deserved primary role in service choreographies as well as defines the functions and artifacts necessary for enabling transparent and efficient data exchange among choreography participants. To implement the introduced life cycle we present the notion of data-aware choreographies through our concepts for Transparent Data Exchange (TraDE) by introducing cross-partner data objects and cross-partner data flows as means to increase runtime flexibility while reducing the complexity of modeling data flows

in service choreographies. The TraDE concepts focus on decoupling the data flow, data exchange and management, from the control flow in service compositions and choreographies. To provide an end-to-end support for the modeling and execution of data-aware choreographies and supporting the respective phases of the choreography management life cycle, we introduce and prototypically implement an overall TraDE ecosystem. This ecosystem comprises a modeling environment for data-aware choreographies as well as the required runtime environment to execute such data-aware choreographies through a new TraDE Middleware and its integration to corresponding Business Process Engines (BPEs). The inherent goal of this work is to simplify the modeling of data and its exchange in choreography models while increasing their runtime flexibility and enabling the transparent exchange and transformation of data during choreography execution.

Zusammenfassung

Diese Arbeit ist motiviert durch die zunehmende Bedeutung und den geschäftlichen Wert von Daten in den Bereichen Business Process Management (BPM), Scientific-Workflows als ein Bereich in eScience und dem Internet der Dinge (IoT), die alle von den jüngsten Entwicklungen in den Bereichen Data Science und Big Data profitieren. Obwohl Service-Choreographien die Möglichkeit bieten, komplexe Konversationen zwischen mehreren interagierenden Parteien aus einer globalen Perspektive und in einer technologieunabhängigen Weise zu spezifizieren, spiegeln sie den Paradigmenwechsel in Richtung Daten und deren Wichtigkeit derzeit noch nicht vollständig wider. Daher liegt der Schwerpunkt dieser Arbeit auf der Identifikation und Auflösung aktueller Einschränkungen, wie z.B. der fehlenden Modellierungsunterstützung für den Datenfluss über Teilnehmergrenzen hinweg oder der Spezifizierung eines Choreographie-Datenmodells als Vertrag über die für die Realisierung der Zusammenarbeit relevanten Geschäftsdaten, auf die sich alle interagierenden Parteien einigen.

Um dieses Ziel zu erreichen, führen wir einen Lebenszyklusmodell für das Choreographie-Management ein, der Daten die ihnen notwendige Rolle in Service-Choreographien einräumt und die Funktionen und Artefakte definiert, die notwendig sind, um einen transparenten und effizienten Datenaustausch zwischen Choreographie-Teilnehmern zu ermöglichen. Zur

Umsetzung des erweiterten Lebenszyklus führen wir den Begriff der datenbewussten Choreographien (data-aware choreographies) durch unsere Konzepte für den transparenten Datenaustausch (kurz: TraDE) ein. Mittels der Einführung und Definition von partnerübergreifenden Datenobjekten (cross-partner data objects) und partnerübergreifenden Datenflüssen (cross-partner data flows) soll sowohl die Komplexität der Modellierung von Daten und Datenflüssen in Service-Choreographien reduziert als auch die Laufzeitflexibilität erhöht werden. Der Schwerpunkt der TraDE-Konzepte liegt dabei auf der Entkopplung der Daten, d.h. Datenfluss, Datenaustausch und Datenverwaltung werden unabhängig vom Kontrollfluss in Service-Choreographien modelliert. Um eine durchgängige Unterstützung für die Modellierung und Ausführung datenbewusster Choreographien zu bieten und die jeweiligen Phasen des eingeführten Lebenszyklus zu unterstützen, wird ein übergreifendes TraDE-Ökosystem eingeführt und prototypisch implementiert. Dieses umfasst eine Modellierungsumgebung und die erforderliche Laufzeitumgebung zur Ausführung solcher datenbewusster Choreographien durch eine neue TraDE-Middleware und deren Integration in entsprechende Business Process Engines (BPE). Das inhärente Ziel dieser Arbeit ist es, die Modellierung von Daten und deren Austausch in Choreographie-Modellen zu vereinfachen und gleichzeitig deren Laufzeitflexibilität zu erhöhen und so den transparenten Austausch und die Transformation von Daten während der Ausführung einer Service-Choreographie zu ermöglichen und zu vereinfachen.

Acknowledgements

First of all, I would like to thank my supervisor Prof. Dr. Dr. h. c. Frank Leymann for giving me the opportunity to work at the Institute of Architecture of Application Systems (IAAS) and do the research culminating in this work as well as for his great support, guidance and advice. I always enjoyed our discussions and really appreciated the freedom given by the trust placed in me and my work.

Many thanks also to Prof. Dr. Dimka Karastoyanova for all her support and guidance in doing research in general and in writing papers in particular as well as shaping the topic of this work and for being the second supervisor of my thesis.

I would also like to thank apl. Prof. Dr.-Ing. Jörg Fehr from the Institute of Engineering and Computational Mechanics for acting as a second co-examiner and for giving me additional insights in another application domain for service choreographies in eScience.

During my time at the IAAS I learned a lot, not only technically also in terms of personal development. Therefore, I would like to thank all my colleagues at the institute for all the fruitful discussions and their support. Special thanks go to Andreas Weiß and Vladimir Yussupov for being great office-mates and always having time for discussing new ideas and concepts as well as providing helpful comments on papers. I would also like to thank

Dr. Oliver Kopp for his support and for never stopping to ask me about the status of this work and therefore introducing additional pressure to finish it. Moreover, special thanks go to Sebastian Wagner for proof-reading and providing helpful comments on this thesis. Finally, I would like to thank my family for their patience and support throughout the whole endeavor of writing this thesis.

Chapter 1

Introduction

Service-oriented Architectures (SOA) [Pap03] have seen widespread adoption. The concept of composing self-contained units of functionality – services – over the network has found application in many research areas and application domains [Zim16], e. g., in Business Process Management (BPM) [LKP10; LRS02; Wes12], Cloud Computing [Ley09; QLDG09], the Internet of Things (IoT) [MSDC12], eScience [HTT+09] and in particular scientific workflows [BG07; TDGS07]. The goal of eScience is to provide IT support for scientists from different scientific fields in order to enable faster scientific exploration and discovery. Therefore, generic approaches and tools are required which support scientists throughout the whole life cycle of their computer-based experiments, simulations and scientific calculations, from data collection and curation, through data processing and analysis to permanent archiving in digital repositories for preservation [HTT+09].

To compose multiple existing services into service compositions a huge variety of modeling languages evolved. These languages can be divided into two general groups each following a different paradigm: service orchestrations and service choreographies. Service orchestrations, also known as workflows or processes, are modeled from the viewpoint of one party that

acts as a central coordinator [DKB08]. The most prominent orchestration modeling languages are the Business Process Model and Notation (BPMN) [BPMN] and the Business Process Execution Language (BPEL) [BPEL] which provide means to specify orchestration-based service compositions, where BPMN also supports the modeling of service choreographies. In contrast to service orchestrations, service choreographies provide a global perspective of the potentially complex conversations between multiple interacting services without relying on a central coordinator. Each party that takes part in the collaboration, as a so-called participant, is able to model its conversations with other involved parties by specifying corresponding message exchanges with them [BDH05; DKB08]. Therefore, participants of a choreography communicate in a direct, peer-to-peer manner without requiring a central coordinator. Since most choreography modeling languages are typically not directly executable a common approach is to transform and refine the choreography participants to a collection of executable workflows that implement the specified choreography [AW01; DKL+08]. To model service choreographies, modeling languages such as BPMN or BPEL4Chor [KLW11] can be used. Decker et al. [DKLW09] provide an overview and comparison of the most prominent choreography modeling languages.

The notion of *workflows* was originally introduced in the area of BPM as a new information processing technology which enables the implementation and automation of business processes as well as easing their adaptation to changing requirements [LR00]. Instead of the term *workflow* often also the more generic term *process* is used in literature. Depending on the underlying definition, there can be slight differences between those terms, however, they are used as synonyms within this work. Workflows provide the means to specify a process based on a set of activities and the order in which they need to be performed. Therefore, a workflow can be seen as a directed graph, where the nodes are the activities or tasks to perform and the edges specify the possible paths between them, also known as control flow. The logic behind the activities themselves is not restricted to a certain technology and can be provided in various manners, e. g., through scripts, applications or services based on the capabilities of the used process modeling notation

and Business Process Engine (BPE) that conducts the modeled workflows.

The concept of workflows is also successfully applied to provide the required IT support for eScience. While for the *conventional workflow technology* in the domain of BPM, a lot of standards and corresponding tools evolved, such as BPMN or BPEL, in the domain of eScience (*scientific workflow technology*) a landscape of different, sometimes domain-specific, Scientific Workflow Management Systems (sWfMSs) were developed, where each of the sWfMSs comes with its own workflow modeling notation [YB05]. The most prominent examples for such sWfMSs are Pegasus [DSS+05], Kepler [ABJ+04], Triana [TSWH07] and Taverna [OAF+04]. The use of standardized notations for the modeling and execution of conventional workflows provides several advantages such as a broad variety of existing, ready-to-use tools, portability of the models, fault handling, forward and backward recovery or monitoring of the workflow execution. To transfer these capabilities to scientific workflows, conventional workflow technology is successfully applied to automate computer-based experiments and scientific calculations in eScience [BG07; GSK+11; SKD10; Slo07; WEB+07]. Various works identified and introduced required functionality to tackle the special challenges of scientific workflows when using conventional workflow technology, e. g., to support the modeling of workflows in a trial-and-error manner or to introduce flexibility concepts to steer simulations during runtime [BG07; GDE+07; SK10; SK11; SK13; Slo07; WEB+07].

Service choreographies have been also already successfully applied to capture collaborations from a global perspective without the necessity to directly specify technical details in both the business [DKB08; MPB+15] and eScience domain (*simulation choreographies*) [BWR09; WAHK15b; WK14a; WK14b; WKMS14]. The efficient exchange of data between the composed services is a crucial factor not only in classical data-centric domains like eScience. With the advent of the fields Big Data and the IoT, the importance of data in terms of its business value and as a driver for gaining advantages over competitors is also increasing in the domain of BPM significantly. The impact of this development on the domain of BPM has already been documented [MSMP11; SMM+14]. Therefore, in recent years there has been

a convergence of approaches from BPM and data-intensive domains such as eScience, and both domains will be able to benefit from future research advances in the other domain.

Through our experience in the fields of BPM and eScience, and based on existing literature, we argue that conventional workflow technology needs to reflect the paradigm shift to data-awareness and provide support for the efficient integration and exchange of heterogeneous data through a central role in the BPM life cycle. However, the current state of the art in service choreographies, despite some promising works trying to improve data-awareness [KPR12a; LN11; MPB+15] and also showcasing performance benefits [BCF06; BW+12; BWV08b; LLW02], fails to provide an adequate solution that allows data to assume its deserved primary role.

To tackle this problem and account for the crucial importance of data in service choreographies, the traditional BPM life cycle has to be extended with data management capabilities. This will enable the automation of further steps of the transformation and refinement of choreography models to executable process models. While most of the existing approaches only utilize the performance benefits from decentralizing the data flow, we want to provide further improvements throughout all life cycle phases and especially during choreography modeling. The inherent goal of this work is to introduce data as a first-class citizen already at the level of the choreography model, to enable the decoupling of data flow, data exchange and data management from the control flow in choreography and process models. This decoupling enables a more flexible exchange and management of data and will support modelers with the data dimension of their service compositions. To execute the resulting *data-aware choreography models*, a middleware layer is required that carries out the exchange of data between the participants of the modeled choreography in an efficient and transparent manner. The ultimate goal is to provide an end-to-end approach and a respective ecosystem of tools and middleware components which supports the modeling and execution of data-aware choreographies. Therefore, the approaches and methods introduced within this work shall remain independent from the various existing modeling languages for choreography and process model specification.

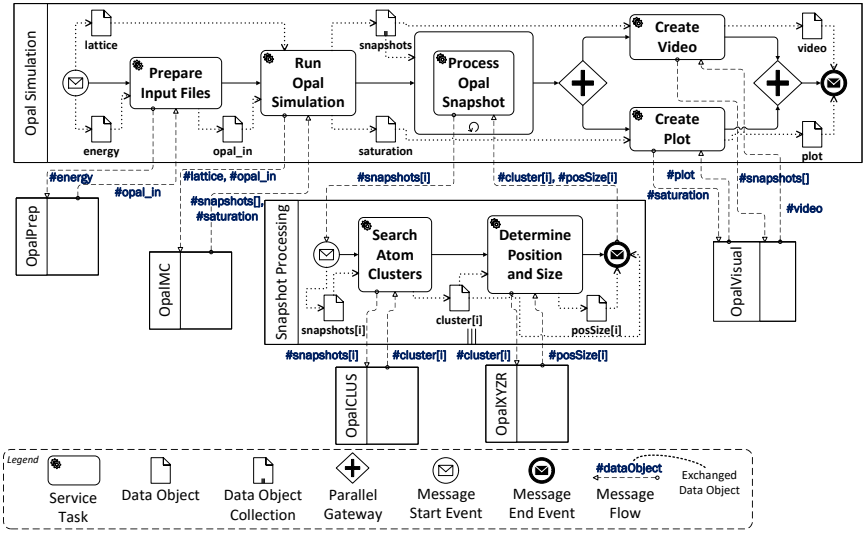

Figure 1.1: Example simulation choreography conducting a thermal aging simulation from material science domain (based on Weiß et al. [WKMS14]).

1.1 Motivation Scenario

To further illustrate and motivate data-aware choreography models, we present in the following a scenario from the domain of material science [BS03; MMC+12; WKMS14]. Figure 1.1 shows a choreography model of a Kinetic Monte Carlo (KMC) simulation using the custom-made simulation software *Ostwald ripening of Precipitates on an Atomic Lattice* (OPAL) [BS03]. The choreography model is expressed in form of a BPMN 2.0 collaboration model. OPAL simulates the formation of copper precipitates, i. e., the development of atom clusters, within a lattice due to thermal aging. The choreography model specifies the conversations between the seven choreography participants which constitute the overall simulation. The *Opal Simulation* and *Snapshot Processing* participants are implemented through corresponding process models and control the conversations between the

other five participants through corresponding service invocation tasks, indicated by small gear wheels in the upper left corner of a task in Figure 1.1. The consumed and produced data is represented through BPMN data objects which are only accessible within the scope of the respective participant. To exchange data between participants, corresponding message exchanges have to be introduced which are depicted as dashed lines in Figure 1.1. The labels on the dashed lines in Figure 1.1 visualize which data objects of the two process-based participants are exchanged during the modeled conversations in form of message payloads. The other five participants are implemented as web services which wrap the different modules of the OPAL simulation software to enable their composition through process models. The wrapping of the OPAL simulation software as services is described in detail by Sonntag et al. [SHK+11]. In the following, we will shortly introduce how the overall simulation is conducted and then describe the shortcomings of the state of the art in service choreographies to motivate the rest of the work.

Whenever the message start event of the *Opal Simulation* participant receives a new request message, a new simulation instance is created. The initial request contains a set of parameters such as the atomic lattice, energy parameters and the number of snapshots to take. The *Prepare Input Files* service task sends these parameters to the *OpalPrep* participant which calculates the energy configuration and replies all required input parameters (opal_in) for the KMC simulation. The *OpalMC* participant is invoked by the *Run Opal Simulation* task to start the actual KMC simulation. According to the specified number of snapshots in the initial request, the *OpalMC* participant saves the current state of the atom lattice at a particular point in time in a snapshot and responds all snapshots together with the saturation data back to the *Opal Simulation* participant. The snapshots are then searched for atom clusters and the position and size of each identified cluster is calculated. Therefore, each snapshot is analyzed by a separate instance of the *Snapshot Processing* participant created by the *Process Opal Snapshot* service task which is contained in a loop task as shown in Figure 1.1. Each *Snapshot Processing* instance invokes first the *OpalCLUS* participant (*Search Atom Clusters* task) with the current snapshot to identify clusters and then uses these clusters

to determine their position and size by invoking the *OpalXYZR* participant (*Determine Position and Size* task). The resulting cluster, position and size data are replied back to the *Opal Simulation* participant. Finally, a video of animated 3d scatter plots of the simulation snapshots (*Create Video* task) and a 2d plot of the saturation function of the precipitation process are created (*Create Plot* task) by the *OpalVisual* participant.

The modeling and execution of the choreography model depicted in Figure 1.1 with state-of-the-art choreography modeling notations and BPEs has from our viewpoint several drawbacks related to data management and exchange. Since the data flow between participants is bound to the exchange of messages, data is unnecessarily routed through several participants instead of directly exchanged in a peer-to-peer manner. For example, the resulting snapshot data is sent from the *OpalMC* participant to the *Opal Simulation* participant and then later forwarded to the *Snapshot Processing* participant from where it is ultimately sent to the *OpalCLUS* participant which actually processes the data. Decoupling data flow from message exchanges, or choreography control flow in general, allows an easier modeling as well as provides more flexibility and further optimization possibilities of the data flow during choreography runtime. Potentially, data can be pro-actively transferred between participants as soon as it is available, independent from the choreography execution, based on knowledge gained from the underlying models, e. g., by analyzing their data dependencies, or monitoring data from previous choreography executions. For example, the snapshot data in Figure 1.1 is available as soon as the *Run Opal Simulation* task is completed and can be therefore directly transferred from the *OpalMC* participant to the *OpalVisual* participant instead of waiting until the *Create Video* task triggers the corresponding message exchange which normally transfers the snapshot data. Introducing data already at the level of choreographies and enabling the definition of explicit data exchange between choreography participants and their activities also reduces the manual refinement effort when transforming and refining the choreography models to executable process models following the Public-to-Private approach introduced by van der Aalst and Weske [AW01]. There, the specified data and data exchanges can be di-

rectly used for automatically generating respective data definitions and data flows within the resulting process models. This will be discussed in detail in Chapter 3. Furthermore, with respect to the presented eScience simulation example, the entry barrier for scientists will be lowered to specify their complex simulations potentially involving and combining different scientific fields or simulation approaches in form of choreography models [WK14b]. From our viewpoint, choreographies are better suited than process models to graphically model complex interactions between multiple components of a simulation. The main reason for that is that choreography models provide means to specify and visualize the big picture of a service composition from an overall global perspective in a technology-agnostic and independent manner. Since the focus is not on defining directly executable models, the specification of technical details regarding the composition of services can be postponed to a later time. Therefore, choreographies allow to specify the components of a simulation, how these components communicate with each other and what data will be produced and consumed by each simulation component. From a generic BPM viewpoint, the same holds, i. e., choreography models provide means to specify and visualize potentially complex conversations of multiple collaborating parties from a global viewpoint.

Despite the modeling dimension, especially in the eScience domain also choreography execution has some drawbacks related to data management and exchange. For example, BPEs are often not designed for or even capable of handling large amounts of data and therefore may become a performance bottleneck when using them for executing simulation choreographies [GSK+11]. Therefore, within this work, we want to introduce concepts and a corresponding middleware, called *TraDE Middleware*, to reduce the workload and storage overhead of BPEs used in data-intensive domains such as eScience by outsourcing the storage and exchange of larger data volumes from the BPEs to the TraDE Middleware. The goal is that only data that is actually required for the navigation of the process model instances will be stored by a BPE. Furthermore, scientists require an easy and intuitive way to provide and access data of their simulations independent of the lifetime of the simulation instances. This means, that simulation input data can be

provided before the simulation is started, and intermediary and final results are still accessible after the execution of the simulation is completed. To avoid the problem of an uncontrolled growth of data, related strategies for garbage collection are discussed in Chapters 4 and 5 as part of our formal model and the TraDE Middleware. In summary, the TraDE Middleware will enable scientists to upload and provide simulation input data, inspect intermediary results during the execution of a simulation or reuse data from previous simulation runs by simply providing a reference to it. The main goal is to decouple the life time of the data from the choreography and therefore simulation life time with the help of the TraDE Middleware, i. e., data specified within a data-aware service choreography will be globally accessible through the TraDE Middleware, e. g., over a Web user interface. Therefore, the TraDE Middleware handles the data exchange automatically and transparently between the collaborating parties in the background.

For example, regarding the OPAL simulation choreography presented in Figure 1.1, scientists are able to provide the initial lattice data by uploading it once to the TraDE Middleware instead of sending it multiple times encapsulated in the initial request message to the *Opal Simulation* participant. Furthermore, if a scientist wants to reuse the same lattice for multiple simulation runs, it does not make sense to send the same data multiple times over the network, if this is not really required for other reasons. The same applies to the resulting video, here the video is automatically transferred from the OpalVisual participant and can then be downloaded from the TraDE Middleware by a scientist, even after the simulation has been completed.

1.2 Research Contributions

Figure 1.2 shows the research contributions of this work which are introduced in more detail in the following.

Contribution 1: Life cycle of data-aware service choreographies

As outlined in the introduction, this contribution introduces an extended business process management life cycle for choreographies where data is

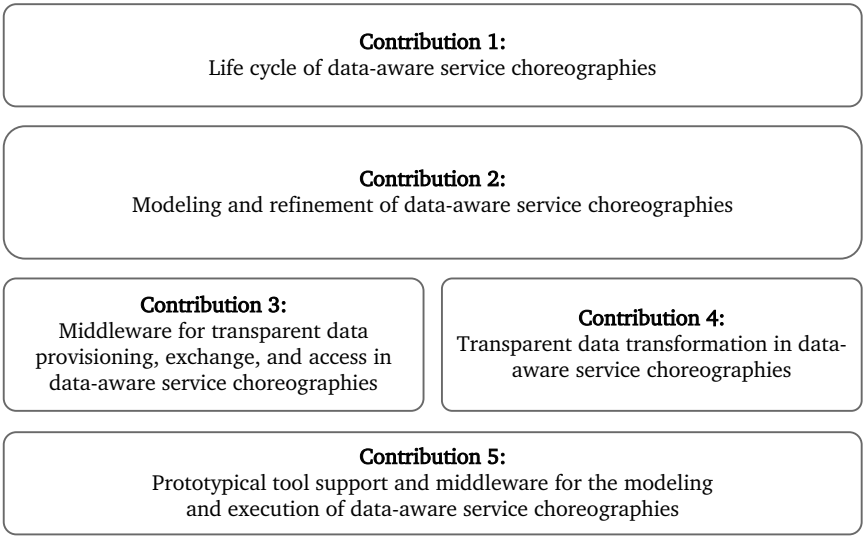

Figure 1.2: Overview of contributions provided in this work.

treated as a first-class citizen by introducing and applying our concepts for data-aware choreographies. Therefore, each of the life cycle phases and its data-related aspects are introduced and described. The goal is to provide a holistic support of data-related aspects throughout the whole life cycle while utilizing within each life cycle phase the knowledge of the involved stakeholders, e. g., domain experts or choreography and process modelers. Furthermore, the life cycle provides an overview and initial requirements regarding required tooling support (e. g., modeling environments) and middleware systems.

Contribution 2: Modeling and refinement of data-aware service choreographies

One of the core contributions of this work is to support and improve the level of data-awareness for choreography modeling and execution. Accordingly, this contribution provides the required support for the enhanced modeling

and refinement life cycle phases based on Contribution 1 by introducing our concept for *Transparent Data Exchange* (TraDE) in service choreographies. This comprises a general introduction of related modeling constructs as well as a complete formal model for data-aware choreographies and their semi-automated transformation and refinement to a collection of executable process models. The underlying goal is to maintain the independence of our conceptual approach and introduced methods from the various existing modeling languages for choreography and process model specification.

Contribution 3: Middleware for transparent data provisioning, access, and exchange in data-aware service choreographies

This contribution provides the required support for the deployment and execution life cycle phases of data-aware choreographies introduced by Contribution 1. Therefore, the underlying concepts of the TraDE Middleware are presented which enables the transparent execution of cross-partner data flows modeled within a data-aware choreography. The actual execution of a data-aware choreography model is based on the generated refined collection of executable process models with our TraDE concepts applied, based on the formal model introduced by Contribution 2. Therefore, the underlying architecture and conceptual model of the TraDE Middleware is presented and its integration with a respective BPE is described.

Contribution 4: Transparent data transformation in data-aware service choreographies

Data exchange and processing in distributed scenarios such as choreographies always introduces data transformation requirements, e. g., to alleviate differences in data formats or splitting and aggregation of data of different participants. This is even more important in data-intensive domains such as eScience. There, large amounts of data in unstructured or at least semi-structured proprietary file-based formats have to be transformed. In addition, data transformations are use case specific and go far beyond simple format translations or data aggregations in eScience.

This contribution presents a transparent data transformation approach, which is build on top of the TraDE concepts introduced by Contribution 2, to enable the specification and modeling of required data transformations already at the level of a choreography model. Despite the modeling, the seamless integration of data transformation logic into the choreography runtime environment is of major importance to enable the transparent execution of data transformations during choreography runtime. Therefore, the focus of this contribution is on the modeling as well as on the integration and transparent execution of heterogeneous data transformation logic provided by experts, e.g., the scientists who also specified the simulation choreography, to support the required transformation of use case specific and unstructured data based on the data transformation logic implemented by an expert.

Contribution 5: Prototypical tool support and middleware for the modeling and execution of data-aware service choreographies

The last contribution of this work is a proof-of-concept of Contributions 1 to 4 for data-aware choreographies and middleware components for transparent data exchange and transformation. Therefore, the main components of our prototypical TraDE ecosystem, namely the data-aware Choreography and Orchestration Modeling Environment, a TraDE-aware Process Engine, the TraDE Middleware, the Data Transformation (DT) Integration Middleware, and a TraDE Web UI are introduced and described. All these components together provide the required support for the respective phases – from modeling to execution – of the life cycle introduced by Contribution 1. Furthermore, the modeling and execution of data-aware choreographies is validated through a case study from the eScience domain. The prototypical implementation of the overall TraDE ecosystem is evaluated through a performance evaluation comparing the execution of a choreography model with and without the TraDE concepts applied.

1.3 Scientific Publications

The following peer-reviewed publications have been published at journals and conferences based on the research conducted in the context of this work.

- M. Hahn, D. Karastoyanova, and F. Leymann. “Data-Aware Service Choreographies through Transparent Data Exchange.” In: *Proceedings of ICWE’16*. Vol. 9671. Lecture Notes in Computer Science (LNCS). Springer International Publishing, 2016, pp. 357–364.
- M. Hahn, D. Karastoyanova, and F. Leymann. “A Management Life Cycle for Data-Aware Service Choreographies.” In: *Proceedings of ICWS’16*. IEEE Computer Society, 2016, pp. 364–371.
- M. Hahn, U. Breitenbücher, O. Kopp, and F. Leymann. “Modeling and Execution of Data-Aware Choreographies: An Overview.” In: *Computer Science - Research and Development* (2017), pp. 1–12.
- M. Hahn, U. Breitenbücher, F. Leymann, and A. Weiß. “TraDE – A Transparent Data Exchange Middleware for Service Choreographies.” In: *Proceedings of OTM 2017 Conferences*. Ed. by H. Panetto, C. Debruyne, W. Gaaloul, M. Papazoglou, A. Paschke, C. A. Ardagna, and R. Meersman. Vol. 10573. Lecture Notes in Computer Science (LNCS). Springer International Publishing, 2017, pp. 252–270.
- M. Hahn, U. Breitenbücher, F. Leymann, M. Wurster, and V. Yussupov. “Modeling Data Transformations in Data-Aware Service Choreographies.” In: *Proceedings of EDOC’18*. IEEE Computer Society, 2018, pp. 28–34.
- M. Hahn, U. Breitenbücher, F. Leymann, and V. Yussupov. “Transparent Execution of Data Transformations in Data-Aware Service Choreographies.” In: *Proceedings of OTM 2018 Conferences*. Vol. 11230. Lecture Notes in Computer Science (LNCS). Springer International Publishing AG, 2018, pp. 117–137.

1.4 Structure of the Document

After the general introduction of the work and its underlying research contributions within this chapter, the rest of the document is structured as follows. Chapter 2 introduces the foundations of this work and presents the state of the art in research in the context of the presented contributions.

Based on that, in Chapter 3 our data-aware choreography methodology is presented which includes our TraDE approach as well as its integration into the traditional BPM life cycle [Wes12] providing a life cycle for data-aware choreographies (cf. Contribution 1).

Chapter 4 introduces a formal model for data-aware service choreographies together with a graphical notation to provide means for the modeling of data-aware service choreographies (cf. Contribution 2) as well as their transformation and refinement to executable process models.

In Chapter 5, the concept underlying the TraDE Middleware is introduced which provides the required runtime support for the execution of data-aware service choreographies based on our TraDE concepts (cf. Contribution 3). To further support modelers in specifying data-related aspects at the level of choreography models, Chapter 6 presents a concept for the modeling and execution of transparent data transformations within choreography models (cf. Contribution 4) as well as a supporting data transformation middleware.

In Chapter 7, the resulting prototypical implementations of all the components introduced in the previous chapters are described (cf. Contribution 5). They form together the TraDE ecosystem that provides an end-to-end support for the modeling and execution of data-aware choreographies by supporting the respective phases of the extended BPM life cycle presented in Chapter 3.

A validation and evaluation of the TraDE concepts and their prototypical implementation is presented in Chapter 8. The validation is provided in form of a case study from the eScience domain which describes and discusses the applicability and use of the presented TraDE concepts to ease modeling and provide additional runtime support regarding data-related aspects such as data exchange and data transformations. In addition, a performance evaluation of the prototypical implementation of the TraDE Middleware is

presented which compares the execution of a choreography model with and without our TraDE concepts applied.

Chapter 9 concludes the work and gives an outlook to future work.

Chapter 2

Background and Related Work

This chapter provides an overview on the background and state-of-the-art in research underlying this work. Therefore, in Section 2.1 a short overview on service compositions is presented. This includes a discussion of their specification following the orchestration or choreography paradigm as well as related modeling languages and their execution aspects. Section 2.2 presents and discusses related work regarding data-awareness in service compositions for both service choreographies and orchestrations. In addition, Section 2.3 presents some alternative modeling and execution approaches with focus on data which are based on alternative modeling paradigms and technologies. Finally, we discuss different approaches for integrating and utilize heterogeneous data transformation logic in service compositions in Section 2.4. Since the ability to transform data is one of the core requirements and aspects of data management in general and therefore also of major importance for introducing concepts for the modeling and execution of data-aware choreographies within this work.

2.1 Service Compositions: Paradigms, Modeling Languages and Execution Aspects

With the advent of Service-oriented Architectures (SOA) [Pap03] as a new architectural style, the concept of services – self-contained units of functionality – is introduced. Base on that, SOA furthermore enables the combination of such services over the network, to create higher-level service compositions [Pel03]. As outlined in Chapter 1, in the domain of Business Process Management (BPM) a variety of modeling languages evolved to specify and model such service compositions by either following the *orchestration* or *choreography* paradigm. Before we will have a closer look into the choreography paradigm in Sections 2.1.1 and 2.1.2, we first provide an overview on both paradigms by discussing their main aspects and their differences.

Service orchestrations, specified in form of workflows or processes [LR00], are modeled from the viewpoint of one party that acts as a central coordinator [DKB08]. Therefore, an orchestration always represents the control, i. e., business logic and task execution order, from one party's perspective required to reach a certain goal, e. g., to book a flight [Pel03]. The involved services which are orchestrated to reach this goal are treated as black boxes and their invocation is represented via respective activities as part of the underlying process model. The most prominent orchestration modeling languages are the Business Process Model and Notation (BPMN) [BPMN] and the Business Process Execution Language (BPEL) [BPEL].

In contrast, service choreographies are modeled from a global perspective without relying on a central coordinator. A choreography represents the potentially complex interactions between the services of multiple, independent parties to achieve an overall goal. Each party that takes part in the collaboration is represented as a so-called participant. The interactions between the services of the involved parties are specified through message exchanges between the choreography participants [BDH05; DKB08]. The resulting set of message exchanges between two or more participants is called a conversation. Therefore, participants of a choreography communicate in a direct, peer-to-peer manner without requiring a central coordinator. To

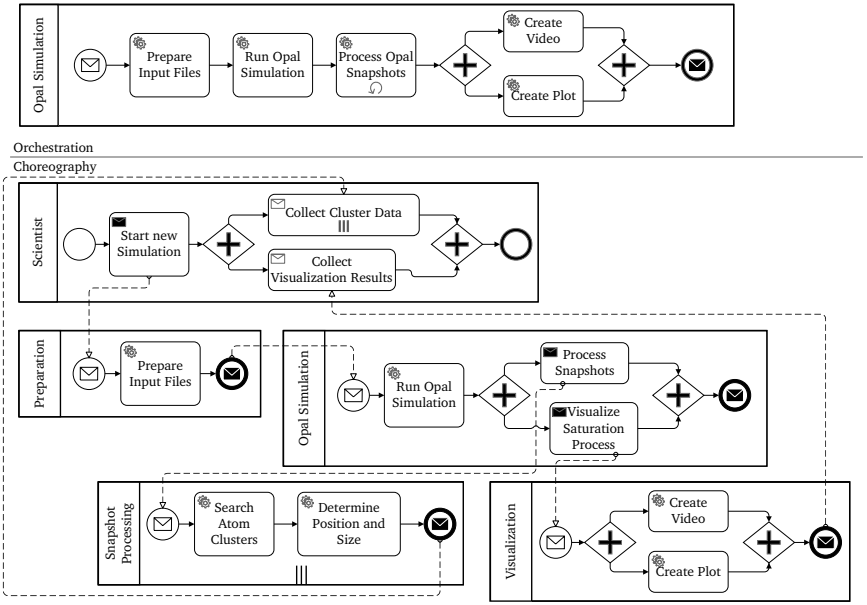

Figure 2.1: Visualizing the orchestration and choreography paradigm based on the eScience simulation example from Section 1.1.

model service choreographies, modeling languages such as BPMN [BPMN] or BPEL4Chor [DKLW07; KLW11] can be used. Decker et al. [DKLW09] or Kopp [Kop16] provide an overview and comparison of the most prominent choreography modeling languages.

While the goal of service orchestrations is to specify executable business processes, the goal of service choreographies is on providing a global view on the potentially complex conversations of multiple interacting parties. Based on that service choreographies are usually not directly executable, e. g., since they lack required technical aspects such as the to be used transport protocol or message format and encoding for conducting the modeled conversations.

Figure 2.1 shows the eScience simulation presented as a motivation example in Chapter 1, specified as an orchestration as well as a choreography using the modeling notations of the BPMN [BPMN] standard. Therefore, the

orchestration is modeled as a BPMN 2.0 process model and the choreography is specified in form of a BPMN 2.0 collaboration model. In our opinion, the core difference between orchestrations and choreographies outlined above, becomes directly clear when visually comparing the example models depicted in Figure 2.1. The orchestration, i. e., the process model, just specifies a sequence of service tasks invoking respective services which are black boxes and therefore not further detailed as part of the model. From an orchestration perspective it is therefore unclear how the conversation-related internal behavior of the services look like regarding the processing of the request message and how the response is formed. This hides also the knowledge if the service itself is again an orchestration or not. In contrast to that, following the choreography paradigm such conversation-related behavior is made visible as part of the choreography model. This allows to specify the interconnections between the collaborating parties and therefore provides the underlying big picture of the service composition enabling each party to be aware of its role and potential implications on their conversations. For example, by introducing the scientist as a participant to the choreography model depicted in Figure 2.1, it is directly clear that the simulation input parameters have to be send to the *Preparation* participant for processing and afterwards the simulation results can be collected by receiving respective messages from the *Snapshot Processing* and *Visualization* participants. By following the modeled message flows, i. e., conversations, between the participants, the overall Opal simulation and its building blocks as well as their interconnection becomes instantly clear in our opinion. Since all such knowledge is abstracted away and moved into the invoked services at the level of an orchestration, it is way more complex or sometimes even impossible to get such an underlying big picture by studying the specified process models.

2.1.1 Choreography Modeling

For the specification of choreography-based service compositions two categories of modeling approaches exist as shown in Figure 2.2: *interaction models* and *interconnected interface behavior models*, or *interconnection models* for

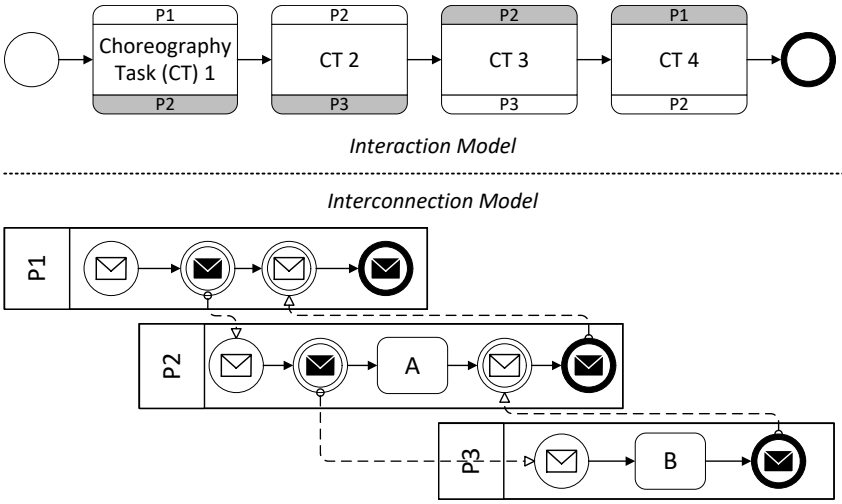

Figure 2.2: Visualizing the interaction and interconnection choreography modeling approaches based on an example specified as BPMN 2.0 choreography and collaboration model.

short [DKB08]. The former enable the specification of interactions between participants from a global perspective only through message exchanges and behavioral dependencies between the interactions. The most prominent examples for modeling languages following the interaction modeling approach are Web Service Choreography Description Language (WS-CDL) [WS-CDL], *Let's Dance* [ZBDH06] or BPMN choreographies [BPMN]. The downside of such interaction models is, that they might be *locally unenforceable* [DKB08]. This is the case, if an interaction model defines behavioral constraints that cannot be enforced locally, i. e., the individual participants cannot guarantee that they are able to fulfill the specified constraints while conducting their message exchanges as defined within the interaction model [ZDH+08]. This local unenforceability results from the fact that individual participants do not know the state of interactions they are not involved in.

The latter, interconnection models allow the specification of control flow per participant and the interaction between the participants through the

specification of message connectors [DKL+08; KLW11]. As shown in the example depicted in Figure 2.2, the BPMN collaboration model following the interconnection modeling approach specifies respective intermediate message throw and catch events within the participants as well as message flows between them. In addition, further control flow constructs such as tasks may be specified at the level of the participants which may have an impact on the overall choreography or are relevant for the other involved parties. The most prominent examples for modeling languages following the interconnected interface behavior, or interconnection modeling approach for short, are BPMN collaborations [BPMN] and BPEL4Chor [DKLW07]. Since the control flow of each participant is explicitly modeled, unenforceability issues cannot arise. However, such models might be incompatible, so that the different participants do not interact correctly at the level of the resulting process models. For example, deadlocks are one of the typical incompatibilities of interconnection models [DKLW09].

Since the inherent goal of this work is to introduce data already at the level of the choreography models to enable the decoupling of data flow, data exchange and data management from the control flow in choreography models, the internal behavior of participants is one of the relevant aspects for introducing data-aware choreography models. However, interaction models do not allow to specify such internal behavior or control flow of participants [KLW11] and therefore we use the interconnection modeling approach as a basis for the contributions presented within this work.

2.1.2 Choreography Execution

Since all standards-based choreography modeling languages do not support the specification of directly executable choreography models [DKB08; KEL+11], a common approach is to transform the specified choreographies to a collection of process models [ALM+08; AW01; DKLW09]. van der Aalst and Weske [AW01] introduce this concept at the level of inter-organizational workflows via the *Public-to-private* (P2P) approach. In relation to this approach, a choreography model can be seen as the *public model* specifying

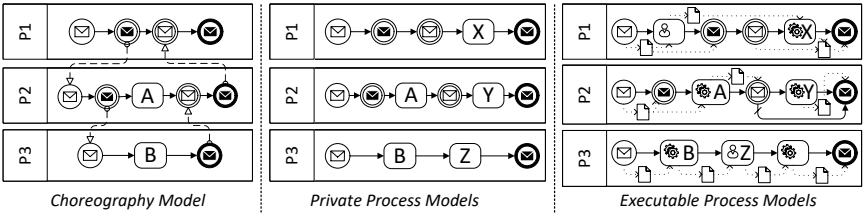

Figure 2.3: The P2P approach translated to choreography models by example

the overall collaboration contract between the interacting parties [AW13]. Such a public model is then transformed into a collection of *private process models* specifying the internal logic of each choreography participant (such as abstract BPEL [BPEL]) while implementing the conversations defined at the level of the choreography model. This approach is used as a basis for the execution of choreography models using private process models within this work and therefore a summary of how the P2P approach is applied to a choreography model is provided in the following.

Figure 2.3 shows an example for applying the P2P approach to a choreography specified as BPMN collaboration model. The resulting private process models are specified as BPMN process models. For each of the modeled choreography participants *P1*, *P2* and *P3* a private process model is generated based on the specified participant's control and data flow as shown in the middle of Figure 2.3. Since the resulting private process models lack required details, such as actual process logic in addition to the specified conversations as well as technical details, they are also not directly executable.

For example, a task type has to be specified for the modeled tasks *A* and *B* shown in Figure 2.3. Required technical details are, for example, the to be applied transport protocol and message encoding for the specified message exchanges or required runtime environment configuration data for successful deployment of the refined process models to respective Business Process Engines (BPEs). Therefore, a manual refinement takes place which uses the private process models as input and produces the executable process models depicted on the right of Figure 2.3 as output. In addition, during the

refinement the process models can be enriched and extended with additional logic that was not relevant from a collaboration perspective but is required to enable an enactment of the choreography, or with participant internal control and data flow that should not be unveiled to the other collaborating parties, e. g., the invocation of internal services.

Such extensions can range from refining already specified modeling constructs or introducing completely new ones. In the example shown in Figure 2.3, new tasks of different types are introduced within each of the three participants and the existing *A* and *B* are refined to service tasks as well as required properties are specified, e. g., the endpoints of the services to be invoked by the task. Finally, the resulting collection of executable process models shown on the right of Figure 2.3 implement the specified choreography model provided as input to the P2P approach, i. e., by executing the process models the overall choreography is conducted. A detailed example how the application of the P2P approach looks like is presented by Reimann, Kopp, et al. [RK+08] using BPEL4Chor as choreography modeling notation and BPEL for the generated private process models.

2.2 Data-Awareness in Service Compositions

In this section, we introduce and discuss related works that provide contributions towards improving data-awareness in service choreographies. The focus is on the modeling and exchange of data regarding our goal of decoupling data flow from control flow and the exchange of messages in service compositions towards introducing data-aware choreography models.

The model-driven approach presented by Meyer et al. [MPB+15] supports the modeling and enactment of data exchange in choreographies using messages. The authors propose an extension of the BPMN modeling language by introducing annotations on BPMN data objects. These annotations are then automatically transformed into SQL queries to specify and enact message extraction from and message storage to local databases. To model and enact data transformations between messages and local data they refer to standard

data query languages, e. g., XML Query Language (XQuery) [XQuery]. This enables the complete automation of data exchange between participants and the enrichment of the model transformations with data-related aspects from the global choreography to the local process model level.

We fully support the authors arguments that the collaborating partners should specify the exchanged data and its structure in a commonly agreed global data model already at the level of a choreography model [MPB+13; MPB+15]. This indeed strengthens the collaboration contract by enhancing it with the data dimension and therefore ensuring correct data exchange in addition to enforcing the specified conversations, i. e., order of messages.

However, our goal is to provide generic concepts and methods that can be applied to different modeling languages and are therefore not directly bound to one specific language such as BPMN. In addition, instead of directly binding data objects to databases at the level of the choreography models, our approach introduces cross-partner data objects and corresponding runtime support in form of the TraDE Middleware as an abstraction layer. The TraDE Middleware can therefore be seen as a data hub from the viewpoint of the interacting choreography participants. Arbitrary data sources can then be connected to the TraDE Middleware to enable the provisioning of data for respective cross-partner data objects using existing generic data provisioning concepts such as the pluggable data management framework SIMPL [Rei17; RRS+11]. This allows us to support also other domains where data is not necessarily stored in databases by default, e. g., in the eScience domain, data is commonly stored and exchanged through files in different formats.

Moreover, our goal is to decouple the exchange of data from the exchange of messages to simplify modeling and improving runtime flexibility of choreographies regarding their data perspective by supporting explicit data flow between choreography participants without the need to exchange messages.

Knuplesch, Pryss, and Reichert [KPR12a; KPR12b] introduce the notion of data-aware process interaction models as a means to model data-aware choreographies. Their goal is to enrich interaction models with a data perspective as a means to explicitly consider data being exchanged through messages between participants and also used for routing decisions while

ensuring correctness of the resulting models. Therefore, they define a formal framework and specific correctness criteria for *Data-Aware Choreographies* (DAChor) while their behavior is described using elements of *Interaction Petri Nets* and *Workflow Nets with Data*.

Although the authors concentrate on the modeling and correctness of data-aware choreographies, our focus is not only on introducing data as a first-class citizen during modeling. Moreover, we want to introduce data-awareness in an end-to-end manner towards increasing runtime flexibility while reducing the complexity of modeling data flows in service choreographies. Similar to our cross-partner data objects, the authors introduce so-called *virtual data objects*. However, the exchange of actual business data is still message-based and virtual data objects only allow to share states across participants to support data-based routing decisions.

Barker, Walton, and Robertson [BWR09] define a new language called *Multiagent Protocols* (MAP) which allows to specify directly executable service choreographies. For the enactment of respective MAP choreographies an open source framework called *MagentA* and the concept of *peers* is introduced. A peer provides extra functionality through a *choreography interface* that enables web services to participate in a choreography without requiring to adapt the underlying service implementations.

Additionally, Barker, Walton, and Robertson [BWR09] present arguments in which scenarios it is beneficial to apply the choreography paradigm to compose services. One such case is the enactment of data-intensive workflows which are very common in the domain of eScience where scientific workflows are used to conduct data-intensive simulations or scientific calculations. The exchange of huge amounts of intermediate data between the composed services through a centralized workflow engine results in potentially unnecessary data transfer, congestion of the engine, and as a result decreases the overall performance of the scientific workflow. Therefore, the authors suggest to model such data-intensive service compositions as choreographies using their MAP language to model and conduct the data exchange in a distributed, peer-to-peer manner directly between the composed services.

Instead of introducing a completely new modeling notation, we rely on

existing choreography notations like BPMN or BPEL4Chor. While BPMN is itself a standardized notation widely-adopted in industry, BPEL4Chor is an extension of the standardized BPEL language which enables the modeling of choreographies. Consequently, this results in reusable and portable models that are supported by a broad variety of existing tools for both modeling and execution. Furthermore, when following the public-to-private approach [AW01] the resulting private process models that implement the choreography participants are also specified through standardized notations like BPMN and BPEL and therefore already provide, for example, established fault or compensation handling capabilities and mechanisms.

Another point is that, in our opinion, the main purpose of a choreography is to provide a global view on the potentially complex conversations of the collaborating parties. Therefore, a choreography model should support the specification of all aspects relevant from this global viewpoint, i. e., conversations between choreography participants expressed by the exchange of messages as well as shared choreography data and its exchange or use within and across the participants. These definitions can then be used to enhance the generation of the resulting process models that represent the modeled choreography participants.

In addition, any unnecessary complexity and information not relevant at the level of the choreography such as technical details regarding the exchange of messages, e. g., transport protocol or message encoding to be used, should be hidden from the modelers at the choreography level. Such technical details can then be provided at the level of the process models by the modelers themselves or by IT specialists that refine and enhance the generated processes to make them executable. If making choreographies directly executable, modelers have to specify the collaborations between the participants in a very fine-grained manner and in addition provide already technical details at the choreography level. In our opinion, this decreases the idea of using choreographies as a global view on the complex conversations between the collaborating parties.

Furthermore, Barker, Weissman, et al. [BW+12] introduce the *Circulate approach* that combines the advantages of both paradigms: orchestration

and choreography. Although the control flow remains orchestration-based, the data flow is conducted in a choreography-based manner. More precisely, the control flow and the service interactions are enacted through a central coordinator, a BPE, and therefore the underlying benefits of such BPEs like robustness and scalability are retained [LR00].

Conversely, the data flow follows the choreography paradigm and therefore participants are allowed to directly exchange data between each other without a central coordinator as intermediary. To enable services to transfer data between each other, *proxies* are introduced to provide the required functionality. Therefore, a proxy acts as an intermediary between the process engine and the actual services during service invocations. Consequently, the process engine triggers the invocation of services and the exchange of data between services through a proxy.

In general, we are following a similar approach by applying the public-to-private approach [AW01] to our data-aware choreography models and introducing the TraDE Middleware to decouple the specified cross-partner data flows from the control flow of participants. However, instead of explicitly modeling the invocation of proxies and data exchange through them, we propose to introduce cross-partner data objects and cross-partner data flows as modeling elements within choreography models. Translated to corresponding annotations at the level of the resulting process models, the process engine is then able to transparently enact the cross-partner data flows together with the TraDE Middleware. As a result, process models are enriched instead of changed and the coordination of cross-partner data flow is outsourced to the TraDE Middleware instead of explicitly specified in process models. The former preserves the portability of the models on runtime environments without TraDE support while enabling an easier specification of data-related aspects. The latter provides more flexibility and optimization possibilities during runtime.

An approach similar to the ideas presented in this work, but with focus at the level of process models and BPEL in particular, is provided by Habich et al. [Hab+08]. They try to overcome the issue of centralized and only implicitly specified data flow in BPEL through variables and assign activities and the

resulting *by value* semantics of data exchange. Therefore, they combine their concept of *Data-Grey-Box* web services with an extension of BPEL through so-called *BPEL data transitions*, or BPEL-DT for short. The former allow to enhance web service interfaces with an explicit data aspect allowing the separation of parameters passed by value and data passed by reference. The latter support the annotation of BPEL processes with explicit data flows between the composed Data-Grey-Box web services. Both concepts together allow to integrate specialized data propagation tools and logic, e. g., specified using Extract Transform Load (ETL) tools, as a means to implement the specified data transitions and act as mediators between Data-Grey-Box web services during runtime to provide and resolve data by reference.

In addition, Khalaf and Leymann [KL06] introduce BPEL-D as a modeling extension for the specification of explicit data flow between activities within process models. Process models containing BPEL-D data flows are not directly executable. They have to be translated into standard BPEL process models where BPEL-D semantics are implemented using standard BPEL constructs. Kopp et al. [Kop+11a] provide an overview of such BPEL extensions, including BPEL-DT and BPEL-D, as well as an evaluation of their conformity to the BPEL standard and its extension mechanisms.

Although the authors propose to introduce explicit data flow at the level of BPEL, or process models in general, we argue that cross-partner data flow can be specified easier and more intuitive at the level of a choreography model representing all participants of a service composition. Furthermore, while we also aim at supporting the exchange of data by reference, our overall goal is to hide as much as possible of the data flow related logic at the level of the process models by outsourcing the required functionality to the TraDE Middleware. Therefore, the TraDE Middleware can be used to propagate and resolve data by reference as well as an integration layer for specialized data propagation tools and logic.

Liu, Law, and Wiederhold [LLW02] introduce a *Flow-based Infrastructure for Composing Autonomous Services* (FICAS) and the *Compositional Language for Autonomous Services* (CLAS) used to express the relationships between collaborating services, i. e., the service composition. The resulting CLAS

programs can then be executed by the FICAS runtime. Based on the analysis of the data dependencies between the services, a CLAS program is decomposed into services and an execution plan for their composition. A central coordinator carries out the execution plan and thus controls the control and data flow and sends the corresponding commands to the services. The decoupling of control commands and data flow happens only at the level of the invoked services through separate data and control queues associated to each service for this purpose. Based on that, the approach supports the decentralized exchange of data among services.

While we also support the idea of coordinating the control flow in a centralized manner and handling data flow in a decentralized fashion, the FICAS approach does not rely on, nor incorporate the capabilities of common web service standards. Furthermore, we follow a model-based approach to specify service compositions on both the choreography as well as the orchestration level instead of specifying the compositions through programs. This lowers the entry barrier and increases the usability of the TraDE approach, especially in combination with a graphical modeling tool that supports the users during the specification of their service compositions.

Binder, Constantinescu, and Faltings [BCF06] introduce the concept of *Service Invocation Triggers* in order to allow for conducting service compositions in a decentralized manner. Such a trigger acts as a proxy for a specific service and collects all input data, invokes the service and routes the output data to successor triggers. In sum, the triggers deal with the data exchange and its ordering on behalf of the services in a service composition. It is our understanding that there is no global process model that is decomposed into such triggers. The triggers themselves, however, contain partial knowledge of the process logic and the data dependencies among services to be invoked.

In contrast to our approach, the application of service invocation triggers requires the decomposition of process models into a set of sequential fragments that are subject to certain restrictions, e. g., they neither can contain loops nor conditional control flow, and the data dependencies have to be encoded into the triggers. Therefore, the data flow related logic specified in the processes is offloaded to the triggers. A trigger fires as soon as all

required input values are available, then actually invokes the service and collects all output data. The output data is then routed to a successor trigger based on the routing information that is encoded into the trigger itself, based on the knowledge gathered from the process model. The successor trigger again waits for all required input data before it fires. Therefore, the order of the triggers is only implicitly defined based on the data being exchanged and the routing information at the triggers.

Monsieur, Snoeck, and Lemahieu [MSL12] present a pattern language for the specification of data flows within service compositions that provides a systematic way for designing the data flow aspects of respective coordination scenarios by composing the introduced patterns as building blocks. The pattern language comprises three classes of patterns: data flow initiation patterns as well as direct-indirect request and data transmission patterns. The data flow initiation patterns reflect from which role a data flow is initiated: service provider, service requestor, or data provider. The request and data transmission patterns reflect if the data is directly or indirectly requested and transmitted, respectively.

While the proposed pattern language enables guiding choreography modelers to select and apply the best data flow patterns for representing their use cases, the target of this work is to introduce data-aware choreographies and to provide an end-to-end approach and a respective ecosystem of tools and middleware components to support their modeling and execution. However, the pattern language introduced by Monsieur, Snoeck, and Lemahieu [MSL12] could be incorporated into the modeling environment of the TraDE ecosystem to further support and guide choreography modelers in defining the required data flow of their collaboration scenarios specified in form of a data-aware choreography model.

Köpke, Franceschetti, and Eder [KFE19] present a flexible, heuristic algorithm for generating *good* implementations, with respect to given quality criteria, for the data flow of inter-organizational processes via message exchanges. For example, the number of required interactions, confidentiality issues or the absence of obsolete data transmissions are such quality criteria for generating optimal data flow implementations. The underlying idea

is that at the beginning a global process model specifies the collaboration between multiple parties and for each modeled process construct, e. g., an activity or transition condition, the consumed and produced data is specified within this global process model. Based on that, the heuristic algorithm automatically derives a collection of local process models based on respective user preferences and the specified data requirements. This collection of local process models together implements the global process model and therefore each of the generated local process models contains respective communication activities that conduct the data-transfer in a message-based manner between the participants.

The resulting message-based data exchange is generated in an optimized way based on the defined data requirements from the global process model. As a result, modelers do not have to take care of how and when respective data has to be exchanged between the participants of a choreography during modeling. They only have to specify the data an individual control flow step requires or produces and the heuristic algorithm generates an optimized set of local process models reflecting the provided definitions.

In contrast to our work, the authors build on the available means for exchanging data across participants via messages. Therefore, they do not require an additional middleware as proposed within this work by introducing the TraDE Middleware for the execution of modeled cross-partner data flows. However, the TraDE Middleware does not only execute the cross-partner data flows in a transparent manner, it further allows to decouple the data exchange from the exchange of messages and therefore from the control flow of the participant's process models which from our viewpoint introduces new abilities for transparent data processing, transformation or the integration of heterogeneous data sources. This may also introduce new optimization potentials during choreography runtime regarding flexible adoption of data transformation and optimized data exchange as well as more efficient data placement and staging. Another aspect why we are not following a message-based implementation of cross-partner data flows is that from our experience in data-intensive domains such as eScience, the exchanged data is often not processed nor used within the process models

and only forwarded as input to a respective service invocation [GSK+11]. Related aspects are also discussed in the motivation example presented in Section 1.1. However, a combination of the concepts introduced within this work and the ones presented by Köpke, Franceschetti, and Eder [KFE19] will be a valuable direction for future research. For example, scenarios from the eScience domain will benefit from the combination of optimized, generated message-based and TraDE-based cross-partner data flows since the best concept can be selected for each data exchange.

Ghilardi et al. [GGMR21] introduce *delta-BPMN* as a language for the modeling and verification of data-aware BPMN process models. For the modeling of data-related aspects, i. e., modeling data and its manipulation, a SQL-based language called *Process Data Modeling and Manipulation Language* (PDMML) is introduced which allows to represent as well as manipulate data within BPMN process models in a verifiable way by using BPMN data objects and data stores as a basis for representing data. The resulting delta-BPMN models can then be verified by a state-of-the-art Satisfiability Modulo Theories (SMT)-based model checker for infinite state systems by translating them into the respective input format. SMT refers to the problem of determining whether a first-order formula is satisfiable with respect to some logical theory [BT18]. To prove the feasibility of the overall approach, the Camunda BPMN modeler is extended to enable the modeling of delta-BPMN process models and allow their automatic translation into the required format for their immediate verification.

Since our focus is on increasing data-awareness already at the level of choreography models, especially to ease and improve the modeling of cross-partner data flows, the concepts introduced by the authors are out of scope of this work. However, the SQL-based PDMML language introduces interesting extensions which can be applied to our cross-partner data flows to enable the modeling and verification of complex data flows between cross-partner data objects as well as participant's control flow constructs such as activities or conditions. This could be an interesting future research direction to further improve the expressiveness of data-aware choreography models.

The *Essential Flow Model* presented by Kopp et al. [KLUW11] tackles

the issue that modelers have to decide early which service composition paradigm they follow when modeling collaborations: orchestration or choreography. This imposes not only restrictions during modeling, but also on the IT infrastructure for the execution of the resulting models specifying the collaborations. The notion of essential flow models allows to defer the decision how to split and organize a collaboration to a later phase. It enables modelers to specify their collaborations by modeling essential flows between tasks and responsible partners. Based on that, an essential flow model can be mapped and implemented in different ways, e. g., as orchestration or as choreography, taking the target IT infrastructure into account. The authors idea of removing the burden from modelers to distinguish between and decide on local or remote control flow, i. e., message flow, is similar to our approach regarding data flow. Although the authors do not take data flow into account, the notion of essential flow models provides a useful abstraction technique and entry point for specifying data-aware service choreographies.

2.3 Alternative Modeling and Execution Approaches with Focus on Data

In the previous sections, the focus is on related works that also follow the goal of improving data-awareness and data-related capabilities for the modeling and execution of choreographies. Therefore, we support the arguments of Meyer et al. [MSMP11] that data-related aspects or data-awareness in general should only be supported to the degree actually required in the domain or scope in which a modeling language is used. Since our focus is on improving and extending the role of data in classical control flow driven choreography and process modeling languages such as BPMN, BPEL4Chor, and BPEL, related work following the same paradigm is presented above. This means, the specification of control flow remains the main part of choreography and process modeling while further support for the specification of data and data flow is introduced. However, within this section we shortly outline other potentially related approaches following another paradigm

and therefore not discussed in detail in the context of this work.

2.3.1 Artifact-centric Business Process Management

Modeling languages following the *artifact-centric* modeling paradigm consider business data and how it evolves or is changed within a process as the main driver. Therefore, the focus changes from modeling control flow with associated data to modeling data with associated control flow, specifying the actions performed on data. Examples for corresponding artifact-centric modeling approaches are *business artifacts* [NC03] or *case handling* [Aal+05].

Lohmann and Wolf [LW10] apply the paradigm of business artifacts at the level of choreographies to model collaborations from a data perspective. Therefore, they provide a systematic approach to specify relevant data as artifacts and a set of agents that are operating on that data in the context of a collaboration. By enhancing artifacts with information about their location and its impact on remote access of agents to them, they are then able to derive an overall interaction model. Furthermore, Lohmann and Nyolt [LN11] investigate to what extend BPMN can be used or requires extensions in order to support modeling of artifact-centric processes and choreographies. Although we support the author's arguments to make data more prominent and increase modeling support in choreography and process models, we still rely on control flow as the main driver for processes and related established standards such as BPMN and BPEL.

2.3.2 Modeling and Enforcing Business Collaborations using Shared Ledger Technologies

Another emerging approach for realizing business collaborations is the use of shared/distributed ledger technology as exemplified by blockchain. Following this paradigm, different works exist which discuss the modeling and enforcing of business collaborations in a choreography or process-based manner with a special attention on the data dimension. For example, Hull et al. [HBC+16] discuss the use of abstractions from business artifacts [NC03] as

the basis for introducing a shared ledger Business Collaboration Language (BCL). The idea is to specify collaborations between multiple parties using such a BCL language and then conduct the modeled collaboration using shared ledger technology by mapping the BCL to respective smart contracts which operate on data where all business-relevant data shared between the collaborating parties is persisted within the shared ledger itself. Combining the business artifacts paradigm with the data-centricity of shared ledgers seems to be a natural fit towards introducing data as a first-class citizen and the main driver in business collaborations.

Ladleif, Weske, and Weber [LWW19] also highlight *shared data* and decision logic defined as *smart contracts* being capabilities of distributed ledger technologies, and blockchains in particular, as promising foundations for realizing blockchain-based choreographies. Therefore, they analyze the underlying assumptions and limitations of the BPMN 2.0 standard [BPMN] as one of the most prominent choreography modeling languages against the possibilities introduced by the shared ledger technology. Based on the results, they present proposals for the extension of BPMN choreography models to align them with and support the execution semantics of shared ledger technology. In accordance with Meyer et al. [MPB+13], they also identified that the specification of a shared data model is an essential building block towards modeling blockchain-based choreographies. The proposal is also implemented as a proof-of-concept on the Ethereum blockchain to test and discuss its expressiveness and practical feasibility.

Similarly, Sturm et al. [SSJS20] propose a novel form of workflow interoperability through *decentralized control* for inter-organizational workflows by also utilizing shared ledger technology. A shared ledger, e. g., a blockchain, acts as a central source of truth and enables decentralization of control by relying on shared data and encapsulating process information and decision logic into smart contracts. The authors therefore present a corresponding decentralized control implementation using extended BPMN modeling elements for specifying inter-organizational processes and execute them on the Ethereum blockchain. A special focus is on the support of data-based routing, i. e., data-based decisions in conditional control flow, using the

shared process data stored in the blockchain.

A general overview of research works regarding the enactment of business collaborations on blockchains is presented by Stiehle and Weber [SW22] based on a taxonomy which itself is derived through a systematic literature review. The resulting taxonomy is structured into supported capabilities and enforced guarantees of blockchain-based process enactment approaches. Based on the taxonomy, the authors classify available research works focusing on blockchain-based process enactment regarding their introduced concepts and approaches for tackling the challenges of inter-organizational processes, namely interoperability, traceability, scalability, flexibility, and correctness.

2.4 Integrating Heterogeneous Data Transformation Logic into Service Compositions

Within this section different approaches for integrating and utilizing heterogeneous data transformation logic in service compositions are discussed. This is of major importance for introducing concepts for the modeling and execution of data-aware choreographies, since the ability to transform data is one of the core requirements and aspects of data management in general. The focus is therefore on related work regarding application reuse and wrapping techniques to specify, package, provision and execute heterogeneous data transformation implementations in the context of data-aware service choreographies. This is especially important to support data transformations within simulation choreographies from the eScience domain, since there often file-based data has to be processed which is only semi-structured or even unstructured and related data transformations are very use case specific and therefore normally implemented by the scientists themselves, e. g., using programming languages such as Python. Data transformation languages such as XQuery [XQuery], XML Path Language (XPath) [XPath] or Extensible Stylesheet Language Transformations (XSLT) [XSLT] established in the domain of BPM and often directly supported by underlying process modeling languages such as BPEL [BPEL] or BPMN [BPMN] as well as re-

lated BPE implementations, are therefore not usable to specify and execute data transformations in choreography models within the eScience domain.

Zdun [Zdu02] introduces an approach for legacy application migration to the web. He describes a method with the following four steps:

- providing an Application Programming Interface (API) using either wrapping or redevelopment approaches,
- implementation of a component responsible for mapping of requests to the legacy API,
- as well as implementation of a component responsible for response generation, and
- the integration of these components into a web server.

Furthermore, a reference architecture supporting the introduced concepts and issues is presented.

Sneed [Sne06] 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. [Afa+13] 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 and Afanasiev [SA14] introduce Everest, a PaaS platform for reusing scientific applications based on the MathCloud platform [Afa+13]. The authors further improve the ideas of providing a uniform interface for task-based execution of applications.

Delaitre et al. [Del+05] propose a black-box approach for deployment of legacy applications written in several languages, e.g., Java, C, and Fortran, as grid services called Grid Execution Management for Legacy Code Architecture (GEMLCA). Legacy applications are described using an XML-based

descriptor which contains information regarding the runtime environment and parameters required for execution.

Glatard et al. [Gla+08] propose a generic web service wrapper which provides a standard service interface for running legacy applications. Additionally, the authors demonstrate how to achieve dynamic service grouping using the generic wrapper and MOTEUR workflow system. The idea is to hide the grid infrastructure behind a standard interface which can then be invoked using any web services specification-compliant client.

Juhnke et al. [Juh+09] present the Legacy Code Description Language framework which allows wrapping legacy code. An extensible legacy code specification model is used as a basis for the generation of executable wrappers. The model stores the information necessary for wrapping binary and source code legacy applications. For instance, the model describes the operations supported by the legacy applications and respective bindings which define the type of the wrapper.

Wettinger et al. [Wet+15] 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.

Hosny et al. [Hos+16] introduce AlgoRun, a container template based on Docker suitable for wrapping CLI-based scientific algorithms and exposing them via a Representational State Transfer (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.

Kim et al. [Kim+17] describe how bioinformatics pipelines can be run using so-called Bio-Docklets, pre-configured Docker containers. With this approach, the complexity of the pipeline is hidden behind a Docker container which exposes only input and output endpoints required for executing the pipeline. Therefore, from a user's perspective the invocation of the complex pipeline does not differ from the invocation of a single application.

While some of the works are used as a basis for the data transformation approach introduced in Chapter 6, none of them fit completely our needs. Since our focus is on enabling the transparent integration and use of heterogeneous data transformation software within data-aware choreographies, the data-related aspects and capabilities of the above presented application reuse and wrapping techniques are of major relevance. The idea to create specifications for legacy applications similar to the ones introduced by Juhnke et al. [Juh+09] and Hosny et al. [Hos+16] is used as a basis for the concepts introduced within this work. Our goal is to provide generic concepts and a supporting middleware for the specification, packaging and provisioning of data transformation applications to enable their use within data-aware service choreographies.

Chapter 3

Data-Aware Choreography Methodology

This chapter introduces our methodology for data-aware choreographies. As outlined in Chapter 1, the goal of the methodology is to increase data-awareness throughout the complete life cycle of choreographies. Therefore, we further motivate the need for improving data-awareness in service choreographies by discussing shortcomings of the state-of-the-art in Section 3.1. Based on that, we introduce our Transparent Data Exchange (TraDE) approach to mitigate the presented shortcomings and improve data-related aspects of service choreographies in Section 3.2. Since in our opinion, data-related aspects are not only relevant during choreography modeling and execution, in Section 3.3 a complete life cycle for data-aware choreographies based on the the traditional Business Process Management (BPM) life cycle [Wes12] is presented to reflect data-related aspects in a seamless fashion. To provide an overview of the TraDE approach and the data-aware choreography methodology as a whole, an example data-aware choreography and its way through the different phases of the extended life cycle is presented.

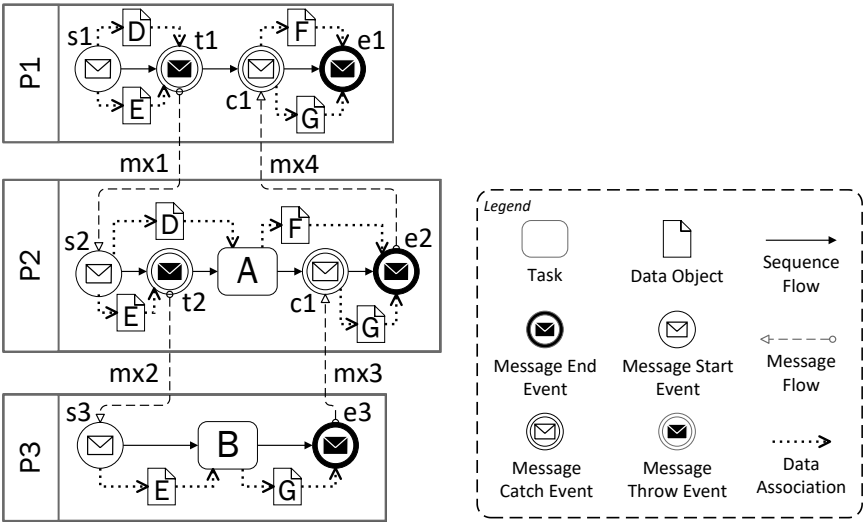

Figure 3.1: Example choreography illustrated as BPMN collaboration model, based on [HBKL17].

3.1 Motivation

To further illustrate and motivate the need to improve data-awareness in service choreographies we first present a motivation example followed by a discussion on shortcomings regarding the modeling and execution of data-aware service choreographies.

We use BPMN as a basis for the following discussion and also to illustrate our concepts within this work. However, our concepts are not bound to BPMN and can be applied to other choreography modeling languages such as BPEL4Chor. The only assumption is, that the underlying choreography modeling language follows the *interconnected interface behavior* modeling approach [DKB08] introduced in Chapter 2. This means, that the choreography model specifies control flow per participant and the interactions between participants through message flows. Figure 3.1 shows an example of such an interconnected interface behavior model illustrated as BPMN collaboration

model with three interacting participants. The conversations between the participants are modeled by message intermediate events and message flows. Data is modeled by BPMN data objects and the reading and writing of data objects from tasks and events is specified through BPMN data associations. As outlined in Chapter 2, a lot of choreography modeling languages are not directly executable, we therefore follow the Public-to-private (P2P) approach introduced by van der Aalst and Weske [AW01]. Based on the P2P approach, a choreography model specifies a public model of a collaboration which is then transformed into a collection of private process models specifying the internal logic of each participant while implementing the conversations defined at the level of the choreography model [ALM+08; DKB08; DKLW09].

As a result, the data-awareness and data modeling capabilities of the used choreography as well as process modeling languages have to be taken into account. Exactly for this purpose, Meyer, Smirnov, and Weske [MSW11] already evaluated business process modeling languages with respect to their level of data-awareness and data modeling capabilities. Therefore, in the context of this work, we reuse their results as a basis to discuss and identify further issues based on the introduced motivation example depicted in Figure 3.1 with focus at the level of choreographies.

One of the major shortcomings is, that data flow across participants has to be modeled differently compared to the data flow inside a participant. The former we call *inter-participant* data flow and the latter *intra-participant* data flow in the following. For example, in BPMN intra-participant data flow can be expressed intuitively by drawing data associations between data objects and tasks as shown in Figure 3.1 between the message start event *s1*, data object *D* and the message intermediate throw event *t1* of participant *P1*. However, to model the exchange of data between two or more participants, i. e., inter-participant data flow, data associations are not allowed anymore and message flow has to be introduced to exchange the required data. For example, in Figure 3.1 data objects *D* and *E* are send from participant *P1* to participant *P2* via the message flow *mx1* specified between the intermediate message throw event *t1* and message start event *s2* of participant P1 and P2, respectively. While from an execution and technical

point of view it is clear that data exchange across participant boundaries needs to be handled differently than local data exchange within a participant, the question is, if this difference has to be explicitly specified within a choreography model using distinct modeling constructs. As shown by the example, introducing message flow to exchange data between participants requires the specification of a whole set of additional modeling constructs. First, a message has to be specified that will transport the data between the participants during runtime. Second, the modeler has to add a corresponding message send task or event (e. g., t1 in Figure 3.1) with data associations on data objects containing the data to exchange, in order to encapsulate data into a message and send it to another participant. On the side of the receiving participant, the modeler has to add a message receive task or event (e. g., s2 of P2 in Figure 3.1) to consume the message and specify data associations to one or more data objects where the data contained in a message should be extracted to. Finally, the modeler has to connect the sending and receiving tasks or events with a message flow (e. g., mx1 in Figure 3.1), to specify the exchange of the previously introduced message. The result is that data produced by one participant is now also accessible at another participant both having their own local, independent copy of the data. We call this type of inter-participant data flow: *message-based data exchange*. As outlined above, the example choreography shown in Figure 3.1 uses this message-based data exchange style to define inter-participant data flows. For example, data objects D and E are exchanged through message flows between participants P1, P2 and P3, respectively. The key point to take away is that data cannot be exchanged between participants without introducing additional control flow and modeling constructs to exchange the data. While Figure 3.1 depicts rather simple conversation scenarios, it already gives an idea of the potential complexity and overhead such message-based data exchange will introduce. This overhead of introducing additional modeling constructs in order share data between participants will increase, e. g., if multiple participants require the same data or received data has to be routed to other participants.

Another drawback of message-based data exchange is the fact that the

same data objects, or slight variations of them, need to be specified in the scope of each participant that interacts with the data. This is also directly visible in Figure 3.1 where participants P1 and P2 define the same four data objects and participant P3 defines its own copy for two of them. However, to technically enable the collaboration between different participants through a choreography they have to define and agree on the structure of the to be exchanged data. This leads to the question if the underlying choreography modeling language allows the specification of a common, globally consolidated and agreed set of data objects representing a data contract between the collaborating parties. Often, this is not supported and therefore modelers have to agree on structures without being able to specify the outcome explicitly and in a central, reusable manner within a choreography model. As a result, each party has to take care of specifying the required set of data objects in the context of all participants it is responsible for while being compliant with the agreed structures. The issue is that if these structures change over time, there is no central point in a choreography model to do this and changes have to be manually reflected in all affected data objects specified in the context of several participants interacting with this data. This makes changes error-prone and might lead to inconsistent specifications of data objects [Lic22]. Furthermore, the data required and produced by the choreography as a whole and of each participant individually has to be identified by analyzing the model instead of being directly visible through its graphical representation. Another potentially useful capability not supported at the moment, is that modelers should be able to express that multiple data objects (semantically) belong together. This will improve the visual expressiveness of the models regarding their data perspective [MSW11].

Moreover, binding data flow between participants to message exchanges potentially results in unnecessary routing of data and blocking of control flow of participants while data is exchanged. The former means that data might be passed through several consecutive message flows across participants instead of directly exchanging it in a peer-to-peer manner. The latter point addresses the fact that while the data is exchanged through message flow the receiving participants are blocked until the message arrives. For example,

data object *E* in Figure 3.1 is send from participant P1 to participant P2 via message flow *mx1* and then routed from there to participant P3 via message flow *mx2* without being processed at participant P2 at all. This is actually the result of an inherent trade-off modelers have to make during choreography modeling. On the one hand, they can improve and optimize the data flow by introducing additional message-based data exchanges to exchange the data as soon as it is available in a peer-to-peer manner. On the other hand, modelers can try to keep the number of message-based data exchanges minimal and only pass all required data to another participant at once, if the exchange of a message is anyhow required in terms of control flow. The first choice results in more complex and conversation-intensive models but reduces the time participants have to wait for required data. The second choice results in less complex and less conversation-intensive models but increases the time participants have to wait for required data because data is only exchanged in blocks at certain points in time.

To sum up our discussion, when applying a message-based data exchange approach the exchange of data during runtime has to be specified completely upfront at modeling time. Therefore, during runtime it is hard to flexibly improve data exchange between participants since all data exchanges are strictly expressed through message flows. The only way to realize (more) dynamic data capabilities using the message-based approach is to introduce corresponding control flow logic already at the level of the choreography models. The main drawback of this approach is that the models are polluted with data management functionality that is not relevant from a business perspective. Therefore, we are arguing that intra- and inter-participant data flow should be expressed within choreography models in a consolidated and control flow independent manner while the underlying data management and exchange functionality should be provided transparently by the runtime environment without the necessity to be explicitly modeled via control flow constructs in a choreography model. Furthermore, choreography models should allow to define a common, globally consolidated and agreed set of data objects representing the data contract between the collaborating parties in form of a choreography data model.

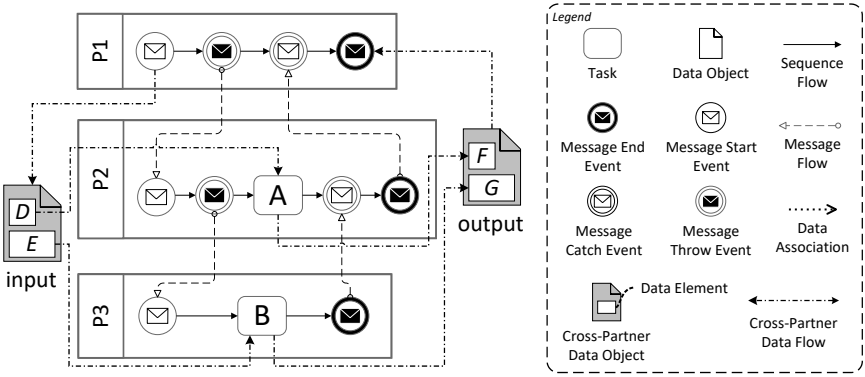

Figure 3.2: Example choreography with TraDE concepts applied, based on [HBKL17].

3.2 The TraDE Approach

In this section, we introduce our concepts for the modeling and execution of data-aware choreographies through TraDE towards mitigating the presented shortcomings and improve data-related aspects of service choreographies. Therefore, we first have a look at the modeling perspective and introduce the notion of cross-partner data objects and data flows in Section 3.2.1 and how they tackle the shortcomings discussed in Section 3.1. Based on that, we will introduce our proposal for required runtime support for these new modeling constructs through a new TraDE Middleware component in Section 3.2.2.

3.2.1 Modeling

Figure 3.2 shows the example choreography model from Figure 3.1 with our TraDE concepts applied. To enable the specification of choreography data, representing a data contract between the participants within a choreography model, we introduce a *choreography data model* (CDM). A CDM provides the foundation for data-awareness and data-related capabilities in choreographies. It enables to specify required data and its structures in a self-contained and consolidated manner as a building block of a cho-

reography model. Therefore, a CDM consists of a set of *cross-partner data objects* that express the commonly agreed data of a choreography shared by and accessible from all participants. To avoid confusion between BPMN 2.0 data objects and cross-partner data objects as a general concept, we use the term *data container* as a language-independent name for a modeling construct that allows the specification of data, e. g., BPMN 2.0 data objects or BPEL variables, within this work. The data-aware choreography model in Figure 3.2 makes it explicit which data the overall choreography as well as each of its participants requires or produces. The major advantage is that data required by multiple participants of a choreography can be itself modeled less redundantly. Moreover, its exchange can be expressed more intuitively using cross-partner data flows instead of specifying messages and corresponding tasks or events to process them. Although each participant can still have its own local data containers, cross-partner data objects allow to model shared data which is associated to the whole choreography instead of a single participant. By using cross-partner data objects, data containers have to be modeled only once which normally need to be specified multiple times at the level of the different participants within the choreography.

A cross-partner data object has a unique identifier and contains one or more *data elements*. A data element has a unique name, from the scope of its surrounding data object, and a reference to a definition of its structure, e. g., using a simple, build-in type system or XML Schema Definitions (XSDs) [XSD1]. The idea of this single level of nesting is that cross-partner data objects can be seen as named envelopes for a collection of typed data containers, namely data elements, which semantically belong together. Therefore, data elements hold the corresponding data values during runtime. We distinguish between cross-partner data objects that can be instantiated only once (single-instance) or that hold a collection of values (multi-instance) during runtime. Where multi-instance cross-partner data objects are useful to hold a collection of identically structured values that can be processed with the same sequence of tasks through a loop, while the number of loop iterations is dynamically bound to the size of the collection during runtime.

Based on top of the notion of cross-partner data objects, we introduce

cross-partner data flow. The idea is that modelers should be able to intuitively specify data flow within and across participants. Furthermore, cross-partner data flow allows to decouple the exchange of data from the exchange of messages and therefore from choreography control flow as shown in Figure 3.2. As participant internal data flow, cross-partner data flow also supports the specification of transformation logic, selective queries and mappings, e. g., to read or write only parts of a data element or cross-partner data object or multiple data elements at once. The only difference between intra-participant and cross-partner data flow is how the specified data flow is conducted during runtime, i. e., if the Business Process Engine (BPE) executing a participants private process or the TraDE Middleware is handling the data exchange. Regarding the modeling perspective, these two extensions, i. e., cross-partner data objects and data flows, provide the basis for introducing data-aware choreographies and enable an easier and more intuitive specification and handling of data.

As already outlined in Section 3.1, the modeled choreographies are transformed to a collection of interconnected process models in order to execute them [AW01; DKB08]. The idea is to translate the introduced cross-partner data objects into standard data containers at the level of the private process models again. For example, using data objects in BPMN 2.0 or variables in BPEL. Using the specified choreography data model as input, this can be done in an automated manner. Based on that, we are able to reduce manual refinement efforts at the level of the resulting private process models by leveraging provided data-related knowledge to generate more complete process models. The overall goal is that modelers refining the private processes should not need to know nor distinguish between local or globally shared data containers. From the viewpoint of each private process model, there is no difference between a data container that is defined locally or globally. As a result, during process refinement, modelers can extend the generated private process models with additional control and data flow as usual.

Since the BPEs executing the private process models need to communicate with the TraDE Middleware to conduct the modeled cross-partner data flow together, data containers which are generated based on cross-partner data

objects need to be identifiable somehow, e. g., by enhancing them with corresponding metadata required by the BPEs. This metadata reflects the context to which a data container belongs, i. e., a data element of a cross-partner data object, and therefore allows to differentiate data containers provided by the BPEs locally and the ones representing shared data provided by the TraDE Middleware in form of cross-partner data objects. To represent this metadata and the linking of a data container to a cross-partner data object at the level of a process model, the modeling construct for data containers of the underlying process language has to be extended. In the case of BPMN 2.0 and BPEL the extensibility of the languages can be used to introduce new attributes and elements that can be associated with the corresponding modeling constructs for BPMN data objects or BPEL variables, respectively. In addition, to enable a more fine-grained level of data staging, i. e., pulling and pushing data of cross-partner data objects between a BPE and the TraDE Middleware, we will introduce and describe so-called *Staging Elements* in Section 3.3 as a process modeling language extension. Both types of language extensions can then be used by the BPEs to communicate with the TraDE Middleware and therefore realize the modeled cross-partner data flow. However, for describing the overall idea of our TraDE approach and how modeled cross-partner data flow is conducted, we will rely on the simple linking of process data containers and cross-partner data objects described above. To enable the TraDE Middleware to provide the defined cross-partner data objects, during transformation all data object related information is bundled into a corresponding *TraDE deployment descriptor*. This deployment descriptor can then be deployed to the TraDE Middleware to make the specified cross-partner data objects available. All related details will be discussed and presented in more detail in the remaining chapters of this work. In the following, a brief overview of the execution of a data-aware choreography model is presented.

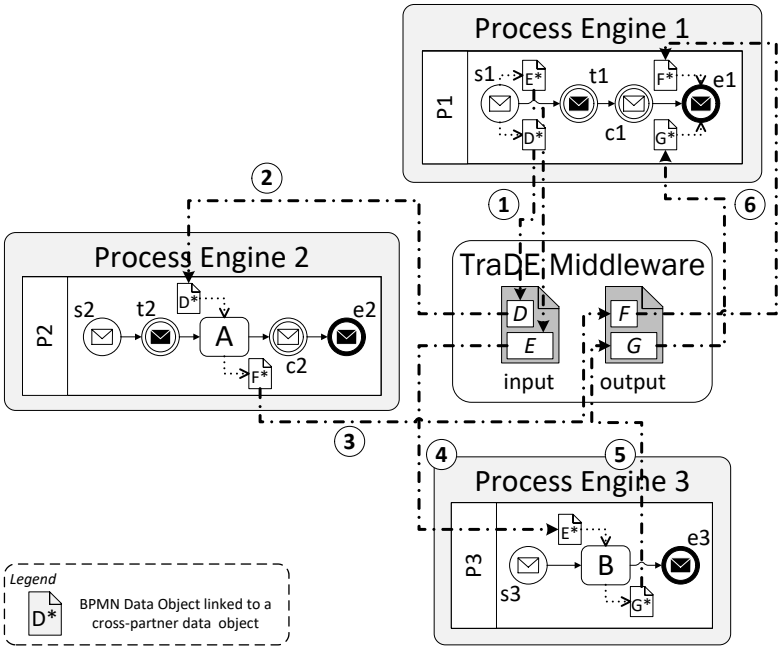

Figure 3.3: Execution of the example choreography from Figure 3.2 based on its refined private process models.

3.2.2 Execution

Figure 3.3 presents the three refined private process models generated out of the example data-aware choreography model shown in Figure 3.2. Each of the process models represents one of the choreography participants and is deployed to a corresponding BPE for its execution. The three process models together implement and therefore conduct the overall data-aware choreography. All modeled cross-partner data objects are made available at the TraDE Middleware by deploying the generated deployment descriptor to the middleware. The generated data containers at the level of the process models are represented through corresponding BPMN data objects. As described above, the data objects are enriched with a link to the respective

cross-partner data objects at the TraDE Middleware, indicated by the * within the data objects shown in Figure 3.3.

By instantiating the deployed process models, e. g., on behalf of a client's request, the overall choreography is executed through the started interrelated process instances which together realize the modeled control and data flow of a choreography. We use the term *choreography instance* introduced by Weiß et al. [WAHK15a] to describe such groups of interrelated process instances conducting a choreography without implying that there is a central coordinator within this work.

For our example choreography shown in Figure 3.3, a new choreography instance is created as soon as process model *P1* at *Process Engine 1* is instantiated through a corresponding request message. This request message contains data values which are extracted by message start event *s1* and stored into the BPMN data objects *D* and *E*. Since these data objects are linked with corresponding cross-partner data objects, the BPE instantaneously forwards the data values to the respective data elements of cross-partner data object *input* at the TraDE Middleware as indicated by step 1 in Figure 3.3. After the data is forwarded, the intermediate message throw event *t1* of P1 sents a corresponding message to process model *P2* deployed at *Process Engine 2* as specified by message flow *mx1* depicted in Figure 3.1. As a result, a new process instance for P2 is created and no data for data objects D and E has to be extracted from the message by message start event *s2* as required for message-based data exchange shown in Figure 3.1 since the data is available at the TraDE Middleware. Next, the intermediate message throw event *t2* of P2 sents a request message to process model *P3* deployed at *Process Engine 3* as specified by the choreography model depicted in Figure 3.1.

Based on that, a new process instance of process model *P3* is created based on the request message received through message start event *s3*. According to the modeled control flow of P3, task *B* is executed next which requires the data of data object E as an input. Therefore, Process Engine 3 requests the data from data element *E* of the cross-partner data object *input* from the TraDE Middleware as indicated by step 4 in Figure 3.3. After task B is successfully completed, it stores its results into data object G where the

process engine directly redirects the result data to data element *G* of cross-partner data object *output* at the TraDE Middleware as indicated by step 5 in Figure 3.3. Finally, the instance of P3 terminates via the modeled message end event *e3* by sending a response message to P2 to inform it about the successful completion of the process instance of P3. While P3 is executed as described above, task *A* of the process instance of P2 is scheduled which requires data object D as an input. Therefore, Process Engine 2 requests the data from data element *D* of the cross-partner data object *input* from the TraDE Middleware as indicated by step 2 in Figure 3.3.

On successful completion of task A, the process engine stores the result data to data element *F* of cross-partner data object *output* at the TraDE Middleware via the linked data object as shown by step 3 in Figure 3.3. As soon as the intermediate message catch event *c2* of P2 receives the response message from P3, the instance of P2 is also terminated via the modeled message end event *e2* which sends a response message to P1. The intermediate message catch event *c1* receives this response message from P2 and terminates itself via the specified message end event *e1* by sending a final choreography response message to the initial requestor. This response message contains the results from task A and B which are requested by Process Engine 1 from the *output* cross-partner data object available at the TraDE Middleware as shown by step 6 in Figure 3.3. As soon as the instance of P1 is terminated, the execution of the choreography instance formed by these three process instances is completed.

In general, whenever a process engine reads or writes data from or to such linked data containers presented in Figure 3.3, it invokes corresponding functionality exposed by the TraDE Middleware to query or forward shared data from or to the middleware, respectively. This allows us to share and exchange data across multiple interacting parties as modeled through the choreography but completely independent and decoupled from message flow. The resulting data containers are only placeholders referring to the actual cross-partner data objects managed by the TraDE Middleware outside of the process engines. The process engines therefore have to be extended to integrate the TraDE Middleware and to support such linked data containers

which will be discussed in more detail in Chapter 5.

By the TraDE Middleware we want to introduce new degrees of freedom regarding the data perspective of service choreographies. In general, TraDE acts as a middleware layer supporting an easier management, exchange, and provisioning of shared data independent of its processing within a service choreography or orchestration. Therefore, all cross-partner data objects are exposed in a web-accessible manner through a Representational State Transfer (REST) API by the TraDE Middleware. The major advantage is that each data object is represented as a resource and can therefore be easily accessed, referenced, and shared with others through a Uniform Resource Locator (URL) [BFM05]. This is especially important in the eScience domain where scientists should be supported with the sharing of their simulation data also independent of the life time of the underlying simulation choreography instances. The TraDE Middleware will enable scientists to upload and provide simulation input data, inspect and observe intermediary results during the execution of a simulation or enable an easier reuse of data from previous simulation runs by simply providing a reference in form of an URL to them.

Furthermore, outsourcing the data to the TraDE Middleware decouples the life time of the data from the underlying process instances and from the availability of the process engines. This allows an easier reuse of data across multiple instances of the same choreography model or even across different choreography models that require common data. Especially the fact that we are able to share the data not only in the context of a choreography and its interacting services, opens up completely new ways of how data can be processed, provided and managed in service choreographies. For example, using other data-centric tools and systems to process or transform the data in parallel to the execution of a choreography to allow their use for other purposes. The logic can therefore be triggered in an event-based manner using the TraDE Middleware as a data flow coordinator.

In contrast to the previously outlined positive effects and advantages of introducing cross-partner data flow and decoupled data exchange through the TraDE Middleware, this causes also some negative side effects. Whenever something is shared in a distributed context, concurrency as well as security

issues will arise. Since in our case data and control flow are running in parallel, the probability for concurrent access of shared data from different, potentially not synchronized participants is much higher than in local scenarios. Therefore, modelers have to pay attention to concurrency issues when specifying cross-partner data flows, e. g., by using respective message exchanges as synchronization points in cases where race conditions between two or more participants on respective cross-partner data objects may exist.

Since the focus of this work is on the notion of data-aware choreographies and to enable their modeling and execution in general, a thorough analysis of potential concurrency issues and cases for which concurrency control mechanisms [BHG87] should be applied are not discussed or presented within this work. For the prototypical implementation of the TraDE concepts within this work, pessimistic concurrency control mechanisms are applied. Therefore, the execution of process instances might be blocked until required data is available at the TraDE Middleware. The identification and optimization of concurrency control as well as security demands in the context of our TraDE concepts are a topic for future research. In the following, the integration of our TraDE approach into the traditional BPM life cycle is presented.

3.3 Life Cycle of Data-Aware Choreographies

To account and increase data-awareness throughout the complete life cycle of choreographies we extend the traditional BPM life cycle [Wes12] with our above introduced TraDE methods. Figure 3.4 presents our proposal for a data-aware service choreography management life cycle that uses the traditional BPM life cycle and available extensions for choreographies introduced by Decker, Kopp, and Barros [DKB08] and Weiß and Karastoyanova [WK14a] as a basis. In the following, we describe each of the life cycle phases, their relations, the artifacts they produce or consume and how each of the phases employs our *TraDE Methods* to support data awareness as a separate concern throughout the whole life cycle of a choreography. The TraDE Methods bundle the above introduced set of data-related modeling extensions

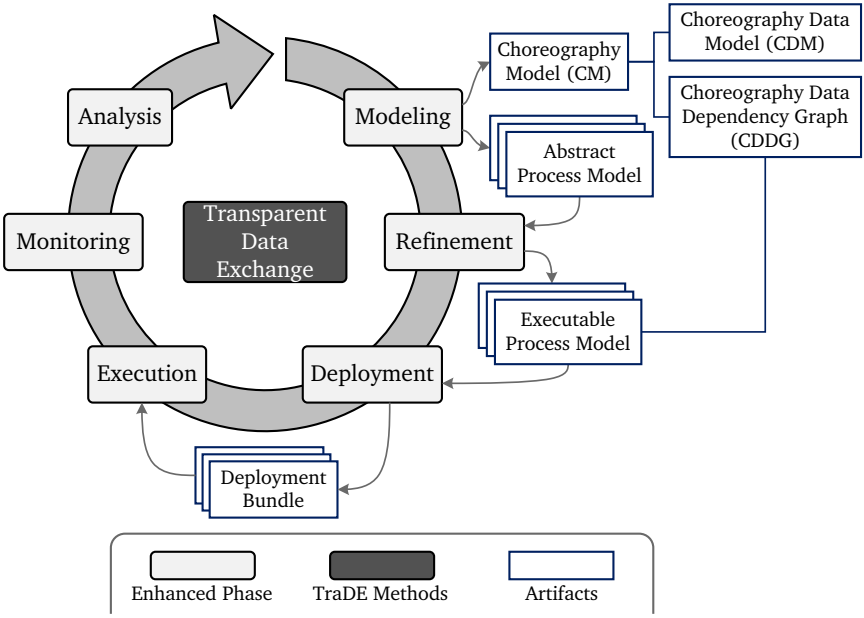

Figure 3.4: Data-Aware Choreography Management Life Cycle, based on [HKL16b].

and methods as well as model transformations to support data awareness throughout the whole life cycle and potential optimizations regarding the data perspective of choreographies.

To abstract away any specific terms or constructs of concrete choreography or process modeling languages, we define our TraDE methods on formal models on both the choreography and the process level within this work. This allows us to maintain the independence of our conceptual approach and introduced methods from the various existing modeling languages for choreography and process model specification. However, the formal models provide the basis to apply our approach to any choreography and process modeling notation by defining a corresponding mapping from the formal model to a concrete modeling notation such as BPMN which we use throughout the whole document for presenting examples of data-aware choreographies in

a graphical manner. Therefore, we use the concept of *Choreography Model Graphs (CM-Graphs)* introduced by Weiß et al. [WAHK15a] and *Process Model Graphs (PM-Graphs)* defined by Leymann and Roller [LR00] as formal metamodels. PM-Graphs provide a formal foundation and specify process models as directed acyclic graphs, where the nodes represent activities and the edges the control connectors (control flow) between the activities. An activity expresses a task or piece of work to perform, like invoking a service or manipulating data. CM-Graphs are based on the PM-Graph definition and introduce the ability to specify the interactions of collaborating parties where each party, i. e., a participant, is represented as a kind of reduced PM-Graph, e. g., by obfuscating all activities that are not involved in the conversations between the interacting parties. We therefore use the term *communication activity* in the following, to refer to the set of activities which are specifying the conversations between the participants, i. e., activities that are sending or receiving messages. The coupling between the PM-Graphs is specified through a set of directed edges where each edge connects a communication activity from one PM-Graph with a communication activity of another PM-Graph. These special interconnection edges are called *message connectors* and represent a conversation between two choreography participants, i. e., a message flow. Chapter 4 introduces our formal model for data-aware service choreographies together with a graphical notation based on extended versions of both, the CM-Graphs and PM-Graphs metamodels. Therefore, required extensions for data-aware choreographies are only briefly introduced where necessary to describe the life cycle phases.

In the following sections, the data-aware choreography life cycle is presented in a top-down manner, i. e., a new choreography model is specified first, based on which a collection of private process models is generated and refined to represent and implement the participants of a choreography model. We assume that during a life cycle iteration, in each phase all successor phases have access to the knowledge and artifacts of the current and all predecessor phases, as well as the history of previous life cycle loops.

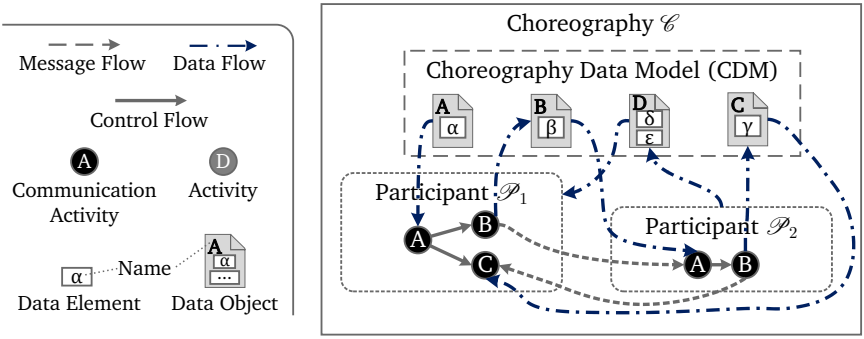

Figure 3.5: Detailed view on the *Modeling* phase, based on [HKL16a].

3.3.1 Modeling

In the *Modeling* phase the different stakeholders, e. g., domain experts of different fields in eScience or business specialists from different companies, who want to collaborate, define their interactions by specifying corresponding participants and their conversations via message exchanges through a choreography model [Kop16]. The resulting choreography model can therefore be seen as a collaboration contract on which all participants agree [MPB+15]. As introduced in Section 2.1, there are a variety of choreography modeling notations. The most prominent examples are BPMN collaboration models [BPMN] or BPEL4Chor [DKLW07] following the *interconnection modeling approach* or BPMN choreography models, Web Service Choreography Description Language (WS-CDL) [WS-CDL] or *Let's Dance* [ZBDH06] following the *interaction modeling approach*. In general, any of these languages can be used as underlying modeling notation to represent choreography models, however, as discussed in Section 2.1, within this work we rely on interconnection models such as BPMN collaboration models and BPEL4Chor models. They allow to specify the conversations between the participants of a choreography on a more fine-grained manner and further enable to provide additional details towards the actual execution of a choreography model. For example, concrete communication activities at the level of the participants

can be specified as part of their control flow to specify and implement the conversations with other participants. Furthermore, the data flow related to the conversations, i. e., extracting and wrapping data from and to message payloads, can be specified as part of the choreography model.

This is exactly where our TraDE approach comes into play. As outlined in Section 3.2, our TraDE approach introduces an explicit data model and data flow across the participants at the level of the choreography with the help of cross-partner data objects and cross-partner data flow in addition to the possibility of exchanging data via messages as part of the modeled conversations. This introduces, in addition to the collaboration contract representing the defined conversations, a *data contract* defining the data to be exchanged and the participants consuming or providing the data. The resulting data contract is represented through a new Choreography Data Model (CDM) and its related data flows as shown in Figure 3.5. Both together provide the foundation for the data-awareness of choreographies and a basis for (semi-)automated phase transitions and model enhancements throughout the life cycle in the following. Since at the level of the choreography model only an extract of the actual participant logic is visible in form of activities, cross-partner data flows and therefore data connectors can be also specified between participant boundaries as shown in Figure 3.5. There, a data connector from participant $\mathscr{P}_2$ to data object D and from data object D to participant $\mathscr{P}_1$ is specified. This type of cross-partner data flow allows to specify which of the choreography data objects a participant requires during runtime to actually provide its part of functionality to the overall choreography without the necessity to disclose any of its participant-internal logic or to add respective placeholder activities.

Our goal is that TraDE-based data exchange can be used as an extension which is seamlessly integrated into choreography modeling. Therefore, message-based data exchange and TraDE-based data exchange provide two alternative yet compatible and therefore interchangeable approaches for modeling data exchange across participants in choreography models, also allowing to translate one to the other type of data exchange specification. We will discuss this in more detail as part of our formal model for data-aware

service choreographies in Chapter 4. The result is, that already modeled choreographies relying completely on message-based data exchange can be analyzed to generate data-aware representations for them, by introducing cross-partner data objects and data flows based on the modeled message exchanges. The resulting data models can be manually refined or extended and are used as input for the TraDE methods in later life cycle phases. Furthermore, choreography models with already explicit cross-partner data flows could be analyzed to identify optimization potentials and incorporate findings from the *Analysis* phase using the collected choreography execution event data from the *Execution* phase of earlier runs of the choreography without doing a whole life cycle loop again.

The result of the Modeling phase, depicted in Figure 3.5, is a manually defined choreography model that comprises two or more participants, their interaction logic and a *CDM* comprising the specified cross-partner data objects and data flow. The interaction logic is specified through communication activities and message flows between the participants. The CDM consists of a set of cross-partner data objects, or *data objects* for short, that are explicitly connected with communication activities, participants or other modeling constructs through so-called *data connectors* defining the choreographies data flow. As shown in Figure 3.5 and described in Section 3.2, a data object has a unique identifier and contains one or more *data elements*.

3.3.2 Transformation

At the end of the modeling phase, a *Transformation* step takes place in which a collection of private process models is generated. Therefore, for each of the specified choreography participants an *abstract process model* is generated based on the specified participant's control and data flow. The generated process models together implement the globally agreed collaboration behavior defined through the choreography model and are used as templates in the *Refinement* phase. Such abstract process models are not directly executable since they lack required details such as actual process logic in addition to the specified conversations as well as technical details, e. g., required runtime

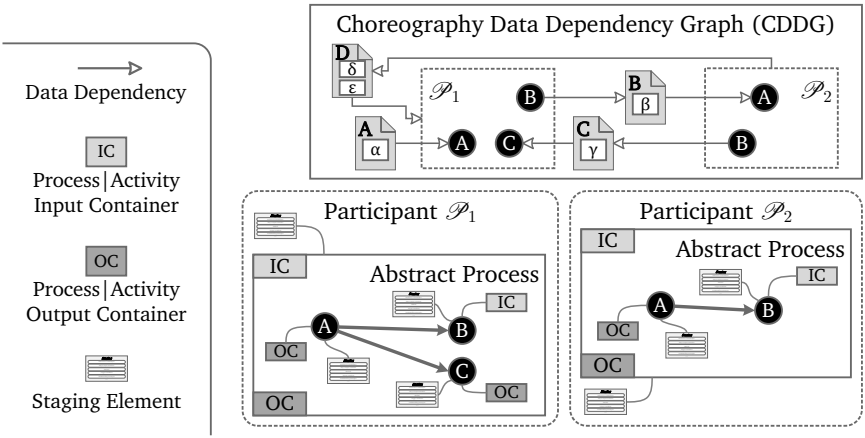

Figure 3.6: Detailed view on the *Transformation* step of the modeling phase, based on [HKL16a].

environment configuration data for successful deployment. BPMN process models or abstract Business Process Execution Language (BPEL) process models can be used to represent such abstract process models. An example of how such a transformation may look like is described by Reimann, Kopp, et al. [RK+08] using BPEL4Chor as choreography modeling notation and BPEL for the generated private process models. In the context of this work, the transformation additionally incorporates the results from applying our TraDE methods to the modeling phase such as the defined CDM and the modeled choreography data flow during the generation of the abstract process models. Therefore, the modeled data flow and the resulting data dependencies between the choreographed participants are extracted and summarized into a Choreography Data Dependency Graph (CDDG). Furthermore, the process models are enriched with so-called *Staging Elements* that reflect the specified cross-partner data exchange between participants from their own viewpoint. The final output of the modeling phase after the transformation step comprises the choreography model (CM), its data model (CDM) and the generated abstract process models with the overall CDDG as

shown in Figure 3.4. All generated models and graphs provide the basis for the manual refinement of the abstract process models to executable process models (also known as *workflows*) with transparent data exchange in the *Refinement* phase of our life cycle. Our goal is to use all available information provided on the choreography level to automatically enrich the generated abstract models depending on the underlying modeling languages.

The transformation itself is performed in three steps. First, an abstract process model in form of a PM-Graph according to our underlying metamodel is created for each of the choreography participants as shown in Figure 3.6. All participant-related definitions such as communication activities, control flow and intra-participant data flow is collected and added to the generated PM-Graphs. Second, the cross-partner data flows specified in the choreography model between cross-partner data objects, participants and activities are analyzed using the TraDE methods and the results are collected in the CDDG as shown in Figure 3.6. We rely on dependency analysis concepts well known from the domain of programming languages and compiler theory where the control and data dependencies between the statements of a program are identified and expressed in form of graphs to optimize the scheduling and execution of a program [KKP+81; Ott78]. The CDDG is a representation of the data dependencies between the choreography participants and we will use it to enrich the generated process models and conduct the modeled cross-partner data flow during later life cycle phases. A CDDG is a directed graph where the nodes are choreography participants, activities and data objects, and the edges the read and write dependencies between them as depicted in Figure 3.6.

During the *last step*, the data-specific model transformations will take place, i. e., the data dependencies collected in the CDDG are incorporated into the abstract process models or PM-Graphs, respectively, in form of the above mentioned *Staging Elements*, resulting in the artifacts shown in Figure 3.6. The structure of a staging element is shown in Figure 3.7.

We distinguish two general cases for staging elements, *pre-staging* and *post-staging*, depending on the time a staging element is executed. Pre-staging takes place before the implementation of an activity is executed and

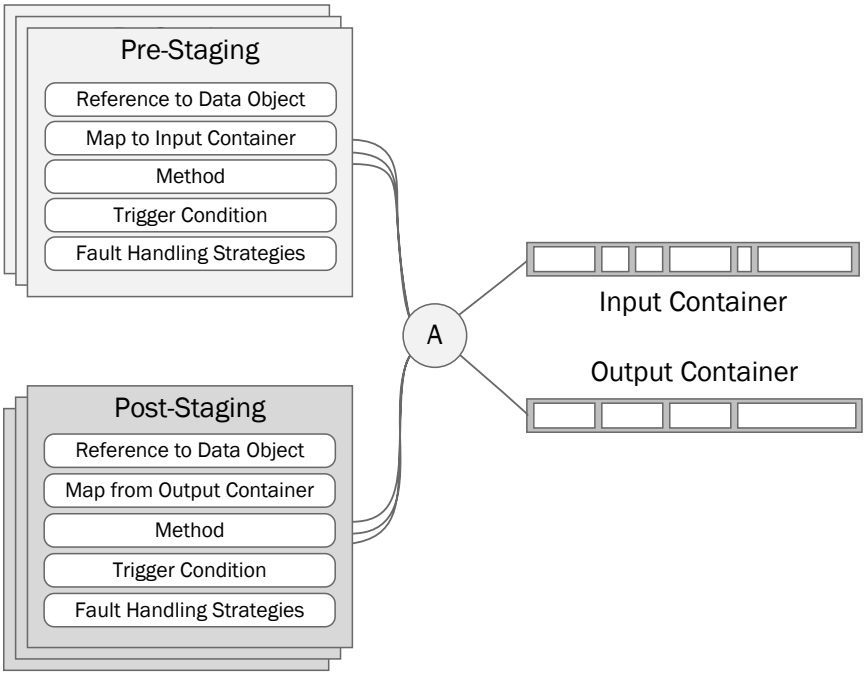

Figure 3.7: Structure of Pre- and Post-Staging Elements and their association to an activity.

therefore allows to use data from a cross-partner data object by materializing it within the input container of the associated activity. Post-staging takes place after the activity implementation is completed and therefore allows writing data from the activity's output container to a cross-partner data object. For each data dependency edge in the CDDG one staging element is generated in a process model and is associated to the activity or participant which is referenced by that edge.

As depicted in Figure 3.7, each staging element contains a *Reference* to a data object, a *Map* between the data elements of the referenced data object and the data elements of the activities' input or output container, the data staging *Method* that specifies if data should be *pulled* from a data object at

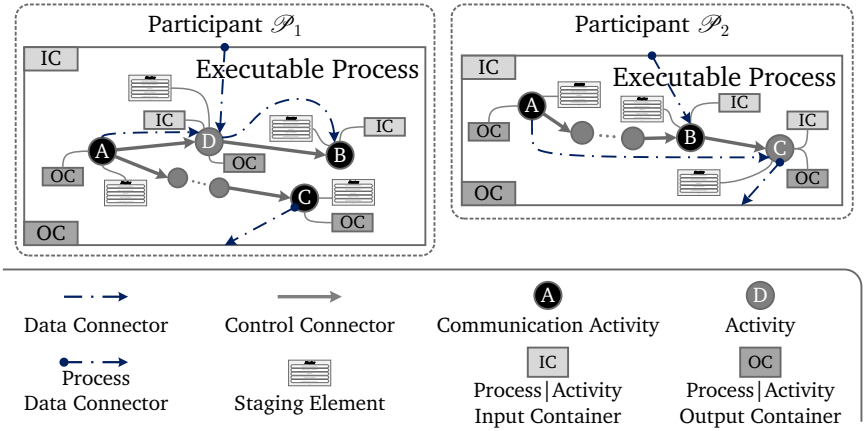

Figure 3.8: Detailed view on the *Refinement* phase, based on [HKL16a].

the TraDE Middleware to the activity's input container (pre-staging) or data from the activity's output container should be *pushed* to a data object at the TraDE Middleware (post-staging). Furthermore, a staging element allows to specify a *Trigger Condition* under which the data staging should take place, e. g., depending on the status of other activities or local data containers, and a set of *Fault Handling Strategies*. The staging elements are used to enact the transparent data exchange in choreographies. Staging elements and their use during choreography execution are discussed and described in full detail in Chapters 4 and 5.

3.3.3 Refinement

During the *Refinement* phase, IT specialists refine the generated abstract process models to make them executable. The result is a collection of executable process models also known as workflow models depicted in Figure 3.8. IT specialists of the different parties responsible for the choreography participants refine the participants internal logic by adding new model constructs, e. g., activities or control and data flow, as well as refine and extent the complete control flow and data flow of the process models. This also comprises the

specification of the actual activity implementations for all activities already specified at the level of the choreographies which are therefore reflected within the abstract process models used as a basis for the refinement.

Regarding the data flow, the data connectors and staging elements, generated as part of the model transformation step in the modeling phase, have to be potentially reconnected or the underlying mapping of data elements (*data maps* in PM-Graphs) from the data containers of the connected source and target constructs have to be refined. For example, when comparing the abstract process models shown in Figure 3.6 with the refined ones in Figure 3.8, the staging element associated to participants $\mathscr{P}_1$ and $\mathscr{P}_2$ are associated to the activities D and C, respectively. This results from the fact, that the underlying cross-partner data flow specified at the level of the choreography is defined between the two participants and data object D as shown in Figure 3.5. Therefore, at the level of the generated abstract process models the staging elements are linked to the participant boundaries, since there is no further information how and where the data will be actually processed within the participants' process models. During refinement, each of the IT specialists responsible for one of the participants has now the required insights to refine this data flow by updating the executable participant process models. Since in process model $\mathscr{P}_1$ a new activity D is introduced which requires data object D as an input, the respective staging element can be moved from the participant boundary and associated to activity D. The same applies to activity C and the related staging element in process model $\mathscr{P}_2$ as depicted in Figure 3.8.

In addition to the described refinements above, IT specialists often need to specify additional required data such as technical details, e. g., the transport protocol and content encoding for the message payloads to be used, and configuration data for the envisaged BPE used to execute the refined process models implementing the overall choreography. Such BPE configuration data of a process model is often summarized in a so-called process model *deployment descriptor*, where the contained information is processed by the BPE during the deployment of the related process model.

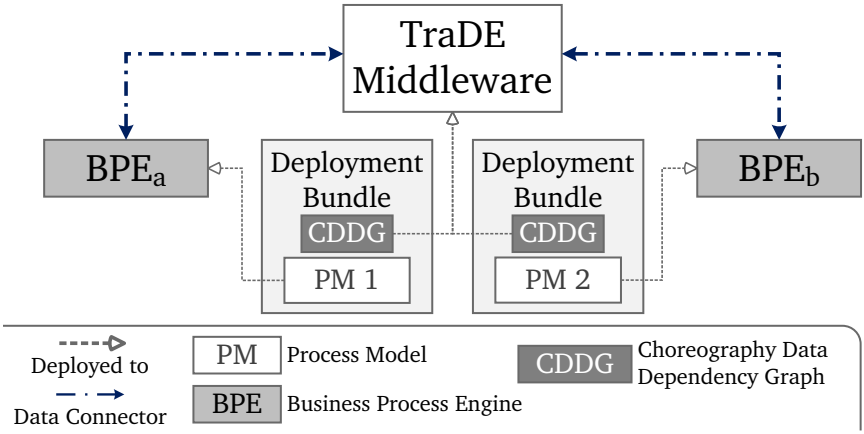

Figure 3.9: Detailed view on the *Deployment* phase, based on [HKL16a].

3.3.4 Deployment

In the *Deployment* phase the executable process models are packaged in the required format and deployed to the target BPE as shown in Figure 3.9. The extracted and generated CDDG is deployed to the TraDE Middleware to make the defined cross-partner data objects available during choreography runtime as described in Section 3.2.2.

Depending on the requirements of the collaborating parties, the deployment distribution can range from deploying all models to one central BPE to the deployment of each individual model to a different BPE node. Beside such static deployment strategies it is possible to defer or even automate the decision where each of the process models has to be deployed by dynamically estimating an optimal distribution that takes the location of invoked services and required data into consideration [BWV08a; CDL+07; DC08]. By leveraging cloud computing capabilities it is even possible to dynamically spin up new BPE nodes for an optimal placement of process models, services and data [Bin+13; Vuk+16]. Therefore, the TraDE methods could be used as a basis to take into consideration all available information, e. g., data dependency graphs, monitoring data or manually defined deployment re-

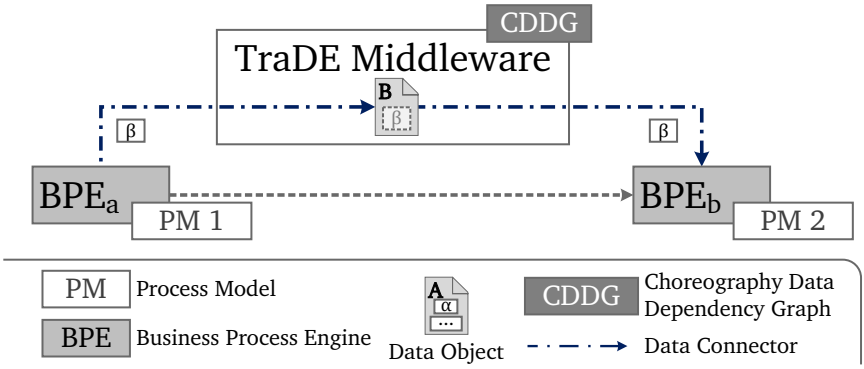

Figure 3.10: Detailed view on the *Execution* phase, based on [HKL16a].

quirements, to find an optimal deployment distribution based on respective state-of-the-art concepts. However, in the context of this work, we rely on the static deployment, whereas dynamic strategies may be considered within future work. At the end of the Deployment phase the choreography is deployed and prepared for execution as depicted in Figure 3.9, i. e., all process models are deployed to corresponding BPEs and the CDDG is deployed to the TraDE Middleware, respectively.

3.3.5 Execution

After the executable process models are deployed, they are ready to enter the *Execution* phase. By instantiating one or more of the deployed models, e. g., on behalf of a client's requests, the overall choreography is executed through the started interrelated process instances which together realize the modeled behavior of the choreography and represent a choreography instance. For details about how the execution of the choreography through such interrelated process instances looks like, we refer to Section 3.2.2.

Regarding the execution of the modeled cross-partner data flow, the respective BPE is pushing or pulling data to and from the defined cross-partner data objects at the TraDE Middleware as depicted as an example for data

element β of data object B in Figure 3.10. During the execution of the choreography instance each of the interrelated process instances produces a set of execution events at the underlying BPE. These events contain information about executed activities, control and data flow, occurred exceptions or faults and a variety of other aspects of process model executions. The same applies for the cross-partner data flow conducted by the TraDE Middleware. There also respective events are collected regarding the reading and writing of data from and to data elements of the cross-partner data objects based on their middleware-internal representation.

Such event data could be analyzed as part of our TraDE methods to detect potential issues as well as optimizations of the cross-partner data flows specified at the level of the choreography in future work. The analysis results can be used, e. g., to identify optimizations for data placement, data transfer and data lifecycle management. Based on predictions calculated from event data and additional monitoring information, data can be transmitted to a BPE in advance. In addition, optimal data lifecycle management ensures that data is only stored for as long as necessary and as short as possible.

3.3.6 Monitoring

The above mentioned execution events, emitted by the BPEs, are collected and analyzed during the *Monitoring* phase. The main target of monitoring is to ensure that the processes are actually carried out according to the assumptions made during the specification of the models [LR00]. Therefore, the monitoring has to detect unintended situations (e. g., exceptions or faults) by tracking the overall system status or analyzing the utilization of the processes and as a result enable corresponding users (e.g. system administrators) or specialized software components to react accordingly, e. g., by adapting process instances during runtime [BDH+08].

To monitor choreography instances based on the execution event data of the interacting process instances, these data need to be collected, analyzed, combined and interpreted. For example, the status of the choreography instance has to be calculated through the combination of the status of all

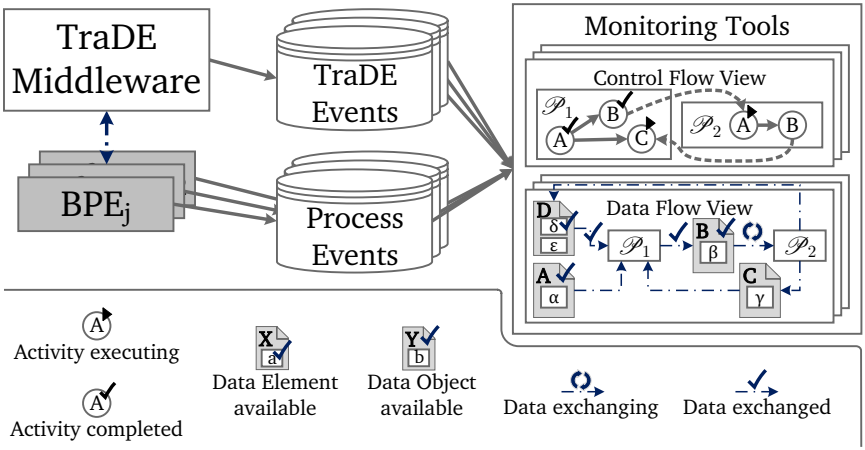

Figure 3.11: Detailed view on the *Monitoring* phase, based on [HKL16a].

underlying process instances. The resulting data can be expressed again in form of higher-level choreography events by corresponding *Monitoring Tools* as shown in Figure 3.11, so that the interpretation and combination has to be done only once and other interested parties are able to directly consume the choreography events instead of interpreting the lower-level process events on their own. This has also the advantage that the higher-level choreography events can be propagated and disclosed to all involved parties of the choreography, while keeping the detailed process events private to avoid the disclosure of the internal logic of the process models or further protected internal data of an involved party. A corresponding environment that enables the monitoring of choreographies is introduced, e. g., by Kopp et al. [KEL+11] and Wetzstein et al. [WKK+10]. For data-aware choreographies the modeled cross-partner data flow and the underlying interactions with the connected BPEs during choreography runtime have to be monitored, too.

This results in the two monitoring views or perspectives shown in Figure 3.11. The classical control flow view is provided by the monitoring capabilities of the BPE and the above mentioned aggregation of process events to choreography events. In addition, a data flow view can be build

by combining the data-related process events with the TraDE events to represent the current execution status of all modeled cross-partner data flows from a choreography viewpoint. Data-related events are emitted and stored to a respective database by the TraDE Middleware to allow the monitoring of the cross-partner data flow and thus to ensure that it is carried out as defined at the level of the choreography models. Since our focus within this work is on the modeling and execution of data-aware choreographies, the monitoring of data-aware choreography instances is not further discussed in the rest of the document.

3.3.7 Analysis

The goal of the *Analysis* phase in conjunction with the modeling phase is to produce choreography and process models that are optimal with respect to a set of requirements or Key Performance Indicators (KPIs). Already existing models from earlier life cycle iterations together with their monitoring data are therefore also taken into account to identify potential optimizations for the models. Established techniques and methods to identify optimization possibilities are, for example, the use of modeling best practices, the detection of so-called anti-patterns [KV07] or the simulation of model alternatives based on quantitative information, also known as *instrumentation* [LR00]. The applied optimizations might have an impact on both the level of the choreography and the process models since the conversations specified in the choreographies are implemented by the processes. Furthermore, the generated CDDG introduced by the TraDE methods might also be considered as a valuable input for the analysis phase towards potential data flow optimizations at the level of the choreography and process models. However, the analysis and optimization of data flow is not in the scope of this work, but may be considered as future work.

Chapter 4

Formal Model for Data-Aware Choreographies

In this chapter, we introduce a formal model for data-aware service choreographies together with a graphical notation. As introduced in Section 2.1 and discussed by Decker, Kopp, and Barros [DKB08], there are two categories of modeling approaches for choreographies: *interaction models* and *interconnected interface behavior models*, or *interconnection models* for short. Within this work, we follow the interconnection modeling approach which allows the specification of control flow per participant and interactions between the participants through message flows. As a result, data-aware choreographies are modeled through a set of interconnected participants where each of the participants specifies its interface and control flow, i. e., the internal behavior relevant for the interaction with others in the context of a choreography. Another important point to clarify is how modeled choreographies are executed.

An overview of possible approaches from related work is presented in Chapter 2, which can be summarized in two categories: the *native execution* of choreography models or following the *public-to-private approach* introduced by van der Aalst and Weske [AW01] as presented in Section 2.1.2. The former introduces corresponding middleware which allows the direct execution of a choreography model. Following the public-to-private approach, a public model, i. e., the choreography model, is transformed into a collection of private process models representing the participants of the choreography model. These private process models are then manually refined and enriched to make them executable on general-purpose Business Process Engines (BPEs). Our formal model follows the public-to-private approach and provides therefore the means for the modeling, transformation and refinement life cycle phases described in Section 3.3. Therefore, transformation and refinement of a data-aware choreography model to a set of interconnected private process models, as means to implement and execute choreography participants, their interactions and data exchanges according to the choreography model, is also presented and discussed as part of this chapter in more detail.

4.1 Overview

To get an overview of all relevant choreography elements and their connection, Figure 4.1 shows an example of a data-aware choreography and its major building blocks using the formal and graphical notation defined within this work in the following. We use this example to shortly introduce the structure and major building blocks of a data-aware choreography and later reuse parts of the example to visualize and explain the definition of respective choreography elements.

For the definition of our formal model for data-aware choreographies, we reuse and build on existing formal frameworks for process models and choreographies, namely Process Model Graphs (PM-Graphs) introduced by Leymann and Roller [LR00] and Choreography Model Graphs (CM-Graphs) introduced by Weiß et al. [WAHK17] and Weiß [Wei18]. Therefore, the

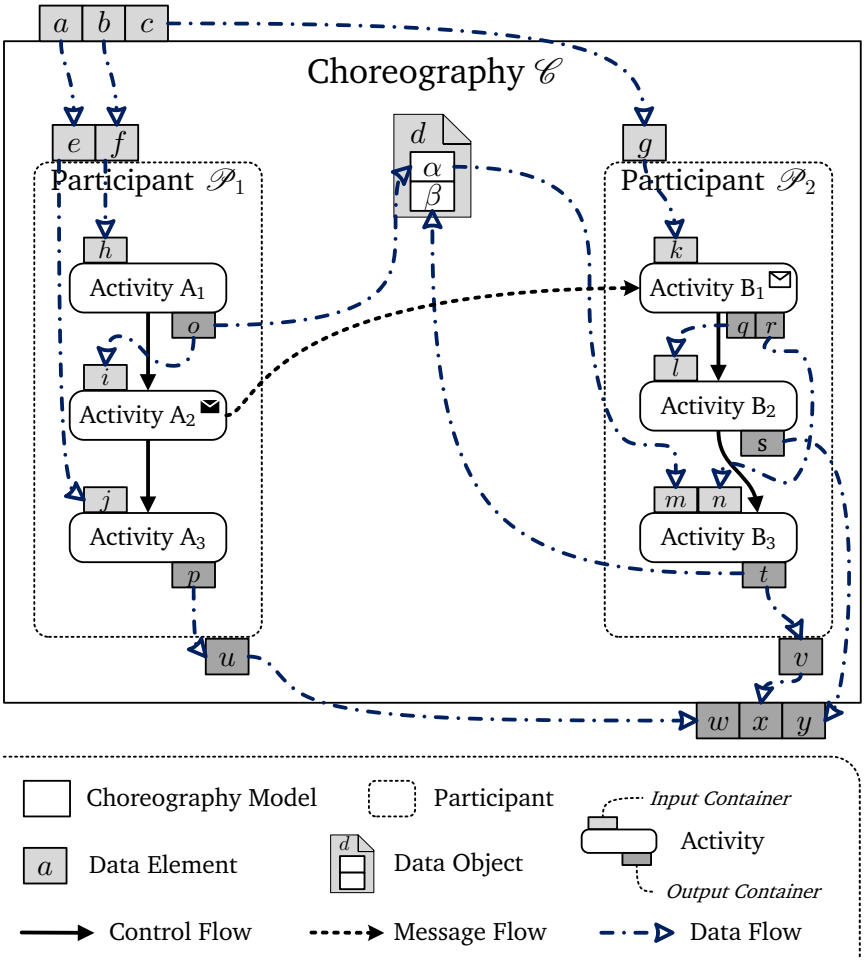

Figure 4.1: Example data-aware choreography model with two participants and a cross-partner data object.

syntax and semantics of our formal model are based on graph theory according to the PM-Graph metamodel introduced by Leymann and Roller [LR00]. A data-aware choreography model is represented as a special kind of directed graph G_C, called *Choreography Model Graph* or CM-Graph for short. A CM-Graph consists of a set of interacting parties, so-called *participants*, e. g., $\mathscr{P}_1$ and $\mathscr{P}_2$ in Figure 4.1. Each participant specifies its control flow through a partial order of *activities*, e. g., $A_1, \ldots, A_3$ in Figure 4.1, providing corresponding process logic linked through *control flow connectors*. Control flow connectors, or *control connectors* for short, are visualized in Figure 4.1 through solid black lines between activities. Choreography data is expressed through so-called *data elements*, e. g., a, b, c in Figure 4.1, and cross-partner data objects or *data objects* for short, e. g., d in Figure 4.1. Data flow between different elements of a choreography, e. g., participants and activities, is realized through copying values of data elements between so-called *data containers* by specifying *data flow connectors* or *data connectors* for short. Data connectors are visualized in Figure 4.1 through dotted dashed blue lines between data elements of data containers of different activities or data objects. The interaction between participants is specified through *message flow connectors*, or *message connectors* for short, by linking activities of two participants. This provides the ability to specify a handover of control and data from one participant to another through the exchange of messages, e. g., invoking other choreography participants or exchanging status or notifications with other participants at certain points in time throughout choreography execution. Message connectors are visualized in Figure 4.1 through dashed black lines connecting two activities of different participants.

According to the choreography depicted in Figure 4.1, Section 4.2 introduces the formal model for data-aware choreographies and their building blocks. Towards the transformation of choreography models to a collection of private process models, in Section 4.3 choreography data dependency graphs are defined as an intermediary model for capturing the data perspective of a choreography in a condensed manner. Furthermore, in Section 4.4 an extended formal model for process models based on the work of Leymann and Roller [LR00] is introduced to finally introduce the transformation and

refinement of a data-aware choreography to a collection of interconnected executable process models in Section 4.5.

4.2 Data-Aware Choreography Models

Following the approach of interconnection models [DKB08] and the public-to-private approach [AW01], a choreography participant can be typically seen as a restricted process model (cf. Figure 4.1). A data-aware choreography model then specifies the interconnections between the participant process models as well as their data and control flow from a global viewpoint. As a result, we can use the formal model of PM-Graphs introduced by Leymann and Roller [LR00] as a basis for the formalization of data-aware choreographies in form of CM-Graphs in a choreography modeling language independent manner. This provides us the basis for the specification of data-aware choreographies as well as their transformation and refinement to a collection of interconnected executable process models.

For all sets or maps from the PM-Graphs metamodel that have to be extended or refined for CM-Graphs, we denote the redefinition with a subscript C. For example, an input data container for an activity A in a CM-Graph will be denoted by $\iota_C(A)$ instead of $\iota(A)$ as defined for PM-Graphs by Leymann and Roller [LR00]. In the following sections, we will introduce the different elements of the formal model for data-aware choreographies. Starting with the definition of data specification in Section 4.2.1 followed by choreography activities and participants with their control and data flow, and finally summarizing all definitions into an overall definition of a data-aware choreography model in Section 4.2.8.

As outlined above, our formal model is based on graph theory. Let $\mathfrak{C}$ denote the set of all choreography models, then one particular choreography model $\mathscr{C} \in \mathfrak{C}$ is represented as a special kind of directed graph, called "CM-Graph" following the notion of a metamodel for process model graphs introduced by Leymann and Roller [LR00]. The node set N of a CM-Graph consists of all activities of a choreography model $\mathscr{C}$. The edge set E of a

CM-Graph prescribes all possible partial orders of executions of activities within participants of $\mathscr{C}$.

4.2.1 Choreography Data

As introduced by Leymann and Roller [LR00], *process data* describes all the information required to properly execute a process model. For example, the inputs and outputs of activities or control flow conditions which are used by the BPE executing a process instance by navigating through the defined control flow connectors, i. e., paths in the PM-Graph, of the underlying process model. Similarly, we introduce *choreography data* to enable the specification of required data and its structure already at the level of choreography models to support our Transparent Data Exchange (TraDE) concepts as introduced in Chapter 3. The goal is to enable modelers to specify all data relevant for actually executing the overall choreography model. Therefore, during execution concrete values of the modeled data represent the current state and context of the choreography and its participants. For example, data to be exchanged between participants, activity input and output data as well as control flow conditions of participants of a choreography model. Furthermore, all available information about choreography data is utilized within the transformation of the participants of a choreography model into a collection of interconnected private process models later. This reduces manual efforts and makes the refinement step less error-prone since the defined choreography data can be automatically incorporated and redefined within each of the generated private process models.

Choreography data has therefore to be defined and associated to a choreography model and its contained participants, activities, conditions as well as data objects and messages.

4.2.1.1 Data Elements

According to the *data elements* definition for PM-Graphs provided by Leymann and Roller [LR00], choreography data is collected in a set V which is

associated with a choreography model $\mathscr{C}$ according to Definition 4.1.

Definition 4.1 (Data Elements)
Let M be a set of names and S be a set of structures. Then,

1. $m \in M \wedge s \in S \Rightarrow < m; s > \in V$, *that means each pair consisting of a name and an already defined structure is a new valid data element.*
2. $\zeta_1, \ldots, \zeta_k \in S$ $(k \in \mathbb{N})$ *where* $\zeta_1, \ldots, \zeta_k$ *represents atomic structures, such as Integer, Float or String.*
3. $d \in V \Rightarrow d(k) \in S$ $(k \in \mathbb{N})$ *where* $d(k)$ *denotes an array of* k *elements over* d.
4. $\overline{V} \subseteq V \Rightarrow \times\overline{V} \in S$ *where* $\times\overline{V}$ *denotes a tuple over* $\overline{V}$. □

A member $v \in V$ of this set is called *data element*. Each data element has an associated name and structure (condition 1 of Definition 4.1). The name of the data element allows its identification and access to it, while its structure is defined based on atomic structures (condition 2 of Definition 4.1) such as *Integer*, *Float* or *String* or by the combination of already defined data elements in form of arrays (condition 3 of Definition 4.1) or tuples (condition 4 of Definition 4.1). While arrays enable the definition of an ordered collection of data elements with the same structure, tuples allow the definition of more complex structures by combining multiple data elements with different underlying structures into one named compound data element [LR00]. Therefore, data elements allow to specify data structures at the level of choreography models. During execution actual data values can be associated to an instance of a data element. For example, as shown in Figure 4.2 after activity A_1 is executed the resulting data values are materialized in the respective data elements a and b of the activity's output data container. From there the values of the data elements can be copied to the input containers of other activities through data flow connectors.

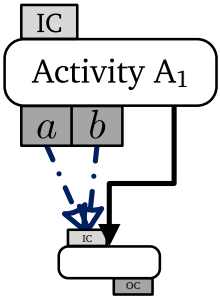

Figure 4.2: Example use of data elements to define data containers of activities.

4.2.1.2 Domains

Based on the specification of the type and structure of choreography data through data elements, we can define creation rules for valid instances of a data element through introducing *domains* according to Leymann and Roller [LR00]. This is summarized in Definition 4.2 for the three types of structures of data elements mentioned above: atomic, arrays and tuples.

Definition 4.2 (Domains)
Each data element $v \in V$ has an associated domain $DOM(v)$ that represents all of its corresponding well-formed values:

1. $DOM(< m; \zeta_i >) := domain(\zeta_i)$*, that means, the domain of an atomic data element is the set of all valid values of the associated atomic structure.*
2. $DOM(< m'; < m; s > (k) >) := DOM(< m; s >)(k)$*, that means, the domain of a data element structured as a k-element array is the set of all arrays with at most k values from the domain of the data element being the base for the array.*
3. $DOM(< m'; \times\{< m_1; s_1 >, \ldots, < m_r; s_r >\} >) := \underset{1 \leq i \leq r}{\times} DOM(< m_i; s_i >)$*, that means, the domain of a tuple-structured data*

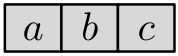

Figure 4.3: Visual representation of a data container.

element is the Cartesian product of the domain of the components of the tuple constructor. □

4.2.1.3 Data Containers

As outlined for the example choreography model presented in Section 4.1, activities and participants as well as choreography models as a whole need input and output data. To specify such inputs and outputs within a process model, Leymann and Roller [LR00] introduce the concept of *data containers* which represent a collection of data elements. Within the context of this work, we transfer this definition to choreography models. The visual representation of a data container representing a collection of data elements as part of our graphical notation is shown in Figure 4.3. Each of the labeled boxes within the overall data container rectangle refers to the name of one data element.

As means to represent *something that happens* at the level of PM-Graphs, i. e., activities and process models, Leymann and Roller [LR00] introduce the set $\mathcal{H}$. According to this definition, we collectively refer to the set of all activities, participants and data-aware choreography models as $\mathcal{H}_C$, i. e., $\mathcal{H}_C = N \cup \mathscr{P}(N) \cup \mathfrak{C}$. V denotes the set of choreography data associated with all CM-Graphs ($\mathfrak{C}$), N the set of all activities, $\mathscr{P}(N)$ the set of all participants, and $\mathcal{C}$ denotes the set of all conditions. Furthermore, let $\wp(X)$ denote the power set of a given set X, i. e., the set of all subsets of X. Then data containers at the level of a CM-Graph are defined according to Definition 4.3.

Definition 4.3 (Data Containers)
The map ι_C assigns to each activity, participant, choreography model

and condition its input container:

$$\iota_C : \mathcal{H}_C \cup \mathcal{C} \rightarrow \wp(V)$$

This means,

$$\forall X \in \mathcal{H}_C \cup \mathcal{C} : \iota_C(X) \subseteq V \text{ with card } \iota_C(X) < \infty$$

The map o_C assigns to each activity, participant and choreography model its output container:

$$o_C : \mathcal{H}_C \rightarrow \wp(V)$$

This means,

$$\forall X \in \mathcal{H}_C : o_C(X) \subseteq V \text{ with card } o_C(X) < \infty \qquad \square$$

This allows us in the following to specify the required input and provided output data of the above mentioned choreography constructs which will be later used as the source and target of specified data flow within a choreography model.

4.2.1.4 Messages

As outlined in the introduction of this chapter, a choreography model has to enable the specification of control flow per participant as well as conversations across participants. Such conversations comprise one or multiple interactions between two or more participants where corresponding collections of data elements, called *messages*, are exchanged. The purpose of a message is therefore to transport data and control from one participant to another at a specific point within a choreography model. As introduced in Section 3.3, this is possible by introducing a special type of activity, called *communication activity*, which allows to send and receive messages across participants at

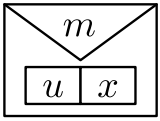

Figure 4.4: Visual representation of a message.

specific points within the control flow of a participant. For the definition of a message itself, it is enough to understand that they are created and consumed by such communication activities when a message is send or received, respectively. All details about communication activities will be introduced in Section 4.2.2. Therefore, messages can be interpreted as a special kind of data container, i. e., collection of data elements, which is transportable between two participants but has only a limited life time. We will discuss the meaning and use of messages as well as communication across participants in more detail when introducing message flow in Section 4.2.5.

To specify messages in a CM-Graph, we introduce, in accordance with the definition of data containers, an additional set of collections of data elements in Definition 4.4. This set represents all possible messages that can be exchanged during a conversation between participants. Therefore, a CM-Graph $\mathscr{C}$ will be associated with a hypergraph $\mathscr{M}$. Each edge of the hypergraph represents one distinguishable collection of data elements which is called a *message*. The visual representation of such a message as part of our graphical notation is shown in Figure 4.4. There we combined a message sketch with a label for the message name and the visual notation of a data container introduced above.

Definition 4.4 (Messages)
Let V be the set of all data elements defined in a choreography model graph $\mathscr{C} \in \mathfrak{C}$, we denote by $\mathscr{M}(V) \subseteq \wp(V) \setminus \emptyset$ the set of all messages defined within $\mathscr{C}$. Each member of $\mathscr{M}(V)$ is given by its name m. □

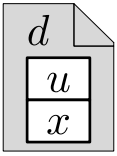

Figure 4.5: Visual representation of a data object.

4.2.1.5 Data Objects

Data containers enable the specification of input and output data of specific choreography elements and messages allow to model data to be exchanged across participants as part of a conversation. However, both of these modeling constructs are associated to one specific element of a model. For example, input and output data containers are defined and associated to one specific activity and a message belongs and is bound to one interaction between two participants. Therefore, as a central part of our TraDE approach, we introduce *cross-partner data objects*, or data objects for short, as outlined in Section 3.2. Such data objects enable the modeling of collections of data elements, in accordance with the definition of data containers in Definition 4.3, but data objects can be specified independent of any other model elements and participants in particular. Moreover, in comparison to messages, they also do not have an a priori restricted life time during choreography execution and therefore provide a durable, shared view on choreography data which is accessible from all participants, i. e., cross-partner data. Data objects allow to model the logical grouping of multiple semantically-related data elements, as a means to enable their identification, referencing them within a choreography model or adding annotations to specify custom properties, for example, regarding their operational semantics to influence their behavior during choreography runtime.

To specify data objects in a CM-Graph, we introduce an additional set of collections of data elements in Definition 4.5. Therefore, a CM-Graph $\mathscr{C}$ will be associated with a hypergraph $\mathscr{D}$. Each edge of the hypergraph represents one distinguishable collection of data elements which we call *data*

object. While the previously introduced data containers are only accessible within the boundaries of a single participant, data objects provide the means of globally shared, participant-independent data containers which enable the exchange of data across participants decoupled from the exchange of messages. The visual representation of such a data object as part of our graphical notation is shown in Figure 4.5. There we combined a document sketch with a label holding the data object's name and the visual notation of a data container introduced above, to visualize the contextual relation between data elements and data objects.

Definition 4.5 (Data Objects)
Let V be the set of all data elements defined in a choreography model graph $\mathscr{C} \in \mathfrak{C}$, we denote by $\mathscr{D}(V) \subseteq \wp(V) \setminus \emptyset$ the set of all data objects defined within $\mathscr{C}$. Each member of $\mathscr{D}(V)$ is given by its name d. □

For the definition and use of the above mentioned property annotations on data objects, we want to introduce two of them as an example, which we use within this work, namely *data object multiplicity* and *data object deletion*. The former allows us to distinguish between data objects that can be instantiated only once (single-instance) and data objects that can be instantiated multiple times (multi-instance) and therefore hold a collection of their data elements' values during runtime. Definition 4.6 introduces data object multiplicity as an example for specifying further properties on data objects during choreography modeling. Multi-instance data objects are very useful to easily model a flexible collection of instances of their contained data elements, which results in a list of identically structured data values that can be processed, for example, within a sequence of activities through a loop, while the loop counter can be dynamically extracted from the number of a data object's instances during runtime.

Definition 4.6 (Data Object Multiplicity)
Let $\mathscr{D}(V) \subseteq \wp(V) \setminus \emptyset$ the set of all data objects defined within a choreography $\mathscr{C}$ and let $\perp$ be a symbol to specify an undefined cardinality. The map $\mu_{\mathscr{D}} : \mathscr{D}(V) \to \mathbb{N} \cup \perp$ associates with each data object $d \in \mathscr{D}(V)$ its multiplicity, i. e., the maximum number of instances during runtime, where $\mu_{\mathscr{D}}(d) = \perp$ means that, the maximum number of data object instances during runtime is not known during modeling time. □

Definition 4.6 enables to specify an upper bound, i. e., the maximum number of instances of a data object during runtime. For example, to specify in a supply chain context that offers for a good should be requested from a predefined number of suppliers, lets say four, and the resulting offers are hold in a multi-instance data object d_{offers}, i. e., $\mu_{\mathscr{D}}(d_{\text{offers}}) = 4$. To express an unknown maximum number of data object instances during runtime, the bottom symbol $\perp$ can be used. This is the default case for *multi-instance data objects* since an exact number of potential instances during runtime is often not known during modeling time. For example, in an auction context bids from an unknown number of bidders have to be collected before successfully completing an auction, i. e., $\mu_{\mathscr{D}}(d_{bids}) = \perp$. To define that exactly one instance of a data object can exist at runtime, the multiplicity is set to 1, e. g., $\mu_{\mathscr{D}}(d) = 1$. This is the default case for *single-instance data objects* and therefore used as a default in our metamodel. The specified multiplicity then has an influence on the runtime behavior of how data objects are managed by the TraDE Middleware, for example, if a data object is specified as being single-instance, the TraDE Middleware blocks all requests trying to create an additional instance of the same data object within the context of a choreography instance. Within the scope of this work, the TraDE Middleware does not react with an error when blocking the creation of additional instances, since an overall fault handling concept for the TraDE Middleware is an open topic for future work. In contrast, for multi-instance data objects the TraDE Middleware allows the flexible creation and deletion of data object instances during runtime while enforcing the

potentially specified maximum number of allowed instances.

As another example for a data object property annotation, we introduce the specification of *data object deletion* strategies in Definition 4.7. This annotation enables the specification of different deletion strategies to enforce that instantiated data objects in context of a choreography instance and their associated data values are treated in a well-defined manner. This is of major importance since by introducing cross-partner data objects at the level of choreography models, the life time of the data is decoupled from the actual choreography instances, i. e., the process instances conducting the overall choreography model. Therefore, our formal model needs to provide the ability to specify how the life time of data objects and their deletion should be handled by the TraDE Middleware to enforce proper operational semantics, avoid ambiguity and enable garbage collection. In the following, we first introduce a formal definition of data object deletion strategies and their association to data objects in Definition 4.7, followed by an informal description of an initial, non-exhaustive set of concrete deletion strategies as well as respective values in form of events, expiration, deadlines, conditions or predefined values as an example.

Definition 4.7 (Data Object Deletion)
Let $\mathscr{D}(V) \subseteq \wp(V) \setminus \emptyset$ the set of all data objects defined within a choreography $\mathscr{C}$, let $\mathbf{P}_M$ be the set of all deletion strategy names, let $\mathbf{P}_V$ be the set of all deletion strategy values, and let $\perp$ be a symbol to indicate an optional or empty value, respectively. We denote by $\mathbf{P} \subseteq \mathbf{P}_M \times (\mathbf{P}_V \cup \{\perp\})$ the set of all deletion strategies, where each deletion strategy has a name and a set of complex defined values.
The map $\rho_{\mathscr{D}} : \mathscr{D}(V) \to \wp(\mathbf{P})$ associates with each data object $d \in \mathscr{D}(V)$ its deletion strategies through specifying a set of pairs each comprising a strategy name $m \in \mathbf{P}_M$ and a complex defined value $v \in \mathbf{P}_V$ or the bottom symbol $\perp$, where

$$\forall d \in \mathscr{D}(V) : (m, v_1), (m, v_2) \in \rho_{\mathscr{D}}(d) \Rightarrow v_1 = v_2.$$

To get an idea how concrete deletion strategies will look like, we introduce $\mathbf{P}_M := \{\text{manual, event, expiration, deadline, auto-archive}\}$ to informally describe an initial set in the following. These strategies have to be implemented within the TraDE Middleware to enable their automated execution. By specifying the *manual* deletion strategy for a data object d, e. g., $\rho_{\mathscr{D}}(d) = \{(\text{manual}, \perp)\}$, the TraDE Middleware will persist the data object until a user is explicitly manually archiving or deleting it. This might be useful for testing, debugging or analyzing choreography executions to identify potentially unintended behavior or data flow, or in addition especially in eScience scenarios to ease ad hoc reuse of intermediate and result data in other choreography instances or even other systems. The *event* strategy allows to specify that a data object d should be deleted on the occurrence of an event within a choreography, for example, if a participant or choreography instance reaches the *finished* life cycle state (cf. Section 3.5.1, "The Life of a Process" [LR00]), or an activity instance enters a specific state. For example, with $\rho_{\mathscr{D}}(d) = \{(\text{event}, \omega(A) = \text{completed})\}$, data object d is deleted as soon as activity A enters the *completed* state, specified through using the *activity state map* $\omega(A)$ as defined by Leymann and Roller [LR00] as deletion strategy value. The strategies *expiration* and *deadline* enable to specify a time interval until a data object expires (expiration) if not being used anymore or a concrete date time (deadline) when to delete a data object and its associated data values. For example, to specify that after ten days of inactivity a data object d should be deleted the following strategy can be associated to a data object d: $\rho_{\mathscr{D}}(d) = \{(\text{expiration}, \text{P10D})\}$. A deadline will look like the following: $\rho_{\mathscr{D}}(d) = \{(\text{deadline}, \text{2021-01-01T08:00:00})\}$. As underlying types for specifying time interval as well as date and time values the XML Schema Duration and DateTime data types can be used [XSD2]. Another approach to handle the garbage collection of data objects is the *auto-archiving* deletion strategy. The idea is that data objects, their instances as well as the associated data values are compressed and bundled into an archive and moved to a backup data store out of the TraDE Middleware. This allows to keep a copy of all relevant data without overloading the TraDE Middleware with the option of reloading the exported archive back to the

middleware, e. g., for reuse or debugging purposes.

Moreover, as defined in Definition 4.7 two or more deletion strategies of different types can be combined to specify a set of alternatives or complex conditions can be used as deletion strategy values to express more complex situations, for example, to enforce that a data object d and its data values will be automatically archived and deleted after not being accessed for two days after the underlying choreography instance is finished: $\rho_{\mathscr{D}}(d) = \{(\text{auto-archive}, \bot), (\text{expiration}, P2D \wedge \omega(\mathscr{C}) = \text{finished})\}$. Concepts for potentially conflicting strategies, their prioritization or composition are out of scope of this work. In general, the introduced data object annotations are just two examples for a potentially huge variety of additional properties that can be defined for cross-partner data objects and their operational semantics provided through the TraDE Middleware in future work.

4.2.2 Choreography Activities

As defined by Leymann and Roller [LR00], activities provide the notion for defining the pieces of work to be done to achieve the particular goal underlying to a process model, e. g., booking a flight. The same applies for activities at the level of a choreography model, however, here the focus is more on the interactions and conversations between the participants forming together the overall choreography and not on the internal logic of the individual participants. Or in other words, the goal of the choreography model is the successful collaboration of the individual participants, where each of them provides another part of the work to be done in order to achieve the overall goal which is not necessarily known by nor visible to each of the involved parties. For example, in a travel agency scenario, the travel agency uses a trip booking choreography model which defines the collaboration of different stakeholders, e. g., hotels, airlines or tour operators and car rental agencies, in form of participants and their conversations in order to fulfill the customer's requirements and book a trip by combining the offers of the involved parties. Therefore, the formal model allows to specify the input and output data of an activity, its underlying implementation, e. g., the piece

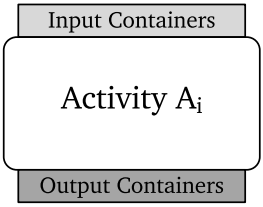

Figure 4.6: Visual representation of a choreography activity.

of software conducting the piece of work the activity represents as well as additional aspects such as staff assignments, in cases where an activity has to be conducted by a human worker. Since our focus is on the data dimension of choreography models, these additional aspects are not discussed within this work. For further details regarding activities at the level of PM-Graphs we refer to the work of Leymann and Roller [LR00].

Definition 4.8 (Choreography Activities)
The set of all activities of a choreography model $\mathscr{C} \in \mathfrak{C}$ is denoted by N. Each member of N is identified by its name. An activity A can be written as an operator: $A : \iota_C(A) \rightarrow o_C(A)$. □

Definition 4.8 summarizes our definition of a choreography activity by adapting the process activity definition of Leymann and Roller [LR00]. The visual representation of a choreography activity as part of our graphical notation is shown in Figure 4.6. There, the input and output data containers of an activity are visually attached to its boundary to graphically represent its operator characteristics, i. e., reflecting the inputs and outputs of the implementation conducting the activity.

Based on the definition of activities as well as defining their input and output data containers (cf. Definition 4.3), we are now able to define how to specify and associate the implementation of an activity. Therefore, we also have to consider the specification of conversations between participants within choreography models. The PM-Graph metamodel already provides

the means that an activity can trigger the invocation of another process model by specifying the process model to invoke as the implementation of the activity. However, this can be seen as kind of a black box approach, because of the underlying operator semantics of the activity. This means, the activity provides an abstraction for the referenced process model for which on activity execution a new process instance is created. The activity's input container is passed to the process instance which is executed until it completes and finally the output data is materialized in the activity's output container. As a result, the complete execution of the referenced process model is done in the background and is therefore not visible at the level of the process model containing the activity.

While the same should be possible at the level of choreography models, i. e., that a choreography activity can trigger another choreography or process model, we need something in addition to specify complex conversations between choreography participants as part of a choreography model. As outlined in Section 4.1, this is realized within our formal model by representing conversations between participants through the exchange of messages between them. Therefore, a message connector between respective activities of two participants has to be defined. Such message connectors together with their source and target activities and their implementations, explicitly represent the initial invocation of participants as well as any complex interactions between two or more participants within a choreography model. This is comprised in Definition 4.9, which allows to associate an implementation to a choreography activity, where a special type of implementations can be referenced for sending and receiving messages between participants.

Definition 4.9 (Activity Implementations)
Let $\mathcal{E}$ denote the set of all possible implementations of all activities, that means an element of $\mathcal{E}$ can be, for example, a program, script, web service, a process or choreography model, or even build-in functionality of a BPE, e. g., logic for sending and receiving messages using a specific transport protocol. The map $\Psi : N \rightarrow \mathcal{E}$ associates with each activity A

its activity implementation $\Psi(A)$.
An activity implementation itself is perceived as a map:

$$\Psi(A) : \underset{v \in \iota_C(A)}{\times} DOM(v) \rightarrow \underset{v \in o_C(A)}{\times} DOM(v) \qquad \square$$

Furthermore, for the same purpose, we introduce so-called *communication activities* in Definition 4.10 as a special kind of activity, with respective activity implementations associated for sending or receiving messages, being the source and target of message connectors, respectively. This provides us the basis for specifying arbitrary conversations between participants within a data-aware choreography model using message connectors which will be defined in detail in Section 4.2.5.

Definition 4.10 (Communication Activities)
Let $\mathcal{E}_{send} \subset \mathcal{E}$ denote the subset of all possible activity implementations that provide functionality to send messages, $\mathcal{E}_{receive} \subset \mathcal{E}$ denote the subset of all possible activity implementations that provide functionality to receive messages, and $m \in \mathcal{M}(V)$ a valid message that will be sent or received by such an activity implementation.
The set N_{send} denotes all sending communication activities, where each member $A_{send} \in N_{send}$ has an associated sending activity implementation. This means,

$$N_{send} := \{A_{send} \in N \mid \Psi(A_{send}) \in \mathcal{E}_{send}\}.$$

The set $N_{receive}$ denotes all receiving communication activities, where each member $A_{receive} \in N_{receive}$ has an associated receiving activity implementation. This means,

$$N_{receive} := \{A_{receive} \in N \mid \Psi(A_{receive}) \in \mathcal{E}_{receive}\}.$$

The set of all communication activities of a choreography model $\mathscr{C} \in \mathfrak{C}$ is denoted by $N_{com} \subseteq N$, where $N_{com} = N_{send} \cup N_{receive}$. □

Such communication activities enable to specify at which point within the participant's control flow an interaction with another participant is required. Therefore, communication activities can be seen as a kind of synchronization or checkpoints within a choreography model at which participants, for example, have to exchange information or notify each other about something that happened. For example, in the travel agency scenario mentioned above, the travel agency will send a flight booking request to one or multiple airlines and then waits for a respective response before starting to send corresponding hotel booking requests to one or more hotels. This can be modeled through a receiving communication activity that blocks until the arrival of a message from another participant, where reaching a specific checkpoint is modeled through a sending communication activity which sends the required message as soon as the control flow reaches the activity. Therefore, as defined in Definition 4.10, a sending communication activity, or more precisely its implementation, wraps the values of the data elements of its input container into a message which is send to the receiving communication activity specified as target of the underlying message connector. The same applies for receiving communication activities, where the values of the received message are extracted and stored into the activity's output data container. Therefore, a receiving communication activity does not have an associated input data container and a sending communication activity does not have an output data container by default.

However, we decided to not restrict the formal model in this manner, since the underlying communication activity implementations might require or provide some data due to technical reasons or which might be required or helpful in terms of fault handling. For example, if using a Hypertext Transfer Protocol (HTTP)-based communication activity implementation, the returned HTTP response status code might be materialized in the output container of a sending communication activity. We will discuss the modeling

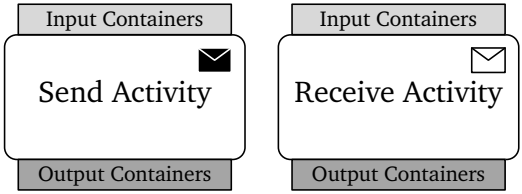

Figure 4.7: Visual representation of sending and receiving communication activities.

of conversations and the exchange of messages in more detail as part of the message flow of choreographies in Section 4.2.5.

Figure 4.7 shows our visual representation for communication activities. Therefore, we follow the visual notation introduced in BPMN 2.0 [BPMN] and add a black message icon to sending communication activities and a white message icon to receiving communication activities. Since the scope of this work is focused on the data dimension of choreography models, we do not introduce definitions for staff assignments as well as exit conditions for activities as provided for PM-Graphs by Leymann and Roller [LR00]. However, these definitions can be transferred one by one into our formal model for CM-Graphs.

4.2.3 Choreography Participants

To represent participants in a CM-Graph $\mathscr{C}$, we associate $\mathscr{C}$ with a hypergraph $\mathscr{P}$. Each edge of the hypergraph is one distinguishable set of choreography activities which represent the globally observable behavior of a *participant* within a choreography following the *interconnected interface behavior* modeling approach [DKL+08]. Definition 4.11 introduces our notion for specifying the participants of a choreography model.

Definition 4.11 (Participants)
Let N be the set of activities defined in a choreography model graph $\mathscr{C}$,

we denote by $\mathscr{P}(N) \subseteq \wp(N) \setminus \emptyset$ the set of all participants defined within $\mathscr{C}$ satisfying the following conditions

1. $\forall R_1, R_2 \in \mathscr{P}(N) : R_1 \cap R_2 \neq \emptyset \Rightarrow R_1 = R_2$,
2. $\forall R_1, R_2 \in \mathscr{P}(N)\ \forall A, B \in N\ \forall p \in \mathcal{C} : (A, B, p) \in E \Rightarrow \{A, B\} \subseteq R_1 \vee \{A, B\} \subseteq R_2$ *with E as the set of control connectors of $\mathscr{C}$ as defined in Definition 4.12,*
3. $\forall R_1, R_2 \in \mathscr{P}(N)\ \forall A, B \in N\ \forall m \in \mathscr{M}(V) : \Delta_{\mathscr{M}}(A, B, m) \neq \emptyset \Rightarrow A \in N_{send} \wedge B \in N_{receive} \wedge A \in R_1 \wedge B \in R_2 \wedge R_1 \cap R_2 = \emptyset$, *with $\Delta_{\mathscr{M}}$ as message connector map defined in Definition 4.13, and*
4. $\forall R_1, R_2 \in \mathscr{P}(N)\ \forall A \in R_1\ \forall B \in R_2 : \Delta_C(A, B) \neq \emptyset \Rightarrow R_1 = R_2$, *with $\Delta_C(A, B)$ as data connector map defined in Definition 4.17.*

Each member of $\mathscr{P}(N)$ is identified by its name R. □

Condition 1 of Definition 4.11 enforces that each activity of a choreography model can be only associated to one participant. Conditions 2-4 restrict the different types of connectors which can be specified between the activities of a choreography. The connectors are defined in detail in the following sections, however, for the sake of completeness their restrictions regarding choreography participants are listed in Definition 4.11 and are shortly summarized in the following. Condition 2 enforces that a control connector cannot cross participant boundaries, i. e., can only be specified between activities associated to the same participant. Condition 3 guarantees that if a message connector between two communication activities exists, the source of the message connector has to be a sending communication activity and the target a receiving communication activity, and in addition the source and target activity of the connector have to be associated to different participants. As a result, message connectors are only allowed to be defined across participant boundaries. Finally, condition 4 ensures that data connectors between activities are only specified within participant boundaries and not between activities associated to different participants.

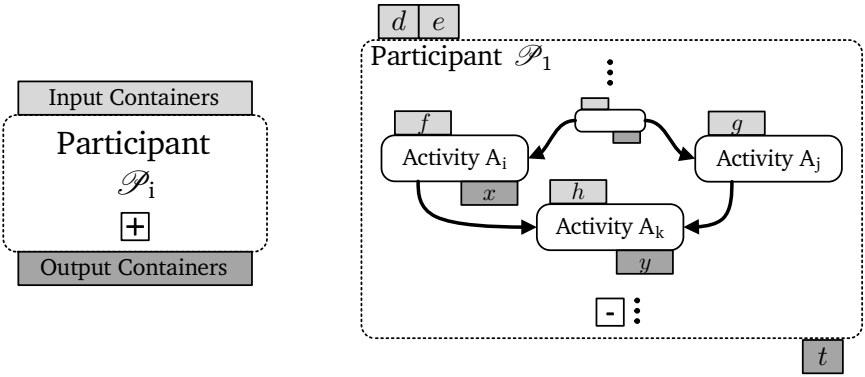

Figure 4.8: Visual representation of a choreography participant.

Our visual representation of a participant is depicted in Figure 4.8. It provides a collapsed as well as expanded representation of a participant obfuscating or showing the participant's activities, respectively. If a participant is collapsed or not is shown by the respective plus or minus sign at the bottom of the shape. The idea of collapsing complex structured modeling constructs is well known from respective modeling tools as a means to reduce the visual complexity and get a better idea of the big picture of a model. For example, collapsing all participants of a choreography model can help to get a better overview of the specified conversations and their associated messages without explicitly showing which activities are implementing the conversations in detail.

4.2.4 Control Flow

As introduced by Leymann and Roller [LR00], control flow allows us to specify valid partial orders of executions of activities, i. e., appropriate partial orders between the pieces of work to be done represented through the modeled activities. Therefore, an activity A can be connected through directed edges with its potential successor activities $A_1, \ldots, A_n$. The decision which of A's successors is executed next depends on business rules controlling the

transitions, also known as *transition conditions*. A transition condition is a predicate in the choreography data or more precisely in the data elements of the transition condition's input container (see Section 4.2.1.3). As a result, the truth value of a transition condition and consequently the actual flow of control through a choreography model $\mathscr{C}$ varies from choreography instance to instance depending on the actual data underlying to the instance. Therefore, the dependency of the control flow between activities and its associated transition conditions is made explicitly visible as part of the notion of control connectors defined in Definition 4.12.

Definition 4.12 (Control Connectors)
The set $E \subseteq N \times N \times \mathcal{C}$ is called the set of control connectors of a choreography model $\mathscr{C} \in \mathfrak{C}$, satisfying the condition

$$\forall R_1, R_2 \in \mathscr{P}(N)\ \forall A, B \in N\ \forall p \in \mathcal{C}:$$
$$(A, B, p) \in E \Rightarrow \{A, B\} \subseteq R_1 \vee \{A, B\} \subseteq R_2.$$

This means, a control connector can only be specified between activities associated to the same participant. For a control connector $(A, B, p) \in E$, the predicate $p \in \mathcal{C}$ is called a transition condition. Each transition condition p is considered as a Boolean function in its input container $\iota_C(p) \subseteq V$:

$$p : \underset{v \in \iota_C(p)}{\times} DOM(v) \rightarrow \{0, 1\}$$

□

Figure 4.9 depicts our visual representation of control connectors within participants of a choreography model.

Based on the definition, control connectors can be specified in an arbitrary manner between activities of a participant, however, Leymann and Roller [LR00] impose two additional restrictions on them which we also apply at the level of CM-Graphs. To avoid ambiguity and keep the control flow simple and comprehensive, the set of control connectors has to be *unified*, i. e., there is at most one control connector with a single transition condition specified

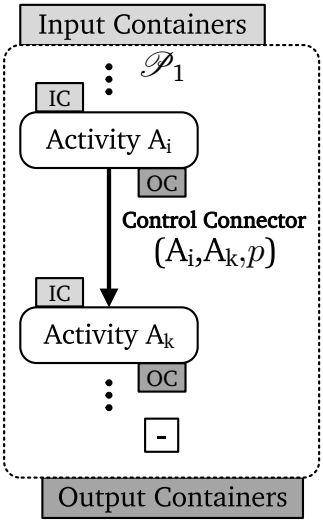

Figure 4.9: Visual representation of the control flow between activities within a choreography through control connectors.

for linking two activities. In addition, the control flow specified through a sequence of control connectors linking activities has to be *acyclic*, i. e., control connectors must not build loops. As a result, the set of control connectors within the participants of a choreography model are both unified and acyclic. For further details, a formal definition of the introduced restrictions as well as additional control flow aspects such as joining and forking the control flow or specifying join conditions as well as dead path elimination concepts, we refer to the work of Leymann and Roller [LR00].

4.2.5 Message Flow

As already outlined in Section 4.2.1.4, in addition to specifying the control flow between activities within a participant, we require a notion for modeling conversations across participants. Therefore, we introduced in Section 4.2.2 communication activities as a special combination of activity and activity implementation which allow us to model the sending and receiving of messages

as part of the participants' control flows. What is still missing is a modeling construct which allows to link message senders and receivers, i. e., the communication activities of different participants, to specify respective message exchanges and therefore model conversations between participants. Therefore, choreography modeling languages such as BPMN 2.0 or BPEL4Chor introduce an additional type of flow: so-called *message flow*. From the perspective of the above introduced definitions, message flow can be seen as a combination of control and data flow within a single connector. While a control connector only forwards control between activities within a single participant and a data connector only forwards data between activities or data objects, a corresponding message connector forwards control from a communication activity of one participant to a communication activity of another participant while also enabling to transfer data as part of the messages exchanged between the sending and receiving communication activities. As discussed in Section 4.2.1.4, a message is therefore similar to a data object, but has a restricted life time within our metamodel. This means as soon as the data contained in the message is materialized at the input container of the receiving communication activity to which the message connector points to, the message itself is discarded. Therefore, message connectors can be defined through a triple comprising two communication activities and a message representing a combined control and data connector across participants. In the following, we will have a closer look on the underlying definition as well as some other aspects regarding the message flow of a data-aware choreography model.

4.2.5.1 Message Connectors

Figure 4.10 depicts our notion for representing message flows within CM-Graphs via so-called *message connectors*. For the definition of message connectors we build on top of the specification of data flow through corresponding data connectors as defined for PM-Graphs by Leymann and Roller [LR00]. Therefore, the definition of a message connector has to prescribe which communication activities expect to send or receive a message to or from which

other communication activities and how the data elements of the messages are composed from data elements of the input containers of the sending communication activities or how the data elements of the output containers of receiving communication activities are composed from data elements of the messages, respectively. The former represents the *message exchanges* or *conversations* within the choreography model and the latter represent the *message data mappings*, i. e., the mapping between data containers of communication activities and the exchanged messages.

Based on that, a message connector is a directed edge between two communication activities with a link to the message to be exchanged as shown in Figure 4.10. The source of a message connector has to be a sending communication and the target a receiving communication activity, respectively. The required message data mappings can be defined within our formal model by specifying corresponding triples of data elements (v_1, v_2, v_3) with $v_1 \in \iota_C(A)$, $v_2 \in m$ and $v_3 \in o_C(B)$ for a sending communication activity A, a message m and a receiving communication activity B. The set of all such triples of data elements specified between two communication activities A, B and a message m is denoted as $\Delta_{\mathcal{M}}(A, B, m)$. The operational semantics of adding a triple (v_1, v_2, v_3) to $\Delta_{\mathcal{M}}(A, B, m)$ is that after the message exchange is performed, the data element v_3 of the output container $o_C(B)$ has received a copy of the actual instance, i. e., the data value, of data element v_1 of the input container $\iota_C(A)$. Therefore, the message is used as a kind of data transport vehicle between the two communication activities. First, the actual instance of data element v_1 of the input container $\iota_C(A)$ is copied to data element v_2 of the message using the specified activity implementation of the sending communication activity A which then sends the message to the other participant. As soon as the receiving communication activity B receives the message via its specified activity implementation the actual instance of data element v_2 of the message is copied to data element v_3 of its output container $o_C(B)$. According to this behavior, an element of the set $\Delta_{\mathcal{M}}(A, B, m)$ is called a *message map*. To bring everything together, our formal model defines a so-called *message connector map* $\Delta_{\mathcal{M}}$ which associates with each triple (A, B, m) the set $\Delta_{\mathcal{M}}(A, B, m)$.

Definition 4.13 (Message Connector Map)
Let $A \in N_{send}$ be a sending communication activity, let $B \in N_{receive}$ be a receiving communication activity, and let $m \in \mathcal{M}(V)$ the message to be exchanged between two participants. The map

$$\Delta_{\mathcal{M}} : N_{send} \times N_{receive} \times \mathcal{M}(V) \rightarrow \bigcup_{A \in N_{send}, B \in N_{receive}, m \in \mathcal{M}(V)} \wp(\iota_C(A) \times m \times o_C(B))$$

satisfying the conditions

1. $\Delta_{\mathcal{M}}(A, B, m) \in \wp(\iota_C(A) \times m \times o_C(B))$,
2. $\forall R_1, R_2 \in \mathcal{P}(N) : \Delta_{\mathcal{M}}(A, B, m) \neq \emptyset \Rightarrow A \in N_{send} \wedge B \in N_{receive} \wedge A \in R_1 \wedge B \in R_2 \wedge R_1 \cap R_2 = \emptyset$,
3. $\forall C \in N_{receive} : \Delta_{\mathcal{M}}(A, B, m) \neq \emptyset \Rightarrow C = B$,
4. $\forall C \in N_{send} : \Delta_{\mathcal{M}}(A, B, m) \neq \emptyset \Rightarrow C = A$,
5. $\forall B \in N_{receive} : (x, m_1, z), (y, m_2, z) \in \bigcup_{A \in N_{send}, m \in \mathcal{M}(V)} \Delta_{\mathcal{M}}(A, B, m) \Rightarrow x = y \wedge m_1 = m_2$

is called a message connector map. An element (v_1, m_1, v_2) is called a message map.
The set of all message connectors $\boldsymbol{E}_{\Delta_{\mathcal{M}}}$ is defined as

$$\boldsymbol{E}_{\Delta_{\mathcal{M}}} := \{(A, B, m, \Delta_{\mathcal{M}}(A, B, m)) \in N_{send} \times N_{receive} \times \mathcal{M}(V) \times \wp(V \times V \times V) \mid \Delta_{\mathcal{M}}(A, B, m) \neq \emptyset\}$$

□

Condition 1 of Definition 4.13 enforces that a message connector specifies only message maps between the input container of the sending communication activity it originates from to the output container of the receiving communication activity it points to using the associated message to transport the values of the data elements in between. Condition 2 guarantees that

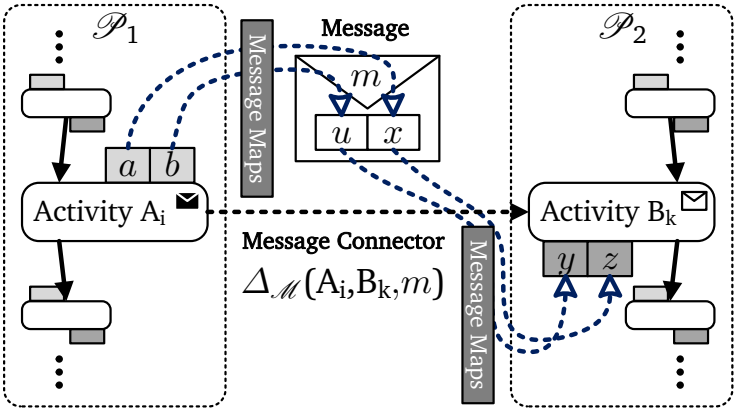

Figure 4.10: Visual representation of a message connector to define the exchange of data element values across participants through a message.

if a message connector between two activities exists, i. e., their message connector map is not empty, the source of the message connector has to be a sending communication activity and the target a receiving communication activity, and in addition each of the activities has to belong to another participant. Condition 3 and 4 prohibit that two message connectors have the same source (condition 4) or target (condition 3), i. e., each sending and receiving communication activity has at most one outgoing or incoming message connector specified, respectively. Finally, condition 5 prohibits two different message maps from having the same data element in the output container of the receiving communication activity as target. As discussed by Leymann and Roller [LR00], this guarantees the determinism of the formal model since it avoids conflicts and ambiguity during runtime when data container instances have to be composed and materialized based on the specified mappings and potentially multiple message maps could provide their source data element instances to copy them to the specified target data element. Therefore, all defined message connectors have to always provide a conflict free set of message maps.

Figure 4.10 shows a concrete example with two participants $\mathscr{P}_1$ and $\mathscr{P}_2$ connected through a message connector via the sending activity A_i, the message m and the receiving activity B_k. Based on the specified message maps, data element $x \in m$ will receive a copy of the instance of data element $a \in \iota_C(A_i)$ and data element $u \in m$ will receive a copy of the instance of data element $b \in \iota_C(A_i)$. The message is then send from participant $\mathscr{P}_1$ to participant $\mathscr{P}_2$, where activity B_k is receiving the message.

Finally, the payload of the message m (data elements u, x) is copied to the output container of activity B_k. Data element $y \in o_C(B_k)$ will receive a copy of the instance of data element $u \in m$ and data element $z \in o_C(B_k)$ will receive a copy of the instance of data element $x \in m$. Thus, it is $(a, x, z), (b, u, y) \in \Delta_{\mathscr{M}}(A_i, B_k, m)$.

4.2.5.2 Choreography Message Connectors

While normal message connector maps specify the message exchanges between participants within a choreography model, the so-called *choreography message connector maps* introduced in Definition 4.14 allow to specify incoming and outgoing messages of the choreography model as a whole. Therefore, they are comparable with the *process data connectors* introduced by Leymann and Roller [LR00] at the level of PM-Graphs to enable the specification of data flow between defined process inputs and outputs and respective activities processing or producing them, respectively. In accordance, at the level of CM-Graphs, choreography message connectors allow the specification of incoming and outgoing messages of a choreography model and by which communication activities they are received for processing them or send with respective data to an external recipient. The main use case for such choreography message connectors is the message-based invocation of a choreography model as well as to return a message to the initial requester on choreography instance termination. Message-based invocation is a well established concept used in choreography and process modeling notations, e. g., Business Process Model and Notation (BPMN) provides message start events or a receive task and Business Process Execution Language (BPEL)

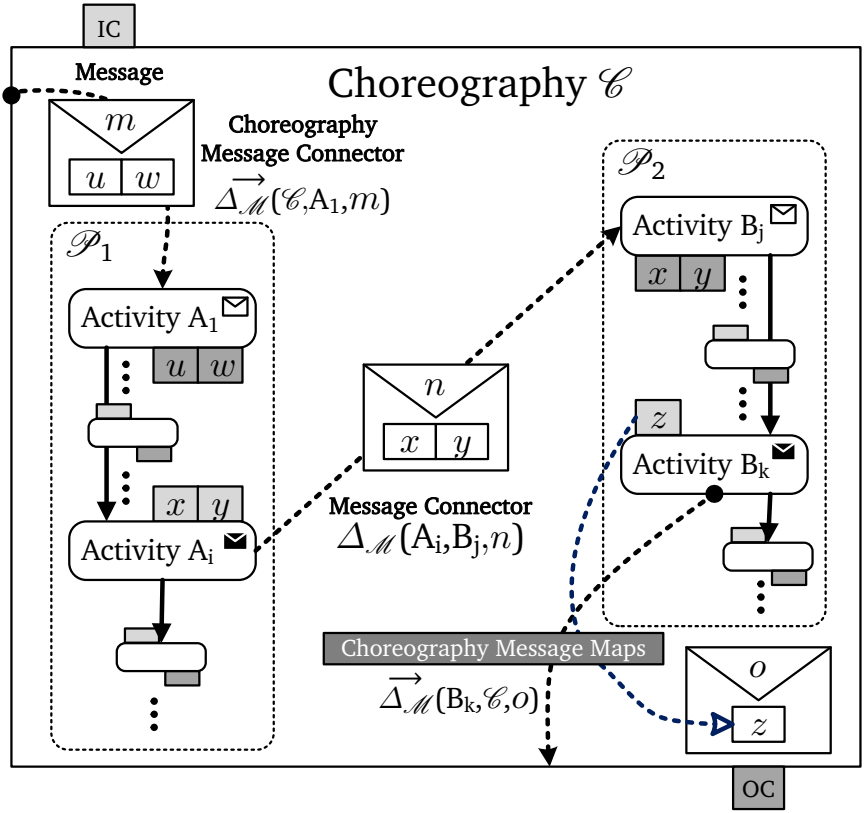

Figure 4.11: Visual representation of a choreography message connector to define the exchange of messages between the choreography as a whole and external entities.

introduces a receive activity with corresponding instance-creation semantics, respectively. We support this concept by introducing choreography message connectors and a respective map $\overrightarrow{\Delta_{\mathscr{M}}}$ within our formal model to enable the modeling of incoming and outgoing messages of a choreography model.

Figure 4.11 shows our visual notation for such choreography message connectors and how they are used to specify incoming and outgoing messages of a choreography model. Therefore, they extend the visual notation

of message connectors also shown in Figure 4.11 by adding a black circle to the source of the arrow connecting either the choreography boundary with a receiving communication activity for incoming messages or a sending communication activity with the choreography boundary for outgoing messages. To model the processing of an incoming message m, a choreography message connector represented by the choreography message connector map $\overrightarrow{\Delta_{\mathscr{M}}}(\mathscr{C}, A_1, m)$, is added between the boundary of the choreography and the receiving communication activity A_1 as depicted in Figure 4.11. For specifying the sending of an outgoing message o, a choreography message connector represented by the choreography message connector map $\overrightarrow{\Delta_{\mathscr{M}}}(B_k, \mathscr{C}, o)$, is added between the sending communication activity B_k and the boundary of the choreography as depicted on the lower right of Figure 4.11.

Definition 4.14 provides the notion for the specification of such choreography message connector maps within our formal model for CM-Graphs.

Definition 4.14 (Choreography Message Connector Map)
Let $\mathscr{C} \in \mathfrak{C}$ be a choreography model, let $A \in N_{receive}$ be a receiving communication activity, let $B \in N_{send}$ be a sending communication activity, and let $m \in \mathscr{M}(V)$ the message to be exchanged with the choreography. The map

$$\overrightarrow{\Delta_{\mathscr{M}}} : (\{\mathscr{C}\} \cup N_{receive}) \times (N_{send} \cup \{\mathscr{C}\}) \times \mathscr{M}(V) \rightarrow \bigcup_{\substack{\mathscr{C} \in \mathfrak{C},\, A \in N_{receive},\\ B \in N_{send},\, m \in \mathscr{M}(V)}} (\wp(m \times o_C(A)) \cup \wp(\iota_C(B) \times m))$$

satisfying the conditions

1. $\overrightarrow{\Delta_{\mathscr{M}}}(\mathscr{C}, A, m) \in \wp(m \times o_C(A))$,
2. $\overrightarrow{\Delta_{\mathscr{M}}}(B, \mathscr{C}, m) \in \wp(\iota_C(B) \times m)$,
3. $\forall C \in N_{receive} : \overrightarrow{\Delta_{\mathscr{M}}}(\mathscr{C}, A, m) \neq \emptyset \Rightarrow C = A \vee \overrightarrow{\Delta_{\mathscr{M}}}(\mathscr{C}, C, m) = \emptyset$,
4. $\forall C \in N_{send} : \overrightarrow{\Delta_{\mathscr{M}}}(B, \mathscr{C}, m) \neq \emptyset \Rightarrow C = B \vee \overrightarrow{\Delta_{\mathscr{M}}}(C, \mathscr{C}, m) = \emptyset$

is called a choreography message connector map. An element (m_1, v_1) *or* (v_2, m_2) *is called a choreography message map.* □

The conditions listed in Definition 4.14 are similar to the ones discussed for message connectors in general in the previous section. Therefore, each of the conditions is only shortly discussed in the following. Conditions 1 and 2 ensure that a choreography message connector specifies only choreography message maps between incoming choreography messages and the output containers of receiving communication activities the connector points to or between the input containers of sending communication activities the choreography message connector originates from and outgoing choreography messages. In addition, conditions 3 and 4 ensure that each incoming message is received and processed by exactly one receiving communication activity and each outgoing message is produced and send by exactly one sending communication activity.

4.2.5.3 Start Activities

Following the above introduced concept of message-based invocations of a choreography model, i. e., a new choreography instance is created based on a received initial request message, a choreography model therefore specifies one or more *start activities*. These start activities represent the entry points of the underlying CM-Graph and therefore immediately enter the executable state when a new instance of the choreography model is created [LR00]. Definition 4.15 introduces the set of all choreography start activities N'.

Definition 4.15 (Choreography Start Activities)
A choreography activity having no incoming control or message connectors and one incoming choreography message connector is called choreography start activity:
Let N be the set of activities of a choreography model $\mathscr{C}$*. We denote by*

N' the set of all start activities of $\mathscr{C}$:

$$A \in N' :\Leftrightarrow \left(\{e \in E \mid \pi_2(e) = A\} \cup \bigcup_{B \in N,\, m \in \mathscr{M}(V)} \Delta_{\mathscr{M}}(B, A, m) \right) = \emptyset$$
$$\wedge \bigcup_{m \in \mathscr{M}(V)} \overrightarrow{\Delta_{\mathscr{M}}}(\mathscr{C}, A, m) \neq \emptyset.$$

□

Where π denotes the projection map between Cartesian products and its indices specify the components of a respective tuple to select, i. e., the domain onto which to project. For example, the projection $\pi_2(e)$ projects an element e of the Cartesian product $E \subseteq N \times N \times \mathcal{C}$ to its second component which is the target activity $A \in N$ of the control connector e.

While at the level of PM-Graphs, there is only one type of start activities, within CM-Graphs we have to distinguish between choreography and participant start activities. The former mark the entry points of the choreography model as a whole as introduced above, while the latter define the entry point of a single participant within a choreography model. Such participant start activities are therefore representing the start activities of the resulting private process models the choreography model is transformed to in order to get executed. Therefore, a process instance of a participant's private process model is created based on the received request message. A choreography start activity is by definition also always a participant start activity, since the choreography is conducted through its participants' private process models and therefore the choreography start activity processing the initial request message resulting into a new choreography instance is defined within one of the choreography's participants.

Figure 4.11 depicts both types of start activities and their relation at the level of a choreography model. There, activity A_1 is both a choreography and a participant start activity because it marks the single entry point of the choreography as a whole as well as for participant $\mathscr{P}_1$. Activity B_j represents a participant start activity because a new instance of participant $\mathscr{P}_2$ will

be created on receiving message n from participant $\mathscr{P}_1$ as specified by the related message connector. The definition of participant start activities within our formal model is summarized in Definition 4.16.

Definition 4.16 (Participant Start Activities)
A choreography activity having no incoming control connectors and an incoming message connector is called participant start activity. In addition, a choreography start activity is by definition also always a participant start activity.
Let N be the set of activities defined in a choreography model $\mathscr{C}$ and let $R \in \mathscr{P}(N)$ be a participant, we denote by $N'(R)$ the set of all start activities of a participant R:

$$\begin{aligned} A \in N'(R) :\Leftrightarrow\ & A \in N' \\ & \vee (A \in R \wedge \{e \in E \mid \pi_2(e) = A\} = \emptyset \\ & \wedge \bigcup_{B \in N,\, m \in \mathscr{M}(V)} \Delta_{\mathscr{M}}(B, A, m) \neq \emptyset) \end{aligned}$$

□

We will further discuss related aspects regarding choreography and process instances in Section 4.2.7 when introducing the correlation of messages and data objects with choreography instances.

4.2.6 Data Flow

After introducing control flow and message flow which allow us to specify partial orders of activities within participants as well as conversations and message exchanges across participants, a notion for the specification of the data flow within a choreography model is missing. Beside the data flow between the activities of a choreography participant which is quite similar to the definitions at the level of PM-Graphs as introduced by Leymann and Roller [LR00], we add our notion for cross-partner data flows which allows us to specify intra-participant as well as inter-participant data flow in a

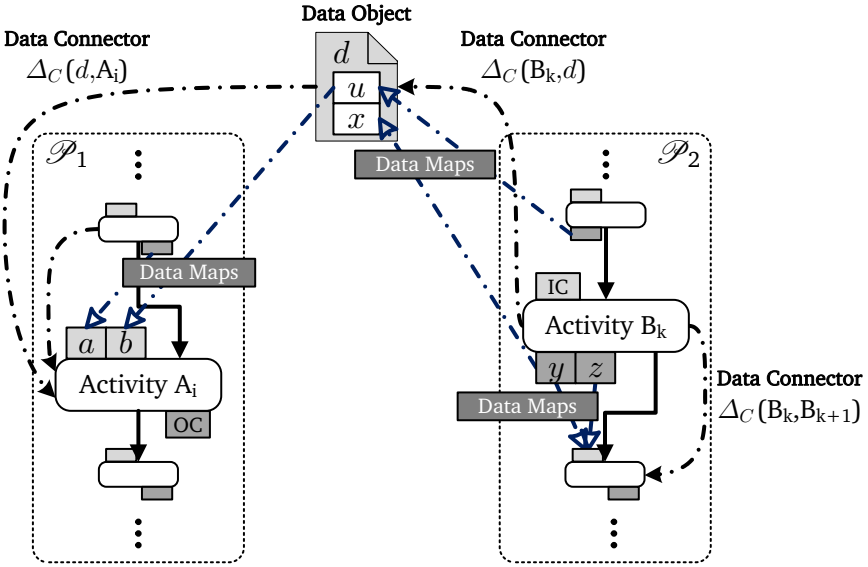

Figure 4.12: Visual representation of data connectors to define the exchange of data element values between activities and data objects.

seamless manner within choreography models as discussed in Section 3.2. Therefore, in this section we introduce the specification of data flow between the data containers of a choreography, participant, activity or predicate introduced in Section 4.2.1.3 and the cross-partner data objects presented in Section 4.2.1.5 as part of our formal model for CM-Graphs.

4.2.6.1 Data Connectors

Figure 4.12 depicts our notion for representing the flow of data within CM-Graphs via so-called *data connectors*. For the definition of data connectors within a CM-Graph, we build on top of the definition of data connectors introduced by Leymann and Roller [LR00] for PM-Graphs. Therefore, the definition of a data connector has to prescribe which activities or predicates expect input data from which other activities or data objects and how the data elements of an input container are composed from data elements of

the output containers of these other activities or the data elements of a data object, respectively. The former represents the *data dependencies* within the choreography model and the latter represent the *data mappings*, i. e., the mapping between data containers of activities, predicates and data objects.

Based on that, a data connector is a directed edge between an activity or data object and another activity, predicate or data object as shown in Figure 4.12. The required data mappings can be defined within our formal model by specifying tuples of data elements (v_1, v_2) with $v_1 \in (o_C(A) \cup d)$ and $v_2 \in (\iota_C(B) \cup e)$ for an activity A, an activity or predicate B and data objects d, e. The set of all such tuples of data elements specified, for example, between activities A and B is denoted as $\Delta_C(A, B)$. The operational semantics of adding a tuple (v_1, v_2) to $\Delta_C(A, B)$ is that when the input container of activity B is materialized at runtime, the data element v_2 of the input container $\iota_C(B)$ will receive a copy of the actual instance, i. e., the data value, of data element v_1 of the output container $o_C(A)$. According to this behavior, an element of the set $\Delta_C(A, B)$ is called a *data map*. To bring everything together, our formal model defines a so-called *data connector map* Δ_C in Definition 4.17 which associates with each tuple (A, B) the set $\Delta_C(A, B)$. The same applies to the specification of data flow between an activity and a data object. For example, the data connector between data object d and activity A_i shown in Figure 4.12 represents a data dependency of the activity on the data object and the underlying data mappings. This is summarized in the data connector map $\Delta_C(d, A_i) = \{(u, b)\}$. Based on this map, data element b of the input container $\iota_C(A_i)$ will receive a copy of the actual instance of data element u of data object d as soon as the input container of activity A_i is materialized at runtime.

Definition 4.17 (Data Connector Map)
Let $A \in N$ be an activity, let $B \in N \cup C$ be an activity or a predicate, and let $d, e \in \mathcal{D}(V)$ be data objects. The map

$$\Delta_C : (N \cup \mathscr{D}(V)) \times (N \cup \mathcal{C} \cup \mathscr{D}(V)) \rightarrow$$
$$\bigcup_{\substack{A \in N,\, B \in N \cup \mathcal{C}, \\ d,e \in \mathscr{D}(V)}} (\wp(o_C(A) \times \iota_C(B)) \cup \wp(d \times \iota_C(B)) \cup \wp(o_C(A) \times d) \cup \wp(d \times e))$$

satisfying the conditions

1. $\forall R_1, R_2 \in \mathscr{P}(N)\ \forall A \in R_1\ \forall B \in R_2 : \Delta_C(A,B) \neq \emptyset \Rightarrow R_1 = R_2$
2. $\forall A \in N\ \forall B \in (N \cup \mathcal{C}) : \Delta_C(A,B) \in \wp(o_C(A) \times \iota_C(B))$,
3. $\forall d \in \mathscr{D}(V)\ \forall B \in (N \cup \mathcal{C}) : \Delta_C(d,B) \in \wp(d \times \iota_C(B))$,
4. $\forall A \in N\ \forall d \in \mathscr{D}(V) : \Delta_C(A,d) \in \wp(o_C(A) \times d)$,
5. $\forall d, e \in \mathscr{D}(V) : \Delta_C(d,e) \in \wp(d \times e)$,
6. $\forall A_1, A_2 \in N : \Delta_C(A_1,A_2) \neq \emptyset \Rightarrow A_2$ *is reachable from* A_1,
7. $\forall A_2 \in N \cup \mathscr{D}(V) : (x,z),(y,z) \in \bigcup_{A_1 \in N \cup \mathscr{D}(V)} \Delta_C(A_1,A_2) \Rightarrow x = y$

is called a data connector map. An element $(v_1, v_2) \in \Delta_C(A,B)$ *is called a data map. The set of all data connectors* $\boldsymbol{E}_{\Delta_C}$ *is defined as*

$$\boldsymbol{E}_{\Delta_C} := \{(A, B, \Delta_C(A,B)) \in (N \cup \mathscr{D}(V)) \times (N \cup \mathcal{C} \cup \mathscr{D}(V)) \times \wp(V \times V) \mid \Delta_C(A,B) \neq \emptyset\} \quad \square$$

As shown in Definition 4.17, certain restrictions apply to data connector maps or data connectors in general which we discuss in the following. Condition 1 ensures that a data connector representing intra-participant data flow only exists between two activities associated to the same participant, since cross-partner data flow is specified through data objects. Conditions 2 ensures that data connectors between activities or an activity and a predicate only specify data maps between the output containers of the activity the data connector originates from and the input container of the activity or

predicate it points to. Conditions 3 and 4 ensure the same for data connectors representing inter-participant, i. e., cross-partner data flow. This ensures that only data maps are specified between the data elements of the data object a data connector originates from to the input container of the activity or predicate it points to or between the output container of the activity it originates from to the data elements of the data object it points to, respectively. In addition, condition 5 ensures that a data connector between two data objects specifies only data maps between the data elements of these data objects. Regarding the data flow between activities as defined by Leymann and Roller [LR00] for PM-Graphs, an activity B can only expect or depend on input data from an activity A, if A is executed before B, i. e., A is a predecessor of B or in other words there is a control flow path from A to B and therefore B is *reachable* from A at the level of the CM-Graph. Therefore, condition 6 in Definition 4.17 enforces that a data connector between two activities can only exist, if there is a control flow specified between the source and the target node of the data connector. Finally, condition 7 prohibits two different data maps from having the same data element as target. As discussed by Leymann and Roller [LR00], this guarantees the determinism of the formal model since it avoids conflicts and ambiguity during runtime when data container instances have to be composed and materialized based on the specified mappings and potentially multiple data maps could provide their source data element instances to copy them to the specified target data element. Therefore, all defined data connectors have to always provide a clear and conflict free set of data maps.

4.2.6.2 Choreography and Participant Data Connectors

While normal data connector maps specify the data flow within and between participants of a choreography model, the so-called *choreography data connector map* introduced in Definition 4.18 allows to specify input and output data of the choreography model as a whole and its participants. Therefore, our definition builds on top of the *process data connectors* introduced by Leymann and Roller [LR00] at the level of PM-Graphs to enable the specification

of data flow between defined process inputs and outputs and respective activities processing or producing them, respectively. In accordance, at the level of CM-Graphs, choreography data connectors allow the specification of inputs and outputs of participants as well as the choreography model and how the data elements of the input containers of the choreography model or its participants are copied to data elements of data objects or the input containers of the choreography's participants, activities or predicates. In the same way, choreography data connectors allow the specification of how data elements of data objects or from the output containers of participants or activities are copied to data elements of the output container of a participant or the choreography model. The main purpose of choreography data connectors is therefore passing of data as an input to an instance of a choreography model during choreography instantiation as well as providing the specified result data on the termination of the choreography instance. Since the message-based invocation of choreography models introduced in Section 4.2.5 encompasses the passing of input and output data, it can be seen as a specialized form of providing choreography input data and returning results to an initial requester or another party. However, the concept of choreography data connectors summarized in Definition 4.18 enables to specify such input and output data dependencies and their related data mappings in an abstract and generic manner and is therefore not bound to any specific concept nor technology on how the data is provided or transported.

Figure 4.13 shows our visual notation for such choreography data connectors and how they are used to specify input and output data of a choreography model and its participants. Therefore, they extend the visual notation of data connectors also shown in Figure 4.13 by adding a black circle to the source of the arrow connecting either the choreography boundary with a data object or participant or the participant boundary with a data object, activity or predicate. For example, to model the data flow from a choreography to a participant input, a choreography data connector represented by the choreography data connector map $\overrightarrow{\Delta_C}(\mathscr{C}, \mathscr{P}_1)$, is added between the boundary of the choreography and participant $\mathscr{P}_1$ as depicted in Figure 4.13. For copying data elements of the output of participant $\mathscr{P}_1$ to a data object

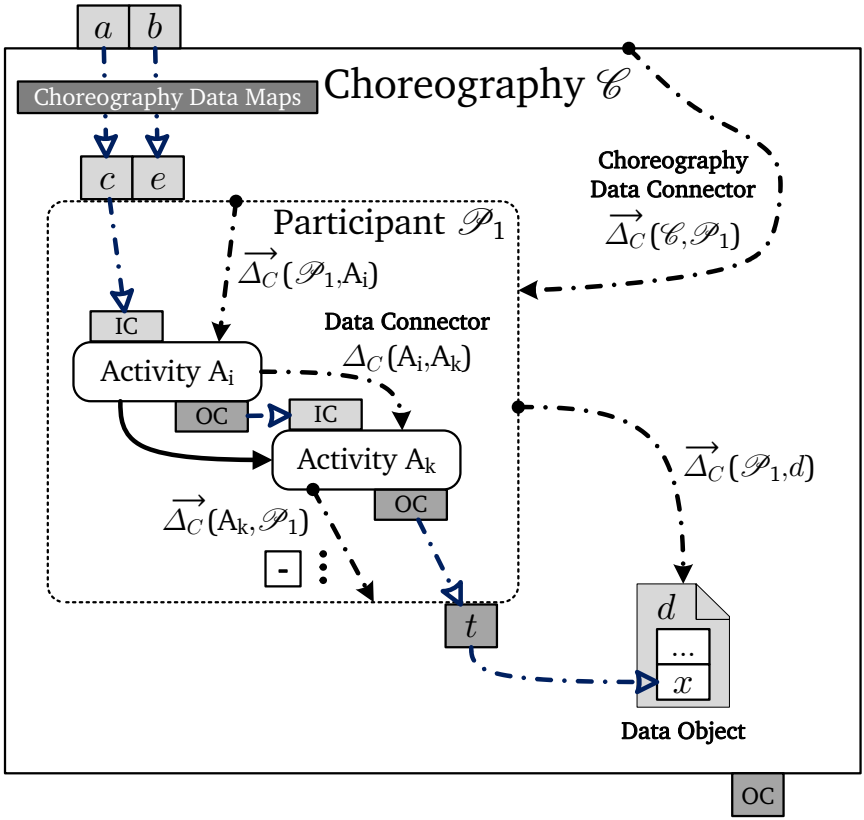

Figure 4.13: Visual representation of choreography data connectors and their data maps to support forwarding of choreography input and output data.

d, a choreography data connector represented by the choreography data connector map $\overrightarrow{\Delta_C}(\mathscr{P}_1, d)$, is added between the boundary of the participant $\mathscr{P}_1$ and the data object as depicted on the lower right of Figure 4.13. We also allow choreography data connectors from data-aware choreography models and their modeled participants to data objects to support the reading and writing of data from choreography and participants' input and output containers from and to data objects, respectively. The idea behind these special type of data connectors is not only to enable the distribution of globally relevant choreography input or output data, it furthermore enables modelers to defer the decision which concrete activity is reading or writing data from or to a data object to the refinement phase (i. e., refinement of generated abstract process models with process internal data flow between activities) discussed in Section 3.3.3. By using the input and output containers of participants as a kind of abstraction layer, modelers can add data elements to these containers, if they know that corresponding data is required or produced by a participant without the necessity to specify which activity is providing or consuming the data. For example, in some cases the corresponding activities are not known or modelers would not like to add them at the level of the choreography. Therefore, they can introduce a data object to specify that specific data will be available for other participants without explicitly stating how this data is consumed or produced. The corresponding activities and logic is therefore only added to the generated process models after the transformation of the choreography by each individual party during the process refinement phase.

Definition 4.18 provides the notion for the specification of such choreography data connector maps within our formal model for CM-Graphs.

Definition 4.18 (Choreography Data Connector Map)
Let $\mathscr{C} \in \mathfrak{C}$ be a choreography model, let N be the set of all activities of $\mathscr{C}$, let $\mathscr{D}(V)$ be the set of all data objects of $\mathscr{C}$, and let $\mathscr{P}(N)$ be the set of all participants of $\mathscr{C}$. The map

$$\overrightarrow{\Delta_C} : (N \cup \mathscr{D}(V) \cup \mathscr{P}(N) \cup \{\mathscr{C}\}) \times (N \cup \mathscr{D}(V) \cup \mathscr{P}(N) \cup \{\mathscr{C}\}) \to$$
$$\bigcup_{\substack{A \in N,\, d \in \mathscr{D}(V),\\ R \in \mathscr{P}(N)}} \begin{aligned} &(\wp(\iota_C(\mathscr{C}) \times \iota_C(R)) \\ &\cup \wp(o_C(R) \times o_C(\mathscr{C})) \\ &\cup \wp(\iota_C(R) \times \iota_C(A)) \\ &\cup \wp(o_C(A) \times o_C(R)) \\ &\cup \wp(d \times \iota_C(R)) \\ &\cup \wp(o_C(R) \times d) \\ &\cup \wp(\iota_C(\mathscr{C}) \times d) \\ &\cup \wp(d \times o_C(\mathscr{C}))) \end{aligned}$$

satisfying the conditions

1. $\forall R \in \mathscr{P}(N) : \overrightarrow{\Delta_C}(\mathscr{C}, R) \in \wp(\iota_C(\mathscr{C}) \times \iota_C(R))$,
2. $\forall R \in \mathscr{P}(N) : \overrightarrow{\Delta_C}(R, \mathscr{C}) \in \wp(o_C(R) \times o_C(\mathscr{C}))$,
3. $\forall R \in \mathscr{P}(N)\ \forall A \in R : \overrightarrow{\Delta_C}(R, A) \in \wp(\iota_C(R) \times \iota_C(A))$,
4. $\forall R \in \mathscr{P}(N)\ \forall A \in R : \overrightarrow{\Delta_C}(A, R) \in \wp(o_C(A) \times o_C(R)$,
5. $\forall R \in \mathscr{P}(N)\ \forall d \in \mathscr{D}(V) : \overrightarrow{\Delta_C}(d, R) \in \wp(d \times \iota_C(R))$,
6. $\forall R \in \mathscr{P}(N)\ \forall d \in \mathscr{D}(V) : \overrightarrow{\Delta_C}(R, d) \in \wp(o_C(R) \times d)$,
7. $\forall d \in \mathscr{D}(V) : \overrightarrow{\Delta_C}(\mathscr{C}, d) \in \wp(\iota_C(\mathscr{C}) \times d)$,
8. $\forall d \in \mathscr{D}(V) : \overrightarrow{\Delta_C}(d, \mathscr{C}) \in \wp(d \times o_C(\mathscr{C}))$,
9. $\forall R \in \mathscr{P}(N)\ \forall d \in \mathscr{D}(V)\ \forall B \in R : (x,z),(y,z) \in \wp(\iota_C(\mathscr{C}) \times \iota_C(R)) \cup \wp(\iota_C(R) \times \iota_C(B)) \cup \wp(d \times \iota_C(R)) \cup \wp(\iota_C(\mathscr{C}) \times d) \cup \bigcup_{A \in N \cup \mathscr{D}(V)} \Delta_C(A,B) \Rightarrow x = y$
10. $(x,z),(y,z) \in \bigcup_{R \in \mathscr{P}(N),\, B \in R} (\wp(o_C(B) \times o_C(R) \cup \wp(o_C(R) \times o_C(\mathscr{C}))) \cup \bigcup_{R \in \mathscr{P}(N),\, d \in \mathscr{D}(V)} (\wp(o_C(R) \times d) \cup \wp(d \times o_C(\mathscr{C}))) \Rightarrow x = y$

is called a choreography data connector map. An element $(v_1, v_2) \in \overrightarrow{\Delta_C}(\mathscr{C}, R)$ *is called a choreography data map.* □

The conditions listed in Definition 4.18 are similar to the ones discussed for data connectors in general in the previous section. Therefore, each of the conditions is only shortly discussed in the following. Conditions 1 and 2 ensure that a choreography data connector specifies only data maps between the input containers of the choreography model it originates from to the input container of the participant it points to or between the output containers of the participant it originates from to the output container of the choreography model it points to, respectively. Conditions 3 and 4 do the same at the level of the participants, there only data maps between the input container of a participant and of one of its contained activities or between the output container of a participant's activity and the output container of the participant itself are specified as part of a choreography data connector, respectively. The third group of conditions, condition 5-9, restrict the specification of choreography data connectors with data objects as source or target. Therefore, conditions 5 and 6 ensure that a choreography data connector specifies only data maps between the data object it originates from to the input container of the participant it points to or between the output container of the participant it originates from to the data object it points to, respectively. To ensure that choreography data connectors specify only data maps between the input container of the choreography they originate from to the data object they point to or between the data object they originate from to the output container of the choreography model they point to, respectively, conditions 7 and 8 are introduced. Finally, conditions 9 and 10 prohibit two different data maps from having the same data element as target to avoid conflicts and ambiguity during runtime and therefore guarantee the determinism of the formal model. In addition, condition 9 enforces that the specified data maps of a choreography data connector are not in conflict, i. e., having the same target data element, with the ones defined as part of the data connectors between the activities within participants.

4.2.7 Correlation of Messages and Data Objects

One of the core concepts of Business Process Management (BPM) is the execution of work according to a specified business process model. Therefore, BPEs provide capabilities to instantiate available business process models multiple times and manage their concurrent execution. Each of the process instances operates on a specific context, i. e., the process data as specified within the underlying model (cf. Section 4.2.1). While inside a BPE, the different instances of a process model are technically identified through an instance identifier (*instance id*) by default, from the outside world and also across multiple distributed BPEs a general mechanism is required to somehow identify the right target process instance, i. e., the context or process data, to route incoming messages to or read from and write to cross-partner data objects. For short, a mechanism for the correlation of messages and data objects to process instances across multiple BPEs is required.

Since within this work, the overall collaboration between multiple, independent participants is modeled through a data-aware choreography model, correlations can and should be modeled as part of the choreography model to enable both, the correlation of data objects and messages to a choreography instance as a whole, as well as, correlation of data objects and messages to process instances within the conversations of the private process models a choreography model is transformed to. This two-level correlation is required, because for each created choreography instance, an instance of each of the interconnected participant process models is created and executed. This requires to identify a choreography instance as a whole, for example, to associate data or send messages from the outside to it, i. e., data or messages being not modeled as part of the private process models, e. g., a choreography instantiation request or a data object used as choreography input (cf. Section 4.2.6.2). We will introduce an example to discuss this in more detail later in the section.

Established BPM standards such as BPMN and BPEL provide already capabilities and mechanisms for message correlation by introducing corresponding modeling constructs, namely *correlation keys* (BPMN) or *correlation sets*

(BPEL). We use the term *correlation set* in the following. The overall idea is to rely on the available business data specified within the process models and being part of messages to be exchanged instead of introducing artificial, implementation-specific correlation tokens. Therefore, modelers can specify a set of characteristical properties, i. e., a subset of defined process data, which allows to uniquely identify a matching context, i. e., a process instance during runtime. Specifying correlation sets can be compared to defining a primary key for a database table.

To enable the matching of messages and process instances during runtime using a correlation set, as a second step, modelers have to specify mappings between messages and a defined correlation set. Therefore, the business data exchanged as content of a message is mapped to one of the properties of the target correlation set. During runtime, this mapping allows to extract all business data relevant for correlation, called *correlation data* from a message, combine it to an instance of the modeled correlation set and to compare it with the correlation set of a process instance. If both correlation set instances contain the same data and therefore are equal, the message is forwarded to this process instance. This enables a BPE to route an incoming message to the right process instance, if multiple instances of the same process model are running in parallel.

Due to the fact that each process model is often not only interacting with one other process model and therefore participates in multiple conversations where different business data are exchanged, often multiple correlation sets have to be specified together with corresponding mappings to messages. In the context of this work, we build on top and reuse the existing concepts for correlation from the BPMN [BPMN] and BPEL [BPEL] standard wherever possible. While abstracting technology specific parts, e. g., correlation property definition using the Web Service Description Language (WSDL) [WSDL] in BPEL, we focus on the core modeling constructs and mechanisms required for enabling correlation at a BPE. For the sake of simplicity, we only introduce correlation sets at the level of the choreography model as a whole. However, this can be extended in future work when necessary to further allow the definition of correlation sets at the level of participants, for example,

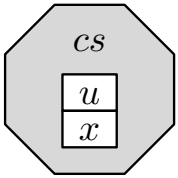

Figure 4.14: Visual representation of a correlation set.

to support use cases where multiple varying correlation sets are required only at the level of single participants to enable instance correlation on an conversation-basis without polluting the overall choreography model with such details. The visual representation of such a correlation set as part of our graphical notation is shown in Figure 4.14.

Definition 4.19 gives our definition of a *correlation set* as a collection of data elements which represent the underlying *correlation properties*.

Definition 4.19 (Correlation Sets and Correlation Properties)
Let V be the set of all data elements defined in a choreography model graph $\mathscr{C} \in \mathfrak{C}$, we denote by $CS(V) \subseteq \wp(V) \setminus \emptyset$ the set of all correlation sets defined within $\mathscr{C}$. Each member of $CS(V)$ is given by its name cs and contains a set of so-called correlation properties, defined through the referenced data elements. □

To specify the above discussed mappings between messages or data objects and a defined correlation set, we introduce a corresponding *correlation set property map* Δ_{CP} in Definition 4.20.

Definition 4.20 (Correlation Set Property Map)
Let $cs \in CS(V)$ be a correlation set, let $m \in \mathscr{M}(V)$ be a message, and let $d \in \mathscr{D}(V)$ be a data object. The map

$$\Delta_{CP} : CS(V) \times (\mathscr{M}(V) \cup \mathscr{D}(V)) \rightarrow \bigcup_{\substack{cs \in CS(V),\, m \in \mathscr{M}(V),\\ d \in \mathscr{D}(V)}} (\wp(cs \times m) \cup \wp(cs \times d))$$

satisfying the conditions

1. $\forall cs \in CS(V)\ \forall m \in \mathscr{M}(V) : \Delta_{CP}(cs, m) \in \wp(cs \times m)$,
2. $\forall cs \in CS(V)\ \forall d \in \mathscr{D}(V) : \Delta_{CP}(cs, d) \in \wp(cs \times d)$,
3. $\forall x \in (\mathscr{M}(V) \cup \mathscr{D}(V)) :$
   $(cp, v_1), (cp, v_2) \in \bigcup_{cs \in CS(V)} \Delta_{CP}(cs, x) \Rightarrow v_1 = v_2$,
4. $\forall cs \in CS(V)\ \forall x \in (\mathscr{M}(V) \cup \mathscr{D}(V)) :$
   $\Delta_{CP}(cs, x) \neq \emptyset \Rightarrow \left(\bigcup_{cp \in cs,\, v \in x} (cp, v) \right) = \Delta_{CP}(cs, x)$

is called a correlation set property map. An element $(cp, v) \in \Delta_{CP}(cs, x)$ *is called a correlation property map.* □

Condition 1 and 2 ensure that a correlation set property map specifies only mappings between the data elements of the correlation set, i. e., its correlation properties, and the data elements of a message or data object, respectively. Condition 3 of Definition 4.20 defines that each correlation property cp of a correlation set cs is mapped to exactly one data element v of a message or data object within a correlation set property map $\Delta_{CP}(cs, x)$. This avoids data conflicts during runtime and guarantees that a correlation property has a well-defined value. Condition 4 defines that if there is a correlation set property map $\Delta_{CP}(cs, x)$ specified between a correlation set cs and a data object or message x, then this map has to contain one correlation property map for each of the correlation properties cp of the correlation set cs. This ensures an all or nothing semantics during runtime, i. e., if a correlation set is initialized, it is guaranteed that all of its correlation properties have a well-defined value to be used for instance correlation.

Finally, a notion for the specification when correlation has to take place during runtime is required. Therefore, the defined correlation sets have to be mapped to the activities which provide or consume the business data to or from messages and data objects to be used for instance correlation which is reflected through the definition of corresponding correlation set property maps between the underlying correlation set and respective messages or data objects. To get a better idea of the overall correlation concept and the interrelation of the different modeling constructs, Figure 4.15 presents an example choreography model we will discuss in more detail in the following.

The depicted correlation set cs has a correlation property u defined as well as respective correlation set property maps which specify which of the data elements of data object d and message m are used for correlation. This means the value of which data elements has to be compared to the values of which correlation properties of the referenced correlation set to identify the right choreography instance the message or data object belongs to. The defined correlation set property maps $\Delta_{CP}(cs, m)$ and $\Delta_{CP}(cs, d)$ therefore provide the respective mappings between the correlation property u of the correlation set cs and the data element u of data object d and message m, respectively. This means, $\Delta_{CP}(cs, m) = \{(u, u)\}$ and $\Delta_{CP}(cs, d) = \{(u, u)\}$. Based on that, the correlation set allows to hold all business data required for instance correlation, i. e., values of defined correlation properties, and the correlation set property maps specify which data elements of a message or data object provide the required correlation property values during runtime. As introduced above, the missing part is when and how a defined correlation set has to be instantiated during runtime to enable its use for instance correlation by comparing the data element instances of the message or data objects referenced in the defined correlation set property maps to the correlation property values of the created correlation set instance. Therefore, we introduce a *correlation map* Δ_{CS} which allows to associate one or more correlation sets to an activity which is either a sending or receiving communication activity, i. e., has an outgoing or incoming message connector, or is reading or writing to a data object, i. e., has an incoming or outgoing data connector with a data object as source or target.

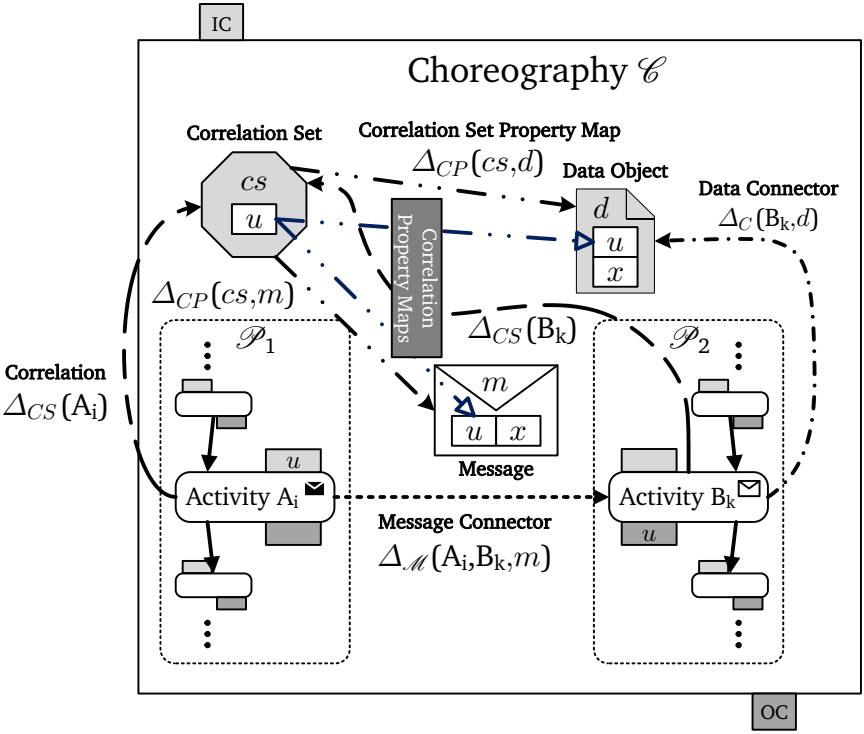

Figure 4.15: Example for a choreography model with a correlation set and correlation property maps to correlate messages and data objects to choreography and process instances.

The underlying operational semantics of such a correlation map are as discussed above. For example, in Figure 4.15 a correlation map $\Delta_{CS}(A_i)$ for the sending communication activity A_i and a correlation map $\Delta_{CS}(B_k)$ for the receiving communication activity B_k is specified. As soon as the sending communication activity A_i is scheduled, its input container is materialized and a copy of the containing data element u is assigned to data element u of message m. In addition, based on the defined correlation map $\Delta_{CS}(A_i)$, the correlation set cs is initialized by copying the value of data element u of the message m to the correlation property u of the correlation set

according to the defined correlation set property map $\Delta_{CP}(cs, m) = \{(u, u)\}$ between the correlation set and the message. The instantiated correlation set cs is then immutable, i. e., all of its correlation property values cannot be changed anymore and become constant values until the choreography instance is terminated. This is important to guarantee a correct and reliable identification of a choreography or process model instance throughout its whole life time. When participant $\mathscr{P}_2$, i. e., the BPE executing the instance of the private process model representing the participant, receives message m, instance correlation takes place in order to identify the right process instance to enable processing message m by activity B_k in the right context. Therefore, the BPE compares the value of data element u of message m with the correlation property u of its instance of the correlation set cs to identify the right process instance to forward the message for processing it. The same applies for the correlation of data object d, here the correlation set is mapped to activity B_k which writes data elements from its output container to the data object as specified through data connector $\Delta_C(B_k, d)$. The correlation of data objects and choreography or process instances during choreography runtime within the TraDE Middleware is discussed in more detail in Chapter 5. As outlined above, in general a choreography instance has one or more correlation sets which enable a correct instance correlation throughout different conversations between the choreography participants through the specified correlation maps. The discussed example choreography depicted in Figure 4.15 only presents one example scenario of a conversation with related correlation to provide a better idea of how the mechanism or concept of correlation works. However, there exist a huge variety of different, also way more complex conversation scenarios, e. g., as discussed by Barros, Dumas, and ter Hofstede [BDH05], which may also have an effect on instance correlation, e. g., when a correlation set has to be instantiated or what happens if required data values are missing. Since this is not in the scope of this work, we refer to the BPMN [BPMN] and BPEL [BPEL] standard modeling notations which provide further insights on the concept of correlation as well as possible cases and how to represent them within a choreography or process model to enable a proper and reliable instance

correlation during model execution.

Definition 4.21 summarizes our notion for the specification of mappings between activities and correlation sets in form of a *correlation map* Δ_{CS}.

Definition 4.21 (Correlation Map)
Let $A, B \in N$ be activities, let $cs \in CS(V)$ be a correlation set, let $m \in \mathcal{M}(V)$ be a message, and let $d \in \mathcal{D}(V)$ be a data object. The map

$$\Delta_{CS} : N \rightarrow \wp(CS(V))$$

satisfying the conditions

1. $\forall A \in N_{send}\ \forall B \in N_{receive}\ \forall m \in \mathcal{M}(V) : \Delta_{\mathcal{M}}(A,B,m) \neq \emptyset \Rightarrow \Delta_{CS}(A) \neq \emptyset \wedge \Delta_{CS}(B) \neq \emptyset \wedge (\Delta_{CS}(A) \cap \Delta_{CS}(B)) \neq \emptyset$,
2. $\forall A,B \in N\ \forall d \in \mathcal{D}(V) : \Delta_C(A,d) \neq \emptyset \wedge \Delta_C(d,B)) \neq \emptyset \Rightarrow \Delta_{CS}(A) \neq \emptyset \wedge \Delta_{CS}(B) \neq \emptyset \wedge (\Delta_{CS}(A) \cap \Delta_{CS}(B)) \neq \emptyset$,
3. $\forall x \in (\mathcal{M}(V) \cup \mathcal{D}(V))\ \forall A \in N : (\bigcup_{B \in N_{com}} \Delta_{\mathcal{M}}(A,B,x) \cup \Delta_C(A,x) \cup \Delta_C(x,A)) \neq \emptyset \Rightarrow \bigcup_{cs \in \Delta_{CS}(A)} \bigcup_{\substack{cp \in cs,\\ v \in x}} (cp,v) \in \Delta_{CP}(cs,x)$

is called a correlation map which associates one or more correlation sets to an activity. □

Condition 1 of Definition 4.21 specifies that, if a message connector map $\Delta_{\mathcal{M}}(A,B,m)$ between a sending communication activity A and a receiving communication activity B exists, a correlation set property map Δ_{CS} has to be specified for both activities and both activities have to map at least to one identical correlation set. The correlation sets to which both activities are mapped through their correlation set property maps are the ones used for message correlation in conversations realized by these activities during choreography runtime. Condition 2 guarantees the same for data objects. This means, if a data connector map $\Delta_C(A,d)$ between an activity A and a data object d (write data) or a data connector map $\Delta_C(d,B)$ between a data

object d and an activity B (read data) exists, a correlation set property map Δ_{CS} has to be specified for both activities. Furthermore, reading and writing activities pointing to the same data object through their data connectors, have to contain at least one matching correlation set in their correlation set property maps. In addition, condition 3 enforces that if a correlation map Δ_{CS} for an activity A is specified, then all correlation properties cp of the associated correlation sets ($cs \in \Delta_{CS}(A)$) have to be mapped to a data element of the message to be send or received or the data object to be read or written by the activity as part of a correlation property map $\Delta_{CP}(cs, x)$. In other words, condition 3 guarantees that all properties of a correlation set can be successfully instantiated during runtime by copying the values of the data elements of a message or data object as referenced in the correlation set property maps specified for the correlation set.

4.2.8 Choreography Model Graph

The combination of all definitions of the previous sections produces the overall definition of a CM-Graph shown in Definition 4.22 that represents the syntax of our formal model of data-aware choreographies. In the scope of this work we will not present definitions for staff assignment as well as exit and join conditions and therefore refer to the PM-Graph metamodel introduced by Leymann and Roller [LR00]. However, these definitions can be easily transferred one to one into our CM-Graph metamodel.

Definition 4.22 (CM-Graphs)
A tuple

$$G_C = (V, \iota_C, o_C, \mathscr{M}(V), CS(V), \mathscr{D}(V), \mu_{\mathscr{D}}, \rho_{\mathscr{D}}, N, \Psi, \mathscr{P}(N), \mu_{\mathscr{P}}, E, \mathcal{C}, \Delta_{\mathscr{M}}, \overrightarrow{\Delta_{\mathscr{M}}}, \Delta_C, \overrightarrow{\Delta_C}, \Delta_{CP}, \Delta_{CS})$$

is called a data-aware choreography model graph, or CM-Graph for short, representing a choreography model $\mathscr{C} \in \mathfrak{C}$. □

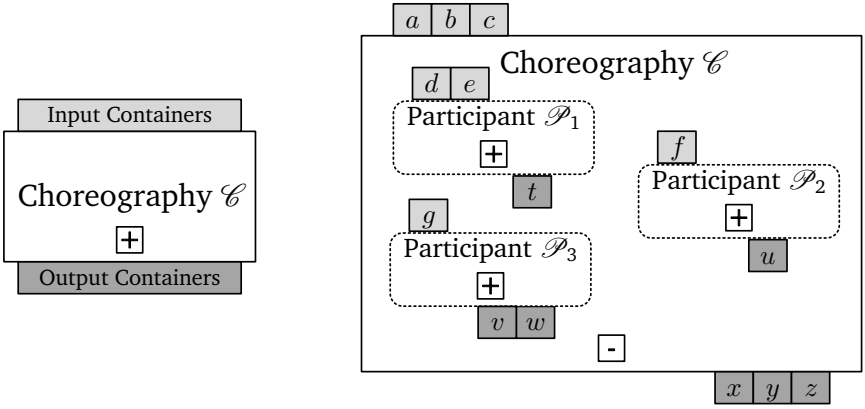

Figure 4.16: Visual representation of a choreography model graph and its participants.

Figure 4.16 depicts our visual representation of a choreography model graph of a choreography $\mathscr{C}$. On the left of the figure, a collapsed choreography shape, indicated by the plus sign at the bottom, with its defined inputs and outputs in form of respective data containers is shown. The other shape shows an expanded choreography with a collapsed version of its participants as well as their input and output data containers as presented in Section 4.2.3. For example, collapsing all participants of a choreography model can help to get a better overview of the specified message connectors and their associated messages without showing all participant-specific details such as source and target activities, participant internal data flow or correlation-related definitions. A more detailed visual representation of a choreography model using our formal model and visual notation is presented in the initial example depicted in Figure 4.1.

According to the definition of control flow in Section 4.2.4, message flow in Section 4.2.5, and data flow in Section 4.2.6, a choreography model graph G_C encompasses three graphs, one for each type of specified connectors. The following list provides a definition for each of those graphs:

1. The control flow graph

$$G_{C_{control}} = (N, E, V, \mathscr{P}(N), \mu_{\mathscr{P}}, \iota_C, o_C, C).$$

2. The message flow graph

$$G_{C_{message}} = (N_{com}, \mathscr{P}(N), \mu_{\mathscr{P}}, \boldsymbol{E}_{\Delta_{\mathscr{M}}}, V, \mathscr{M}(V), CS(V), \Delta_{\mathscr{M}}, \overrightarrow{\Delta_{\mathscr{M}}}, \Delta_{CP}, \Delta_{CS}),$$

   where the edge set $\boldsymbol{E}_{\Delta_{\mathscr{M}}}$ represents message connectors according to Section 4.2.5.1.

3. The choreography data flow graph

$$G_{C_{data}} = (N, \boldsymbol{E}_{\Delta_C}, V, \mathscr{D}(V), \mu_{\mathscr{D}}, \rho_{\mathscr{D}}, \Delta_C, \overrightarrow{\Delta_C}),$$

   where the edge set $\boldsymbol{E}_{\Delta_C}$ represents data connectors according to Section 4.2.6.1.

4.3 Choreography Data Dependency Graphs

The formal model for CM-Graphs introduced in Section 4.2 enables the specification of data-aware choreography models with our TraDE concepts applied in form of cross-partner data objects and cross-partner data flows. Within this section we introduce the notion of a *choreograpyh data model* (CDM) and a *choreography data dependency graph* (CDDG) as introduced in Chapter 3 as a prerequisite for the transformation of a CM-Graph to a collection of PM-Graphs in Section 4.5. Therefore, in Section 4.3.1 we define a choreography data model based on the specified data objects of a choreography model. In Section 4.3.2, we introduce how data dependencies across participants, i. e., cross-partner data flows, specified in form of data connectors between activities and data objects, are represented in form of a choreography data dependency graph.

4.3.1 Choreography Data Model

Within a CM-Graph data elements are defined to be used at the level of the choreography, i. e., in form of data objects to be independent of participants, as well as at the level of participants to define input and output containers of participants and activities. While the data flow within participants will be conducted by respective BPEs executing the refined participant process models, the data flow across participants, i. e., the choreography data flow, has to be handled from a global perspective, i. e., by our TraDE Middleware, as described in Section 3.2. Therefore, in Definition 4.23 we introduce a so-called *choreography data model* which consists of all defined cross-partner data objects and the data elements on which they are build on.

Definition 4.23 (Choreography Data Model)
Let $\mathscr{C} \in \mathfrak{C}$ be a choreography model, let $V_{\mathscr{D}} \subseteq V$ be the subset of all choreography data elements used as part of a data object definition within $\mathscr{C}$, let $\mathscr{D}(V)$ be the set of all data objects of $\mathscr{C}$, i. e., $\mathscr{D}(V_{\mathscr{D}}) = \mathscr{D}(V)$, and let $\mu_{\mathscr{D}}$ and $\rho_{\mathscr{D}}$ the defined data object multiplicity and deletion maps.
A tuple

$$C_{DM}(\mathscr{C}) = (V_{\mathscr{D}}, \mathscr{D}(V_{\mathscr{D}}), \mu_{\mathscr{D}}, \rho_{\mathscr{D}})$$

is called a Choreography Data Model (CDM) representing the data model of a given choreography model $\mathscr{C}$, where

$$V_{\mathscr{D}} := \{v \in V \mid \exists d \in \mathscr{D}(V) : v \in d\}$$ □

4.3.2 Choreography Data Dependencies

To represent the data dependencies of a particular choreography model $\mathscr{C} \in \mathfrak{C}$, we introduce a special kind of directed, edge-labeled multigraph $G_{C_{DDG}}$ based on our introduced choreography data model $C_{DM}(\mathscr{C})$, called Choreography Data Dependency Graph (CDDG). The node set $\mathcal{N}$ of $G_{C_{DDG}}$

consists of *dependency nodes* which are data objects, activities, conditions, participants or the overall choreography model itself as shown in Definition 4.24. To distinguish between data dependency nodes that process data (data consumers and producers, i. e., activities, conditions, participants, and choreographies) from the ones that hold data, i. e., data objects, we introduce the set of *data processors* Π.

Definition 4.24 (Data Dependency Nodes, Data Processors)
Let $\mathscr{C} \in \mathfrak{C}$ be a choreography model, let $\mathscr{D}(V_{\mathscr{D}})$ be the set of all data objects, let N be the set of all activities, let C be the set of all conditions, and let $\mathscr{P}(N)$ be the set of all participants of $\mathscr{C}$, we define

- *$\Pi = N \cup C \cup \mathscr{P}(N) \cup \{\mathscr{C}\}$ as the set of all data consumers and producers of a choreography model $\mathscr{C} \in \mathfrak{C}$, called data processors, and*
- *$\mathcal{N} = \mathscr{D}(V_{\mathscr{D}}) \cup \Pi$ as the set of all nodes being source or target of data flow within $\mathscr{C}$, which we call data dependency nodes, and*
- *V_{Π} as the set of all data elements being referenced as source or target of a data connector map between an input or output data container of a data processor Π and a cross-partner data object $d \in \mathscr{D}(V_{\mathscr{D}})$, and*
- *$\hat{V} = V_{\mathscr{D}} \cup V_{\Pi}$ as the set of all data elements associated with a data dependency node $\mathcal{N}$.* □

The edge set $E_{C_{DDG}}$ of the choreography data dependency graph denotes all identifiable data dependencies within $\mathscr{C}$, specifying their source and target data elements as edge labels, i. e., the read and write dependencies between its nodes, as given by Definition 4.25.

Similarly to the choreography data model introduced above, the focus is on cross-partner data flow and therefore data dependencies across participants through cross-partner data objects. Participant-internal data flow, i. e., data

connectors between activities or conditions, are therefore out of scope.

Definition 4.25 (Data Dependency Edges)
The set $E_{C_{DDG}} \subseteq \mathcal{N} \times \mathcal{N} \times \hat{V} \times \hat{V}$ *is called the set of data dependency edges of a choreography data dependency graph* $G_{C_{DDG}}$*. For a data dependency edge* $(A, B, s, t) \in E_{C_{DDG}}$ *from a node* A *to a node* B *of the node set,* $s \in \hat{V}$ *is called source data element and* $t \in \hat{V}$ *is called target data element and the tuple* (s, t) *represent the label of the dependency edge. Each data dependency edge denotes that during runtime the data value of source data element* s *of source node* A *needs to be available at target data element* t *of target node* B*.*
Therefore, $(A, B, s, t) \in E_{C_{DDG}} :\Leftrightarrow$

- $A \in \mathscr{D}(V) \wedge B \in (N \cup C) \wedge (s, t) \in \Delta_C(A, B)$*, or*
- $A \in N \wedge B \in \mathscr{D}(V) \wedge (s, t) \in \Delta_C(A, B)$*, or*
- $A \in \mathscr{D}(V) \wedge B \in \mathscr{D}(V) \wedge (s, t) \in \Delta_C(A, B)$*, or*
- $A \in \mathscr{D}(V) \wedge B \in (\mathscr{P}(N) \cup \{\mathscr{C}\}) \wedge (s, t) \in \overrightarrow{\Delta_C}(A, B)$*, or*
- $A \in (\mathscr{P}(N) \cup \{\mathscr{C}\}) \wedge B \in \mathscr{D}(V) \wedge (s, t) \in \overrightarrow{\Delta_C}(A, B)$. □

However, if participant-internal data flow is relevant, for example, when analyzing potential optimizations for data placement, it can be easily added by simply enriching the choreography data depencency graph with additional data dependency edges without changing anything else at the level of the metamodel. The same applies for message-based data flow, i. e., data elements passed through the exchange of messages between communication activities of different participants. This can be an interesting topic for future work, e. g., to evaluate collections of choreography models regarding their data exchange through message flows towards identifying modeling best practices in which scenarios the exchange of data across participants should be modeled through cross-partner data flows or message flows.

According to Definition 4.25, each edge of the choreography data depen-

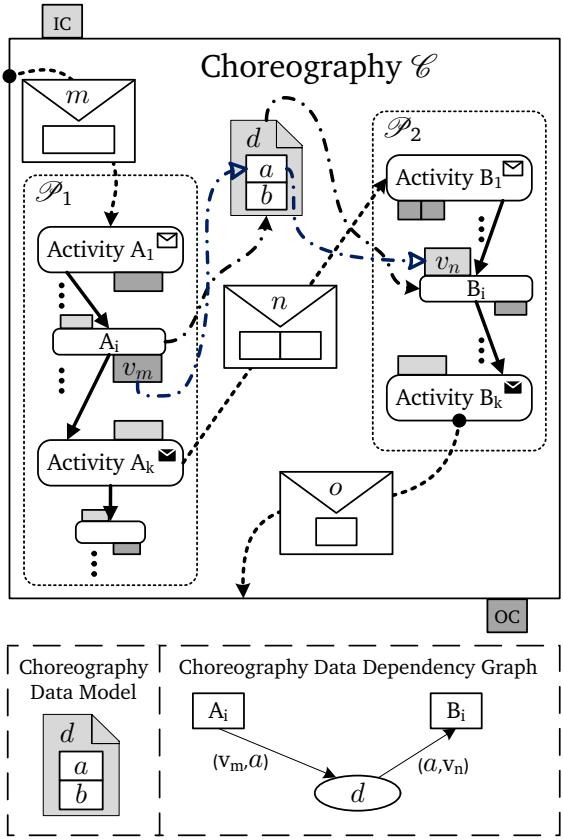

Figure 4.17: Example data-aware choreography model and its choreography data dependency graph.

dency graph denotes a data dependency, i. e., read or write access, between a data processor and a data element of a cross-partner data object specified through a corresponding data connector or choreography data connector within a CM-Graph. Therefore, for each data map of a data connector a corresponding data dependency edge is constructed following the rules listed in Definition 4.25. We denote data elements as sources and targets of a data dependency edge since their instances contain the actual data values during

runtime of an activities' data container or a data object. Therefore, the data maps of the specified data connectors are rolled out in order to identify and represent all data dependencies defined within a data-aware choreography model during transformation which we will further look into in Section 4.5.

This results in the directed multigraph $G_{C_{DDG}}$ shown in Definition 4.26 where potentially multiple directed edges between the dependency nodes of the graph can exist, while each edge reflects another data map of a data connector from or to a data object, i. e., a choreography data dependency.

Figure 4.17 shows an example choreography model and its respective choreography data depencency graph. The choreography model contains cross-partner data flows between participants $\mathscr{P}_1$ and $\mathscr{P}_2$ which is modeled through the two data connectors between activities A_i, B_i and data object d. The data maps underlying the two data connectors define that data element v_m is copied from the output container of activity A_i to data element a of data object d and from there data element a is copied to data element v_n of the input container of activity B_i. The data dependencies between the two participants are reflected via respective data dependency nodes (cf. Definition 4.24) and data dependency edges (cf. Definition 4.25). The resulting data dependency graph is shown at the bottom of Figure 4.17.

Definition 4.26 (Choreography Data Dependency Graph)
The tuple

$$G_{C_{DDG}} = (\Pi, E_{C_{DDG}}, V_\Pi, \overbrace{V_{\mathscr{D}}, \mathscr{D}(V_{\mathscr{D}}), \mu_{\mathscr{D}}, \rho_{\mathscr{D}}}^{C_{DM}(\mathscr{C})})$$

is called a Choreography Data Dependency Graph, or CDDG for short, representing all data dependencies and the data model $C_{DM}(\mathscr{C})$ of a choreography model $\mathscr{C} \in \mathfrak{C}$, where $G_{C_{DDG}}$ is a directed multigraph with node set $\mathcal{N} = \Pi \cup \mathscr{D}(V_{\mathscr{D}})$ and edge set $E_{C_{DDG}}$. □

4.4 Process Model Graphs and Staging Elements

The last prerequisite to introduce the transformation of a CM-Graph to a collection of PM-Graphs in Section 4.5 is to shortly introduce the PM-Graphs metamodel defined by Leymann and Roller [LR00] as well as required extensions. The main focus is here on the staging elements outlined in Section 3.3.3 to represent cross-partner data flows and references to the related cross-partner data objects at the level of a PM-Graph to provide the required runtime support for conducting cross-partner data flows via the TraDE Middleware as outlined in Section 3.2.2. Therefore, within this section we first introduce the notion of staging elements within our formal model in Section 4.4.1 and based on that provide a summary of the extended PM-Graph metamodel in Section 4.4.2.

4.4.1 Staging Elements

When transforming a data-aware choreography model to a collection of private process models as introduced in Chapter 3, we have to translate and represent the specified cross-partner data objects and the cross-partner data flow at the level of the resulting process models. Therefore, we have to extend the PM-Graph metamodel defined by Leymann and Roller [LR00] to reflect the defined cross-partner data flows at the level of a process model. As described in Chapter 3, the execution of the defined cross-partner data flows is handled together by the TraDE Middleware and the BPE executing the participant process models. To enable the BPEs to do so, the provided participant process models need to contain required information about the source or target data objects provided by the TraDE Middleware to exchange data with. In the following, we therefore extent the PM-Graph metamodel with so-called *staging elements* as outlined in Section 3.3.2 in the context of the transformation phase of the choreography management life cycle. These staging elements allow to specify required metadata within process models to conduct the modeled cross-partner data flows.

Definition 4.27 summarizes the notion of staging elements. The set of all

staging elements is denoted as Σ. Each staging element has to specify the source or target data object of the cross-partner data flow it represents. In addition, the mapping between the data elements of the referenced data object and the data elements of the activities' input or output container, the staging element is associated to, has to be specified. If data should be *pulled* from a data object at the TraDE Middleware to the associated activity's input container or data from the activity's output container should be *pushed* to a data object at the TraDE Middleware, is represented as *staging method*. To enable the correlation of the underlying data object instances at the TraDE Middleware and process instances at a BPE during choreography runtime, as discussed in Section 4.2.7, corresponding correlation information in form of a correlation set property map has to be specified as part of the staging element. Furthermore, a staging element should support the specification of a *trigger condition* and a *fault handling strategy*. The former allow to specify under which conditions the data staging should take place, e. g., depending on the status of other activities or local data containers. The latter enables the specification of fault handling behavior in cases where the data staging, i. e., copying data element values between data objects and activity data containers is not successful. In summary, a staging element $s \in \Sigma$ is a 6-tuple comprising a reference to a data object $d \in \mathscr{D}(V)$, a set of data maps $(v_1, v_2) \in \wp(V \times V)$, a staging method $p \in \Sigma_\Psi$, a correlation set property map (see Definition 4.20), and an optional trigger condition $t \in \mathcal{C}_T$ and fault handling strategy $f \in F$.

Definition 4.27 (Staging Elements)
Let $\mathscr{D}(V)$ be the set of all data objects and V be the set of data elements defined within a choreography model $\mathscr{C}$, let Σ_Ψ be the set of staging methods, let $CP \subseteq \Delta_{CP}$ be the map of correlation set properties for a referenced data object, let $\mathcal{C}_T \subseteq \mathcal{C}$ be the set of trigger conditions, and let F be the set of fault handling strategies with F_M as the set of fault handling strategy names and F_V as the set of corresponding fault handling strategy values, we denote Σ as the set of all staging elements

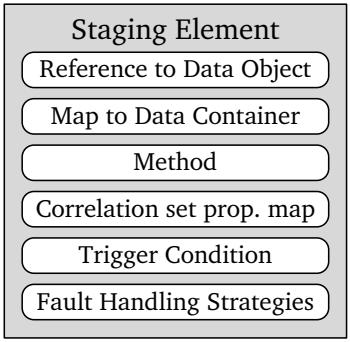

Figure 4.18: Visual representation of a Staging Element.

$$\Sigma \subseteq \mathscr{D}(V) \times \mathscr{P}(V \times V) \times CP \times \Sigma_{\Psi} \times (\mathcal{C}_T \cup \bot) \times (F \cup \bot)$$

where

1. $\Sigma_{\Psi} := \{push, pull\}$,
2. *each trigger condition* $t \in \mathcal{C}_T$ *is considered as a Boolean function in its input container* $\iota_C(t) \subseteq V$*:*

$$t : \underset{v \in \iota_C(t)}{\times} DOM(v) \to \{0,1\}$$

   or the bottom symbol $\bot$*, if the trigger condition is undefined,*

3. $F \subseteq (F_M \times F_V) \cup \bot$*, using the bottom symbol* $\bot$ *to represent an undefined fault handling strategy.* □

Figure 4.18 shows the visual representation of a staging element based on which we introduce its different parts. The reference to a data object d uniquely identifies one of the cross-partner data objects defined within the choreography model as a source or target for data staging. The map to

a data container is a set of data maps, i. e., a set of corresponding tuples $(v_1, v_2) \in \wp(V \times V)$ which specify a mapping of the data elements of an activity's input or output container to the data elements of a data object to read from or write to, respectively. To enable the correlation of data object instances as described in Section 4.2.7, the staging element contains the required correlation set property map. This map defines how the data elements of the referenced data object relate to the correlation properties of a defined correlation set.

With data elements of the data containers of an activity are copied from or to a data object is specified via the staging method $p \in \Sigma_\Psi$. As presented in Definition 4.27 the set of staging methods comprises two methods: *push* and *pull*. These methods specify how the staging between the referenced data object and the input or output container of an activity should take place during process execution. While push enables to copy values of a data element value from an activity's data container to a data object, pull supports the copying of data element values from a data object to an activity's data container. As already mentioned, it is possible that multiple staging elements can be associated to the same activity. This provides the ability to specify reading and writing from or to different data objects and in addition enables different data exchange and staging strategies by specifying corresponding trigger conditions to support more complex scenarios.

A trigger condition $t \in \mathcal{C}_T$ provides the means to define conditions that can be evaluated to true or false and therefore to specify under which condition a staging element should be scheduled during runtime. The underlying Boolean expression can therefore refer to a variety of input sources, e. g., activity data containers, data objects or even the execution state of activities. For example, a possible data-related trigger condition can be an expression that checks if the value of a data element of an activity's output container is above a certain threshold and only if this is the case the respective staging element is triggered which copies the value of the data element to the referenced data object. An example for a control-related trigger condition could be an expression that incorporates the state of other activities, e. g., *activity B is completed*. Trigger conditions may also introduce optimization

potentials for future work, since they provide an entry point to introduce the anticipatory and therefore proactive staging of data during runtime. Furthermore, monitoring data from previous choreography executions may be used to automatically generate or refine existing trigger conditions of corresponding staging elements. If the trigger condition is undefined, i. e., $t = \perp$, by default the staging element is scheduled on every value change of the referenced source data elements during runtime.

A fault handling strategy $f \in F$ provides the means to specify how the BPE or the TraDE Middleware should react if an exception or fault occurs during data staging. Since fault handling of cross-partner data flow is not in the scope of this work, we introduce in Definition 4.27 only how fault handling strategies and respective values can be specified as part of a staging element. To still get an idea how concrete fault handling strategies may look like, we introduce $F_M := \{\text{retry}, \text{manual}, \text{handle}\}$ to informally describe an initial set of strategies in the following. The *retry* strategy allows to specify a simple retry counter that defines how often the staging should be retried until the fault has to be escalated to the process level. Therefore, a fault handling strategy based on *retry* may look like the following: $f = (\text{retry}, 3)$.

By specifying a *manual* fault handling strategy for a staging element s, the BPE will contact an agent or human user for the resolution of the fault or a potentially underlying problem. To specify the target audience that may be contacted in terms of a fault, a so-called *staff query* introduced within the PM-Graph metamodel by Leymann and Roller [LR00] can be used as a value. While the selected agent, e. g., an administrator, tries to identify, and if possible solve, the underlying problem causing the fault, e. g., a problem with the surrounding infrastructure so that maybe the TraDE Middleware is not reachable by the BPE, the process instance is suspended and can be continued after the problem is solved. A resulting fault handling strategy f with a *manual* strategy may then look like the following: $f = (\text{manual}, q)$. Where q denotes a staff query which will return at any point in time $i \in \mathbb{N}$ a set of agents $q(i) \in \wp(\mathscr{A})$ with $\mathscr{A}$ as the set of all agents [LR00].

The *handle* strategy allows to reference complex fault handling logic that, e. g., tries to rerun one or more activities which produce the required data

or to resolve the underlying problems based on which the fault occurs. Such complex fault handling logic can be specified, for example, in form of a subprocess [KELU10], i. e., a process model with respective activities and control flow specified as part of the parent process representing the choreography participant. Therefore, the process modeling and fault handling capabilities of corresponding process modeling languages and the related BPE, e. g., using the fault handling mechanisms of BPMN [BPMN] or BPEL [BPEL], respectively, can be used to specify *data fault handlers* providing customized fault handling logic. An overview of fault handling within the different layers of the web service stack is provided by [Kop+10]. In terms of BPEL, Kopp et al. [Kop+11a] present a classification of a variety of BPEL extensions which may be used to specify such complex fault handling logic to associate it as a fault handling strategy value to a staging element. A resulting fault handling strategy f with a *handle* strategy may then look like the following: $f = (\text{handle}, P)$, where $P \in \mathcal{P}$ is a process model specified using the PM-Graph metamodel which defines respective fault handling logic. If the fault handling strategy is undefined, i. e., $f = \perp$, by default no explicit staging-related fault handling takes place during runtime. Therefore, staging-related faults will be propagated to the level of the process instance, e. g., if the BPE tries to read from uninitialized data elements which should have been initialized during staging by copying values from a data object, the BPE may terminate the overall process instance with status *faulted*.

After introducing staging elements in Definition 4.27, they have to be associated to respective data consumers or producers, i. e., activities, conditions, or participants as a whole, as already outlined above. Therefore, we introduce a *staging element map* σ which allows to associate one or multiple defined staging elements to respective data consumers or producers. The reason that more than one staging element can be associated is that cases where data from one or more data objects is copied from or to a data container of a data consumer or producer have to be supported accordingly. For example, activity B_3 of the choreography model shown in Figure 4.1 has a reading data dependency and a writing data dependency to data object d. Therefore, two staging elements with a reference to data object d and respec-

tive data maps (α, m) and (t, β) are created and associated to the activity during the transformation of the CM-Graph to a collection of PM-Graphs. Before we have a closer look into the translation of data dependencies from the level of a choreography model to a set of staging elements within the resulting process models, we provide and discuss the definition of the above introduced staging element map σ presented in Definition 4.28.

Definition 4.28 (Staging Element Map)
Let $\Pi = N \cup \mathcal{C} \cup \mathcal{P}(N) \cup \{\mathcal{C}\}$ be the set of all data consumers and producers of a choreography (cf. Definition 4.24), and let Σ be the set of all staging elements. The map

$$\sigma : \Pi \to \wp(\Sigma)$$

satisfying the condition

$$\forall A \in \Pi : \left(\bigcup_{d \in \mathcal{D}(V)} \Delta_C(A, d) \cup \Delta_C(d, A) \right) \neq \emptyset \Rightarrow \sigma(A) \neq \emptyset$$

is called staging element map and associates with each activity $A \in N$, condition $p \in \mathcal{C}$, or participant $R \in \mathcal{P}(N)$ its data staging elements, where a staging element has the following form: $(d, \{(v_1, v_2)\}, pull, t, f) \in \Sigma$. □

The condition in Definition 4.28 guarantees that if a data consumer or producer node at the level of the choreography model has an incoming or outgoing data connector with a data object on the other end, a corresponding staging element has to be associated to this node. The translation of data connectors to staging elements will be discussed in Section 4.5 in detail.

Figure 4.19 shows a concrete example of a resulting staging element associated to an activity A_i based on the respective data connector $\Delta_C(d, A_i)$ specified at the level of the choreography model depicted on the left of the figure. Following the conditions of Definition 4.28 described above, the

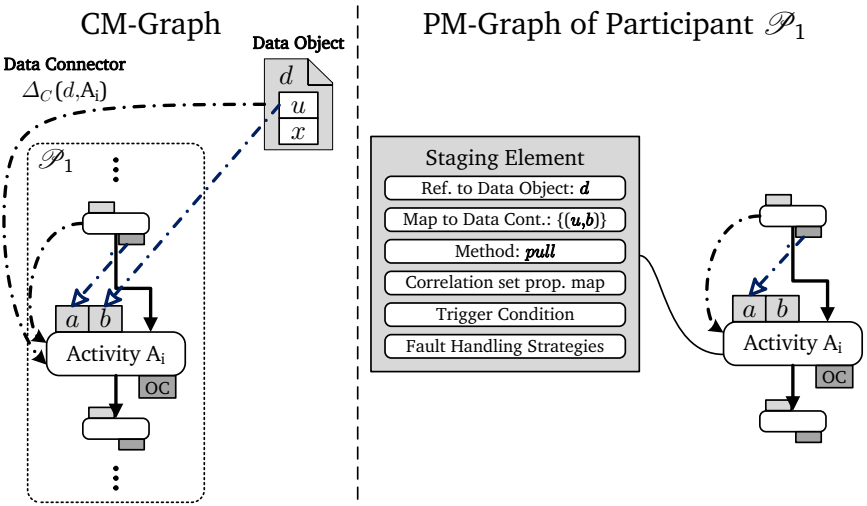

Figure 4.19: Example of a Staging Element and its association to an activity within a process model based on data connectors defined at the level of the choreography model.

staging element references data object d and contains the specified data map (u, b) of the data connector as well as indicates via the *pull* staging method that data element values have to be requested at the TraDE Middleware to copy them to the activities input container. Further details and dependencies will be discussed in detail in the context of the transformation algorithm presented in Section 4.5.

4.4.2 Process Model Graphs

Before the transformation is described in detail in Section 4.5, we first provide a short summary of the adapted version of the formal model for PM-Graphs defined by Leymann and Roller [LR00]. To reflect cross-partner data flow as well as message exchanges across participants at the level of the resulting process models, we add the respective elements introduced in the previous sections to the PM-Graph model. Since we have not introduced definitions

for staff assignment as well as exit and join conditions at the level of CM-Graphs, they are also omitted from the compressed PM-Graph definition presented in the following. Therefore, we refer to the work of Leymann and Roller [LR00] for further details as well as the complete PM-Graph definition. The PM-Graph definition shown in Definition 4.29 is reduced to only those elements also used or introduced within this work.

Definition 4.29 (PM-Graph)
A tuple

$$G = (V, \iota, o, \mathscr{M}(V), CS(V), \Delta_{CP}, \Delta_{CS}, N, \Psi, \mathcal{C}, E, \Delta, \overrightarrow{\Delta}, \Sigma, \sigma)$$

is called a process model graph, or PM-Graph for short, representing a process model $P \in \mathcal{P} :\Leftrightarrow$

1. V *is a finite set of data elements (see Definition 4.1).*
2. $\iota : N \cup \mathcal{C} \cup \{G\} \to \wp(V)$ *is called an input container map (cf. Definition 4.3).*
3. $o : N \cup \{G\} \to \wp(V)$ *is called an output container map (cf. Definition 4.3).*
4. $\mathscr{M}(V)$ *is a finite set of messages send or received by the process model (see Definition 4.4).*
5. $CS(V)$ *is a finite set of correlation sets (see Definition 4.19).*
6. Δ_{CP} *is the correlation set property map (see Definition 4.20).*
7. Δ_{CS} *is the correlation map (see Definition 4.21).*
8. N *is a finite set of activities (cf. Definition 4.8).*
9. $\Psi : N \to \mathcal{E}$ *is called an activity implementation map (cf. Definition 4.9).*
10. $\mathcal{C}$ *is a finite set of conditions, where each condition* p *is a map:* $\underset{v \in \iota(p)}{\times} DOM(v) \to \{0, 1\}$.

11. $E \subseteq N \times N \times \mathcal{C}$ *is a set of control connectors (cf. Definition 4.12) with the following properties:*
    a) *E is unified, i. e.,* $\forall e, e' \in E : \pi_{1,2}(e) = \pi_{1,2}(e') \Rightarrow \pi_3(e) = \pi_3(e')$,
    b) *(N,E) is acyclic.*
12. $\Delta : N \times (N \cup \mathcal{C}) \rightarrow \bigcup_{A \in N,\, B \in N \cup \mathcal{C}} \wp(o(A) \times \iota(B))$ *is the data connector map (cf. Definition 4.17), where*
    a) $\Delta(A, B) \in \wp(o(A) \times \iota(B))$,
    b) $\Delta(A, B) \neq \emptyset \Rightarrow B$ *is reachable from A,*
    c) $\forall B \in N : (x, z), (y, z) \in \bigcup_{A \in N} \Delta(A, B) \Rightarrow x = y$.
13. $\overrightarrow{\Delta} : N \rightarrow \bigcup_{A \in N} (\wp(\iota(P) \times \iota(A)) \cup \wp(o(A) \times o(P)))$ *is the process data connector map (cf. Definition 4.18), where*
    a) $\forall A \in N : \overrightarrow{\Delta}(A) \in \wp(\iota(P) \times \iota(A)) \cup \wp(o(A) \times o(P))$,
    b) $\forall B \in N : (x, z), (y, z) \in \wp(\iota(P) \times \iota(B) \cup \bigcup_{A \in N} \Delta(A, B) \Rightarrow x = y$,
    c) $(x, z), (y, z) \in \bigcup_{B \in N} \wp(o(B) \times o(P)) \Rightarrow x = y$.
14. Σ *is a finite set of staging elements (see Definition 4.27).*
15. $\sigma : N \rightarrow \Sigma$ *is the staging element map (see Definition 4.28).* □

4.5 Transformation of a CM-Graph to a Collection of interconnected PM-Graphs

Based on our formal model for CM-Graphs introduced in Section 4.2 as well as the extended PM-Graph metamodel presented in Section 4.4, we are now able to specify the transformation of a data-aware choreography model to a collection of interconnected process models to support the *transformation*

phase of the choreography management life cycle introduced in Section 3.3. Before we go into details of the different steps of the overall transformation process, we first describe the process as a whole as well as its source and resulting artifacts. Figure 4.20 shows an example choreography model $\mathscr{C}$ used as input to the transformation. Our goal regarding the transformation is that data-related aspects specified at the level of a choreography model are utilized wherever possible to generate and enrich the resulting process models to reduce manual refinement efforts and avoid error-prone manual translations between models. While within this work the application of correctness checks, or model verification approaches in general, are out of scope, such techniques may be applied in future work as a quality gate before transformation. For example, [FMW22] introduce an approach for checking the soundness of data-aware processes based on data petri nets which takes the interplay of data and control flow into account. Such data-aware model verification approaches can help to further reduce manual efforts and detect errors as early as possible within the choreography management life cycle.

The transformation itself is performed in three steps as described in Section 3.3. First, all cross-partner data flows, i. e., data objects and data connectors, modeled within a choreography model are analyzed and distilled into a corresponding choreography data model (see Definition 4.23) and a choreography data dependency graph (see Definition 4.26) as shown at the top right of Figure 4.20. The generated CDDG provides a straight-forward representation of all data dependencies between choreography participants and is therefore used later to enrich the generated process models regarding their data perspective. For example, the data dependencies between activity A_i of participant $\mathscr{P}_1$ and data object d specified through a corresponding data connector between them is represented within the generated CDDG. Therefore, for each of the data maps of the data connector a corresponding data dependency edge between respective dependency nodes is added to the graph. For the example shown in Figure 4.20, a data dependency between the source data element v_m of the output container of activity A_i and the target data element a of data object b is added in form of a labeled edge between the activity and data object node (cf. Section 4.3.2).

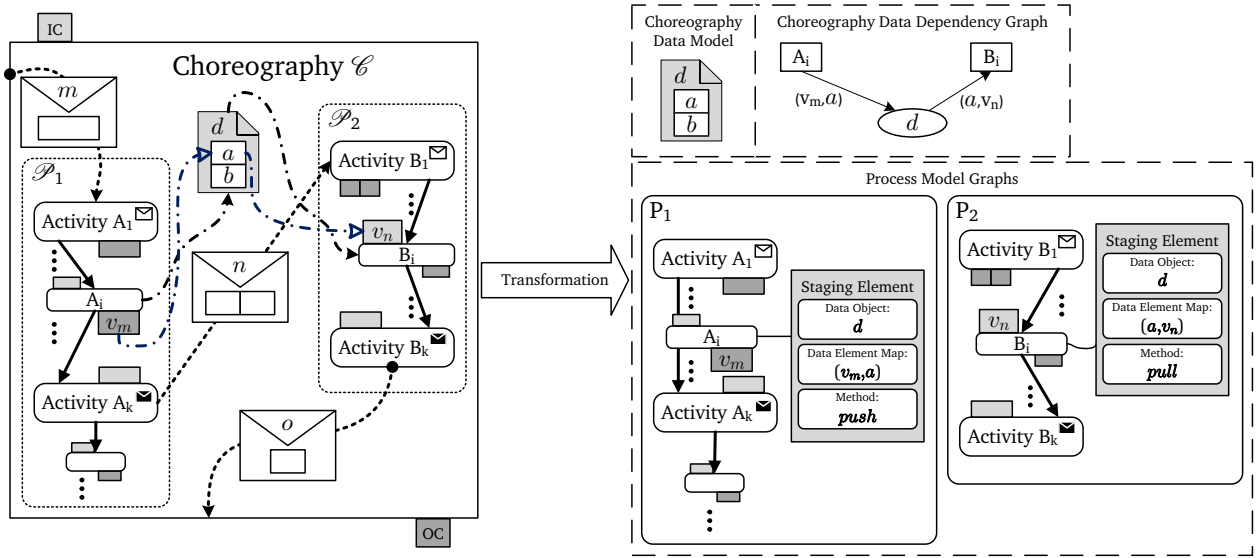

Figure 4.20: Transformation of an example data-aware choreography model to a collection of private process models and a choreography data dependency graph.

In the second step, for each participant of the choreography model a new PM-Graph is generated and the modeled control and message flows, e. g., activities, conditions, messages or control connectors, are copied into the respective participant process model represented by a PM-Graph. For the choreography shown in Figure 4.20, a process model P_1 for choreography participant $\mathscr{P}_1$ and a process model P_2 based on participant $\mathscr{P}_2$ are generated. From the perspective of the underlying metamodels, this can be seen as some kind of a view transformation since the actual data remains the same, only the viewpoint of the definition changes.

On the level of the choreography model, the focus is on the global collaboration and interconnection of the modeled participants and at the level of the process models, the focus changes to how an individual process model interacts with external black-box entities (e. g., process models, services, etc.) only specifying its internal control flow and based on that which messages have to be sent to or received from other parties.

Finally, within the last step, all data dependencies to cross-partner data objects collected in the CDDG are incorporated into the generated PM-Graphs by enriching them with respective staging elements as described in Section 4.4.1. Therefore, as shown in Figure 4.20 one staging element is added to each of the activities A_i and B_i to represent their respective data dependencies to data object d according to the generated CDDG of the choreography model. In the following three sections, each of the introduced steps is defined and presented in more detail.

4.5.1 Generating a Choreography Data Dependency Graph for a CM-Graph

Within the first step of the transformation process the cross-partner data flows, i. e., data objects and data connectors, specified within a choreography model are analyzed and distilled into a corresponding choreography data model and a choreography data dependency graph. Therefore, Algorithm 4.1 defines how a choreography data model is generated from a CM-Graph of an underlying choreography model.

As described in Section 4.3.1, a choreography data model C_{DM} of a given

Algorithm 4.1 Generating a choreography data model C_{DM} for a given CM-Graph G_C representing a choreography model $\mathscr{C}$

1: **procedure** GENERATECDM(G_C)
2: $V_{\mathscr{D}} \leftarrow \{v \in \overbrace{\pi_1(G_C)}^{V} \mid \exists d \in \overbrace{\pi_6(G_C)}^{\mathscr{D}(V)} : v \in d\}$
3: $\mathscr{D}(V_{\mathscr{D}}) \leftarrow \pi_6(G_C)$
4: $\mu_{\mathscr{D}} \leftarrow \overbrace{\pi_7(G_C)}^{\mu_{\mathscr{D}}}$
5: $\rho_{\mathscr{D}} \leftarrow \overbrace{\pi_8(G_C)}^{\rho_{\mathscr{D}}}$
6: $C_{DM}(G_C) \leftarrow (V_{\mathscr{D}}, \mathscr{D}(V_{\mathscr{D}}), \mu_{\mathscr{D}}, \rho_{\mathscr{D}})$
7: **return** $C_{DM}(G_C)$ ▷ The choreography data model
8: **end procedure**

CM-Graph G_C consists of all defined cross-partner data objects and the data elements they contain. This results in a very straight-forward algorithm for the generation of such a C_{DM} for a choreography model. First, in line 2 of Algorithm 4.1 the set of data elements which are used as part of a data object definition are identified and collected in the set $V_{\mathscr{D}}$. Therefore, we collect all data elements from the set V which are used as part of a data object definition, i. e., $v \in d$, with $d \in \mathscr{D}(V)$. To get the respective sets V and $\mathscr{D}(V)$ of the CM-Graph G_C we use the projection $\pi_i(G_C)$ to get the i-th element of the underlying CM-Graph tuple. In addition, to improve the readability and ease the understanding to which set or map a projection refers to, we add a bracket on top of each projection with the respective target set or map throughout all algorithms in the following. The set of data objects can just be used as is by extracting it via the projection $\pi_6(G_C)$ from the tuple of the CM-Graph G_C passed as input to the algorithm.

The same applies for the defined data object multiplicity and deletion maps which are also extracted via projection on lines 4 and 5 of the algorithm. Finally, in line 6, the resulting choreography data model can be created by wrapping the collected sets and maps into a respective 4-tuple $C_{DM}(G_C)$ which is then returned as a result.

Based on that, we can then define how a choreography data dependency

Algorithm 4.2 Generating a choreography data dependency graph $G_{C_{DDG}}$ for a given CM-Graph G_C representing a choreography model $\mathscr{C}$

procedure GENERATECDDG(G_C)
  $C_{DM}(G_C) \leftarrow$ GENERATECDM(G_C)
  $V_\Pi, E_{C_{DDG}} \leftarrow \emptyset$ ▷ Initialize required sets
  $\Pi_{read} := \{A \in (\overbrace{\pi_9(G_C)}^{N} \cup \overbrace{\pi_{14}(G_C)}^{\mathcal{C}} \cup \overbrace{\pi_{11}(G_C)}^{\mathscr{P}(N)} \cup \{\mathscr{C}\})$
    $\mid \exists d \in \overbrace{\pi_6(G_C)}^{\mathscr{D}(V)} : \Delta_C(d,A) \cup \overrightarrow{\Delta_C}(d,A) \neq \emptyset\}$
    ▷ Data processors **reading** from a data object
  $\Pi_{write} := \{B \in (\pi_9(G_C) \cup \pi_{11}(G_C) \cup \{\mathscr{C}\})$
    $\mid \exists d \in \pi_6(G_C) : \Delta_C(B,d) \cup \overrightarrow{\Delta_C}(B,d) \neq \emptyset\}$
    ▷ Data processors **writing** to a data object
  $\Delta_C = \pi_{17}(G_C)$ ▷ Data connector map of G_C
  $\overrightarrow{\Delta_C} = \pi_{18}(G_C)$ ▷ Choreography data connector map of G_C
  for all $A \in \Pi_{read}$; $d \in \pi_6(G_C)$ **do** ▷ Extract *read* data dependencies
    for all $(u,x) \in \Delta_C(d,A) \cup \overrightarrow{\Delta_C}(d,A)$ **do**
      $V_\Pi \leftarrow V_\Pi \cup \{x\}$ ▷ Add data elements of data processors to V_Π
      $E_{C_{DDG}} \leftarrow E_{C_{DDG}} \cup \{(d,A,u,x)\}$ ▷ Add a data dependency edge
    end for
  end for
  for all $B \in \Pi_{write}$; $d \in \pi_6(G_C)$ **do** ▷ Extract *write* data dependencies
    for all $(y,v) \in \Delta_C(B,d) \cup \overrightarrow{\Delta_C}(B,d)$ **do**
      $V_\Pi \leftarrow V_\Pi \cup \{y\}$
      $E_{C_{DDG}} \leftarrow E_{C_{DDG}} \cup \{(B,d,y,v)\}$
    end for
  end for
  for all $d,e \in \pi_6(G_C)$ **do** ▷ Extract *read/write* data dependencies
    for all $(w,z) \in \Delta_C(d,e)$ **do**
      $E_{C_{DDG}} \leftarrow E_{C_{DDG}} \cup \{(d,e,w,z)\}$
    end for
  end for
  $\Pi = \Pi_{read} \cup \Pi_{write}$ ▷ Set of data processors
  $G_{C_{DDG}} \leftarrow (\Pi, E_{C_{DDG}}, V_\Pi, C_{DM}(G_C))$
  return $G_{C_{DDG}}$ ▷ The choreography data dependency graph
end procedure

graph is generated which comprises the extracted choreography data model as well as all specified data dependencies as outlined in Section 4.3.2. Therefore, in line 2 of Algorithm 4.2 the introduced *GenerateCDM* algorithm (Algorithm 4.1) is executed to generate the required choreography data model. Next, in line 3 required sets are defined to incrementally create and add respective elements to them, i. e., the set of data elements of data processors V_{Π} as well as the set of data dependency edges $E_{C_{DDG}}$. In line 4 of Algorithm 4.2 all data processors reading from a data object, i. e., being the target of a data connector or choreography data connector originating from a data object, are identified and collected in the set Π_{read}. The same happens for all data processors writing to a data object, i. e., being the source of a data connector or choreography data connector targeting a data object, which are identified and collected in the set Π_{write} in line 5 of the algorithm. In line 6 and 7 the respective data connector and choreography data connector maps are read from the provided CM-Graph G_C to extract respective data dependencies in the following. For a better understanding and to ease the description, we split the extraction of data dependencies into three groups: *read* (lines 8-13), *write* (lines 14-19) and *read/write* (lines 20-24) dependencies on data objects.

To extract all *read* data dependencies, the algorithm iterates over all identified data processors with a read dependency contained in the set Π_{read} and all data objects in line 8 and creates for each data map defined between a data processor and a data object as part of a data connector or choreography data connector (line 9) a respective data dependency edge in line 11. Each resulting dependency edge (d, A, u, x) is then added to the edge set $E_{C_{DDG}}$. In addition in line 10, the involved data element x of the data processor being the target of the underlying data map and therefore also of the resulting data dependency is added to the set V_{Π}. The extraction of all *write* data dependencies works exactly the same way using the set of all identified data processors with a write dependency Π_{write} as shown in Algorithm 4.2. The last group of *read/write* dependencies is somehow special in the sense that it focuses on the data dependencies between two data objects represented through corresponding data connectors with a data object as source and

target. Therefore, in line 20 of Algorithm 4.2, the algorithm iterates over all defined data objects and creates for each data map defined between two data objects as part of a corresponding data connector (line 21) a respective data dependency edge in line 22 which is then added to the edge set $E_{C_{DDG}}$. In line 25 of Algorithm 4.2, the set of all identified data processors Π is created by composing the sets of reading and writing data processors Π_{read} and Π_{write}, respectively. Finally, all created or extracted sets are combined into the overall 4-tuple $G_{C_{DDG}}$ which represents the CDDG of the CM-Graph G_C given as input to the algorithm and is then returned as a result.

4.5.2 Generating PM-Graphs based on a CM-Graph

By introducing the generation of a Choreography Data Model (CDM) and a CDDG for a given CM-Graph G_C, the remaining part of the transformation results shown in Figure 4.20 is the generation of a process model P in form of a PM-Graph G for each defined participant of the choreography model represented by the CM-Graph G_C. Therefore, the modeled control flow, data flow and message flow defined in the scope of a participant as well as entering or leaving the participant has to be extracted from the choreography model and represented accordingly within the respective participant process models by adding corresponding modeling constructs to the resulting PM-Graphs. Finally, all data dependencies collected in the CDDG provided as result of Algorithm 4.2 are incorporated into the generated PM-Graphs by enriching them with respective staging elements as described in Section 4.4.1.

In the following, we will discuss both remaining steps in more detail and present a respective transformation algorithm. Due to the size of the algorithm it is split into three parts shown in Algorithms 4.3 - Part 1 to 4.3 - Part 3. Algorithm 4.3 - Part 1 takes the previously generated CDDG $G_{C_{DDG}}$ and the CM-Graph G_C itself as an input. The algorithm starts on line 2 with the initialization of the set of all to be generated PM-Graphs $\mathcal{P}_C$ used during the algorithm to collect all created PM-Graphs and return the final transformation result at the end of the algorithm.

Next, the actual transformation starts by iterating over the set of specified

Algorithm 4.3 - Part 1 Generating PM-Graphs G_i based on the defined participants R_i of a given CM-Graph G_C and its data dependency graph $G_{C_{DDG}}$

1: **procedure** GENERATEPROCESSMODELGRAPHS(G_C, $G_{C_{DDG}}$)

2: $\mathcal{P}_C \leftarrow \emptyset$ ▷ Initialize set of PM-Graphs

3: **for all** $R_i \in \overbrace{\pi_{11}(G_C)}^{\mathscr{P}(N)}$ **do** ▷ Loop over participants of CM-Graph

$$\mathscr{M}(V_{G_i}) \leftarrow \{m \in \overbrace{\pi_4(G_C)}^{\mathscr{M}(V)} \mid \exists A \in R_i, B \in \overbrace{\pi_9(G_C)}^{N} \cup \{\mathscr{C}\} :$$

4: $$(A,B,m) \in \overbrace{\pi_{15}(G_C)}^{\Delta_{\mathscr{M}}} \cup \overbrace{\pi_{16}(G_C)}^{\overrightarrow{\Delta_{\mathscr{M}}}} \vee$$
$$(B,A,m) \in \pi_{15}(G_C) \cup \pi_{16}(G_C)\}$$ ▷ Messages

5: $N_{G_i} \leftarrow \{A \in R_i\}$ ▷ Activities

6: $\Psi_{G_i} \leftarrow \bigcup_{A \in N_{G_i}} \overbrace{\pi_{10}(G_C)}^{\Psi}(A)$ ▷ Activity implementations

7: $E_{G_i} \leftarrow \{e \in \overbrace{\pi_{13}(G_C)}^{E} \mid \pi_{1,2}(e) \in N_{G_i} \times N_{G_i}\}$ ▷ Control connectors

8: $\mathcal{C}_{G_i} \leftarrow \bigcup_{e \in E_{G_i}} \pi_3(e)$ ▷ Transition conditions

9: $\iota_{G_i} \leftarrow \bigcup_{A \in N_{G_i} \cup \mathcal{C}_{G_i} \cup R_i} \overbrace{\pi_2(G_C)}^{\iota_C}(A)$ ▷ Input containers

10: $o_{G_i} \leftarrow \bigcup_{B \in N_{G_i} \cup R_i} \overbrace{\pi_3(G_C)}^{o_C}(A)$ ▷ Output containers

11: $CS(V_{G_i}) \leftarrow \bigcup_{A \in N_{G_i}} \overbrace{\pi_{20}(G_C)}^{\Delta_{CS}}(A)$ ▷ Correlation sets

12: $\Delta_{CP_{G_i}} \leftarrow \bigcup_{cs \in CS(V_{G_i}),\, m \in \mathscr{M}(V_{G_i})} \overbrace{\pi_{19}(G_C)}^{\Delta_{CP}}(cs,m)$ ▷ Correlation set property map

13: $\Delta_{CS_{G_i}} \leftarrow \{(A, \overbrace{\pi_{20}(G_C)}^{\Delta_{CS}}(A)) \mid A \in N_{G_i} \wedge \pi_{20}(G_C)(A) \neq \emptyset\}$ ▷ Correlation map

choreography participants $\mathscr{P}(N)$ in line 3. Therefore, within each loop iteration $i \in \mathbb{N}$ a new PM-Graph G_i representing the participant R_i is created and added to the set of PM-Graphs $\mathcal{P}_C$. According to our adapted PM-Graph metamodel presented in Definition 4.29, a PM-Graph G_i is a 15-tuple for which we have to extract and define each of the required sets and maps from the provided CM-Graph G_C. Algorithm 4.3 - Part 1 starts on line 4 with the extraction of the set of messages $\mathscr{M}(V_{G_i})$ relevant in the context of participant R_i. Where relevant means, that all messages defined at the level of the choreography within the set $\mathscr{M}(V)$ are copied to the set $\mathscr{M}(V_{G_i})$, if they are associated to an outgoing or incoming message connector or choreography message connector which originates from or points to one of the participant's activities.

Next, the set of activities N_{G_i} on line 5 can be simply created by adding all participant activities being the elements of the participant R_i according to the participant definition presented in Section 4.2.3. The extraction of all relevant activity implementation maps Ψ_{G_i} in line 6 of Algorithm 4.3 - Part 1 is also straightforward, since just all maps associating an activity implementation to one of the participant's activities $A \in N_{G_i}$ have to be copied. Regarding the set of control connectors E_{G_i} of the PM-Graph, the algorithm extracts all control connectors specified between two participant activities from the set of all control connectors E specified at the level of the CM-Graph. Since we do not discuss other types of conditions such as exit conditions or join conditions, and have only considered transition conditions, the set of $\mathcal{C}_{G_i}$ of the PM-Graph is defined by simply adding all the defined transition conditions from the control connectors to it. If other types of conditions will be reflected at the level of choreography models in the future, they also have to be added to the set of conditions of the PM-Graph accordingly. To create the input container and output container maps at line 9 and 10 of Algorithm 4.3 - Part 1, the relevant maps from the CM-Graph are copied similar as done for activity implementations. Therefore, all defined input container maps for the participant itself as well as its activities and conditions, and all defined output container maps for the participant itself and its activities are copied, respectively.

In line 11, we collect all correlation sets $CS(V_{G_i})$ relevant for the current participant R_i by aggregating all correlation sets associated to an activity $A \in R_i$ based on the specified correlation maps Δ_{CS} at the level of the CM-Graph (see Definition 4.21). Next, in line 12 the respective correlation set property maps $\Delta_{CP_{G_i}}$ for the above collected correlation sets $CS(V_{G_i})$ can be extracted from the CM-Graph. Since the data object related correlation set property maps are added as part of the staging elements as described in Section 4.4.1, only the message-related property maps have to be added to $\Delta_{CP_{G_i}}$. Finally, in line 13 the correlation map $\Delta_{CS_{G_i}}$ can be created by collecting all mappings between participant activities and respective correlation sets defined at the level of the CM-Graph.

The second part of the transformation algorithm is presented in Algorithm 4.3 - Part 2 where data-related aspects such as data connectors and the staging elements are added to the PM-Graph G_i. Therefore, in line 14 of the algorithm the set of data connectors is introduced by extracting all respective data connector maps between two activities or an activity and a condition specified within the participant at the level of the CM-Graph G_C. The same applies for the process data connectors in line 15. Since the notion of a *process data connector* $\overrightarrow{\Delta}$ defined at the level of PM-Graphs by Leymann and Roller [LR00], and shown in the PM-Graph summary in Definition 4.29, slightly differs to the choreography data connector $\overrightarrow{\Delta_C}$ introduced in Section 4.2.6.2, we have to translate the mappings into the correct process data connector format. The difference results from the fact, that at the level of a PM-Graph there is only one process for which corresponding process data connectors can be specified, to copy either data elements from the process input container to an activity input container or from an activity output container to the process output container. Therefore, the process itself has not to be reflected within a process data connector at all, since the source or target activities and the respective data maps are enough to specify all required information.

On the level of a choreography model, we need to be able to specify data connectors between choreography as well as participant inputs and outputs and respective activities, therefore we decided to introduce one consolidated

Algorithm 4.3 - Part 2 Generating PM-Graphs G_i

14: $\Delta_{G_i} \leftarrow \bigcup_{A \in N_{G_i},\, B \in N_{G_i} \cup \mathcal{C}_{G_i}} \overbrace{\pi_{17}(G_C)}^{\Delta_C}(A,B)$ ▷ data connector map

15: $\overrightarrow{\Delta}_{G_i} \leftarrow \{(A, \overrightarrow{\Delta}_{G_i}(A)) \mid A \in N_{G_i} \wedge \overrightarrow{\Delta}_{G_i}(A) = \overbrace{\pi_{19}(G_C)}^{\overrightarrow{\Delta_C}}(R_i,A) \cup \overbrace{\pi_{19}(G_C)}^{\overrightarrow{\Delta_C}}(A,R_i)\}$ ▷ process data connector map

$V_{G_i} \leftarrow \{v \in \overbrace{\pi_1(G_C)}^{V} \mid v \in (\mathcal{M}(V_{G_i})$

16: $\bigcup_{A \in N_{G_i} \cup \mathcal{C}_{G_i} \cup R_i} \iota_{G_i}(A) \cup$ ▷ Data elements

$\bigcup_{B \in N_{G_i} \cup R_i} o_{G_i}(B) \cup CS(V_{G_i}))\}$

17: **for all** $X \in (N_{G_i} \cup \mathcal{C}_{G_i} \cup \{P_i\})$ **do** ▷ Data processors and consumers

18: **for all** $e \in \overbrace{\pi_2(G_{C_{DDG}})}^{E_{C_{DDG}}} \mid \pi_1(e) = X \vee \pi_2(e) = X$ **do**

19: **if** $\pi_1(e) = X$ **then** ▷ Write dependency: $X \rightarrow d$

20: $d \leftarrow \pi_2(e)$

21: $\delta \leftarrow (\overbrace{\pi_3(e)}^{v_1 \in o_{G_i}(X)}, \overbrace{\pi_4(e)}^{v_2 \in d})$

22: $s_{\Sigma_\Psi} \leftarrow$ push

23: **else** ▷ Read dependency: $X \leftarrow d$

24: $d \leftarrow \pi_1(e)$

25: $\delta \leftarrow (\overbrace{\pi_3(e)}^{v_1 \in d}, \overbrace{\pi_4(e)}^{v_2 \in \iota_{G_i}(X)})$

26: $s_{\Sigma_\Psi} \leftarrow$ pull

27: **end if**

28: $CP \leftarrow \bigcup_{cs \in CS(V_{G_i})} \overbrace{\pi_{19}(G_C)}^{\Delta_{CP}}(cs,d)$

29: $s \leftarrow (d, \delta, CP, s_{\Sigma_\Psi}, \perp, \perp)$ ▷ Create a new staging element

30: $\Sigma_{G_i} \leftarrow \Sigma_{G_i} \cup s$ ▷ Add it to the set of staging elements

31: $\sigma_{G_i}(X) \leftarrow \sigma_{G_i}(X) \cup \{s\}$ ▷ Associate it to data consumer X

32: **end for**

33: **end for**

choreography data connector map which supports all such specifications by explicitly referring to the source and target elements as introduced in Section 4.2.6.2. The participants explicitly stated within a choreography data connector have therefore to be omitted when creating a corresponding process data connector as shown in line 15 of Algorithm 4.3 - Part 2.

To finalize the second step of the transformation process, in line 16 the set of data elements V_{G_i} relevant in the context of the participant can be created by collecting all data elements used as part of a defined message ($v \in m$ with $m \in \mathcal{M}(V_{G_i})$), input container map ($v \in \iota_{G_i}$), output container map ($v \in o_{G_i}$) or a correlation set ($v \in cs$ with $cs \in CS(V_{G_i})$). At this point, the generated PM-Graph G_i fully reflects all the specified control and message flows of choreography participant R_i.

However, only participant or process internal data flow via the translated data connectors and process data connectors is represented and therefore a representation of the specified cross-partner data flow is missing. Within the last step of the transformation, therefore, the PM-Graph is enriched with respective staging elements to incorporate and represent all data dependencies to cross-partner data objects collected in the CDDG as described in Section 4.4.1. As indicated by the conditions in Definition 4.28, we have to distinguish between read and write dependencies to data objects to set the correct staging method and data maps. Therefore, in line 17, Algorithm 4.3 - Part 2 iterates over the set of all data consumers and producers which are part of the participant the generated PM-Graph G_i represents. This comprises all activities, conditions and the process model P_i itself, since choreography data connectors support the specification of data dependencies between data objects and a participant as a whole as described in Section 4.2.6.2.

For the sake of simplicity we generate one staging element for each data dependency edge defined within the CDDG and associate it to the corresponding activity, condition or process model. Another option is to collect all read data dependencies and write data dependencies between a specific activity, condition or process model X and a specific data object d and then aggregate them into one staging element for the read dependencies and one for the write dependencies between X and d, respectively. In this case,

each of the two staging elements contains multiple data maps representing how to copy the respective data elements between the data containers of an activity, condition or process model and the data object. However, both approaches are identical in terms of their semantics and only vary in the number of staging elements generated and associated for a given activity, condition or process model with a data dependency to or from a data object.

Therefore, in the for-each loop starting on line 18 of Algorithm 4.3 - Part 2, staging elements for all data dependencies of a data consumer or producer X are generated and associated. Since the CDDG is created in a way, so that for each data dependency one edge within the graph exists, we simply have to loop over the set of all edges $E_{C_{DDG}}$ and identify the ones which originate from or point to the current element X selected within the surrounding loop. In accordance with the definition of staging elements in Section 4.4.1, the focus during transformation is on setting the reference to a data object d, a data map $\delta = (v_1, v_2)$ specifying how to copy data values between the data object and a data consumer or producer, a map of correlation set properties CP for the referenced data object, and a staging method $s_{\Sigma_\Psi} \in \Sigma_\Psi$. Corresponding trigger conditions and fault handling strategies can be added by a modeling expert during process model refinement, therefore, we use the bottom symbol $\perp$ introduced in Definition 4.27 during staging element generation to mark these elements as undefined. Potentially, this can be enhanced in future work by enabling the specification of corresponding aspects already at the level of a choreography model which can then be used to automatically create or associate related trigger conditions or fault handling strategies to the generated staging elements.

The if-then-else block starting on line 19 identifies if the current data dependency edge represents a read or write dependency of X on a data object. This is relevant since this has an influence on how to extract the related data from the underlying dependency edge e. For write data dependencies, the data object can be extracted from the target of the data dependency edge as shown on line 20. In addition, we can extract the required data map $\delta := (v_1, v_2)$, which specifies the source and target data element and therefore which data element value of the data producer's output container

has to be copied to the data object, from the data dependency edge as shown on line 21 of Algorithm 4.3 - Part 2. Finally, the respective staging method is set to s_{Σ_Ψ}, i. e., *push* for write dependencies. Read dependencies are handled accordingly starting from line 23 as part of the else branch. Only the projection indices change, since the data object is now the source of the edge and as staging method *pull* is set to request the respective data at the TraDE Middleware at runtime.

The rest of the staging element generation is identical for read and write dependencies. Therefore, the respective map of correlation set properties CP for data object d is extracted from the correlation set property map Δ_{CP} of the CM-Graph G_C as shown on line 21 of Algorithm 4.3 - Part 2. Based on that, a respective staging element s can be created through the combination of the introduced elements in line 29 and finally associated to the respective data consumer or producer X via the staging element map $\sigma_{G_i}(X)$. This is repeated until all data dependencies of the process model P_i are reflected through a corresponding staging element.

Algorithm 4.3 - Part 3 Generating PM-Graphs G_i

34: $G_i \leftarrow (V_{G_i}, \iota_{G_i}, o_{G_i}, \mathcal{M}(V_{G_i}), CS(V_{G_i}), \Delta_{CP_{G_i}}, \Delta_{CS_{G_i}}, N_{G_i}, \Psi_{G_i}, \mathcal{C}_{G_i}, E_{G_i}, \Delta_{G_i}, \overrightarrow{\Delta}_{G_i}, \Sigma_{G_i}, \sigma_{G_i})$ ▷ Combine elements into a PM-Graph

35: $\mathcal{P}_C \leftarrow \mathcal{P}_C \cup \{G_i\}$ ▷ Add PM-Graph to result set

36: **end for**

37: **return** $\mathcal{P}_C$ ▷ Set of PM-Graphs

38: **end procedure**

The third and final part of the transformation algorithm is presented in Algorithm 4.3 - Part 3. There everything is wrapped up by combining the collected and translated sets and maps from the other parts into an PM-Graph G_i which specifies the process model P_i implementing choreography participant R_i in line 34 of Algorithm 4.3 - Part 3. The resulting PM-Graph G_i is then added in line 35 to the set of PM-Graphs $\mathcal{P}_C$ which is returned as the final result of the algorithm as soon as a process model for each choreography

participant is generated and therefore the for-each loop ends. Based on that, the overall transformation process is completed and all artifacts depicted in Figure 4.20 are available for refinement by corresponding modeling experts of the stakeholders involved in the choreography as described in Section 3.3.

Chapter 5

A Middleware for Data-Aware Choreography Models

In Chapter 3, we motivate and introduce our vision for Transparent Data Exchange (TraDE) in choreographies by introducing the notion of data-aware choreographies and a respective life cycle reflecting data-related aspects throughout the complete choreography management life cycle. Based on that, in Chapter 4 the focus is on the modeling and refinement phases of the life cycle by introducing Choreography Model Graphs (CM-Graphs) as a modeling notation for data-aware choreographies with our TraDE concepts applied. In this chapter, we will further describe how to support the execution life cycle phase, i. e., executing data-aware choreography models and their specified cross-partner data flows based on our CM-Graph metamodel. Therefore, in Section 5.1 we provide a short recap on how the execution of a data-aware choreography model with the help of the TraDE Middleware as a data hub looks like based on an example choreography as outlined in

Section 3.2.2. Afterwards, we will go into more detail on how the TraDE Middleware actually conducts the modeled cross-partner data flows together with respective Business Process Engines (BPEs), e. g., by presenting the internal conceptual model of the middleware in Section 5.2, its underlying architecture in Section 5.4 or its integration with BPEs in Section 5.5.

5.1 Overview

Figure 5.1 shows the resulting artifacts of the transformation of the example data-aware choreography model depicted in Figure 3.1 to a collection of process models, a Choreography Data Model (CDM) and a Choreography Data Dependency Graph (CDDG) as described in Section 4.5. The resulting three participant process models are represented with our formal model and visual notation as introduced in Chapter 4 where each of the process models is deployed to a respective BPE. The CDM and the CDDG representing the modeled cross-partner data objects and data flows are deployed to the TraDE Middleware as shown on the left of Figure 5.1. The dependencies of activities on cross-partner data objects are represented within the process models through corresponding staging elements as introduced in Section 4.4.1 which are added to each process model as a result of the choreography to process model transformation described in Section 4.5.2. The resulting staging elements therefore represent the data dependencies on cross-partner data objects of the summarized CDDG deployed to the TraDE Middleware. To keep the process models as simple as possible, we omitted the process internal data flow, i. e., the data connectors between the activities.

As soon as BPE 1 receives an incoming request for process model P_1, a new choreography instance is started by creating a new instance of the process model which consumes the incoming request through its *Receive Chor. Request* receiving communication activity. The activity extracts the message payload and copies the respective data element values to its output container. Afterwards, BPE 1 conducts the modeled cross-partner data flow represented through the two staging elements 1a and 1b shown in Figure 5.1. Therefore,

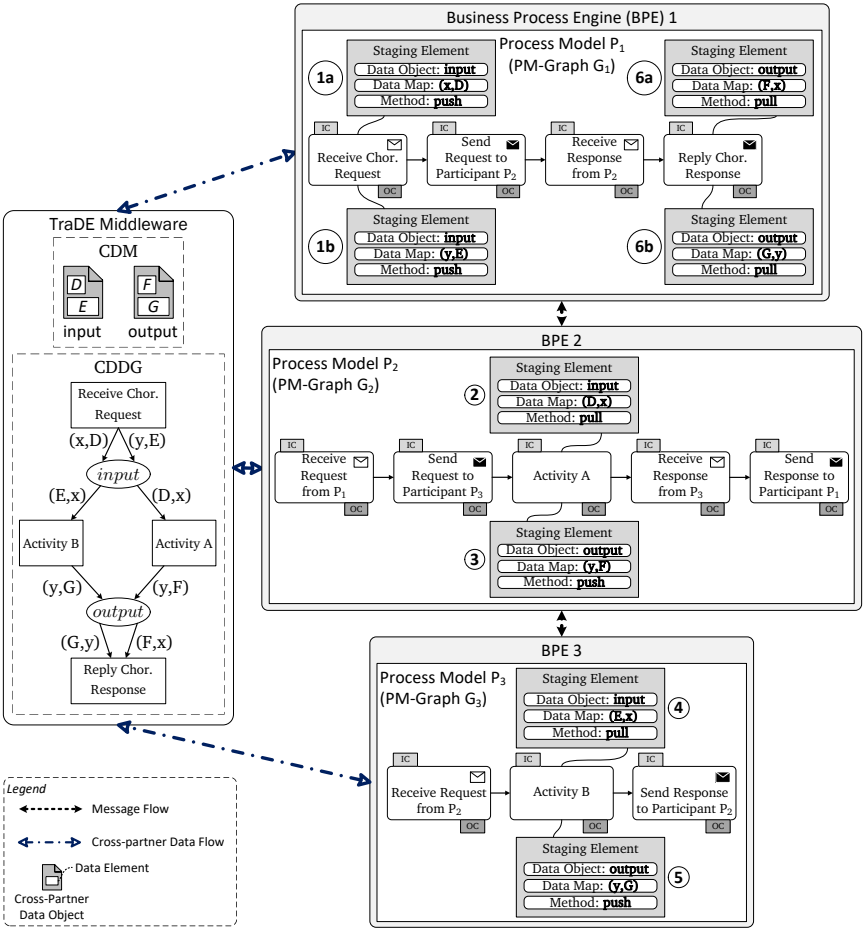

Figure 5.1: Execution of a data-aware choreography model with its specified cross-partner data flows using the TraDE Middleware, based on the choreography presented in Section 3.2.

the extracted message payload available at the output container of the receive activity is pushed by the BPE to the *input* data object at the TraDE Middleware using the middleware's Application Programming Interface (API). According to the specified data maps within the staging elements, in step 1a, data element x of the activity's output container is copied to data element D of data object *input* and data element y is copied from the output container to data element E of data object *input* in step 1b, respectively. After the receive activity completes, data object *input* is completely initialized and has respective values set to be used in the following. Next, the *Send Request to Participant* P_2 sending communication activity is scheduled which invokes process model P_2 by sending a request message to BPE 2. Upon receipt of the message, a new process instance is created which consumes the request message through the modeled *Receive Request from* P_1 receive activity. The *Send Request to Participant* P_3 send activity then directly invokes process model P_3 by sending a request message to BPE 3. There again a new process instance is created upon receipt of the message which consumes the request message through the modeled *Receive Request from* P_2 receive activity. In the following, the process instances of process model P_2 and P_3 are running in parallel, executing activities A or B next, respectively. Activity A of process model P_2 therefore pulls data element D from data object *input* at the TraDE Middleware to store it to data element x of activity A's input container as specified by staging element 2 depicted in Figure 5.1. After the output data of the executed activity implementation of activity A is materialized in the activity's output container, the data value of data element y is pushed to data element F of data object *output* to the TraDE Middleware as depicted by staging element 3 in Figure 5.1. In parallel, activity B of process model P_3 pulls data element E from data object *input* at the TraDE Middleware to store it to data element x of activity B's input container as specified by staging element 4 depicted in Figure 5.1. Afterwards, the results are pushed from data element y of the output container of the activity to data element G of the *output* data object as specified in staging element 5. By sending a response message back to participant P_2 via the modeled *Send Response to Participant* P_2 send activity, the process instance of process model P_3 is

terminated. The same applies to the process instance of process model P_2 which receives the response message from P_3 via the *Receive Response from* P_3 activity and then also terminates after sending its final response to P_1 via the modeled *Send Response to Participant* P_1 send activity. Finally, activity *Receive Response from* P_2 of process P_1 consumes the response message and then prepares the final response message of the choreography by pulling the results of activities A and B from the *output* data object at the TraDE Middleware. Therefore, according to staging elements 6a and 6b shown in Figure 5.1, data elements *F* and *G* of data object *output* are requested from the TraDE Middleware and copied to data elements *x* and *y* of the input container of activity *Reply Chor. Response*, respectively. From there the values are wrapped into the final response message which is then send back to the client who started the choreography instance with its initial request and then also the instance of process model P_1 is terminated which results in the completion of the overall choreography instance.

5.2 Conceptual Model of the Middleware

Since the middleware and its underlying concepts should not be bound to any specific choreography and process modeling languages or related runtime environments, we introduced our formal model for CM-Graphs and an extended Process Model Graph (PM-Graph) metamodel [LR00] in Chapter 4. Based on that, the TraDE Middleware comes with its own internal metamodel shown in Figure 5.2 which represents the choreography data dependency graphs (CDDG) and choreography data models (CDM) introduced in Chapter 4 with respective cross-partner data objects and data elements as well as any related information in a choreography and process modeling language independent format within the middleware.

The entry point of the conceptual model is the *ChoreographyDataDependencyGraph* entity which represents a CDDG as introduced in Section 4.3.2. To uniquely identify such a CDDG within the middleware it has a *name* and *namespace* attribute associated as depicted in Figure 5.2. Using namespaces

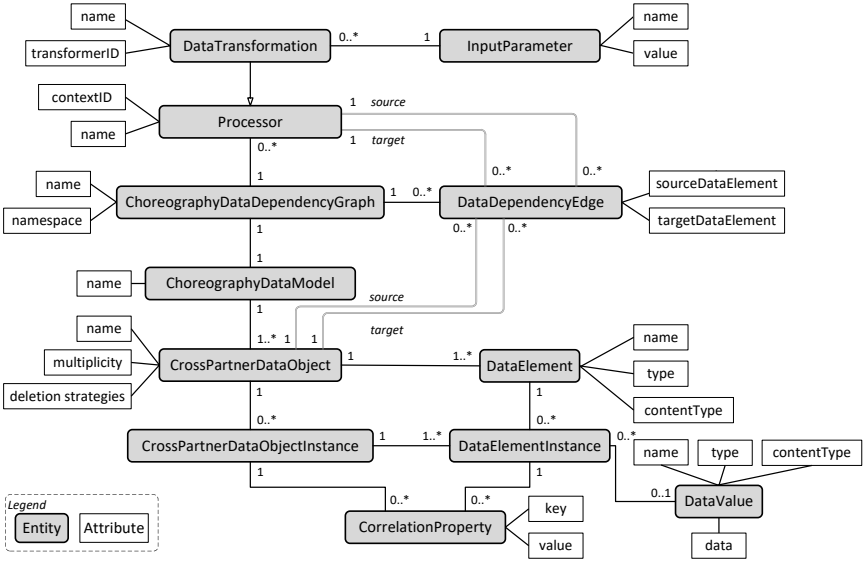

Figure 5.2: Metamodel of a Choreography Data Dependency Graph and its Choreography Data Model within the TraDE Middleware, based on [HBLW17].

to specify the context for which the underlying model and its contained elements are defined, is an established approach. For example, namespaces are heavily used within eXtensible Markup Language (XML) by using XML namespaces [XML-NS] to uniquely identify models and their elements within and across XML documents. A namespace itself is represented in form of a Uniform Resource Identifier (URI) [BFM05] that can be added to a model, i. e., persisted within its document format, and therefore allows to uniquely identify the model as well as its contained elements. Therefore, different models can use the same element names without introducing any conflicts as long as their respective namespaces are not overlapping or in conflict. Since both Business Process Model and Notation (BPMN) and BPEL4Chor are XML-based modeling notations, the namespace specified for the choreography model can be simply set also to the CDDG during its generation while

the name can be set constant, e. g., to *CDDG* or also to the name specified for the choreography model. This enables us to identify which CDDG deployed to the TraDE Middleware belongs to which choreography model what is especially important during runtime in terms of instance correlation, i. e., finding the right data object instance to conduct the modeled cross-partner data flow of a choreography model.

As introduced in Section 4.3.2, a CDDG contains a choreography data model (CDM) comprising a set of data objects as well as corresponding data processors and their data dependencies on the data objects. All those elements are reflected within the conceptual model depicted in Figure 5.2 through corresponding entities and relations between them. Therefore, a *ChoreographyDataDependencyGraph* entity refers to zero or more *Processor* entities which can be the source or target of a *DataDependencyEdge*.

To uniquely identify a data processor, e. g., a participant, activity or condition, a Processor entity has a *contextID* and *name* attribute. While the name is simply the name of the data processor as specified in the underlying choreography model, the contextID specifies the related context in which the processor is defined. The value of the contextID can therefore be provided by simply specifying the name of the underlying participant or even an XML Path Language (XPath) [XPath] for XML-based modeling languages can be used to provide a reference to the right context. According to the definition in Section 4.3.2, a *DataDependencyEdge* is a 4-tuple referring to the source and target entity as well as representing a data map between a data element of the source entity and a data element of the target entity which are represented through the *sourceDataElement* and *targetDataElement* attributes shown in Figure 5.2, respectively.

A special kind of data processors are represented through *DataTransformation* entities which allow to specify data transformations between two cross-partner data objects conducted by the TraDE Middleware as part of the specified cross-partner data flow. As shown in Figure 5.2, a *DataTransformation* entity has a *name* and *transformerID* attribute. The former holds the name of the data transformation specified as part of the choreography model and the latter contains the specified reference to a corresponding transfor-

mation implementation. Provided input parameters of a data transformation specified at the level of a choreography model are represented through one or multiple associated *InputParameter* entities. An InputParameter has a unique *name* and a corresponding *value*. The whole concept of TraDE data transformations will be introduced in Chapter 6 in detail.

The collection of cross-partner data objects specified within a choreography model is represented by a *ChoreographyDataModel* entity in accordance with the definition of a CDM presented in Section 4.3.1. A ChoreographyDataModel has a *name* attribute for providing its qualified name and contains one or more *CrossPartnerDataObject* entities. A CrossPartnerDataObject has a *name*, *multiplicity* and *deletion strategies* attribute as well as a reference to its ChoreographyDataModel and contains one or more *DataElement* entities. Again the name attribute holds the name of the data object specified at the level of the choreography model. The multiplicity and deletion strategies attributes represent the respective property values of the data object as introduced as part of our formal model for data objects in Section 4.2.1.5. Therefore, the multiplicity attribute specifies how many instances of the data object may exist during choreography runtime, ranging from one instance over a specific upper bound to an unknown, potentially unlimited number of instances. In addition, the specified deletion strategies define if and at which point in time the created data object instances and their associated values should be deleted by the TraDE Middleware in terms of automatic garbage collection. This information is used by the TraDE Middleware to enforce the specified properties regarding the number of data object instances and their deletion during the execution of the underlying choreography.

As outlined above, each DataObject entity contains one or more DataElement entities. Each DataElement has a *name*, a *type* and a *contentType* attribute. The name attribute again represents the data element's name as specified within the choreography model. The type attribute specifies the data element's structure, in accordance with the formal model as described in Section 4.2.1.1, in a concrete manner for a respective data format, e. g., using XML Schema Definition (XSD) [XSD1] for defining XML-based [XML] data structures or using JavaScript Object Notation (JSON) Schema [JSON

Schema] for defining JSON-based [Bra17] or YAML-based [YAML] data structures, respectively. In addition, the contentType attribute represents the kind of data hold by the data element and therefore provides the means for the interpretation of the data independent of its structure. Based on that, the middleware can handle arbitrary binary data without parsing it while still being aware of its content and how to represent it, e. g., to human users. This allows us to support various types of data, e. g., structured and unstructured data, videos or pictures, as well as data formats and representations, e. g., XML, plain text, MPEG, or PNG. Such content type information can be specified, e. g., using *Media Types* [IANA].

At this point all model related entities of the conceptual model shown in Figure 5.2 are introduced and described. However, since multiple instances of a choreography model can run in parallel, we need corresponding entities to reflect the data related to each individual choreography instance for conducting the specified cross-partner data flows. To represent and manage the data of multiple instances of a choreography model referring to the same CrossPartnerDataObject and DataElement entities during runtime, we apply the well-known concept of *model instances* from BPM to our metamodel. Therefore, we introduce corresponding *CrossPartnerDataObjectInstance* and *DataElementInstance* entities which allow us to represent concrete instances of cross-partner data objects and their data elements for one specific instance of a choreography model. The creation of a new CrossPartnerDataObjectInstance also creates all related DataElementInstances according to the relation between the associated CrossPartnerDataObject and DataElement entities. In order to correlate the data managed by the TraDE Middleware, i. e., CrossPartnerDataObjectInstance and DataElementInstance entities, with a choreography model instance, corresponding correlation information have to be supported and provided by the metamodel. Therefore, we associate a set of *CorrelationProperty* entities to the DataObjectInstance and DataElementInstance entities in accordance with the correlation of data objects introduced as part of our formal model in Section 4.2.7. These CorrelationProperty entities allow to uniquely identify a choreography instance at the level of a BPE as well as to identify the data that belongs to this instance

at the level of the TraDE Middleware. Since the concept of property-based correlation is well known in the domain of BPM, we therefore reuse existing correlation mechanisms as provided, for example, by BPMN [BPMN] or Business Process Execution Language (BPEL) [BPEL] in order to enable the instance correlation between BPEs and the TraDE Middleware. To enable the reuse of data across choreography instances, concrete data should not be bound directly to one DataElementInstance entity. Therefore, the actual data is represented and provided by an independent *DataValue* entity as shown in Figure 5.2. This allows us to reuse and share DataValue entities across multiple DataElementInstance entities by referencing them. Moreover, it enables the manual creation of DataValue entities and therefore the upload of data to the middleware independent of a choreography instance. A DataValue has a *name*, a *type* defining the structure of its data and a *contentType* definition similarly as for DataElement entities. Furthermore, the instances of corresponding DataValue entities hold the concrete *data* values within the TraDE Middleware. In the Section 5.4, we will have a detailed look into the architecture of the TraDE Middleware and how it implements the introduced conceptual model.

5.3 TraDE Event Models

The entities introduced as part of the conceptual model in Section 5.2 reflect the cross-partner data flows and cross-partner data objects specified at the level of the underlying data-aware choreography model within the TraDE Middleware. Corresponding instances of the conceptual model are created during the deployment of a CDDG of a choreography model to the TraDE Middleware. The execution of the specified cross-partner data flows of a choreography results in respective *state changes* of the involved entities within the TraDE Middleware. For example, a new instance of a data object is *created* and then *initialized*, i. e., instances of its contained data elements are created and the provided values are stored to respective data values which are associated to the data elements, and as soon as it is not required

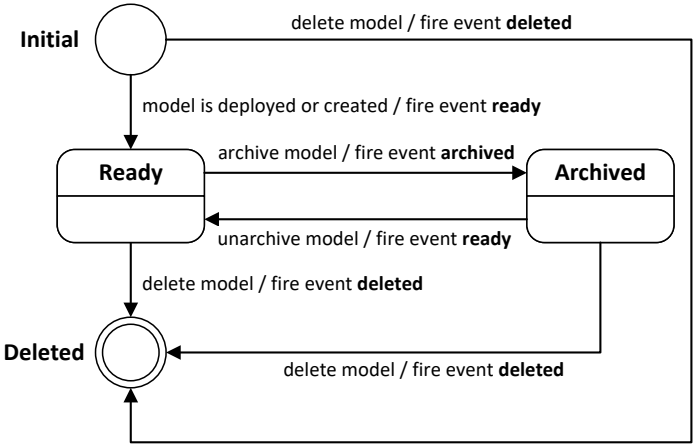

Figure 5.3: Event model underlying to all model entities within the TraDE Middleware represented as UML state diagram.

anymore the instance is *deleted* again. For auditing and monitoring purposes the emitted events during data flow execution can be stored in an audit log or may be published to a topic via messaging where interested parties can subscribe for receiving the events for monitoring the data flow execution.

In the following, we introduce the event models of the entities of the conceptual model used within our TraDE Middleware. We use the BPEL event model and its underlying syntax introduced by Kopp et al. [Kop+11b] as a basis for introducing our TraDE event models. Therefore, the event models are presented in form of Unified Modeling Language (UML) state diagrams [UML] specifying the possible states and transitions between them. Each state transition specifies its activating trigger and the event which is fired on the state change. In general, all entities of the conceptual model shown in Section 5.2 can be grouped into two categories, namely *models* and *model instances*, each following a common event model which we will introduce in the following.

Figure 5.3 shows the common event model for all entities being part of to the *models* group, i. e., ChoreographyDataDependencyGraph, Choreogra-

phyDataModel, CrossPartnerDataObject and DataElement. All such model entities are created within the middleware based on the deployment of a corresponding CDDG to the TraDE Middleware. Therefore, each model entity is in an *Initial* state on creation and as soon as all related entities are ready, the model entity also changes its state to *Ready* and a corresponding *ready* event is fired. For example, a DataObject entity becomes ready as soon as all of its contained DataElement entities have reached the *Ready* state. All model entities being in the *Ready* state are fully operational and can be instantiated. If a model entity, e. g., a CDDG and its contained model entities, are not actively required anymore, they can be archived or deleted.

This is reflected within the event model shown in Figure 5.3 by the corresponding *Archived* and *Deleted* states, respectively. When a model entity is archived, a corresponding *archived* event is fired and a *ready* event if it is unarchived again. In addition, a model entity can directly change from any introduced state to the *Deleted* state, if the respective entity is not required anymore. On every state change to *Deleted* a corresponding *deleted* event is fired within the middleware.

Figure 5.4 shows the common event model for all entities associated to the *model instances* group, i. e., CrossPartnerDataObjectInstance, DataElementInstance and DataValue. Whenever a new model instance entity for a CrossPartnerDataObject or DataElement entity or a new DataValue entity is created within the middleware, the state of the resulting instance entity changes from *Initial* to *Created* and a corresponding *created* event is fired. The *Created* state therefore reflects that a new entity is available which is ready to be used for conducting the modeled cross-partner data flow. As soon as a model instance entity is fully initialized, it changes its state to *Initialized* which is reflected by firing a respective *initialized* event. The meaning of fully initialized depends on the underlying type of model instance entity. As shown in Figure 5.4, a DataValue entity is initialized as soon as data is set to it. A DataElementInstance is marked as initialized as soon as the DataValue entity associated to it is initialized or an already initialized DataValue is directly associated to it. Finally, a DataObjectInstance is fully initialized if all of its associated DataElementInstance entities are

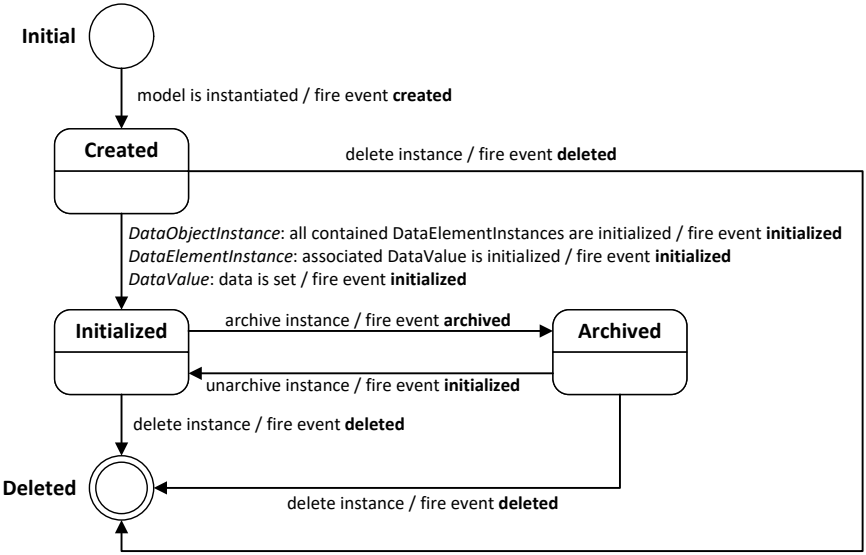

Figure 5.4: Event model underlying to all model instance entities within the TraDE Middleware represented as UML state diagram.

initialized. Similar to model entities, also model instance entities can be archived and unarchived again represented through the *Archived* state and the firing of respective *archived* and *initialized* events. In addition, a model instance entity can directly change from any introduced state to the *Deleted* state, if the respective entity is not required anymore. On every state change to *Deleted* a corresponding *deleted* event is fired within the middleware.

In addition to the two common event models presented above, Figure 5.5 shows the event model for the data held by corresponding DataValue entities, i. e., concrete data set to a data value during choreography runtime. This event model builds the basis for reflecting corresponding state changes and events in result to data changes during choreography runtime. While a DataValue entity is always in state *Unset* when being created, it changes its state to *Initialized* and fires a corresponding *initialized* event as soon as concrete data is set to it. As described above, this initial setting of data

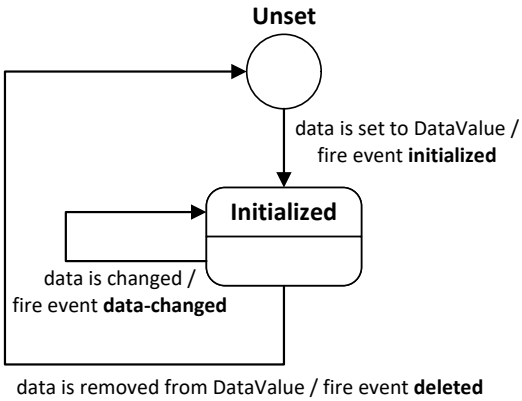

Figure 5.5: Event model representing the data perspective of DataValue entities represented as UML state diagram.

to a DataValue entity also changes the state of the DataValue entity to *Initial*. In addition, the data of a DataValue entity can change over time, i. e., depending on the modeled cross-partner data flow, different values can be set to a data element of a cross-partner data flow throughout the execution of a choreography. Whenever the data changes it remains in state *Initialized* and a *data-changed* event is fired to indicate that a new value is set to the DataValue entity. Another possibility for a data change is that the data is removed from the DataValue entity. This results in a state change back to the *Unset* state and firing a *deleted* event. Corresponding *initialized* and *data-changed* events are also used to realize the above mentioned event-based triggering of data transformations since a corresponding event is emitted within the middleware whenever the data of a DataValue changes. Such changes are the result of a conducted cross-partner data flow, i. e., a process engine is pushing data for a cross-partner data object to the TraDE Middleware or if a human user manually changes related data values.

5.4 Architecture of the Middleware

Figure 5.6 presents the architecture of the TraDE Middleware. The overall design follows the three layer architecture pattern [Fow02] where we introduce each of the layers and its components in a top-down manner.

The *Presentation* layer exposes the functionality of the TraDE Middleware through a corresponding *Web User Interface* (UI) and a *Representational State Transfer (REST) API*. Both components enable the interaction with the TraDE Middleware. While the focus of the REST API is more on the integration of the middleware with other systems, i. e., machine-to-machine interactions, the Web UI provides a graphical interface for human users on top of the REST API to support also human-to-machine interactions. By following the REST architectural style [Fie00], each entity of our internal TraDE metamodel presented in Section 5.2 is represented as a resource and can therefore be easily accessed, referenced, and shared through a respective Uniform Resource Locator (URL) [BFM05]. For example, BPEs can use the REST API to integrate with the middleware in order to push or pull data values to or from data objects, respectively. Modeling tools can use the REST API to support modelers with the direct deployment of a generated CDDG of a choreography model to the TraDE Middleware to make the specified data dependencies and cross-partner data objects available at the middleware, so that they can be used by BPEs during choreography runtime for conducting the modeled cross-partner data flows. Best practices and established patterns for designing such a REST API are presented, for example, by Allamaraju [All10] or Masse [Mas11]. In addition, human users can access and inspect the deployed CDDGs and their components as well as available data values or manually upload their data to the TraDE Middleware via the Web UI to make it available for use within choreographies, e. g., as input data.

The *Business Logic* layer provides the core functionality of the middleware made available through the presentation layer. Figure 5.6 shows the components providing the functionality of the middleware which will be introduced in the following. The *TraDE Instance Models* component implements the metamodel of the middleware introduced in Section 5.2 to represent concrete

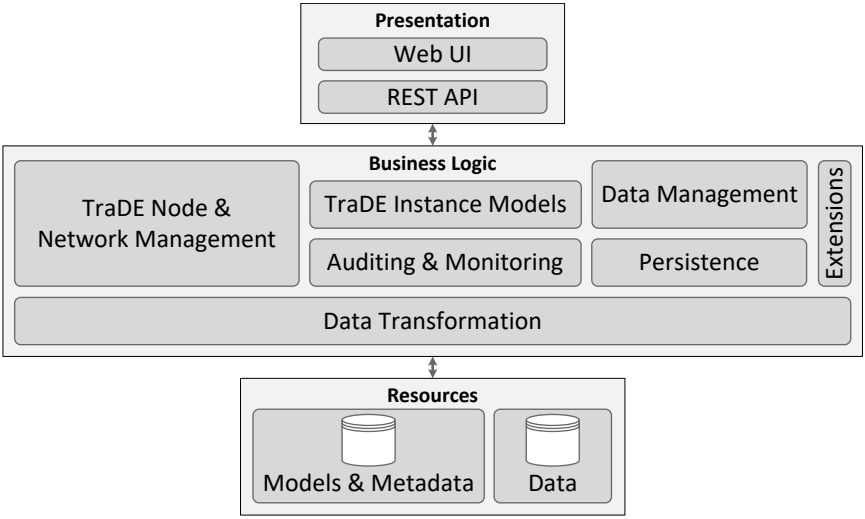

Figure 5.6: Architecture of the TraDE Middleware, based on [HBLW17].

instances of the metamodel's entities, i. e., a deployed CDDG, its contained CDM with all defined cross-partner data objects and data elements as well as instances of them and respective data values within the middleware. All functionality related to data management is provided by the *Data Management* component which is one of the core components of the middleware. It provides the access and inspection of data associated to corresponding cross-partner data objects through the REST API. Furthermore, it handles the upload and retrieval of data for a corresponding DataValue or DataElementInstance entity. Related to that, it also handles the correlation of the TraDE-internal metamodel instances with respective choreography instances in order to enable the BPEs to access and retrieve the correct entity instances of the metamodel and their associated data as described in detail in Section 4.2.7. Moreover, it is responsible for the management of the life cycle of the TraDE-internal metamodel instances. Therefore, the component uses the event models and life cycle operations described in Section 5.3 which are implemented by the *Auditing* & *Monitoring* component for each of the

metamodel entities shown in Figure 5.2. The Data Management component is also responsible to enforce and conduct the specified deletion strategies of data objects introduced in Section 4.2.1.5 which define at which point in time the created data object instances and their associated values should be deleted by the TraDE Middleware.

The *TraDE Node & Network Management* component is responsible for enabling a decentralized deployment of multiple middleware nodes and their connection into networks to allow more efficient data placement and staging as well as further optimizing the data exchange between the choreography participants. Within this work, we use a single TraDE Middleware node as default deployment scenario, i. e., all BPEs participating in the execution of respective process models implementing an underlying choreography communicate with one centralized TraDE Middleware. A decentralized, multi-instance deployment, e. g., having one TraDE Middleware node per BPE, may introduce new possibilities as well as improvements in terms of flexibility or further optimizing data staging and transfer between the interacting parties also during choreography runtime. However, this is not within the focus of this work, but maybe an interesting topic for future work.

To decouple the life time of the data within the TraDE Middleware from the choreography instances it belongs to, the *Persistence* component provides the required functionality to store both instances of the internal metamodel of the middleware as well as the managed data in an underlying data source in order to guarantee its availability for later (re)use. As outlined above, the *Auditing & Monitoring* component provides an associated life cycle based on a respective event model for each of the entities of the middleware's metamodel. This enables the auditing and monitoring of all entities by emitting corresponding events whenever the state of an entity changes. Furthermore, these internal events can be consumed by any interested component within the middleware, for example, allowing the Data Management component to trigger corresponding actions on state changes in order to realize the life cycle management of all TraDE Instance Models. The whole middleware is designed to be extensible in order to integrate new or adapt existing components. Therefore, the *Extensions* component provides corresponding

logic and mechanisms to plug-in new functionality as well as extensions or variants of existing components. For example, the default persistence component can be replaced by a new implementation that uses a different technology stack by adding it as an extension to the middleware.

To conduct the specified data transformations between cross-partner data objects outlined as part of the TraDE metamodel in Section 5.2, the *Data Transformation* component of the middleware shown in Figure 5.6 provides required functionality to trigger respective transformation logic in an event-based manner. Therefore, the component builds on top of the event models introduced in Section 5.3 in order to transparently trigger the specified data transformations on respective state changes of the underlying data element instances or their associated data values, respectively. For example, as soon as a data value is available, i. e., a value is set to the respective DataValue instance, the *Auditing* & *Monitoring* component of the TraDE Middleware emits a corresponding *initialized* state change event. The *Data Transformation* component listens to such emitted events and checks if a data transformation is specified, if so it triggers the specified data transformation logic. How this exactly looks like and all underlying concepts for the modeling and execution of TraDE data transformations and their event-based triggering through the TraDE Middleware will be introduced in Chapter 6 in detail.

The *Resources* layer contains all required resources used within the Business Logic layer. This comprises data sources for the persistence of TraDE Instance Models and related metadata about nodes and networks as depicted by the *Models* & *Metadata* data source in Figure 5.6 as wells as a data source for the actual data managed by the TraDE Middleware as presented by the *Data* data source in Figure 5.6. Furthermore, all other TraDE Middleware nodes belonging to the same network can be seen as a resource from the viewpoint of a single middleware node.

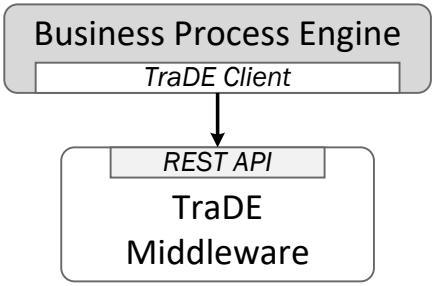

Figure 5.7: TraDE-aware approach for the integration of the TraDE Middleware with a BPE, based on [HBLW17].

5.5 Integration with Process Engines

In the following, we want to briefly discuss how the TraDE Middleware is integrated with a BPE within this work as shown in Figure 5.7 and in addition outline an alternative approach shown in Figure 5.8.

The *TraDE-aware integration* approach depicted in Figure 5.7 follows our formal model introduced in Chapter 4 where corresponding staging elements are generated at the level of the process models implementing a choreography to represent the modeled cross-partner data flows. Therefore, a corresponding BPE needs to know when and how to interact with the TraDE Middleware in order to conduct the specified data exchange based on a deployed process model. As a result, the implementation of the BPE has to be extended in two dimensions. First, the TraDE Middleware has to be integrated by adding a corresponding *TraDE Client* to the BPE which uses the REST API of the middleware to integrate the required functionality for pushing and pulling data values to or from the middleware. In addition, required logic to parse and conduct the specified staging elements of deployed process models has to be added to the BPE implementation which then uses the *TraDE Client* to communicate with the TraDE Middleware during process execution. The advantage of this approach is that the BPEs remain in control of the overall process executions, i. e., the control flow dimension of the choreography. The main disadvantage is that the BPE implementation

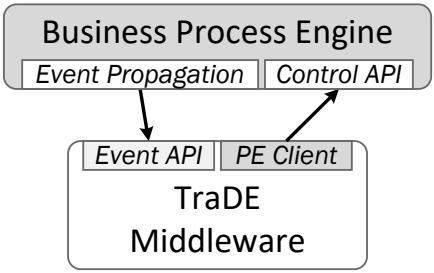

Figure 5.8: Two-way approach for the integration of the TraDE Middleware with a BPE, based on [HBLW17].

has to be extended in order to integrate the client and introduce required functionality to interpret and conduct the specified staging elements to push and pull data to or from cross-partner data objects at the TraDE Middleware. Especially if the collaborating partners use different BPE solutions this integration approach requires the same integration effort for each of the individual BPE implementations.

In contrary, the *two-way integration* approach depicted in Figure 5.8 integrates the BPE and the TraDE Middleware in a loosely coupled manner through corresponding APIs. The basic idea of this integration approach is to extract any data-related knowledge from the data-aware choreography models, e. g., in form of a CDDG that represents which participant requires or produces which cross-partner data objects, to move the control of executing specified cross-partner data flows completely to the TraDE Middleware. However, therefore the TraDE Middleware has to be able to control the process execution, e. g., by blocking the control flow to transfer required data from a data object to the BPE so that it can be consumed by an activity as input. Therefore, the BPEs have to expose the execution state of the process model instances through a corresponding event propagation mechanism, e. g., using messaging. The emitted state change events can then be consumed by an *Event API* at the TraDE Middleware, so that it is always aware of the current execution state of the overall choreography instances for which it executes the cross-partner data flows. Furthermore, the BPE implementations have

to expose a *Control API* which allows external parties, such as the TraDE Middleware, to control the process execution as well as retrieve and write data from and to data containers of the process model instances executed by a BPE. Khalaf, Karastoyanova, and Leymann [KKL07] present a respective pluggable framework for BPEL-based BPE solutions which supports both required extensions: emitting the execution status via an event propagation mechanism as well as enabling the control of a process instance execution from outside the BPE. The advantage of this integration approach is that the BPE implementation is not directly coupled with the TraDE Middleware. Instead it has to be only extended with generic event propagation functionality and expose required control as well as data management capabilities through an API. A respective event model for BPEL is presented by Kopp et al. [Kop+11b]. A BPE implementation combining the pluggable framework of Khalaf, Karastoyanova, and Leymann and the event model of Kopp et al. is introduced by Steinmetz [Ste08]. In any case, the required extensions are not only bound to the integration of the TraDE Middleware and may be potentially also useful in other application scenarios. The main disadvantage of this approach is that the TraDE Middleware has to keep track of the execution state of all choreography instances to fully take over control of the execution of the cross-partner data flows which increases the complexity of the TraDE Middleware implementation and requires to a certain extent control over and access to the BPEs as described above. Therefore, within this work we follow the *TraDE-aware integration* approach to integrate our prototypical implementation of the TraDE Middleware with a corresponding BPE which will be described in more detail in Chapter 7.

Chapter 6

Transparent Data Transformation in Data-Aware Choreographies

In the previous chapters, we introduced our concepts for Transparent Data Exchange (TraDE) as a means for the modeling and execution of data-aware choreographies. However, participants in service choreographies often rely on the composition of already existing business logic and therefore come up with their own internal data formats and models. These participant data models have to be integrated and consolidated with the overall choreography data model all participants rely on, i. e., the data contract they agreed upon choreography modeling time. Therefore, data transformation capabilities are needed to mediate between the potentially different data formats, structures and representations of the data used by the collabo-

rating parties. For example, one participant requires the aggregated data of another participant where the data has to be transformed accordingly before exchanging it. The generic solution is to explicitly model required data transformations as tasks within the choreography models or rely on established solutions such as Enterprise Service Buses (ESBs) [Cha04] and their message transformations capabilities [HW04]. However, an explicit modeling as tasks 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 the XML Query Language (XQuery) [XQuery] or Extensible Stylesheet Language Transformations (XSLT) [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 participant's business perspective, but technically required to realize the communication between participants of a choreography. In addition, the underlying transformation implementations become a part of the resulting process models implementing the overall choreography. As a consequence, the same transformation implementations may be spread across multiple different process models or maybe even choreography models which hinders their reuse and makes it harder to maintain them in a consistent manner. This is especially problematic when data formats of choreography participants change over time, as underlying choreography and process 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 make them invocable from their choreography models. Furthermore, this introduces an entry barrier for data-intensive application domains such as eScience, where data transformation logic goes far beyond simple format transformations or data aggregation. Using a service-based integration approach would require that scientists need to

manually wrap their data transformations as services and integrate them to the choreography runtime environment. On the other hand, relying on established message transformation capabilities such as provided by an ESBs, instead of explicitly modeling required data transformations as tasks within the choreography models, has also its limitations if the required data transformation logic goes far beyond simple format transformations or becomes very domain or use-case specific. For example, as mentioned above in data-intensive application domains such as eScience, often simulation-specific data transformation logic implemented by respective experts is required, e. g., to prepare simulation input data or to visualize results.

Therefore, in this chapter, the focus is on providing an overall, generic concept regarding the aforementioned data transformation capabilities to enable the specification of data transformations at the level of data-aware service choreographies as well as supporting their transparent execution during choreography runtime. The goal is that the specification of data transformations within choreography models and their binding to a respective transformation implementation during runtime is only loosely coupled while supporting and easing reuse. Therefore, an overall concept for *TraDE Data Transformations (TDTs)* is introduced. This concept comprises a TraDE *Data Transformation (DT)* modeling extension as well as a generic, technology-independent integration middleware which together enable to provide and invoke data transformation implementations in an easy and automated manner to realize an end-to-end support for the modeling and execution of data transformations in service choreographies.

Therefore, in Section 6.1 the TDT approach is outlined and further motivated. Based on that, in Section 6.2 a DT modeling extension and its underlying formal model are introduced. Finally, in Section 6.3 the required runtime support for modeled data transformations is presented by introducing a data transformation integration middleware and a concept for providing and invoking data transformation implementations as well as its integration with the TraDE Middleware described in Chapter 5.

6.1 The TraDE Data Transformation Approach

This section introduces the TraDE Data Transformation approach, for short *TDT*, which provides the underpinnings for an end-to-end support for the modeling and execution of data transformations within service choreographies. An overview and motivation of the approach and its building blocks is followed by a detailed description of the individual parts in the next sections.

6.1.1 Overview

As depicted by the left model shown in Figure 6.1, applying our TraDE concepts still forces modelers to manually specify data transformations by adding transformation tasks to a choreography model. Furthermore, it even requires modelers to introduce additional cross-partner data flows connecting the transformation tasks and their inputs and outputs represented by cross-partner data objects leading to more complex models. While the TraDE concepts allow to decouple data from participants by specifying cross-partner data objects, something similar for data transformations is missing, i. e., decoupling data transformations from concrete participants. Therefore, our goal is to provide an end-to-end support for the modeling and execution of data transformations in service choreographies independent of participants directly between modeled cross-partner data objects. The choreography model on the right of Figure 6.1 presents our idea on modeling data transformations in service choreographies in a seamless and straightforward manner. There, the transformation tasks T_1 and T_2 are replaced by corresponding cross-partner data flows with associated data transformation logic between the data objects E and G (for task T_1) as well as F and K (for task T_2). The underlying software, e. g., services, scripts or executables, that provide the data transformation logic and are invoked by the transformation tasks are called *DT Implementations* in the following. To enable the reuse and fully automate the deployment and execution of such DT Implementations referenced in a choreography model, concepts for their specification and packaging in so-called *DT Bundles* are introduced in Section 6.1.2.

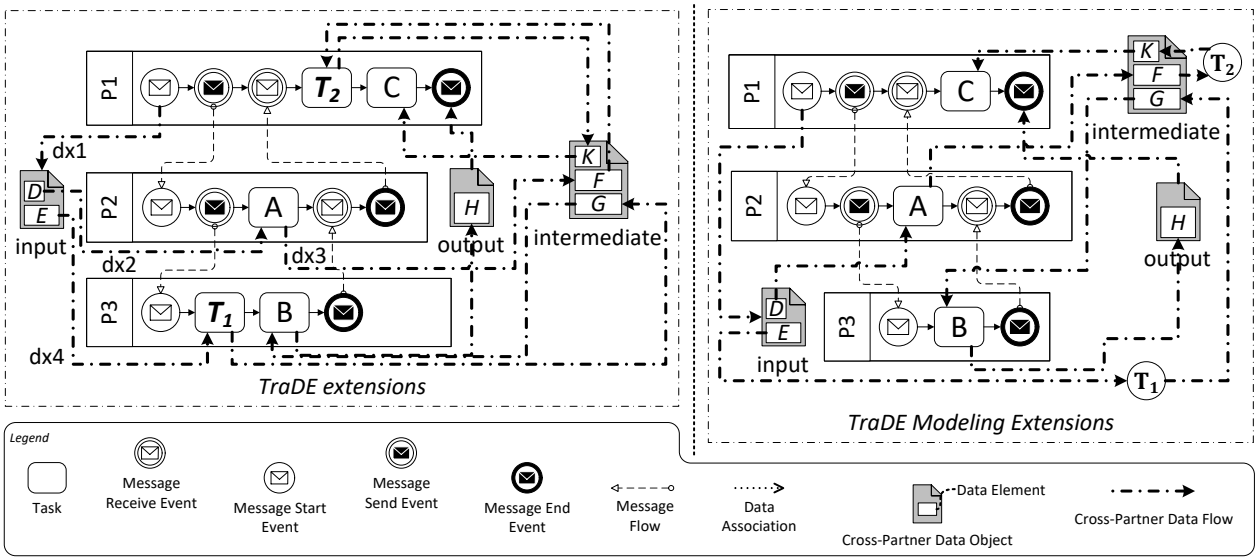

Figure 6.1: Comparison of an example data-aware choreography with task-based data transformations (left) and with TraDE data transformations (right) (based on [HBL+18]).

These concepts should abstract away any specifics of concrete technologies and standards to allow modelers to easily provide new DT Implementations without requiring previous knowledge on certain technologies or standards. The execution of DT Implementations based on the concepts defined in this chapter is supported through a new *DT Integration Middleware* presented in Section 6.1.3. This middleware allows modelers to publish their DT Implementations to make them available for use within service choreographies. Therefore, the DT Integration Middleware provides means for the provisioning of DT Implementations in form of DT Bundles and enables their uniform task-based invocation by exposing them through a generic interface.

Based on these concepts, a corresponding TraDE *Data Transformation* (DT) modeling extension is introduced in Section 6.2 which supports modelers with the specification of data transformations by combining DT Implementations and our TraDE concepts. Since our goal is to enable the transparent execution of such data transformations, i. e., without the necessity to explicitly model corresponding tasks in a choreography that integrate required logic, the underlying choreography runtime environment has to natively support the introduced DT modeling extension. The required transformation logic can be specified through these modeling extension by simply referencing DT Implementations as depicted in the choreography model on the right of Figure 6.1. For example, the cross-partner data flow between data elements F and K references transformation T_2 that is provided as a DT Implementation. Finally, in Section 6.3, the execution of modeled data transformations during choreography runtime based on the presented TDT approach is described.

6.1.2 Specification and Packaging of DT Implementations

To support the execution of data transformations specified within a choreography model, the referenced transformation implementations (DT Implementation) need to be integrated into the choreography execution environment somehow. In addition, data transformations are often implemented in different programming languages or restricted to certain execution en-

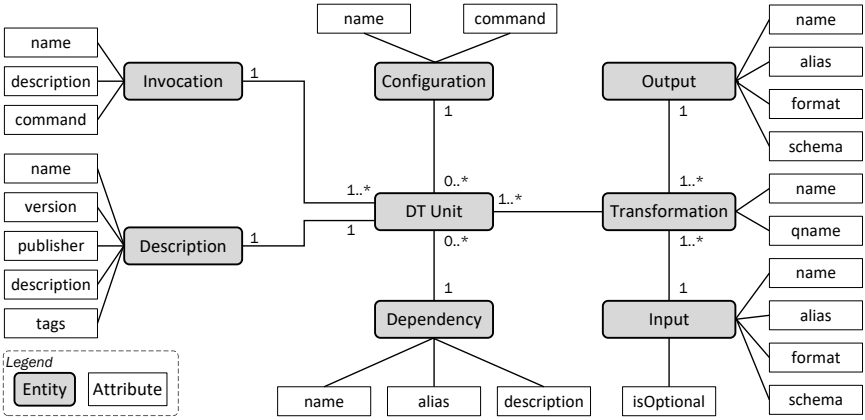

Figure 6.2: A conceptual metamodel for specifying DT Units [HBLY18].

vironments. To tackle such heterogeneity, we present concepts for the technology-agnostic specification and packaging of DT Implementations in so-called DT Bundles in the following. 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.

Therefore, this section introduces a metamodel for the specification and packaging of DT Implementations which provides means for storing, searching, provisioning, and automatically executing DT Implementations. Based on the findings of related work discussed in Chapter 2 and a 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 use cases and functionalities.

Figure 6.2 shows our proposal for an extensible conceptual metamodel for the specification of *data transformation units* (DT Units). In contrast to the already introduced DT Bundles, 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 executable scripts, configuration files or installation scripts to setup the required tools or frameworks for executing respective transformations. As outlined previously, DT Implementations can vary in different dimensions and can be highly use case specific, we therefore employ a black-box approach, i. e., considering DT Implementations as atomic reusable entities. Since DT Implementations should be usable within service choreographies, we assume that they are runnable in a fully automated manner and do not rely on any user interactions.

Apart from general information such as name, version, or publisher specified as *Description* entity shown in Figure 6.2, a DT Unit has several more important characteristics. First, any DT Unit might support one or more transformations, e. g., transforming textual data into several different image formats. This is specified through the *Transformation* entities shown in Figure 6.2 which represent DT Implementations. A Transformation has a name and a unique fully-qualified name (QName) which can be used, e. g., for searching transformations at the DT Integration Middleware or to link them as implementations within our TraDE data transformations modeling extensions as part of a choreography model. This linking will be described in detail in see Section 6.2. Each transformation is described by one or more inputs and one or more outputs specified through corresponding *Input* and *Output* entities depicted in Figure 6.2. Each input or output has a name and must be uniquely identifiable by some alias, which can later 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 or Representational State Transfer (REST) resources. The inputs and outputs of a transformation have a specific format and might provide a schema file describing their data format. Furthermore, some of the inputs of a transformation can be specified as optional.

Every DT Unit can have a particular set of dependencies that have to be satisfied to execute their transformations, represented as *Dependency* entities in Figure 6.2. 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 required. Such dependencies 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 required installation commands.

Moreover, the execution of some preparation steps or 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 as shown in Figure 6.2. Finally, every DT Unit has to specify how to invoke its provided transformations represented through *Invocation* entities in Figure 6.2. For example, the underlying transformations can be invoked by sending a request to an Application Programming Interface (API) or executing a command using the Command Line Interface (CLI) of an OS. Such invocation commands might need to reference other model's entities, e. g., inputs. Therefore, a predefined format for inserting aliases into the invocation command has to be used.

To package specified 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 within the DT Integration Middleware later. A packaged DT Unit, i. e., a DT Bundle, consists therefore of three distinct parts namely *unit*, *dependencies* and *schemas*, and in addition the *DT Unit specification* file. The *unit* part contains all DT Unit-related files, e. g., DT Implementation artifacts such as scripts or executables. All required dependencies such as software, libraries or (configuration) files are grouped into a *dependencies* part. The *schemas* part contains all provided schema files which define the structure of transformation inputs and outputs. Finally, the *DT Unit specification* file specifies all entities of the DT Unit and its relations and therefore provides an instance of the conceptual model presented in Figure 6.2 for a concrete DT Unit materialized as a file, e. g., as JSON file.

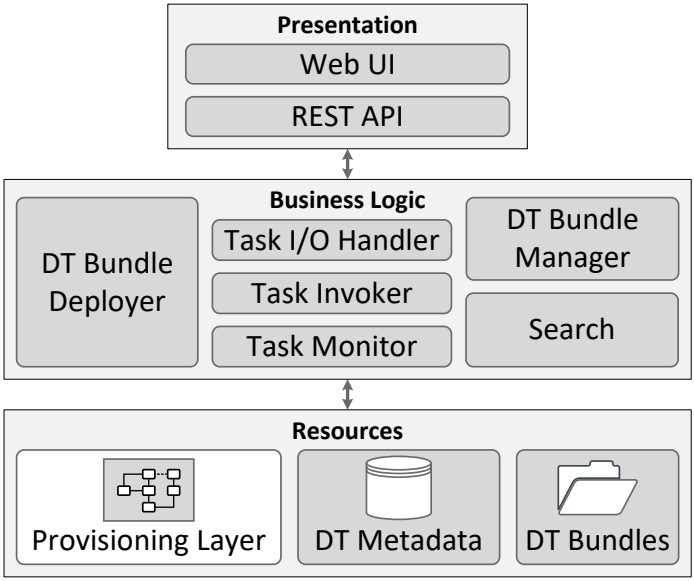

Figure 6.3: Architecture of the DT Integration Middleware [HBLY18].

6.1.3 Architecture of the DT Integration Middleware

Figure 6.3 presents the architecture of the DT Integration Middleware which allows providers to publish their DT Bundles to make the contained data transformation implementations available for consumers, e. g., to use them within choreographies. The middleware itself acts as a registry between the providers and consumers of DT Bundles. To allow a broad variety of implementations of the presented architecture, it is defined in a generic and technology-independent manner. Therefore, the focus is on the description of the logical building blocks, their functionality and related user interactions. Those building blocks can then be combined or implemented, e. g., through well-established middleware solutions like ESBs [Cha04] known for their integration and transformation capabilities in the context of Service-oriented Architectures (SOA). The prototypical implementation developed as part of this work is presented in Chapter 7.

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 user, i. e., providers as well as consumers of DT Bundles, 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 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 Section 6.1.2) in the file system (*DT Bundles* in Figure 6.3). The related metadata of all managed DT Units and DT Bundles is persisted in a database (*DT Metadata* in Figure 6.3) 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 [Bin+13], to setup and provide the specified runtime environment and software dependencies of a DT Bundle.

Whenever a DT Bundle provider registers a new DT Bundle at the middleware, it has to be prepared for provisioning first to make its specified DT Implementations invokable for consumers. Since the specification of a DT Unit contained in a DT Bundle can have references to remote resources, e. g., files or software dependencies, these references need to be materialized, i. e., resolved and persisted locally, to preserve the state and behavior of the DT Bundle within the middleware, as referenced resources may change over time leading to different variants of a bundle. For example, if a certain software dependency is specified, exactly the version of the software specified has to be present to guarantee that the underlying data transformations implementations are working properly. If materialization happens, the DT Unit specification has to reflect the changes affecting materialized references, i. e., the references are updated to refer to the resolved and locally persisted artifacts. 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 Figure 6.3 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 a provisioning-ready specification can be provided in form of, e. g., a Dockerfile or a TOSCA [TOSCA] topology. Which target format is used is determined by the Provisioning Layer used by the middleware implementation or if more than one is supported, the provider or consumer of a DT Bundle may be allowed to specify its preferences. The *Search* component allows DT Bundle consumers to search and identify suitable transformations of available DT Bundles by utilizing the metadata provided through the available DT Unit specifications. Search can employ various techniques from a trivial unique name search to the composition of multiple transformations together to produce a desired output from the provided input. The former allows consumers to find a specific transformation via its identifier while the latter allows to search for possible sequences of transformations to realize the requested data transformation. For example, if transformation A allows to transform data from XML to JSON format and transformation B allows to transform data from JSON to PDF format, the sequence of transformation A and B, in addition, allows to transform data from XML to PDF format. The *DT Bundle Deployer* is responsible for deploying DT Bundles of providers 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 provider's or consumer'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 invoked on a consumer's request. Therefore, a consumer issues a new *transformation task* 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 Section 6.1.2). 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 on behalf of a consumer's request. Therefore, inputs can be received in a pull or push-based manner. In the former case, inputs are provided as references within a consumer's requests and need to be downloaded and prepared for the invocation of a data transformation. 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 as specified within the related DT Unit specification (*Invocation* in Section 6.1.2). During the execution of the transformation, the *Task Monitor* component allows consumers 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 consumer.

The DT Integration Middleware has to be capable of handling various types of inputs and outputs (e. g., files, messages, or data streams) as well as supporting different invocation mechanisms (e. g., CLI or HTTP) and monitoring concepts for different types of data transformation. Therefore, the middleware has to support the integration of various implementations of the *Task I/O Handler*, *Task Invoker*, and *Task Monitor* components in a pluggable manner. To automate the execution of data transformations in choreographies, the middleware is integrated with the TraDE Middleware which is further discussed in Section 6.3.

The TDT approach enables us to use the introduced DT Bundles and their contained DT Implementations for the modeling and execution of data

transformations in service choreographies. Therefore, in the following a modeling extension for data transformations and its execution based on the introduced DT Bundles and our TraDE concepts are described.

6.2 Modeling Data Transformations in Service Choreographies

Figure 6.4 depicts our data transformation (DT) modeling extension for the specification of data transformations in data-aware service choreography models. Modelers can use the new DT element to specify a required data transformation directly between a set of cross-partner data objects independent of participants. All sources and targets of such a DT element have to be cross-partner data objects. The rationale behind that restriction is twofold. On one hand, this guarantees that there always exists an independent container (i. e., cross-partner data object) for all input and output data of a data transformation which will be materialized during runtime. This is required since the specified cross-partner data objects define the choreography data in terms of data formats, structures and related properties. Furthermore, cross-partner data objects represent the data that will be produced and consumed by the choreography's tasks and therefore reflect the potential inputs for data transformations at the level of a choreography. Therefore, a cross-partner data flow connecting a cross-partner data object and a DT element is well-defined, while for a cross-partner data flow between an activity and a DT element it is unclear what data can be expected as the activities' output. On the other hand, this allows modelers to graphically specify the inputs and outputs of a data transformation by simply connecting cross-partner data objects via cross-partner data flows with them. This restriction will be also valuable for providing more advanced functionalities in future, e. g., data provenance, sharing of transformation results, or monitoring of data exchange and transformations. However, there may be cases where it will be beneficial to enable the specification of data transformations as part of a data connector to enable the execution of data transformations in a transient manner. This means, the specified data transformation logic is

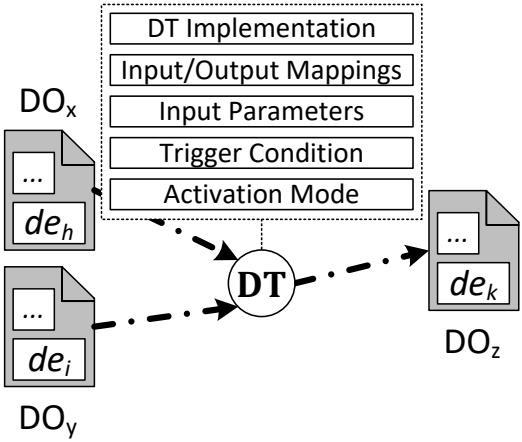

Figure 6.4: Example of a data transformation through a new *DT* modeling extension and cross-partner data flows [HBL+18].

executed based on the demand of the data consumer without persisting the transformation result in a dedicated data object. Introducing this variant for the specification and execution of data transformations may be an interesting topic for future work.

Figure 6.4 shows an example for a data transformation defined through a DT modeling element between the cross-partner data objects DO_x, DO_y and DO_z connected through corresponding cross-partner data flows. The DT element contains a reference to the software that provides the related data transformation logic referred to as *DT Implementation*. For example, specifying the fully-qualified name of a Transformation of a DT Bundle available at the DT Integration Middleware as introduced above. 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*. The definition of such

mappings might differ based on the type of the underlying DT Implementation. Therefore, modelers might be also graphically supported through the choreography modeling environment by utilizing available knowledge about required inputs and outputs of a DT Implementation, e. g., by extracting related information from the DT Unit specification of the referenced DT Implementation. 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 specify constant values for the configuration or initialization of the underlying DT Implementation, e. g., to select the output file format if multiple target formats are supported by the 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 has to be evaluated to *true* before the referenced DT Implementation is executed. The activation mode enables modelers to specify when the data transformation should be conducted: *on-read* or *on-write*. This will be discussed in more detail in the context of the execution semantics of the DT element in Section 6.3.

The binding of modeled data transformations to concrete logic, specifying a DT Implementation with a DT modeling element, is not necessarily 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. By supporting the definition of data transformations independent of choreography participants' control flow directly between cross-partner data objects, the open question is how to provide, integrate

and invoke the underlying data transformation implementations for modeled data transformations within service choreographies during choreography execution. Before we discuss the execution of TraDE data transformations in Section 6.3 in detail, Section 6.2.1 presents the means for the dynamic definition of data transformation input parameters in form of a TraDE Query Language and Section 6.2.2 formalizes our data transformation modeling extension to enable its integration into our formal model for data-aware choreographies presented in Chapter 4.

6.2.1 A TraDE Query Language

Listing 6.1 shows our proposal of a grammar for a TraDE Query Language (TQL) that allows to specify transformation parameter values in a dynamic way. The grammar is provided in a simple Extended Backus-Naur Form (EBNF) notation as used, for example, in the eXtensible Markup Language (XML) specification [XML]. The structure of the query language builds on top of the internal, choreography language independent metamodel of the TraDE Middleware presented in Section 5.2.

For a better understanding, we provide a brief recap of it before introducing the TQL on top of it. All cross-partner data objects and their contained data elements are represented through corresponding *CrossPartnerDataObject* and *DataElement* entities at the TraDE Middleware. A DataElement can refer to a single or a collection of data values. In addition to the model perspective, the metamodel of the TraDE Middleware provides related entities to represent instances of cross-partner data objects and data elements to reflect the runtime perspective of data-aware choreographies, i. e., manage the data of choreography instances. For each choreography instance corresponding *CrossPartnerDataObjectInstance* and *DataElementInstance* entities will be created at the TraDE Middleware with associated *CorrelationProperty* entities that enable to uniquely identify to which choreography instance the data object and data element instances belong. The actual data is provided through *DataValue* entities which are referenced by one or more *DataElementInstance* entities. A DataElementInstance entity associated to a multi-instance data

object always refers to a collection of *DataValue* entities.

As shown in Listing 6.1, a query always starts with a **$** character to enable the TraDE Middleware to distinguish between constant input parameter values and queries. After that, two possible types of query strings that follow an Uniform Resource Locator (URL)-like structure can be specified. A *DataElementQuery* enables modelers to refer to a property of a data element. Therefore, it contains a reference to a data object (*DataObjectRef*) followed by a reference to a data element (*DataElementRef*) and a selection of a property (*PropertyName*) as shown in Listing 6.1 (line 3-4). While the queries are specified at the level of the choreography models, i. e., referencing data objects and data elements in the modeling tool, they will be evaluated during runtime for each individual choreography instance at the TraDE Middleware. All specified references to data objects and data elements within the queries will be resolved to corresponding DataObjectsInstance, DataElementInstance and DataValue entities at the TraDE Middleware during choreography runtime. This will be discussed in more detail in Section 6.3. Valid property names are *size* and *url* at the moment while this list can be extended in future (cf. line 10 in Listing 6.1). A query like $\texttt{\$dataObject}_x\texttt{/dataElement}_i\texttt{?size}$ returns the number of data values associated to an instance of dataElement_i of dataObject_x. If the referenced data element is defined as part of a multi-instance data object (see Section 4.2.1.5), the current number of associated data values is returned, else the size is always *one*. The *url* property returns the URL under which the DataElementInstance entity can be retrieved at the REST API of the TraDE Middleware. This allows to easily share and distribute the resolved URLs as a means to provide references to data objects and the data values they contain.

```
1 Query             ::= '$' ( DataElementQuery |
    DataValueQuery )
  DataElementQuery  ::= DataObjectRef '/' DataElementRef
    '?' PropertyName
3 DataValueQuery    ::= DataObjectRef '/' DataElementRef
    '/' DataValueRef ('?' PropertyName)?
  DataObjectRef     ::= ReferenceName
```

```
5  DataElementRef    ::= ReferenceName
   DataValueRef      ::= '/value' ('[' Index ']')?
7  PropertyName      ::= 'size' | 'url'
   Index             ::= [1-9] Number* | 'first' | 'last'
9  Number            ::= [0-9]
   ReferenceName     ::= StartCharacter Character*
11 StartCharacter    ::= [a-zA-Z]
   Character         ::= StartCharacter | Number | '_' |
       '-' | '[]' | '#'
```

Listing 6.1: EBNF-like grammar of the TraDE Query Language (TQL).

A *DataValueQuery* enables modelers to refer to an actual data value associated to an instance of a data element (cf. line 5-6 in Listing 6.1). Therefore, again a data object and a data element are referenced followed by a reference to a data value (*DataValueRef*) and an optional property selection as shown in Listing 6.1 (line 9). A *DataValueRef* always starts with a `/value` string which specifies that the remaining part of the query is evaluated against a data value. Furthermore, an optional `index` value can be specified by adding a number larger than one or the fixed index values `first` or `last` surrounded by square brackets (cf. line 11 in Listing 6.1). The index only has an effect on the query evaluation if the referenced data element is defined as part of a multi-instance data object, i. e., has probably more than one associated data value during runtime. A query like $\texttt{\$dataObject}_x\texttt{/dataElement}_i\texttt{/value[last]}$ returns the data of the last data value associated to an instance of dataElement_i of dataObject_x. The use of the property selector introduced in Listing 6.1 is similar as for the *DataElementQuery*. The *url* property will return an URL referring to the data value and the *size* property will by default always return one for data values.

6.2.2 Formal Model for TraDE Data Transformations

This section presents the formal model for the data transformation (DT) modeling element introduced above by extending our formal model for data-aware service choreographies introduced in Chapter 4.

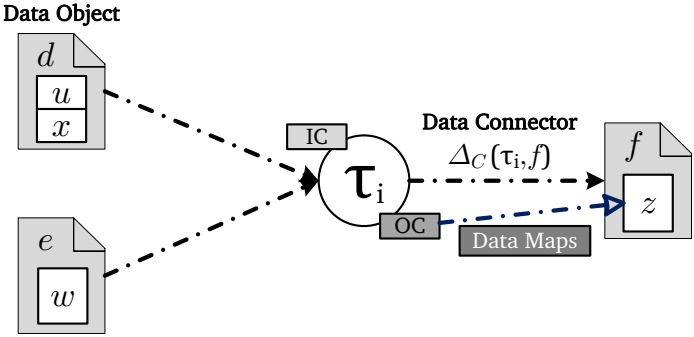

Figure 6.5: Visual representation of a data transformation and corresponding data connectors to source and target data objects.

From a conceptual viewpoint, we introduce data transformations within our formal model as a special kind of *data-driven activity* which is not bound nor part of the control flow of the choreography model. Instead, data transformations are only connected to data objects via corresponding data connectors and the referenced transformation logic is triggered as part of the data flow conducted by the TraDE Middleware during choreography runtime. Following this approach, a data transformation also has an associated implementation (cf. Definition 4.9) as well as input and output data containers (cf. Definition 4.3) specifying the inputs and outputs of the associated data transformation implementation. Therefore, the required extensions of the related definitions introduced in Chapter 4 together with new definitions for data transformations are introduced in the following. This allows us to reflect all the required properties of the DT modeling extension depicted in Figure 6.4 based on our formal model. Before required extensions of existing definitions are discussed, the notion of data transformations is introduced in Definition 6.1. The visual representation of a data transformation as part of our graphical notation is shown in Figure 6.5.

Based on this definition, we are now able to associate *DT Implementations* as outlined in Figure 6.4 to a modeled data transformation. Therefore, Definition 6.2 introduces a data transformation implementation map which

is aligned to the activity implementation map introduced in Definition 4.9.

Definition 6.1 (Data Transformations)
Let V be the set of data elements defined within a choreography model $\mathscr{C}$, let $\mathcal{C}_T \subseteq \mathcal{C}$ be the set of trigger conditions, and let $\mathscr{A}$ be the set of activation modes, we denote τ as the set of all data transformations

$$\tau \subseteq \mathcal{C}_T \times \mathscr{A}$$

where

1. *a trigger condition $t \in \mathcal{C}_T$ is considered as a Boolean function in its input container $\iota_C(t) \subseteq V$:*

$$t : \underset{v \in \iota_C(t)}{\times} DOM(v) \rightarrow \{0,1\}$$

   to define a Trigger Condition as depicted in Figure 6.4,

2. *and $\mathscr{A} := \{on\text{-}read, on\text{-}write\}$ allows to specify the Activation Mode shown in Figure 6.4.* □

To represent the remaining properties shown in Figure 6.4 for data transformations, namely specifying *Input Parameters* and their values, *Input/Output Mappings* as well as connecting data objects as source and target of a data transformation, corresponding extensions of the related definitions provided in Chapter 4 are discussed in the following. To avoid redundancy, we try to keep the updated definitions as short as possible and only present the extended definition as well as potentially new restrictions or specifics introduced through the extension.

Definition 6.2 (Data Transformation Implementations)
Let $\mathcal{E}_\tau$ denote the set of all possible implementations of all data trans-

formations, that means a member of $\mathcal{E}_\tau$ can be a program, script, web service or even build-in data transformation functionality of a process engine. The map $\Psi_\tau : \tau \to \mathcal{E}_\tau$ associates with each data transformation T its activity implementation $\Psi_\tau(T)$. A data transformation implementation itself is perceived as a map:

$$\Psi_\tau(T) : \underset{v \in \iota_C(T)}{\times} DOM(v) \to \underset{v \in o_C(T)}{\times} DOM(v) \qquad \square$$

As already outlined above, the inputs and outputs of data transformations and their underlying implementations will be specified using data containers as shown in Figure 6.5. Therefore, we have to extend the set $\mathcal{H}_C$ introduced in Section 4.2.1.3 accordingly, which is used as a basis for the definition of data containers in Definition 4.3. That means, data transformations are added to the set, so that $\mathcal{H}_C = N \cup \tau \cup \mathscr{P}(N) \cup \mathfrak{C}$. This directly allows us to specify input and output data containers for data transformations based on Definition 4.3 without requiring any extensions of the definition itself.

To enable the specification of *Input Parameters* and related input values for a data transformation which cannot be provided through cross-partner data objects directly as outlined in Section 6.2, Definition 6.3 is introduced. This enables modelers to specify data values for specific data elements within a data transformation's input container, i. e., input parameters, in form of constant values or a TraDE query using the above introduced TQL.

Definition 6.3 (Input Parameter Map)
Let V be the set of data elements defined within a choreography model $\mathscr{C}$, let $T \in \tau$ be a data transformation, let $v \in \iota_C(T) \subseteq V$ be a data element within the input container of T, let $DOM(v)$ be the domain of valid values of v, and let C_{TQL} be a finite set of TQL expressions according

to Listing 6.1. The map

$$\chi_\tau : \tau \times V \rightarrow \left(\bigcup_{v \in V} DOM(v) \cup \mathcal{C}_{TQL} \right)$$

satisfying the conditions

1. $\forall T \in \tau, \forall v \in V : \chi_\tau(T, v) \neq \emptyset \Rightarrow v \in \iota_C(T)$,
2. $\forall T \in \tau, \forall v \in \iota_C(T) : \chi_\tau(T, v) \in (DOM(v) \cup \mathcal{C}_{TQL})$,
3. $\forall T \in \tau, \forall v \in \iota_C(T) : \chi_\tau(T, v) = x \wedge x \in \mathcal{C}_{TQL} \Rightarrow x$ *evaluates to a value* $y \in DOM(v)$

is called an input parameter map. □

Condition 1 of Definition 6.3 enforces that an input parameter map for a data transformation can only refer to data elements being defined as part of the transformation's input container. Conditions 2 and 3 enforce that the domain of the specified constant value or the evaluated TQL expression is identical to the one specified for the targeted data element. For example, if an input parameter map $\chi_\tau(T, v) = x$ is specified, and the data element is of type integer, i. e., $DOM(v) = \text{domain(Integer)}$, then x has to be either an integer value or the specified TQL expression has to evaluate to an integer value. The TQL expression $\texttt{\$dataObject}_\texttt{x}\texttt{/dataElement}_\texttt{i}\texttt{?size}$ introduced above as an example, fulfills this requirement by returning the number of data values associated to a data element of a multi-instance data object during runtime.

Finally, the specification of data connectors between data objects and data transformations will be supported by extending Definition 4.18 in the following. Definition 6.4 presents the extended data connector map. The association of data containers to data transformations as well as the extended data connector map realize the *Input/Output Mappings* shown in Figure 6.4 and enable to specify the related data flow to finally support data transformations within Choreography Model Graphs (CM-Graphs) as depicted in Figure 6.5.

Definition 6.4 (Data Connector Map - extended)
Let $A \in N \cup \tau$ be an activity or data transformation, let $B \in N \cup \mathcal{C} \cup \tau$ be an activity, a predicate or data transformation, and let $d, e \in \mathcal{D}(V)$ be data objects. The map

$$\Delta_C : (N \cup \tau \cup \mathcal{D}(V)) \times (N \cup \mathcal{C} \cup \tau \cup \mathcal{D}(V)) \rightarrow \bigcup_{\substack{A \in N \cup \tau,\ B \in N \cup \mathcal{C} \cup \tau,\\ d,e \in \mathcal{D}(V)}} (\wp(o_C(A) \times \iota_C(B)) \cup \wp(d \times \iota_C(B)) \cup \wp(o_C(A) \times d) \cup \wp(d \times e))$$

satisfying the conditions

1. $\forall d \in \mathcal{D}(V)\ \forall B \in (N \cup \mathcal{C} \cup \tau) : \Delta_C(d, B) \in \wp(d \times \iota_C(B))$,
2. $\forall A \in N \cup \tau\ \forall d \in \mathcal{D}(V) : \Delta_C(A, d) \in \wp(o_C(A) \times d)$,
3. $\forall A_2 \in N \cup \tau \cup \mathcal{D}(V) : (x, z), (y, z) \in \bigcup_{A_1 \in N \cup \tau \cup \mathcal{D}(V)} \Delta_C(A_1, A_2) \Rightarrow x = y$,
4. $\forall A \in (N \cup \tau \cup \mathcal{D}(V))\ \forall T \in \tau : \Delta_C(A, T) \neq \emptyset \Rightarrow A \in \mathcal{D}(V)$,
5. $\forall T \in \tau\ \forall B \in (N \cup \mathcal{C} \cup \tau \cup \mathcal{D}(V)) : \Delta_C(T, B) \neq \emptyset \Rightarrow B \in \mathcal{D}(V)$

is called a data connector map. □

Conditions 1-3 are just extended versions of the conditions presented in Definition 4.18 taking data transformations into account. The conditions 4 and 5 guarantee that data connectors ending or originating at a data transformation are always only connected to data objects and no other modeling elements. This restriction is required to decouple the data transformations from the participants control flow as outlined in Section 6.2. Since all the sources and targets of a data transformation are either cross-partner data objects or specified via an input parameter map, and are therefore not bound to a participant and its runtime environment, data transformations can be executed as part of the cross-partner data flow managed by the TraDE Middleware in an independent and flexible manner.

The introduced and extended definitions for data transformations have to be reflected at the level of CM-Graphs as summarized in Definition 6.5.

Definition 6.5 (CM-Graphs - extended)
A tuple

$$G_C = (V, \iota_C, o_C, \mathscr{M}(V), CS(V), \mathscr{D}(V), \mu_{\mathscr{D}}, \rho_{\mathscr{D}}, N, \Psi, \mathscr{P}(N), \mu_{\mathscr{P}}, E, \mathcal{C}, \Delta_{\mathscr{M}}, \overrightarrow{\Delta_{\mathscr{M}}}, \Delta_C, \overrightarrow{\Delta_C}, \Delta_{CP}, \Delta_{CS}, \tau, \Psi_\tau, \chi_\tau)$$

is called a data-aware choreography model graph, or CM-Graph for short, representing a choreography model $\mathscr{C} \in \mathfrak{C}$. □

To reflect the above formalized data transformations (cf. Definition 6.1) as part of the Choreography Data Dependency Graph (CDDG) of a choreography model $\mathscr{C} \in \mathfrak{C}$ introduced in Section 4.3, we have to extend the node set $\mathcal{N}$ of $G_{C_{DDG}}$. In addition to data objects, activities, conditions, participants or an overall choreography model, data transformations are also valid *dependency nodes*. Since they actually process data, they can be added to the set of *data processors* as summarized in Definition 6.6.

Definition 6.6 (Data Dependency Nodes - extended)
Based on Definition 4.24, let $\mathscr{C} \in \mathfrak{C}$ *be a choreography model, let* $\mathscr{D}(V_{\mathscr{D}})$ *be the set of all data objects, let* N *be the set of all activities, let* $\mathcal{C}$ *be the set of all conditions, let* τ *be the set of all data transformations, and let* $\mathscr{P}(N)$ *be the set of all participants of* $\mathscr{C}$*, we define*

- $\Pi = N \cup \mathcal{C} \cup \tau \cup \mathscr{P}(N) \cup \{\mathscr{C}\}$ *as the set of all data consumers and producers of a choreography model* $\mathscr{C} \in \mathfrak{C}$*, called data processors, and*
- $\mathcal{N} = \mathscr{D}(V_{\mathscr{D}}) \cup \Pi$ *as the set of all nodes being source or target of data flow within* $\mathscr{C}$*, which we call data dependency nodes.* □

In addition to the node set of a CDDG also its edge set has to be extended to reflect the data connectors between data objects and data transformations as summarized in Definition 6.7.

Definition 6.7 (Data Dependency Edges - extended)
The set $E_{C_{DDG}} \subseteq \mathcal{N} \times \mathcal{N} \times \hat{V} \times \hat{V}$ *is called the set of data dependency edges of a choreography data dependency graph* $G_{C_{DDG}}$*. For a data dependency edge* $(A, B, s, t) \in E_{C_{DDG}}$*,* $s \in \hat{V}$ *is called source data element and* $t \in \hat{V}$ *is called target data element of the dependency edge.*
Each data dependency edge denotes that during runtime the data value of source data element s *of source node* A *needs to be available at target data element* t *of target node* B*.*
Therefore, $(A, B, s, t) \in E_{C_{DDG}} :\Leftrightarrow$

- $A \in \mathscr{D}(V) \wedge B \in (N \cup \mathcal{C} \cup \tau) \wedge (s, t) \in \Delta_C(A, B)$*, or*
- $A \in (N \cup \tau) \wedge B \in \mathscr{D}(V) \wedge (s, t) \in \Delta_C(A, B)$*, or*
- $A \in \mathscr{D}(V) \wedge B \in \mathscr{D}(V) \wedge (s, t) \in \Delta_C(A, B)$*, or*
- $A \in \mathscr{D}(V) \wedge B \in (\mathscr{P}(N) \cup \{\mathscr{C}\}) \wedge (s, t) \in \overrightarrow{\Delta_C}(A, B)$*, or*
- $A \in (\mathscr{P}(N) \cup \{\mathscr{C}\}) \wedge B \in \mathscr{D}(V) \wedge (s, t) \in \overrightarrow{\Delta_C}(A, B)$. □

Finally, the CDDG generation algorithm introduced in Algorithm 4.2 has to be extended to extract all data dependencies between data objects and data transformations from a CM-Graph.

Based on the generated CDDG, the modeled data transformations can then be triggered as part of the cross-partner data flow at the TraDE Middleware which is discussed in more detail in the following section.

Algorithm 6.1 Extended version of Algorithm 4.2 taking data transformations into account

1: **procedure** GENERATECDDG(G_C)
2: $C_{DM}(G_C) \leftarrow$ GENERATECDM(G_C)
3: $V_\Pi, E_{C_{DDG}} \leftarrow \emptyset$ ▷ Initialize required sets
4: $\Pi_{read} := \{A \in (\overbrace{\pi_9(G_C)}^{N} \cup \overbrace{\pi_{14}(G_C)}^{\mathcal{C}} \cup \overbrace{\pi_{21}(G_C)}^{\tau} \cup \overbrace{\pi_{11}(G_C)}^{\mathscr{P}(N)} \cup \{\mathscr{C}\})$
$\mid \exists d \in \overbrace{\pi_6(G_C)}^{\mathscr{D}(V)} : \Delta_C(d,A) \cup \overrightarrow{\Delta_C}(d,A) \neq \emptyset\}$
▷ Data processors reading from a data object
5: $\Pi_{write} := \{B \in (\pi_9(G_C) \cup \pi_{21}(G_C) \cup \pi_{11}(G_C) \cup \{\mathscr{C}\})$
$\mid \exists d \in \pi_6(G_C) : \Delta_C(B,d) \cup \overrightarrow{\Delta_C}(B,d) \neq \emptyset\}$
▷ Data processors writing to a data object
6: $\Delta_C = \pi_{17}(G_C)$ ▷ Data connector map of G_C
7: $\overrightarrow{\Delta_C} = \pi_{18}(G_C)$ ▷ Choreography data connector map of G_C
8: **for all** $A \in \Pi_{read}$; $d \in \pi_6(G_C)$ **do** ▷ Extract *read* data dependencies
9: **for all** $(u,x) \in \Delta_C(d,A) \cup \overrightarrow{\Delta_C}(d,A)$ **do**
10: $V_\Pi \leftarrow V_\Pi \cup \{x\}$ ▷ Add data elements of data processors to V_Π
11: $E_{C_{DDG}} \leftarrow E_{C_{DDG}} \cup \{(d,A,u,x)\}$ ▷ Add a data dependency edge
12: **end for**
13: **end for**
14: **for all** $B \in \Pi_{write}$; $d \in \pi_6(G_C)$ **do** ▷ Extract *write* data dependencies
15: **for all** $(y,v) \in \Delta_C(B,d) \cup \overrightarrow{\Delta_C}(B,d)$ **do**
16: $V_\Pi \leftarrow V_\Pi \cup \{y\}$
17: $E_{C_{DDG}} \leftarrow E_{C_{DDG}} \cup \{(B,d,y,v)\}$
18: **end for**
19: **end for**
20: **for all** $d,e \in \pi_6(G_C)$ **do** ▷ Extract *read/write* data dependencies
21: **for all** $(w,z) \in \Delta_C(d,e)$ **do**
22: $E_{C_{DDG}} \leftarrow E_{C_{DDG}} \cup \{(d,e,w,z)\}$
23: **end for**
24: **end for**
25: $\Pi = \Pi_{read} \cup \Pi_{write}$ ▷ Set of data processors
26: $G_{C_{DDG}} \leftarrow (\Pi, E_{C_{DDG}}, V_\Pi, C_{DM}(G_C))$
27: **return** $G_{C_{DDG}}$ ▷ The choreography data dependency graph
28: **end procedure**

6.3 Transparent Execution of Data Transformations

Before we describe how the modeled data transformations are actually executed during choreography runtime, we first have a short look on the deployment artifacts of a data-aware choreography and how they are distributed and utilized by the different components of the overall TraDE ecosystem shown in Figure 6.6. As described in Section 6.1.2 the actual transformation logic of a DT Implementation is provided in form of a DT Bundle which comprises one or more transformations, an optional set of dependencies and schema definitions, as well as a DT Unit specification file. Defined DT Bundles are published to the DT Integration Middleware to make them available within the TraDE ecosystem. The TraDE data transformation modeling extension introduced in Section 6.2 then 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 QName (see Section 6.1.2). This allows the TraDE Middleware to trigger a new transformation task at the DT Integration Middleware in a generic task-based manner as means to conduct a modeled transformation by executing the referenced transformation implementation of a 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*. Following the public-to-private approach [AW01], the resulting data-aware choreography models are then transformed into a collection of private process models to enable the execution of the choreography (see Section 4.5). During the transformation process, the cross-partner data objects will be translated to respective staging elements at the level of the private process models as discussed in Section 5.1. The resulting private process models can then be manually refined by adding corresponding internal logic for each participant. The refined private process models are finally packaged together with related files, e. g., process engine deployment descriptors, or interface

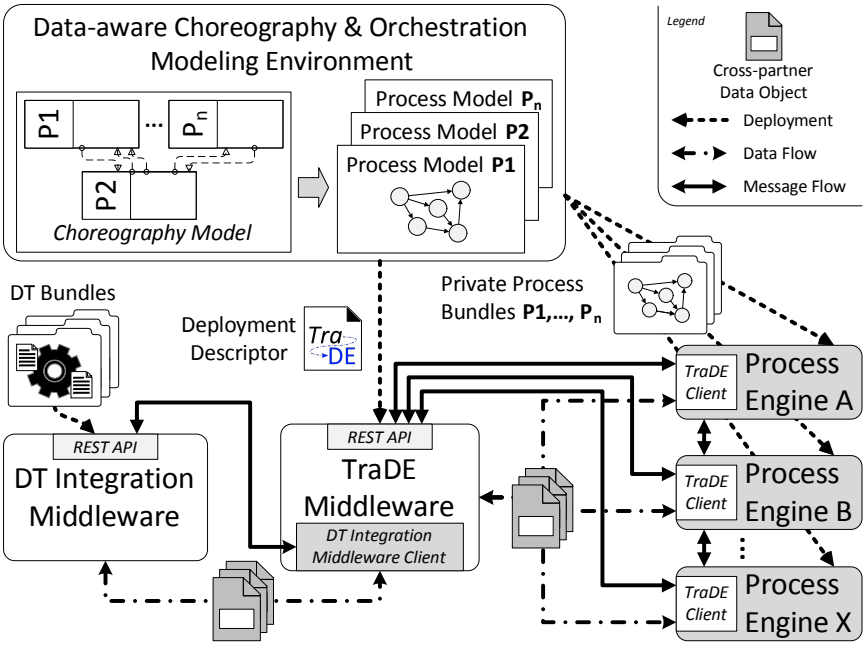

Figure 6.6: Integrated system architecture and deployment artifacts of the TraDE ecosystem [HBLY18].

and schema definitions, as *Private Process Bundles* for the deployment on respective *Process Engines* as depicted in Figure 6.6. Furthermore, the CDDG introduced in Section 4.3.2 including the choreography data model, i. e., all modeled cross-partner data objects, as well as all data dependencies originating from the modeled cross-partner data flows, are exported to a *TraDE Deployment Descriptor* file. This also includes the modeled data transformations. They are exported together with their specified properties to the deployment descriptor. This comprises all the information outlined in Section 6.2, i. e., the QName of a DT Implementation available at the DT Integration Middleware and mappings of cross-partner data objects to inputs and outputs of such a DT Implementation as well as input partameters and an optional trigger condition and activation mode. The exported deployment

descriptor is uploaded to the TraDE Middleware to make all these information accessible there. On upload, the deployment descriptor is compiled into the middleware's internal metamodel presented in Section 5.2. As a result, the middleware provides and exposes all specified cross-partner data objects and data elements as resources through its REST API as described in Chapter 5 as well as supports the triggering of specified data transformations when conducting the cross-partner data flow of a choreography model. This separation of concerns, i. e., the participants' business logic is specified in private process models and the transformation logic is referenced within the 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 at the level of the private process models of a choreography.

Since the realization of the data exchange across participants, i. e., between their private process models, through integrating the TraDE Middleware with the underlying process engines is described already in detail in Chapter 5, in the following, our focus is on the interactions between the DT Integration Middleware and the TraDE Middleware to conduct the modeled data transformations. Therefore, we first want to have a closer look on the internal representations and behavior of the TraDE Middleware. The TraDE Middleware comes with its own internal, choreography language independent metamodel as presented in Chapter 5. For the sake of conciseness, we only provide a brief recap of it. All cross-partner data objects and their contained data elements are represented through corresponding *CrossPartnerDataObject* and *DataElement* entities at the TraDE Middleware. In addition to the model perspective, the metamodel of the TraDE Middle-

ware provides further entities to represent instances of cross-partner data objects and data elements to reflect the runtime perspective of data-aware choreographies, i. e., manage the data of choreography instances. For each choreography instance corresponding *CrossPartnerDataObjectInstance* and *DataElementInstance* entities will be created at the TraDE Middleware with associated *CorrelationProperty* entities that enable to uniquely identify to which choreography instance the data object and data element instances belong. The actual data is provided through *DataValue* entities which are referenced by one or more *DataElementInstance* entities.

The TraDE Middleware provides an event model for each of the introduced entity types, i. e., a life cycle with states and transitions, introduced in Section 5.3. This allows firing an event whenever an entity changes its state, e. g., a DataValue is *initialized*, i. e., an initial value is set. Based on these event models, the TraDE Middleware supports an event-based mechanism to trigger actions on respective events. This event-based mechanism 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. Based on the example shown on the right of Figure 6.1, this means that transformation T_1 is triggered as soon as data element *E* of cross-partner data object *input* is initialized or whenever it is modified. While this is the default trigger behavior of the TraDE Middleware, this can be influenced by the specification of more fine-grained *trigger conditions* under which a data transformation should be executed.

The invocation of the respective DT Bundle's transformation itself is realized in a straightforward manner. First, all required information are collected to execute a specified DT Bundle's transformation by sending an invocation request to the DT Integration Middleware that triggers the task-based execution of the transformation. This comprises the resolution of corresponding DataValue entities for a specific choreography instance that hold the input data for the transformation. For the example depicted in Figure 6.1, this means that the TraDE Middleware has to first identify the DataElementInstance entity of data element *E* related to the running choreography instance

based on its correlation properties. Based on that, the DataValue entity associated to the resulting data element instance can be resolved to get the actual data to transform. Instead of passing the data by value within the invocation request to the DT Integration Middleware, the data is passed by reference. Therefore, the URL pointing to the respective resource exposing the required DataValue entity at the TraDE Middleware's REST API is added to the request for each transformation input based on the specified Input Mappings. The same applies for the transformation outputs. Instead of retrieving the transformation output in a response message, a DataValue resource URL is added to the invocation request for each transformation output based on the specified Output Mappings. After all inputs are resolved the specified DT Bundle's transformation is invoked by sending the prepared invocation request to the DT Integration Middleware. As soon as the transformation is completed, the DT Integration Middleware then uploads all results to the TraDE Middleware by pushing them to the DataValue resources specified as output in form of respective URLs in the invocation request, making the data available at the TraDE Middleware.

As outlined above, by default a data transformation is triggered whenever data is written to a DataValue associated to one of the transformation's input cross-partner data objects. The *trigger condition* and *activation mode* introduced in Section 6.2 enable modelers to influence the underlying behavior of the TraDE Middleware during choreography execution. By default, the TraDE Middleware applies the *on-write* activation mode. There, 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 triggers a DT Bundle's transformation implementation at the DT Integration Middleware. 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 requested. For example, data transformation T_1 in Figure 6.1 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. However, this is not in the focus of this work and therefore not further detailed.

Chapter 7

System Architecture and Implementation

To prove the technical feasibility of the concepts introduced in this work, we prototypically implemented them as part of our TraDE ecosystem. Within this chapter, we therefore describe the underlying system architecture of the TraDE ecosystem and the implementation of its components.

Section 7.1 presents the system architecture and provides an overview of the ecosystem components as well as their interrelations. Based on that, the prototypical implementation of the individual components is shortly summarized in Section 7.2.

7.1 System Architecture of the TraDE Ecosystem

Figure 7.1 shows the system architecture of the complete TraDE ecosystem. The goal of the TraDE ecosystem and its individual components is to provide an end-to-end support for the TraDE concepts introduced in this work, namely transparent data exchange via cross-partner data objects and cross-partner

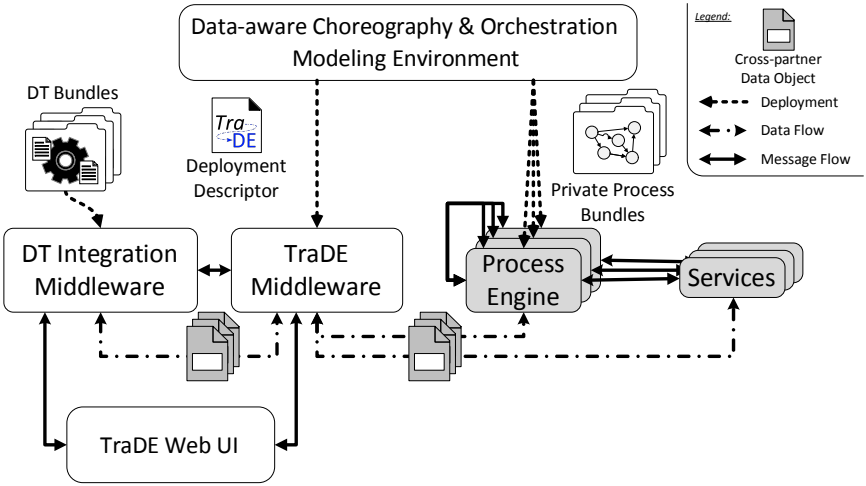

Figure 7.1: System Architecture of the complete TraDE ecosystem and its components, based on [HBKL17; HBLY18].

data flows as well as transparent data transformations. Therefore, the ecosystem components support the phases of the data-aware choreography management life cycle introduced in Section 3.3.

The *Data-aware Choreography & Orchestration Modeling Environment* at the top of Figure 7.1 implements the modeling, refinement and partially the deployment phase of the data-aware choreography management life cycle introduced in Section 3.3. Therefore, new data-aware choreography models can be specified and automatically transformed to a set of *private process models* and a *TraDE deployment descriptor*. The generated process models directly reflect all data-related aspects specified at the level of the choreography model as a result of the transformation process as defined in Section 4.5.2. With the help of the orchestration-part of the modeling environment, modelers can then refine the generated process models to make them executable and directly package and deploy them from the modeling environment to a connected *process engine* in form of *private process bundles*. The TraDE deployment descriptor is an eXtensible Markup Language (XML)

document which represents all defined cross-partner data objects and related data flows and therefore reflects the Choreography Data Model (CDM) and Choreography Data Dependency Graph (CDDG) introduced in Section 4.3 in a machine-readable format. The Data-aware Choreography & Orchestration Modeling Environment provides corresponding functionality to automatically generate such TraDE deployment descriptors from a specified data-aware choreography model as defined in Section 4.5.1. Furthermore, the modeling environment allows to directly deploy such descriptor files to the *TraDE Middleware* using its Representational State Transfer (REST) API.

As described in detail in Chapter 6, defined *DT Bundles* providing data transformation logic which can be referenced in a data-aware choreography model using our TraDE modeling extensions, can be deployed to the *DT Integration Middleware* to make them invocable from the TraDE Middleware to conduct the modeled data transformations on cross-partner data objects. Therefore, the TraDE Middleware and the DT Integration Middleware are integrated to exchange the cross-partner data objects which have to be transformed in an efficient manner. The TraDE Middleware triggers a data transformation by sending an invocation request with all required information to the DT Integration Middleware that starts the task-based execution of the transformation using the specified DT Bundle. Instead of passing the actual data within the invocation request to the DT Integration Middleware, a Uniform Resource Locator (URL) is added for each required transformation input which points to the respective cross-partner data object at the TraDE Middleware. The same applies for the transformation results, the invocation request contains one or more predefined URLs reflecting the cross-partner data object(s) at the TraDE Middleware to push the result data to.

On the other side, the TraDE Middleware is integrated with the process engines that are responsible for executing the private process models implementing an underlying data-aware choreography. There the process engines and the TraDE Middleware together realize and conduct the modeled cross-partner data flows by pushing and pulling data to and from the defined cross-partner data objects managed by the TraDE Middleware. In addition, services being invoked by the private process models can be also connected

to the TraDE Middleware to improve the data exchange by passing references to cross-partner data objects available at the TraDE Middleware instead of embedding potentially large data values in requests and responses between the process engines and invoked services.

To further support users in providing and uploading their data into the ecosystem and provide some initial monitoring support, we implemented a *TraDE Web UI*. The web UI is connected to the Application Programming Interfaces (APIs) of the TraDE Middleware and the DT Integration Middleware and allows, e. g., to manually upload TraDE deployment descriptors and DT bundles, inspect registered deployment descriptors as well as their defined data objects and data elements or to upload, download or even preview the data values of available cross-partner data objects. This feature is especially interesting in the context of eScience simulations since it allows to inspect intermediary simulation results while the simulation is still running and therefore allows to take appropriate action, e. g., by utilizing *Model-as-you-go* concepts [Son16; Wei18]. For example, changing data of a cross-partner data object and rewinding the simulation execution to re-execute parts of it using the updated data.

7.2 Prototypical Implementation

For the modeling of data-aware choreographies based on our TraDE concepts, the choreography modeling language BPEL4Chor [DKLW09; Kop16] is used and extended accordingly based on our formal model for data-aware choreographies introduced in Chapter 4. However, BPMN 2.0 collaboration models with respective TraDE modeling extensions can be also used instead of BPEL4Chor as shown by the different data-aware choreography model examples throughout the document. Since we want to build on existing works and prototypes, we decided to use BPEL4Chor for choreography modeling and the Business Process Execution Language (BPEL) for the modeling and execution of the resulting private process models conducting the overall choreography model. Based on the system architecture of the TraDE

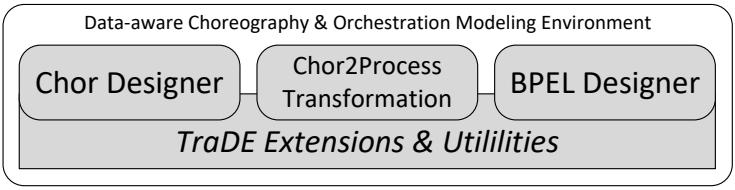

Figure 7.2: Overview of the TraDE Modeling Environment

ecosystem introduced in the previous section, we want to provide a short overview of the prototypical implementation of each individual component in the following.

7.2.1 Data-aware Choreography & Orchestration Modeling Environment

The Data-aware Choreography & Orchestration Modeling Environment shown in Figure 7.2 builds on established tools and prototypes developed and used within several projects at the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart together with our TraDE extensions. To enable the modeling of data-aware choreographies with our TraDE concepts, we extended the *Chor Designer* [WAG+13] — an Eclipse-based BPEL4Chor modeling tool.

Figure 7.3 shows a screenshot of the resulting TraDE-aware Chor Designer with a data-aware choreography model. The palette on the right provides all introduced TraDE modeling constructs such as cross-partner data objects (Data Object and Data Element) as well as cross-partner data flows (Data Connector). Furthermore, we added a *Data Dependency Graph Viewer* to automatically visualize the data dependencies while modeling data flows within the choreography model and provide further utilities to support modelers, e. g., automatic generation of a TraDE deployment descriptor of a data-aware choreography model.

The transformation of data-aware BPEL4Chor choreography models to a collection of BPEL process models is also provided through the modeling environment by a TraDE-aware version of the *Chor2Process Transformation*

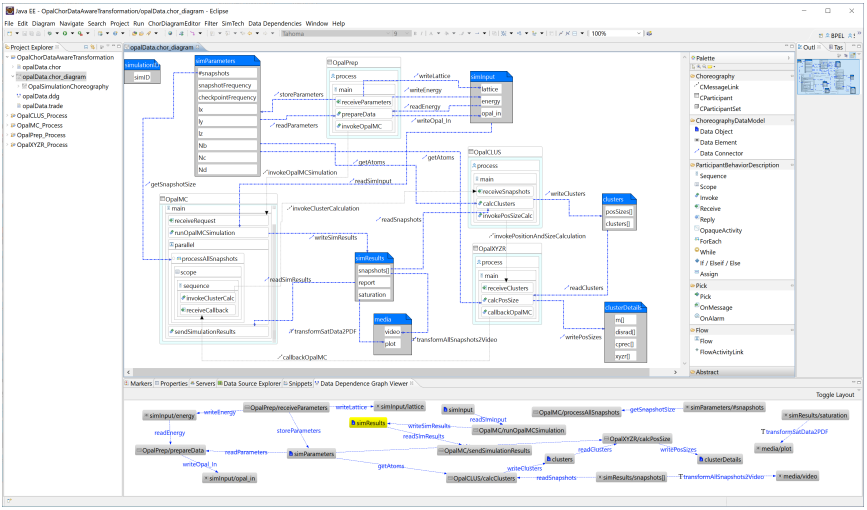

Figure 7.3: Overview of the TraDE ChorDesigner.

module shown in Figure 7.2 which is based on the work of Reimann, Kopp, et al. [RK+08]. For the prototypical implementation we followed the simple mapping of process data containers and cross-partner data objects as presented in Section 3.2. Therefore, the transformation generates a set of BPEL variables representing corresponding cross-partner data objects at the level of the process models based on the CDDG. This means, for each cross-partner data object read or written by the participant represented through the process model, a respective BPEL variable with a reference to the cross-partner data object it represents is added. These references are specified through corresponding TraDE annotations which provide the means to specify the data element of a cross-partner data object the underlying variable represents. The TraDE annotations are added to respective BPEL variable elements by using BPEL's extensibility feature.

For process modeling and especially the refinement of the generated private process models the modeling environment provides an extended

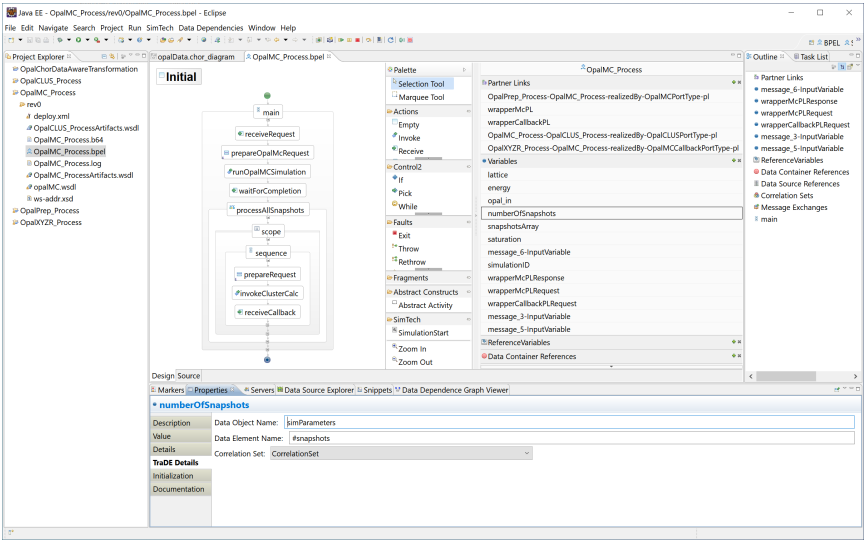

Figure 7.4: Overview of the TraDE BPEL Designer.

version of the Eclipse *BPEL Designer*[1]. There each of the generated process models can be refined to an executable BPEL process model, e. g., by adding additional activities from the palette or provide missing technical details for the generated activities. Furthermore, the mapping of BPEL variables to cross-partner data objects created during transformation, as outlined above, can be inspected and also missing or further details can be specified as shown in Figure 7.4. For example, a BPEL correlation set can be associated which is used during runtime to correlate the right data object instances at the TraDE Middleware to a process instance execution, if multiple instances of the choreography model will be run in parallel.

The complete Data-aware Choreography & Orchestration Modeling Environment is available on GitHub[2] in form of an Eclipse Update Site containing all Eclipse plug-ins for installing the TraDE-aware ChorDesigner and BPEL

[1] BPEL Designer: `https://www.eclipse.org/bpel/`
[2] TraDE: `https://github.com/traDE4chor/trade-chordesigner`

Designer.

7.2.2 TraDE Middleware

The main component of the TraDE ecosystem is the TraDE Middleware introduced in detail in Chapter 5. It is responsible for providing cross-partner data objects defined during choreography modeling and to execute the specified cross-partner data flows together with the connected process engines. The TraDE Middleware itself is implemented as a Java-based web application that exposes its functionality through a REST API which is specified using OpenAPI[1] and implemented with the *Jersey* RESTful Web Services framework[2]. The TraDE internal representations and the actual data processed within the choreographies, can be persisted either using MongoDB or the local file system. To support the event-based triggering of DT Bundles, the TraDE Middleware is extended with corresponding functionality implemented using Apache Camel[3] to send requests to the REST API of the DT Integration Middleware. For basic concurrency control a simple pessimistic locking approach is applied to block concurrent requests on cross-partner data objects already at the level of the process engines.

The TraDE Middleware is accessible as open source on GitHub[4].

7.2.3 TraDE-aware Process Engine

As Process Engine an extended version of the open source BPEL engine Apache Orchestration Director Engine (ODE)[5] is used. To enable the execution of cross-partner data flows, i. e., pushing and pulling data to or from cross-partner data objects at the TraDE Middleware, the middleware has to be integrated into Apache ODE. Therefore, we followed the *TraDE-aware integration* approach introduced in Section 5.5 by adding and integrating a

[1] OpenAPI Initiative: `https://www.openapis.org/`
[2] Eclipse Jersey: `https://eclipse-ee4j.github.io/jersey/`
[3] Apache Camel: `https://camel.apache.org/`
[4] TraDE: `https://github.com/traDE4chor/trade-core`
[5] Apache ODE: `https://ode.apache.org/`

respective REST API client to Apache ODE. Based on this client, Apache ODE is now able to interact with the TraDE Middleware. Moreover, the logic for reading and writing data from or to BPEL variables is extended to support the reading and writing of cross-partner data objects using the specified mappings of BPEL variables to cross-partner data objects created during the choreography to process model transformation. This means, whenever Apache ODE tries to read or write data from or to a BPEL variable, it has to be checked if the variable refers to a cross-partner data object or not. If the latter is the case, a corresponding request to the REST API of the TraDE Middleware is sent instead of using the default variable handling mechanism of Apache ODE. Therefore, Apache ODE is extended to support the introduced TraDE annotations at the level of BPEL variables, i. e., enable their identification and interpretation to support the reading and writing of data to the referenced cross-partner data objects. The TraDE-aware Apache ODE process engine is provided as open source on GitHub[1].

7.2.4 DT Integration Middleware

The DT Integration Middleware, introduced in detail in Chapter 6, is implemented as a web application using Python Flask[2] and also exposes its functionality through a REST API. The REST API is again specified using OpenAPI. For storing DT Bundles a combination of MongoDB and the local 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 and 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 might be supported using TOSCA [TOSCA]

[1] TraDE-aware Apache ODE: `https://github.com/traDE4chor/ode`
[2] Pallets Flask: `https://palletsprojects.com/p/flask/`

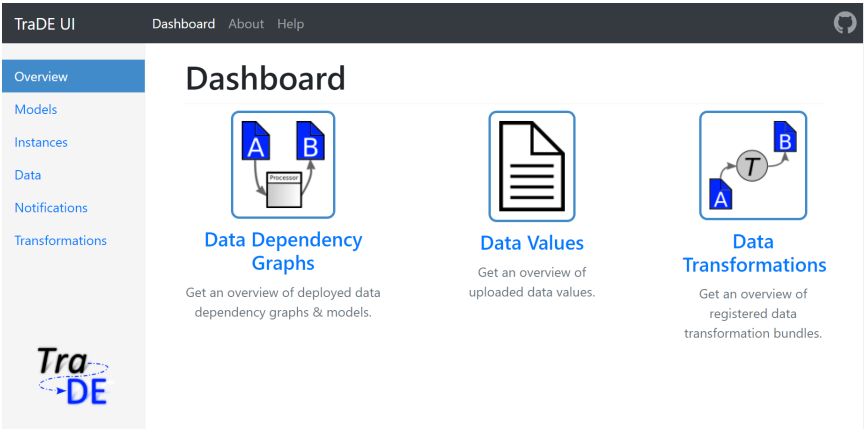

Figure 7.5: Entry page of the TraDE Web UI.

and OpenTOSCA [Bin+13] as provisioning layer in the future.

The DT Integration Middleware is accessible as open source on GitHub[1].

7.2.5 TraDE Web UI

To support the interaction of human users with both the TraDE Middleware as well as the DT Integration Middleware a TraDE Web UI is provided. The web UI is implemented as a single-page web application using the Angular[2] framework. It connects to the REST APIs of both middleware components and allows to query and inspect any available data as well as to manually upload data or register new artifacts, e. g., new TraDE deployment descriptors or DT bundles. Figure 7.5 shows a screenshot of the entry page of the web UI. The menu on the left allows to navigate to different views comprising all available choreography data *models* and their content (data objects and data elements) registered through uploaded TraDE deployment descriptors, *instances* of data objects and data elements created during choreography executions, as well as actual *data* reflected by data values associated to a

[1] DT Middleware: `https://github.com/traDE4chor/hdtapps-prototype`
[2] Angular: `https://angular.io/`

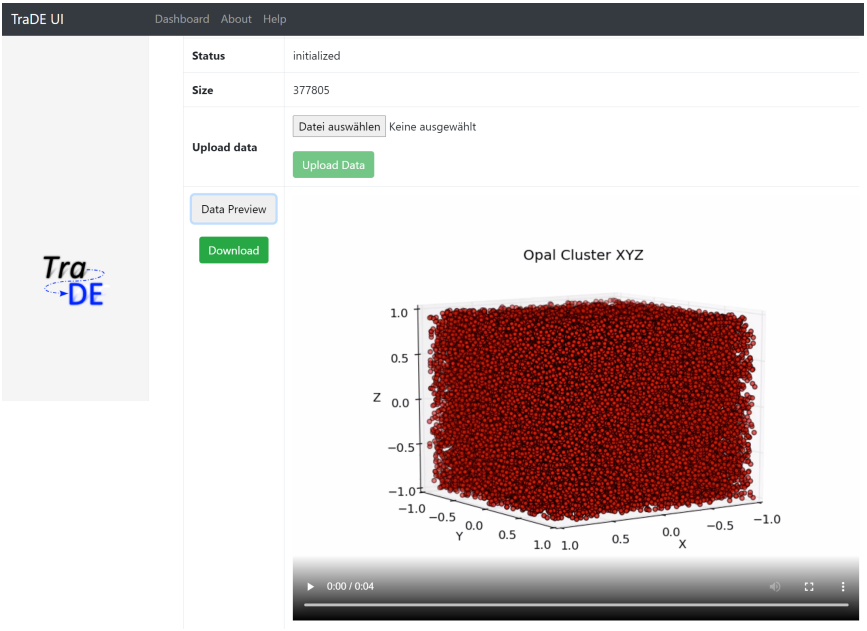

Figure 7.6: Preview of data values using the TraDE Web UI.

data object instance and registered *transformations* in form of DT bundles available at the DT Integration Middleware. The dashboard in the middle of the page provides three direct entry points to the data dependency graphs, i. e., the TraDE Middleware representation of the data contained in a TraDE deployment descriptor, data values as well as data transformations. From there all other resources can be found by dynamically traversing through the web UI via the provided links in the details pages.

Figure 7.6 shows such a details page for data values which allows to inspect the underlying metadata (e. g., media-type, status or size of the data), the related data object instance where the data value is used or originates from or provides the ability to download or even show an inline preview of the current data. This preview is possible for a predefined collection of media-types such as PDFs, videos, images or XML and plain text, for which the web

UI is able to directly render an inline preview as shown for the video data in Figure 7.6. The TraDE Web UI is accessible as open source on GitHub[1].

[1] TraDE Web UI: `https://github.com/traDE4chor/trade-web-ui`

Chapter 8

Validation and Evaluation

This chapter provides an validation and evaluation of the TraDE concepts and their prototypical implementation introduced in this work. Therefore, we first present a case study from the eScience domain, where we applied our TraDE concepts to a simulation choreography to ease modeling and provide additional runtime support regarding data-related aspects such as data exchange and data transformations. In addition, we validate and evaluate the prototypical implementation of the overall TraDE environment described in Chapter 7 through a performance evaluation comparing the execution of a choreography model with and without our TraDE concepts applied. The individual results are already published as part of different publications [HBKL17; HBLW17; HBLY18] based on the research conducted in the context of this thesis as outlined in Section 1.3.

8.1 OPAL Case Study

Figure 8.1 shows a choreography model of a Kinetic Monte Carlo (KMC) simulation using the custom-made simulation software *Ostwald ripening of Precipitates on an Atomic Lattice* (OPAL) [BS03] presented in Section 1.1.

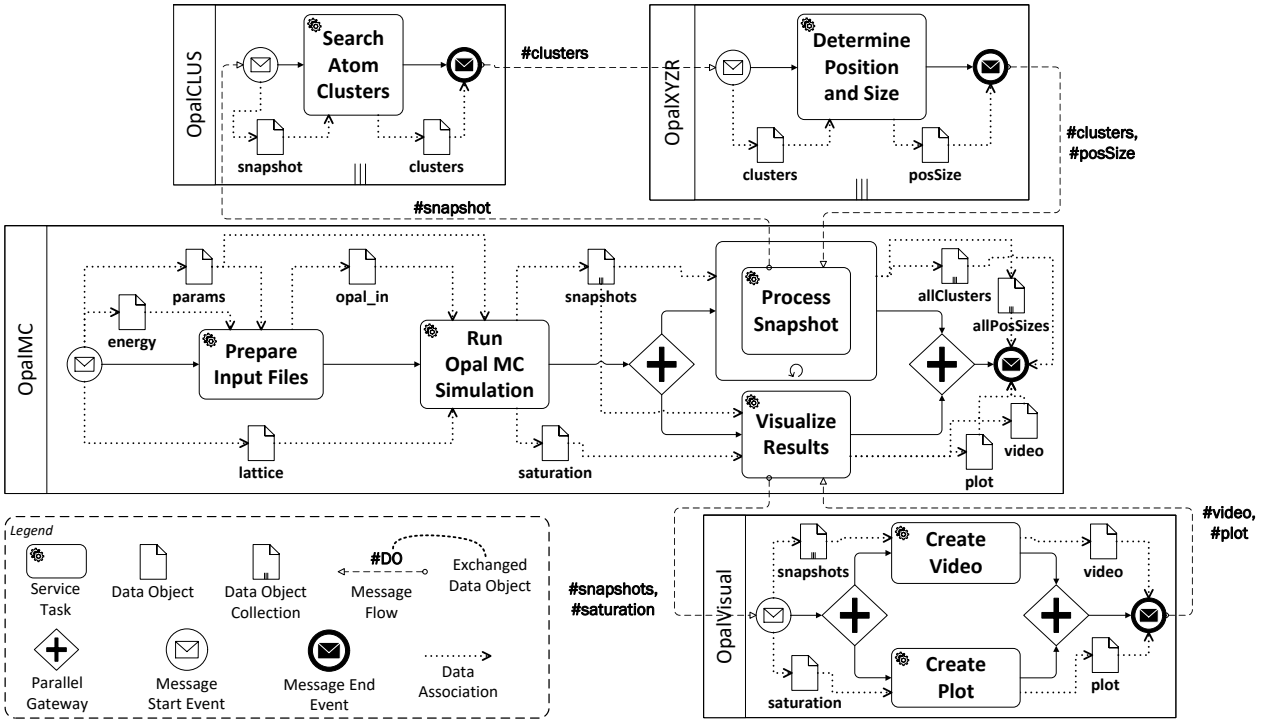

Figure 8.1: Opal simulation choreography model specifying a thermal aging simulation from material science domain (based on Hahn et al. [HBKL17]).

OPAL simulates the formation of copper precipitates, i. e., the development of atom clusters, within a lattice due to thermal aging. The simulation consists of four major building blocks which are reflected as participants of the choreography depicted in form of an BPMN 2.0 collaboration model in Figure 8.1. Before we describe the details of each of the choreography participants and how they together conduct an OPAL simulation run, we shortly introduce their purpose towards implementing the OPAL simulation.

The *OpalMC* participant reflects the underlying three phases of the OPAL simulation: pre-processing, simulation, post-processing. In the pre-processing phase, all data required for the simulation is collected and prepared in the required simulation input format by the *Prepare Input Files* task. Based on that, a Kinetic Monte Carlo simulation is conducted as part of the simulation phase reflected by the *Run Opal MC Simulation* task. Finally, in the post-processing phase, the simulation results are analyzed to identify formed clusters as part of the *OpalCLUS* participant, their positions and sizes as part of the *OpalXYZR* participant, as well as visualize the simulation results as part of the *OpalVisual* participant. Following the choreography paradigm, the conversations between the four participants are specified in a peer-to-peer manner through the exchange of messages. The consumed and produced data is represented through BPMN data objects, abbreviated with *DO* in the following. To exchange data between participants corresponding message flows are specified. The labels on the message flows in Figure 8.1 are added to visualize which data objects of a participant are exchanged during a conversation as part of the underlying message. The invocations of the different modules of the OPAL simulation software are modeled through corresponding service tasks within the participants. Since the OPAL simulation software is implemented in Fortran, where data between the modules is exchanged via files on the local file system, the software is provided as a set of executables. To provide these executables as services, corresponding service wrappers are manually implemented to enable their service-based invocation [SHK+11] and therefore enable their composition via the defined choreography model as described in the following.

Whenever the *OpalMC* participant receives a new request message, a new

OPAL simulation instance is created. The initial request contains an atom lattice (*lattice* DO), the initial energy configuration (*energy* DO) of the atoms, as well as a set of parameters (*params* DO), e. g., the number of snapshots of the lattice to create during the simulation. These snapshots are later used during the post-processing phase to identify and calculate clusters and their variation over time. The *Prepare Input Files* service task sends the parameters and the energy configuration to a service which takes the data to consolidate and transform it into the right input format (*opal_in* DO) for starting the Kinetic Monte Carlo (KMC) simulation.

The *Run Opal MC Simulation* service task invokes the provided service wrapper for the underlying Fortran executable which conducts the KMC simulation based on the available input data hold by the respective data objects. According to the specified number of snapshots in *params*, 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* collection DO) as well as cluster saturation data (*saturation* DO). After that, the snapshots are analyzed and visualized in parallel. Based on the number of snapshots, multiple instances of the *OpalCLUS* and *OpalXYZR* participants are created through the *Process Snapshots* service task. Each instance of the two participants is analyzing one particular snapshot, i. e., the position and size of clusters formed so far. The *OpalCLUS* participant takes a snapshot and its *Search Atom Clusters* service task invokes the provided service wrapper for the underlying Fortran executable which calculates and finally replies a list of identified clusters (*clusters* DO) of the snapshot. This cluster information is then forwarded to the *OpalXYZR* participant via a message exchange where the *Determine Position and Size* service task invokes the respective service wrapper for the underlying Fortran executable which is capable of calculating the position and size of each previously identified cluster (*posSize* DO). After that, the resulting cluster and position data is replied back to the *Process Snapshot* task of the *OpalMC* participant which collects the snapshot-related results within the collection data objects *allClusters* and *allPosSizes*.

In parallel to analyzing the snapshots, the *Visualize Results* service task triggers the visualization of the snapshot and saturation data through the

OpalVisual participant. The required data is passed through a message and then used by the *Create Video* and *Create Plot* service tasks to invoke corresponding visualization services. Based on the collection of snapshots, a video of animated 3D scatter plots (*Create Video* task) is created and the saturation data is used to create a 2D plot of the saturation function of the precipitation process (*Create Plot* task). Finally, the resulting media data is replied back to the *OpalMC* participant which then completes the execution of the OPAL simulation instance. The overall result data, i. e., the content of the data objects *allClusters* and *allPosSizes* as well as *plot* and *video*, is then replied to the requestor as part of the response message.

Based on the introduced OPAL simulation choreography model, in the following two sections, we discuss how the application our TraDE concepts for data-aware choroegraphies enable an easier modeling and execution of data-related aspects.

8.1.1 OPAL Simulation Choreography with TraDE Concepts

Figure 8.2 presents the OPAL simulation choreography model with our TraDE concepts applied. All data relevant for the choreography model is specified independent of any participant and in a shared and reusable manner through cross-partner data objects (DO) defined in Section 4.2.1.5. This makes the data required and produced by the choreography and each of its participants visible and therefore potentially easier to identify by human readers. Furthermore, data objects relevant for more than one participant do not have to be specified on the level of each individual participant which reduces the amount of data object definitions and participant internal data flow. The grouping of data elements into cross-partner data objects directly allows modelers to visually reflect that data is semantically related, as for example the *sim_input* DO contains all data elements providing required simulation input data. Another benefit of this grouping is that the whole collection of data elements can be specified as the source or target of a data flow. For example, the data flow between the start activity of the *OpalMC* participant and the *sim_input* DO specifies that the request message contains

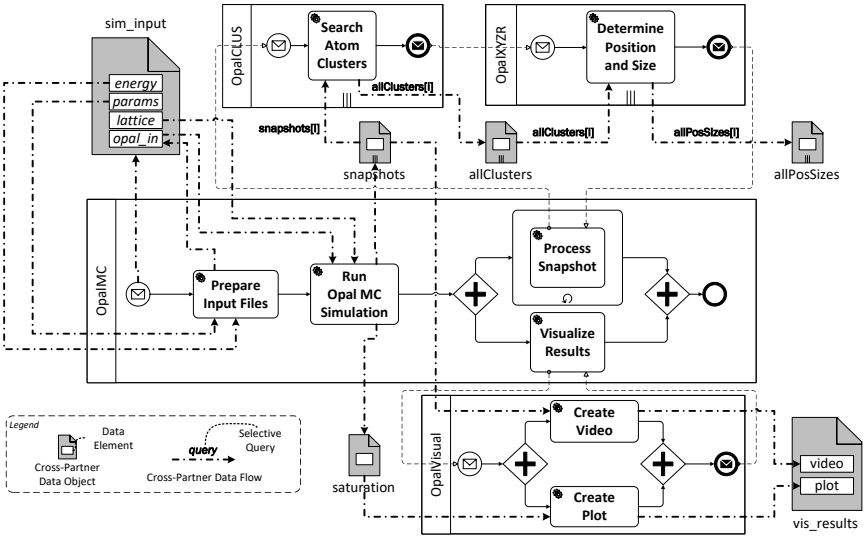

Figure 8.2: Data-aware OPAL simulation choreography model after applying our TraDE concepts (based on Hahn et al. [HBKL17]).

data for multiple data elements. The data flow therefore specifies a set of mappings how the data received via the message should be copied to the respective data elements. Furthermore, selective queries can be attached to the cross-partner data flow in order to specify that only parts of a cross-partner data object are required as defined in Section 4.2.6.1. For example, each instance of the *OpalCLUS* participant processes one snapshot from the whole collection of snapshots (*snapshots* cross-partner DO) which is specified through the selective query attached to the data flow (*snapshots[i]*).

The routing of data through consecutive participants is improved compared to the classical version of the model shown in Figure 8.1. Instead of routing cluster data from the *OpalCLUS* participant through the *OpalXYZR* participant to the *OpalMC* participant, with the TraDE approach such data can be directly stored in the globally accessible *allClusters* cross-partner data object. The advantage is that other services requiring this data can directly retrieve it from the cross-partner data object instead of waiting for a corre-

sponding message. Furthermore, by using the introduced capabilities of the TraDE approach, scientists are able to inspect each of the identified clusters as soon as they are added to the *allClusters* data object while the processing of the remaining snapshots is still running. For example, to inspect intermediary simulation results via the TraDE Web UI as presented in Section 7.2.5. Instead of sending all required input data encapsulated in the initial request message to trigger a new simulation instance, scientists are also able to upload data such as the initial lattice to the TraDE Middleware beforehand and then only pass a reference to this data within the request message. This is especially useful if the same data is used for multiple simulation runs, i. e., choreography instances executed in parallel. For example, in the context of a parameter study where it does not make sense to send the same data multiple times over the network, if this is not really required in terms of other reasons. The same applies the other way around, e. g., for the resulting video. Instead of routing data through several participants, the video will be directly stored from the *OpalVisual* participant to the shared *vis_results* DO from where it is directly accessible or downloadable at the TraDE Middleware even after the simulation instance itself is already completed.

8.1.2 OPAL Simulation Choreography with TraDE Data Transformations

Finally, we apply our TraDE Data Transformation (TDT) approach introduced in Chapter 6 to the Opal simulation choreography model as an example for easier modeling of data transformations and enabling their provisioning and execution in a technology-independent and transparent manner. The resulting model is depicted in Figure 8.3.

Compared to the TraDE-aware version of the model depicted in Figure 8.2, data transformations are modeled using the TraDE data transformation modeling extension introduced Section 6.2. Therefore, required data transformations can be modeled between cross-partner data objects instead of specifying them through respective transformation tasks, e. g., service tasks invoking transformation logic provided as a service. The service tasks *Prepare Input Files*, *Create Video* and *Create Plot* depicted in the TraDE-aware

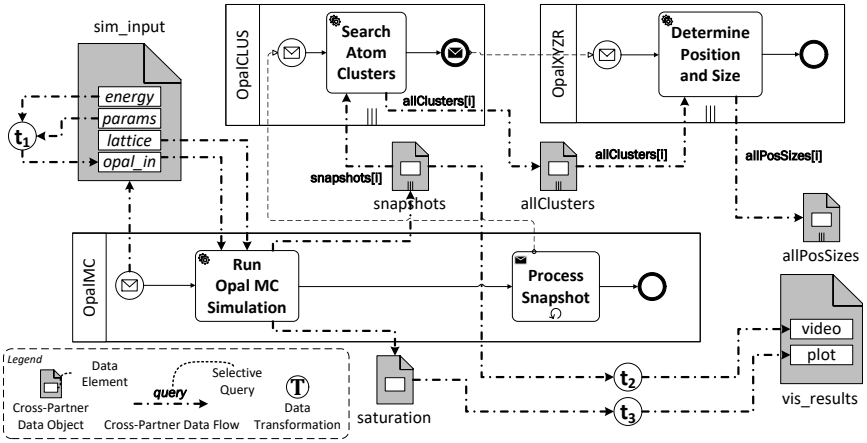

Figure 8.3: Opal simulation choreography model with TraDE data transformation modeling extension applied (based on Hahn et al. [HBLY18]).

model in Figure 8.2 are such transformation tasks since they invoke related transformation services using the specified data objects as input values. For example, the *Create Video* service task takes the collection of snapshots and transforms them into a video of animated 3D scatter plots.

According to our motivation for introducing data transformation capabilities on top of our TraDE concepts, as discussed in Chapter 6, the Opal simulation choreography model shown in Figure 8.2 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 (vis_results) of simulation data. Moreover, the simulation specific transformation implementations have to be manually wrapped as services to enable their invocation from the choreography. This requires expertise and additional effort since the transformation implementations required for the OPAL simulation are provided in form of a shell script (*Prepare Input Files*) or Python (*Create Video*) and Gnuplot scripts (*Create Plot*).

Figure 8.3 depicts the resulting OPAL choreography model with our TraDE

Data Transformation (TDT) approach applied. The definition of data transformations between cross-partner data objects allows to substitute the explicitly modeled transformations tasks: *Prepare Input Files*, *Create Video* and *Create Plot*. Instead, TraDE data transformations t_1, t_2, t_3 are defined to transform the simulation 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., *Prepare Input Files*) or in a way the underlying process engine enforces. Modelers are now able to specify and integrate their transformation implementations without manual wrapping effort, in the form of DT Bundles to enable their transparent execution within the TraDE ecosystem as presented in Chapter 6. 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. Finally, these bundles have to be published to the DT Integration Middleware to enable their use by the TraDE Middleware to conduct the specified transformation of cross-partner data objects.

The complete case study and any related artifacts to instantly run the case study in a dockerized environment, e. g., BPEL process models, a Docker Compose file and a JMeter test plan, are available on GitHub[1].

8.2 Evaluation

In the following, we introduce a performance evaluation comparing cross-partner data flows with the classical exchange of data through messages within service choreographies. Therefore, we first present the underlying evaluation methodology we apply, followed by a description of the experimental setup and finally a discussion of the evaluation results.

[1] TraDE Opal Case Study: `https://github.com/traDE4chor/trade-core-evaluation/tree/master/opal-case-study`

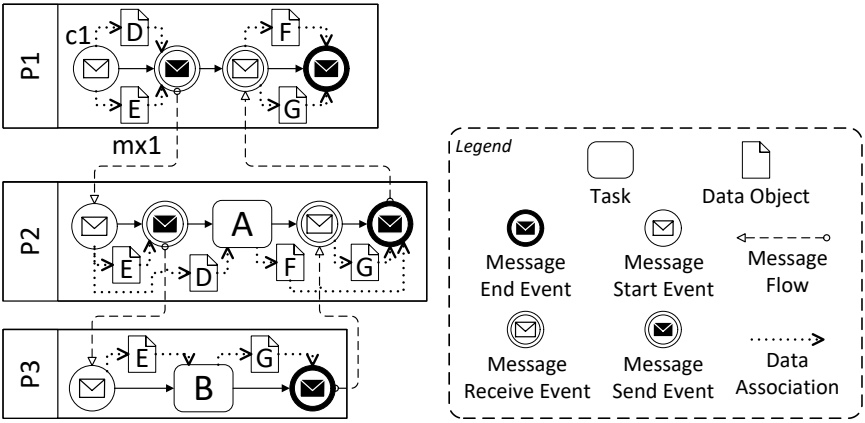

Figure 8.4: Choreography model used as baseline without our TraDE concepts applied (based on [HBLW17]).

8.2.1 Evaluation Methodology and Experimental Setup

The focus of the evaluation is at analyzing the performance variation when introducing our TraDE concepts, i. e., cross-partner data objects and cross-partner data flows. Therefore, two versions of a simple choreography model are used to measure variations of the response time perceived by a user triggering the execution of the choreography. Response time here refers to the time between sending the initial request and receiving the final response message from the choreography.

Figure 8.4 depicts the standards-based choreography model used as baseline for the evaluation in form of an BPMN 2.0 collaboration model. The conversations between the participants are modeled by BPMN message intermediate events and message flows, e. g., *mx1* in Figure 8.4. Each of the participants is instantiated through a corresponding BPMN message start event, e. g., *c1* in P1, which consumes an incoming request message and extracts the contained data for processing it within the choreography. Following the standards-based approach of modeling data and data exchange between participants, required data is modeled in form of BPMN data objects

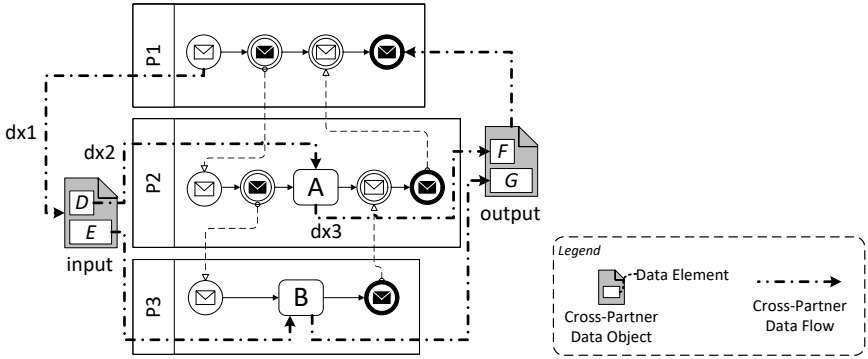

Figure 8.5: Choreography model with our TraDE concepts applied: *cross-partner data objects* and *cross-partner data flow*.

and their exchange is handled via message flows. For example, BPMN data objects *D* and *E* are passed from participants P1 to P2 via message flow *mx1*, i. e., the data contained in the data objects is put into a message and send to participant P2 where the data is extracted again and stored into related BPMN data objects *D* and *E* of participant P2. To introduce some actual data processing load at the level of the participants, the BPMN tasks *A* and *B* duplicate their input data by its concatenation and store the result in respective BPMN data objects.

Figure 8.5 depicts the data-aware choreography model with our TraDE concepts applied which is compared against the baseline choreography model introduced above. Choreography data is modeled through our cross-partner data objects, e. g., *input* in Figure 8.5, 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* in Figure 8.5.

While in the standards-based choreography model, data between participants is transferred through the exchange of messages within conversations, the notion of cross-partner data objects and cross-partner data flows allows us to decouple the exchange of data from the exchange of messages. For example, instead of forwarding the data of the initial request from partici-

pant P1 to participant P2 through message flow mx1, we can directly specify a cross-partner data flow to task *A* of participant P2 where the data is processed. The same applies to the result data of task A, instead of forwarding it to other participants through the exchange of messages, it is directly stored in cross-partner data object *output* via the specified cross-partner data flow *dx3* as shown in Figure 8.5.

To conduct the evaluation, both choreographies are modeled as BPEL4Chor [DKLW09] choreography models using the Data-aware Choreography & Orchestration Modeling Environment presented in Section 7.2.1. The resulting choreography models are transformed [RK+08] to three private process models implemented using the Business Process Execution Language (BPEL). Finally, each of the generated process models is manually refined, resulting in three executable BPEL process models which provide the basis to conduct the evaluation scenarios. To avoid confusion between BPMN data objects as language-specific constructs and cross-partner data objects as a general concept, in the following, we use the generic term *data container* for modeling constructs that allow the specification of data at the level of a specific modeling language, e. g., BPMN data objects or BPEL variables. The BPMN tasks A and B of participant P2 and P3 shown in Figure 8.5 are therefore refined to BPEL assign activities. Each of them contains respective assignment expressions which duplicate the random data contained in data containers D and E by its concatenation and store the result in data containers F and G. To guarantee that for both scenarios, baseline as well as TraDE, the underlying data has to be actually transferred through the exchange of messages or by uploading or downloading it from the TraDE Middleware, respectively, we deploy each of the resulting executable private process models to a separate process engine instance. As process engine the TraDE-aware version of Apache ODE introduced in Section 7.2.3 with its default configuration is used. As described in Section 7.2.1, the BPEL process models generated for the data-aware choreography model shown in Figure 8.5 are extended with TraDE annotations during the transformation, so that the process engine is aware of the linking of the BPEL variables with the cross-partner data objects managed by the TraDE Middleware.

To also measure a potential impact of the size of the data being processed, we introduce three scenarios with increasing data size for each of the input data elements (data object *input*, data elements *D* and *E*): 1KB, 128KB and 256KB. While for the baseline scenarios all data is stored locally in corresponding data containers at the process engines, in the TraDE evaluation scenarios the data are uploaded once to the cross-partner data objects managed by the TraDE Middleware and retrieved directly from there only when required by the process engines as outlined in Section 5.1. For each of the six scenarios summarized in Table 8.1, a workload consisting of randomly generated request messages with the above mentioned data element sizes is created. Based on previous experience from the experiments presented in [HSA+14; SASL13], the workload is distributed among a *warm-up phase* ($w(t_0)$) with 10 requests followed by an *experimental phase* with a total of 310 requests. Performing a warm-up phase prior to conducting the actual measurement is intended to reduce outliers during the initialization phase of the system under test. The requests are sent concurrently in five load bursts (i) to the process engine, to which the private process model of participant P1 is deployed, according to the following function over time:

$$m(t_i) = w(t_o) + \sum_{i=1}^{5} 2^{i-1} \cdot k \mid k = 10, w(t_0) = 10$$

The idea behind splitting the total load of 310 requests into five load bursts is to steadily increase the load on the system while measuring the variation of the user-perceived performance (average response time) based on the introduced load. Therefore, each of the load bursts follows an exponential function of base 2, with an initial burst of 10 followed by bursts with 20, 40, 80, and 160 concurrent requests.

The experimental environment is set up in an on-premise private cloud infrastructure on two virtual machines (VM) as shown in Figure 8.6. The evaluation VM is configured with 8 virtual CPUs, Intel® Xeon® CPU E5-2690 v2 3.00GHz, 32GB RAM, 120GB disk space, and is running an Ubuntu

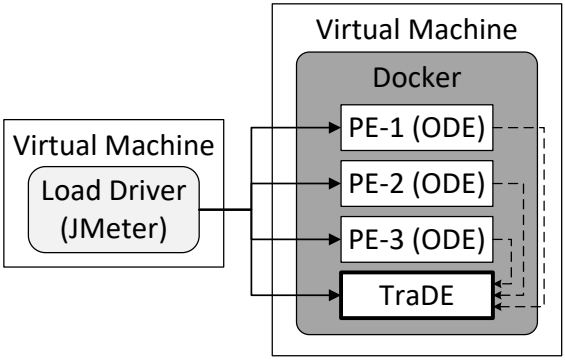

Figure 8.6: Experimental Setup.

14.04.4 64bit server distribution. We use Docker[1] within this VM to deploy the three separate instances of Apache ODE and in addition one TraDE Middleware instance in the TraDE scenarios. The idea behind this level of nesting and using Docker for the deployment of the evaluation environment is that we want to have a clean and therefore identical setup for each of the conducted experiments towards creating reproducible evaluation results.

To setup the evaluation environment and to conduct the workload for each of the defined evaluation scenarios, we use Apache JMeter 3.2[2] as load driver which is deployed in a separate VM, with the following configuration: 2 virtual CPUs, Intel® Xeon® CPU E5-2690 v2 3.00GHz, 4GB RAM, 40GB disk space, running an Ubuntu 14.04.2 64bit desktop distribution.

We created a JMeter test plan for each of the defined six scenarios, i. e., three baseline scenarios and three TraDE scenarios with data sizes of 1KB, 128KB and 256KB each, which concurrently sends the above defined workload for five concurrent users to the endpoint of the BPEL process model implementing participant P1. To alleviate the effect of outliers in the experimental results, we execute ten rounds of each scenario and calculate the average response time for each load burst while excluding the samples

[1] Docker Container Runtime: `https://www.docker.com/products/container-runtime`

[2] Apache JMeter: `http://jmeter.apache.org/`

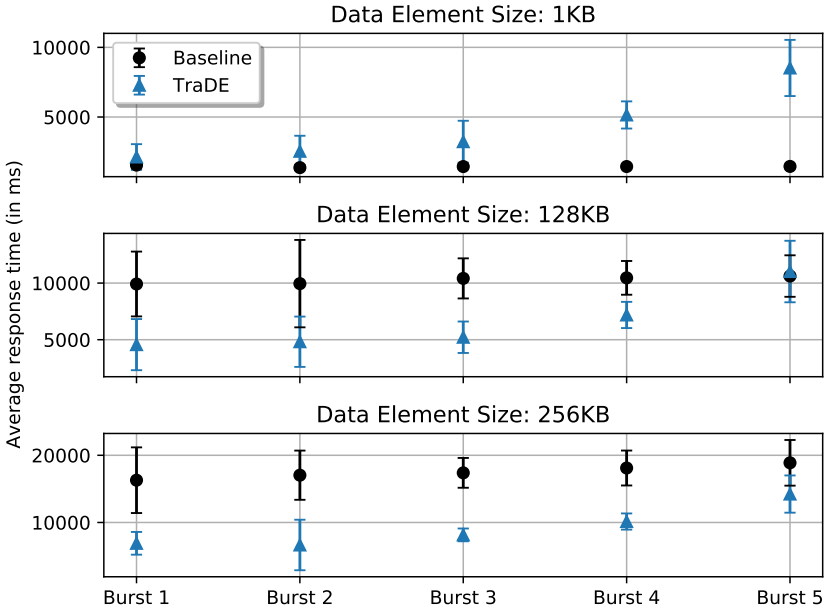

Figure 8.7: Evaluation results comparing the average response time in milliseconds (ms) for the five load bursts of all scenarios [HBLW17].

which are timed-out at the process engine.

8.2.2 Experimental Results

Figure 8.7 shows the experimental results comparing the user-perceived performance (average response time) of the load bursts of all scenarios. If we compare the baseline with the TraDE scenarios, there exists an overall beneficial impact to the user-perceived performance when introducing cross-partner data flows. However, this impact greatly varies on the size of the data being exchanged as well as on the workload applied throughout the load bursts. Comparing the scenarios with 256KB data size as shown in Figure 8.8, the performance is improved by approximately 90% in total. When we

have a look at the different load bursts in detail depicted in the last row of Figure 8.7, this improvement decreases from approximately 136% in burst 1 to approximately 32% in the last load burst. Therefore, we can assume that when increasing the load on the middleware, the improvement will further degrade and actually convert to an overall performance deterioration.

This is also underpinned when comparing the 128KB scenarios shown in Figure 8.8 where the performance is still improved by approximately 56% in total. However, again the performance alters from an improvement of approximately 117% in burst 1 to a small performance degradation of approximately 1% in the last load burst as shown in the second row of Figure 8.7. Comparing the scenarios with 1KB data size shown in Figure 8.8, the performance is degraded by approximately 66% in total when introducing the middleware and cross-partner data flows. There the overhead of introducing additional communication between the process engines and the TraDE Middleware to conduct the cross-partner data flows is higher than the improvements gained by reducing the amount of data to be exchanged.

Furthermore, Figure 8.7 shows that for the baseline scenarios with message-based data exchange, the performance maintains quite stable in terms of increasing the workload across the load bursts but decreases significantly when increasing data element sizes. This fact is also underpinned when comparing the overall average response time among all load bursts of the six scenarios based on the processed data element sizes as shown in Figure 8.8. In contrast, for the TraDE-based scenarios with cross-partner data flows, the performance maintains quite stable in terms of data element sizes as shown in Figure 8.8, but decreases significantly when increasing the workload throughout the five load bursts as shown in Figure 8.7.

Both types of scenarios are not fully able to process the complete workload in all load bursts without having a set of samples that timeout at the process engine. By default, Apache ODE is configured to timeout incoming requests after 120 seconds. For the baseline scenarios this is especially the case in the third scenario with a data element size of 256KB, where about 30% of the samples in load burst 5 result in timeouts, after approximately 556 successful samples are processed. A reason for this behavior might be the

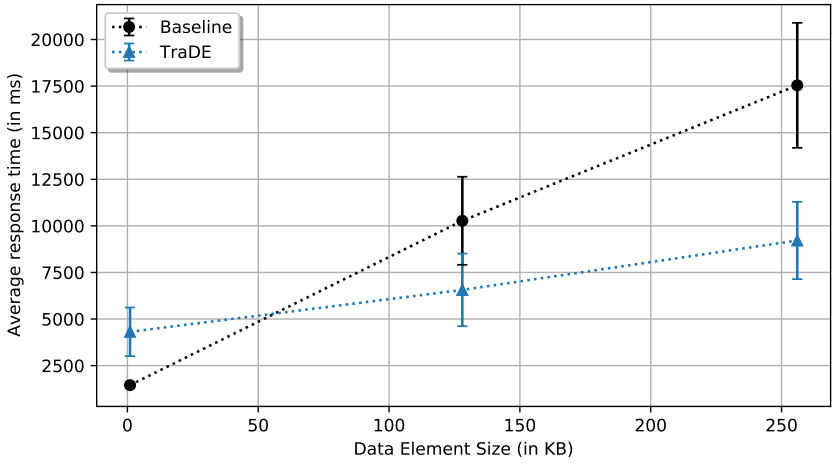

Figure 8.8: Evaluation results comparing the average response time in milliseconds (ms) based on the data element size (based on [HBLW17]).

large amount of data (1550 instances $*$ 3840KB $\approx$ 5.9GB) ODE is not capable of handling in its default configuration at a certain point in time. For the TraDE scenarios such samples causing timeouts are randomly distributed across nearly all scenarios and load bursts, but also with a peak in load burst 5. The reason for such an unpredictable behavior is most probably related to the resolution of required data from the TraDE Middleware through the process engine. To correlate process instances and data object instances and to finally retrieve data element values, the process engines poll the middlewares' Representational State Transfer (REST) API by sending repeated requests every second as long as the process instance is not timed-out. These requests are queued up at the TraDE Middleware while throttling its performance for a certain amount of time which again results in timeouts at the process engines. The average amount of timed-out samples as well as a summary of the scenarios and their average response times is shown in Table 8.1.

In summary, the evaluation results show that introducing a TraDE Middle-

Table 8.1: Summary of the experimental evaluation scenarios and their results.

Scenario	Data Element Size (in KB)	Total Data Size (in KB/instance)	Timed-out Req. (in %)	Avg. Resp. Time (in ms)
Baseline	1	15	0.04	1451.58
	128	1920	0.32	10272.86
	256	3840	6.49	17540.36
TraDE	1	6	0.20	4313.86
	128	768	0.47	6560.86
	256	1536	0.61	9214.08

ware layer and applying our concept for cross-partner data flows in service choreographies provide valuable performance improvements already for relatively small data sizes above 128KB. To alleviate the performance degradation when increasing the load at the middleware, the prototypical implementation and its integration with Apache ODE has to be improved, so that its performance maintains stable when increasing the workload.

Therefore, further experiments could help to investigate current capacity limitations of the TraDE Middleware when increasing the data sizes as well as the number of concurrent users and requests. The complete evaluation result data and any related material, e. g., BPEL process models and JMeter test plans, are available on GitHub[1].

[1] TraDE Evaluation: `https://github.com/traDE4chor/trade-core-evaluation/tree/master/initial-evaluation`

Chapter 9

Conclusions and Outlook

The motivation underlying to this work is driven by the increasing importance and business value of data in the fields of business process management and eScience as a data-intensive domain, all of which profiting from the recent advances in data science and Big data. As discussed in Chapter 1, service choreographies provide means to model and conduct collaborations between multiple parties specifying potentially complex conversations and interactions from a global perspective and in a technology-agnostic manner. However, in our opinion the paradigm shift towards data-awareness is not fully reflected by the state of the art in the domain of service choreographies at the moment. Therefore, within this work we present respective shortcomings in Chapter 1 which are then translated to the research contributions defined in Section 1.2 and tackled throughout the rest of the work. In the following, Section 9.1 summarizes and concludes the presented research contributions and Section 9.2 provides an outlook on potential future research directions in the context of data-aware service choreographies.

9.1 Conclusion

Towards the overall goal of increasing data-awareness in service choreographies, in Chapter 3, we present our overall methodology for data-aware service choreographies. The example presented in Section 3.1 further illustrates and motivates the need to improve data-awareness and discusses the shortcomings regarding the modeling and execution of service choreographies in more detail. For example, that data flow across participants has to be modeled differently compared to the data flow inside a participant or that choreography models do not support the definition of a common, globally consolidated and agreed set of data objects representing the data contract between the collaborating parties in form of a choreography data model. Based on that, we introduce our Transparent Data Exchange (TraDE) approach to mitigate the presented shortcomings and improve data-related aspects of service choreographies in Section 3.2. Therefore, we present the notion of cross-partner data objects and cross-partner data flows as well as required runtime support through the TraDE Middleware. The goal of the TraDE approach and its underlying concepts is to help to decouple the data flow, data exchange and management, from the control flow in service compositions and choreographies. Section 3.3 summarizes our methodology for data-aware choreographies by presenting how the TraDE approach can be integrated into the traditional Business Process Management (BPM) life cycle [Wes12], to increase data-awareness throughout all life cycle phases. The resulting life cycle for data-aware choreographies represents Contribution 1 and provides the basis for the rest of the work regarding the modeling and execution of data-aware choreographies.

Following the life cycle phases, Contribution 2 tackles the modeling and refinement of data-aware service choreographies. Therefore, in Chapter 4 we introduce a formal model for data-aware service choreographies together with a graphical notation based on our TraDE approach. The resulting Choreography Model Graph (CM-Graph) metamodel introduced in Section 4.2 supports the modeling of data-aware choreographies through cross-partner data objects and cross-partner data flows. Therefore, the metamodel follows

the *interconnection modeling approach* discussed in Section 2.1.1 which allows the specification of control flow per participant and interactions between participants through message connectors. Beside this decision, the CM-Graph metamodel is completely independent of any concrete choreography modeling notation such as Business Process Model and Notation (BPMN) [BPMN] collaboration models or BPEL4Chor models [DKLW07]. Following the *public-to-private approach* introduced by van der Aalst and Weske [AW01] and presented in Section 2.1.2, a data-aware choreography will be transformed to a collection of private process models. The resulting private process models reflecting the participants of a data-aware choreography model are then refined and can be finally executed to conduct the data-aware choreography they represent. For representing the private process models and their data-related aspects we rely on the Process Model Graph (PM-Graph) metamodel introduced by Leymann and Roller [LR00] and extend it with the notion of *staging elements* in Section 4.4. These staging elements specify the cross-partner data flows and references to the related cross-partner data objects at the level of the private process models, i. e., a PM-Graph, to provide the required runtime support for conducting cross-partner data flows via the TraDE Middleware as outlined in Section 3.2.2. To reflect the data perspective of a CM-Graph during the execution life cycle phase, i. e., at the TraDE Middleware, we introduce the notion of a *choreograpyh data model* (CDM) and a *choreography data dependency graph* (CDDG) in Section 4.3. Finally, the transformation of a CM-Graph to a collection of PM-Graphs is presented in Section 4.5.

In Chapter 5, we introduce a middleware for transparent data provisioning, exchange, and access for data-aware service choreographies. The resulting TraDE Middleware represents Contribution 3 and provides the required runtime support for the execution life cycle phase, i. e., executing data-aware choreography models and their specified cross-partner data flows based on our CM-Graph and extended PM-Graph metamodels. Therefore, the TraDE Middleware has to expose and manage the defined cross-partner data objects of a data-aware choreography model as well as actually conduct the modeled cross-partner data flows together with the Business Process

Engines (BPEs) executing the private process models representing the data-aware choreography model. The TraDE Middleware can be therefore seen as kind of a data hub from the perspective of the individual private process models. To get a better idea on how this actually looks like, a respective overview for the execution of a data-aware choreography model is presented in Section 5.1. This is followed by the definition of the internal conceptual model of the middleware in Section 5.2, its underlying architecture and main components in Section 5.4, and its integration with BPEs in Section 5.5. Based on that, the resulting TraDE Middleware provides the required runtime support for data-aware choreographies.

Since participants in service choreographies often rely on the composition of already existing business logic and therefore come up with their own internal data formats and models, Contribution 4 is about concepts for transparent data transformation in data-aware service choreographies. Chapter 6 introduces an overall TraDE Data Transformation (TDT) concept where the focus is on providing the aforementioned data transformation capabilities to enable the specification of data transformations at the level of data-aware service choreographies as well as supporting their transparent execution during choreography runtime. Regarding the modeling of data transformations, a TraDE *Data Transformation (DT)* modeling extension is presented in Section 6.2 and its integration into the overall formal model for data-aware choreography models is defined. The required runtime support for modeled data transformations is presented in Section 6.3 by introducing a data transformation integration middleware (DT Integration Middleware), its integration with the TraDE Middleware and a concept for providing and invoking data transformation implementations in form of *DT Bundles*.

The last contribution, Contribution 5, introduces the prototypical implementation of respective tools and middleware components for data-aware service choreographies introduced within this work. The resulting TraDE ecosystem provides an end-to-end support for the modeling and execution of data-aware choreographies and supports the respective phases of the data-aware life cycle presented in Section 3.3. Chapter 7 introduces the ecosystem which is comprised of a modeling environment for data-aware

choreography and process models as well as the required runtime environment to execute data-aware choreographies through the TraDE Middleware and its integration to corresponding BPEs as well as the DT Integration Middleware. A validation and evaluation of our concepts for data-aware choreographies and their prototypical implementation through the TraDE ecosystem is presented in Chapter 8. The validation is provided in form of a case study from the eScience domain which describes and discussed the applicability and use of the presented TraDE concepts to ease modeling and provide additional runtime support regarding data-related aspects such as data exchange and data transformations. In addition, a performance evaluation of the prototypical implementation of the TraDE Middleware is presented which compares the execution of a choreography model with and without our TraDE concepts applied. This means, comparing TraDE-based cross-partner data flows to classical message-based data exchange as discussed in Section 3.1 regarding the shortcomings of the state of the art in specifying and conducting data exchange in service choreographies. The evaluation results show some interesting performance improvements for relatively small data sizes which indicates that introducing a TraDE Middleware layer and applying our concept for cross-partner data flows in service choreographies can be beneficial, especially for data-intensive use cases. However, the current TraDE Middleware prototype shows capacity limitations when increasing data sizes as well as the number of concurrent requests for reading or writing data objects.

9.2 Outlook

As already outlined within the different chapters of the work, our goal is to describe and provide an overall end-to-end support for the modeling and execution of data-aware choreographies. This required us often to focus only on the most important core aspects to show the feasibility of the overall TraDE concepts and our idea of data-aware choreography models within this work. Therefore, there exist several future research directions for the

concepts presented in this work as well as for the context of data-aware choreographies in general. Some of them will be outlined in the following.

Weiß et al. [Wei+16] and Weiß [Wei18] introduce the concept of *Model-as-you-go* for choreographies and an underlying *ChorSystem* middleware which can be integrated within our TraDE ecosystem. The presented Model-as-you-go approach together with the middleware allow the exploratory and therefore flexible modeling and execution of choreography models. This flexibility is provided by introducing user-driven control based on concepts for rewinding and repeating choreography logic during choreography runtime. By combining the Model-as-you-go approach with our TraDE concepts, respective flexibility support regarding the data flow perspective of choreographies can be introduced. This will allow modelers to leverage the capabilities of the Model-as-you-go approach and the ChorSystem middleware to increase flexibility while modeling and executing data-aware choreographies, and to ease and improve their deployment and management in the future.

Within this work, we assumed that cross-partner data objects managed by the TraDE Middleware are either initialized with respective data through a request sent to the choreography itself or being manually uploaded to the TraDE Middleware by a human user. For example, in the context of a simulation choreography, e. g., as presented in Section 8.1, a scientist uploads a set of input data to respective data objects at the TraDE Middleware and then starts one or more choreography instances operating on that data. Therefore, another future research direction could be on how to automate such data provisioning steps, e. g., to upload or even link required data directly from arbitrary data sources. Especially for such data provisioning tasks within simulation workflows, Reimann [Rei17] and Reimann et al. [RRS+11] introduce the so-called *SIMPL framework*. By integrating the SIMPL framework with the TraDE Middleware and enabling the modeling of data provisioning for cross-partner data objects at the level of data-aware choreography models, required data provisioning steps can be fully automated and transparently executed by the TraDE Middleware during the execution of a data-aware choreography model. To integrate data provisioning logic a similar approach as presented for data transformations in Chapter 6 may be applicable.

Another future research direction is to introduce and enable distributed, multi-node deployments of the TraDE Middleware towards the goal of identifying and enabling further data flow optimization possibilities during choreography runtime as outlined in Chapter 3. For example, event data of previous choreography executions could be analyzed based on our TraDE concepts to detect potential modeling issues as well as runtime optimizations for the cross-partner data flows specified at the level of the choreography model. Based on such event data the probabilities for each control flow path of the choreographed process models can be calculated, e. g., to estimate when and where data of a specific cross-partner data object is required. This information can be used to predictively start the data staging and exchange as soon as the data or parts of it are available, so that in the best case, the data is accessible at the required TraDE node when it is needed without influencing or even blocking the process execution at all. Therefore, a set of optimization strategies have to be defined which leverage the available knowledge to target different optimization goals, e. g., cost, performance or resource consumption. Since such a distributed TraDE Middleware node network, or *TraDE network* for short, increases the level of concurrency from a data perspective, a thorough analysis of potential application cases and their concurrency issues has to be conducted to identify and define corresponding synchronization and scheduling mechanisms and related protocols. A future research direction related to the idea of introducing TraDE networks, is the use and integration of different *data interchange layers* to guarantee a robust, reliable and secure data exchange between the TraDE Middleware nodes within a network. This is especially interesting in scenarios where different parties want to collaborate and therefore have to exchange data, however, do not trust each other. Therefore, distributed ledger technologies such as blockchains may be used as a data interchange layer to introduce a layer of trust within TraDE networks or directly between choreography participants. For the latter, the concepts introduced by Falazi et al. [FHBL19] may be integrated and mapped to our TraDE concepts to support the transparent reading and writing of data from or to a distributed ledger directly with cross-partner data objects and data flows.

Bibliography

[Aal+05] W. M. P. van der Aalst et al. "Case handling: a new paradigm for business process support." In: *Data & Knowledge Engineering* (2005), pp. 129–162 (Cited on p. 47).

[ABJ+04] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, S. Mock. "Kepler: an extensible system for design and execution of scientific workflows." In: *Proceedings of SSDBM'04*. IEEE. 2004, pp. 423–424 (Cited on p. 15).

[Afa+13] A. Afanasiev et al. "MathCloud: publication and reuse of scientific applications as RESTful web services." In: *Proceedings of PaCT'13*. 2013 (Cited on p. 50).

[All10] S. Allamaraju. *RESTful Web Services Cookbook*. O'Reilly Media, Inc., 2010 (Cited on p. 183).

[ALM+08] W. M. P. van der Aalst, N. Lohmann, P. Massuthe, C. Stahl, K. Wolf. "Multiparty Contracts: Agreeing and Implementing Interorganizational Processes." In: *The Computer Journal* 53.1 (2008), pp. 90–106 (Cited on p. 34, 55).

[AW01] W. M. P. van der Aalst, M. Weske. "The P2P Approach to Interorganizational Workflows." In: *Proceedings of CAiSE'01*. Ed. by K. R. Dittrich, A. Geppert, M. C. Norrie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 140–156 (Cited on p. 14, 19, 34, 39, 40, 55, 61, 84, 87, 218, 257).

[AW13] W. M. P. van der Aalst, M. Weske. “Reflections on a Decade of Interorganizational Workflow Research.” In: *Seminal Contributions to Information Systems Engineering*. Berlin, Heidelberg: Springer, 2013 (Cited on p. 35).

[BCF06] W. Binder, I. Constantinescu, B. Faltings. “Decentralized Orchestration of Composite Web Services.” In: *Proceedings of ICWS'06*. Sept. 2006, pp. 869–876 (Cited on p. 16, 42).

[BDH+08] M. Bitsaki, O. Danylevych, W.-J. van den Heuvel, G. Koutras, F. Leymann, M. Mancioppi, C. Nikolaou, M. Papazoglou. “An Architecture for Managing the Lifecycle of Business Goals for Partners in a Service Network.” In: *Towards a Service-Based Internet*. Vol. 5377. LNCS. Springer Berlin Heidelberg, 2008, pp. 196–207 (Cited on p. 80).

[BDH05] A. Barros, M. Dumas, A. H. M. ter Hofstede. “Service Interaction Patterns.” In: *Proceedings of BPM'05*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 302–318 (Cited on p. 14, 30, 134).

[BFM05] T. Berners-Lee, R. T. Fielding, L. M. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. Jan. 2005. URL: https://rfc-editor.org/rfc/rfc3986.txt (Cited on p. 66, 174, 183).

[BG07] R. Barga, D. Gannon. “Workflows for e-Science: Scientific Workflows for Grids.” In: Springer London, 2007. Chap. Scientific versus Business Workflows, pp. 9–16 (Cited on p. 13, 15).

[BHG87] P. A. Bernstein, V. Hadzilacos, N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987 (Cited on p. 67).

[Bin+13] T. Binz et al. “OpenTOSCA – A Runtime for TOSCA-based Cloud Applications.” In: *Proceedings of ICSOC'13*. 2013, pp. 692–695 (Cited on p. 78, 201, 234).

[BPEL] OASIS. *Web Services Business Process Execution Language Version 2.0*. OASIS Standard. Apr. 2007. URL: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html (Cited on p. 14, 30, 35, 49, 129, 134, 149, 178).

[BPMN] Object Management Group (OMG). *Business Process Model And Notation (BPMN) Version 2.0.2*. Object Management Group, Inc. Jan. 2014. URL: https://www.omg.org/spec/BPMN/ (Cited on p. 14, 30, 31, 33, 34, 48, 49, 70, 104, 129, 134, 149, 178, 257).

[Bra17] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC8259. Dec. 2017 (Cited on p. 177).

[BS03] P. Binkele, S. Schmauder. "An atomistic Monte Carlo simulation of precipitation in a binary system." In: *Zeitschrift für Metallkunde* (2003) (Cited on p. 17, 237).

[BT18] C. Barrett, C. Tinelli. "Satisfiability Modulo Theories." In: *Handbook of Model Checking*. Ed. by E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem. Springer International Publishing, 2018. URL: https://doi.org/10.1007/978-3-319-10575-8_11 (Cited on p. 45).

[BW+12] A. Barker, J. B. Weissman, et al. "Reducing Data Transfer in Service-Oriented Architectures: The Circulate Approach." In: *IEEE Transactions on Services Computing* 5.3 (2012), pp. 437–449 (Cited on p. 16, 39).

[BWR09] A. Barker, C. D. Walton, D. Robertson. "Choreographing Web Services." In: *IEEE Transactions on Services Computing* 2.2 (Apr. 2009), pp. 152–166 (Cited on p. 15, 38).

[BWV08a] A. Barker, J. B. Weissman, J. Van Hemert. "Eliminating the Middleman: Peer-to-Peer Dataflow." In: *Proceedings of HPDC'08*. ACM. 2008, pp. 55–64 (Cited on p. 78).

[BWV08b] A. Barker, J. B. Weissman, J. Van Hemert. "Orchestrating Data-Centric Workflows." In: *Proceedings of CCGRID'08*. IEEE. 2008, pp. 210–217 (Cited on p. 16).

[CDL+07] A. Chervenak, E. Deelman, M. Livny, et al. "Data placement for Scientific Applications in Distributed Environments." In: *Proceedings of Grid'07*. IEEE Computer Society. 2007, pp. 267–274 (Cited on p. 78).

[Cha04] D. Chappell. *Enterprise Service Bus*. O'Reilly Media, Inc., 2004 (Cited on p. 192, 200).

[DC08] E. Deelman, A. Chervenak. “Data Management Challenges of Data-Intensive Scientific Workflows.” In: *Proceedings of CCGRID’08*. IEEE. 2008, pp. 687–692 (Cited on p. 78).

[Del+05] T. Delaitre et al. “GEMLCA: Running legacy code applications as grid services.” In: *Journal of Grid Computing* (2005) (Cited on p. 50).

[DKB08] G. Decker, O. Kopp, A. Barros. “An Introduction to Service Choreographies.” In: *Information Technology* 50.2 (2008) (Cited on p. 14, 15, 30, 33, 34, 54, 55, 61, 67, 83, 87).

[DKL+08] G. Decker, O. Kopp, F. Leymann, K. Pfitzner, M. Weske. “Modeling Service Choreographies using BPMN and BPEL4Chor.” In: *Advanced Information Systems Engineering*. Springer. 2008, pp. 79–93 (Cited on p. 14, 34, 104).

[DKLW07] G. Decker, O. Kopp, F. Leymann, M. Weske. “BPEL4Chor: Extending BPEL for Modeling Choreographies.” In: *Proceedings of ICWS’07*. IEEE, 2007 (Cited on p. 31, 34, 70, 257).

[DKLW09] G. Decker, O. Kopp, F. Leymann, M. Weske. “Interacting Services: From Specification to Execution.” In: *Data & Knowledge Engineering* 68.10 (2009), pp. 946–972 (Cited on p. 14, 31, 34, 55, 228, 248).

[DSS+05] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, et al. “Pegasus: A framework for mapping complex scientific workflows onto distributed systems.” In: *Scientific Programming* 13.3 (2005), pp. 219–237 (Cited on p. 15).

[FHBL19] G. Falazi, M. Hahn, U. Breitenbücher, F. Leymann. “Modeling and execution of blockchain-aware business processes.” In: *SICS Software-Intensive Cyber-Physical Systems* 34.2 (2019), pp. 105–116 (Cited on p. 261).

[Fie00] R. T. Fielding. “Architectural Styles and the Design of Network-based Software Architectures.” PhD thesis. University of California, Irvine, 2000 (Cited on p. 183).

[FMW22] P. Felli, M. Montali, S. Winkler. *Soundness of Data-Aware Processes with Arithmetic Conditions*. 2022. URL: https://arxiv.org/abs/2203.14809 (Cited on p. 154).

[Fow02] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002 (Cited on p. 183).

[GDE+07] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, J. Myers. "Examining the challenges of scientific workflows." In: *Computer* 40.12 (2007), pp. 24–32 (Cited on p. 15).

[GGMR21] S. Ghilardi, A. Gianola, M. Montali, A. Rivkin. "Delta-BPMN: A Concrete Language and Verifier for Data-Aware BPMN." In: *Proceedings of BPM'21*. Ed. by A. Polyvyanyy, M. T. Wynn, A. Van Looy, M. Reichert. Springer International Publishing, 2021, pp. 179–196 (Cited on p. 45).

[Gla+08] T. Glatard et al. "A Service-Oriented Architecture enabling dynamic service grouping for optimizing distributed workflow execution." In: *Future Generation Computer Systems* (2008) (Cited on p. 51).

[GSK+11] K. Görlach, M. Sonntag, D. Karastoyanova, F. Leymann, M. Reiter. "Guide to e-Science: Next Generation Scientific Research and Discovery." In: Springer London, 2011. Chap. Conventional Workflow Technology for Scientific Simulation, pp. 323–352 (Cited on p. 15, 20, 45).

[Hab+08] D. Habich et al. "BPEL^{DT} - Data-Aware Extension for Data-Intensive Service Applications." In: *Emerging Web Services Technology*. 2008 (Cited on p. 40).

[HBC+16] R. Hull, V. S. Batra, Y.-M. Chen, A. Deutsch, F. F. T. Heath III, V. Vianu. "Towards a Shared Ledger Business Collaboration Language Based on Data-Aware Processes." In: *Proceedings of ICSOC'16*. Springer International Publishing, 2016, pp. 18–36 (Cited on p. 47).

[HBKL17] M. Hahn, U. Breitenbücher, O. Kopp, F. Leymann. “Modeling and Execution of Data-Aware Choreographies: An Overview.” In: *Computer Science - Research and Development* (2017), pp. 1–12 (Cited on p. 25, 54, 59, 226, 237, 238, 242).

[HBL+18] M. Hahn, U. Breitenbücher, F. Leymann, M. Wurster, V. Yussupov. “Modeling Data Transformations in Data-Aware Service Choreographies.” In: *Proceedings of EDOC’18*. IEEE Computer Society, 2018, pp. 28–34 (Cited on p. 25, 195, 205).

[HBLW17] M. Hahn, U. Breitenbücher, F. Leymann, A. Weiß. “TraDE – A Transparent Data Exchange Middleware for Service Choreographies.” In: *Proceedings of OTM 2017 Conferences*. Ed. by H. Panetto, C. Debruyne, W. Gaaloul, M. Papazoglou, A. Paschke, C. A. Ardagna, R. Meersman. Vol. 10573. Lecture Notes in Computer Science (LNCS). Springer International Publishing, 2017, pp. 252–270 (Cited on p. 25, 174, 184, 187, 188, 237, 246, 251, 253).

[HBLY18] M. Hahn, U. Breitenbücher, F. Leymann, V. Yussupov. “Transparent Execution of Data Transformations in Data-Aware Service Choreographies.” In: *Proceedings of OTM 2018 Conferences*. Vol. 11230. Lecture Notes in Computer Science (LNCS). Springer International Publishing AG, 2018, pp. 117–137 (Cited on p. 25, 197, 200, 219, 226, 237, 244).

[HKL16a] M. Hahn, D. Karastoyanova, F. Leymann. “A Management Life Cycle for Data-Aware Service Choreographies.” In: *Proceedings of ICWS’16*. IEEE Computer Society, 2016, pp. 364–371 (Cited on p. 25, 70, 73, 76, 78, 79, 81).

[HKL16b] M. Hahn, D. Karastoyanova, F. Leymann. “Data-Aware Service Choreographies through Transparent Data Exchange.” In: *Proceedings of ICWE’16*. Vol. 9671. Lecture Notes in Computer Science (LNCS). Springer International Publishing, 2016, pp. 357–364 (Cited on p. 25, 68).

[Hos+16] A. Hosny et al. “AlgoRun: a Docker-based packaging system for platform-agnostic implemented algorithms.” In: *Bioinformatics* (2016) (Cited on p. 51, 52).

[HSA+14] M. Hahn, S. G. Sáez, V. Andrikopoulos, D. Karastoyanova, F. Leymann. “Development and Evaluation of a Multi-tenant Service Middleware PaaS Solution.” In: *Proceedings of the 7th International Conference on Utility and Cloud Computing, UCC 2014, 8-11 December 2014, London, United Kingdom*. IEEE Computer Society, 2014 (Cited on p. 249).

[HTT+09] T. Hey, S. Tansley, K. M. Tolle, et al. *The fourth paradigm: data-intensive scientific discovery*. Vol. 1. Microsoft research Redmond, WA, 2009 (Cited on p. 13).

[HW04] G. Hohpe, B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004 (Cited on p. 192).

[IANA] Internet Assigned Numbers Authority (IANA). *Media Types*. URL: `https://www.iana.org/assignments/media-types/media-types.xhtml` (Cited on p. 177).

[JSON Schema] *JSON Schema*. Online. URL: `https://json-schema.org/` (Cited on p. 176).

[Juh+09] E. Juhnke et al. “LCDL: an extensible framework for wrapping legacy code.” In: *Proceedings of iiWAS'09*. 2009 (Cited on p. 51, 52).

[KEL+11] O. Kopp, L. Engler, T. van Lessen, F. Leymann, J. Nitzsche. “Interaction Choreography Models in BPEL: Choreographies on the Enterprise Service Bus.” In: *Proceedings of S-BPM ONE*. Vol. 138. Communications in Computer and Information Science. Springer-Verlag, 2011, pp. 36–53 (Cited on p. 34, 81).

[KELU10] O. Kopp, H. Eberle, F. Leymann, T. Unger. “The subprocess spectrum.” In: *Proceedings of ISSS and BPSC*. Ed. by W. Abramowicz, R. Alt, K.-P. Fähnrich, B. Franczyk, L. A. Maciaszek. Gesellschaft für Informatik e.V., 2010, pp. 267–279 (Cited on p. 149).

[KFE19] J. Köpke, M. Franceschetti, J. Eder. “Optimizing data-flow implementations for inter-organizational processes.” In: *Distributed and Parallel Databases* 37 (2019), pp. 651–695 (Cited on p. 43, 45).

[Kim+17] B. Kim et al. “Bio-Docklets: Virtualization Containers for Single-Step Execution of NGS Pipelines.” In: *bioRxiv* (2017) (Cited on p. 51).

[KKL07] R. Khalaf, D. Karastoyanova, F. Leymann. “Pluggable Framework for Enabling the Execution of Extended BPEL Behavior.” In: *Service-Oriented Computing - ICSOC 2007 Workshops*. Ed. by E. Di Nitto, M. Ripeanu. Springer Berlin Heidelberg, 2007, pp. 376–387 (Cited on p. 189).

[KKP+81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, M. Wolfe. “Dependence Graphs and Compiler Optimizations.” In: *Proceedings of POPL’81*. ACM, 1981, pp. 207–218 (Cited on p. 74).

[KL06] R. Khalaf, F. Leymann. “Role-based Decomposition of Business Processes using BPEL.” In: *International Conference on Web Services (ICWS 2006)*. IEEE Computer Society, 2006 (Cited on p. 41).

[KLUW11] O. Kopp, F. Leymann, T. Unger, S. Wagner. “Towards The Essential Flow Model.” In: *Proceedings of ZEUS’11*. Ed. by D. Eichhorn, A. Koschmider, H. Zhang. Vol. 705. CEUR Workshop Proceedings. CEUR-WS.org, 2011, pp. 26–33 (Cited on p. 45).

[KLW11] O. Kopp, F. Leymann, S. Wagner. “Modeling Choreographies: BPMN 2.0 versus BPEL-based Approaches.” In: *Proceedings of EMISA’11*. Lecture Notes in Informatics. GI, Sept. 2011 (Cited on p. 14, 31, 34).

[Kop+10] O. Kopp et al. “Fault handling in the web service stack.” In: *Proceedings of ICSOC’10*. Springer. 2010, pp. 303–317 (Cited on p. 149).

[Kop+11a] O. Kopp et al. “A Classification of BPEL Extensions.” In: *Journal of Systems Integration* 2.4 (2011), pp. 3–28 (Cited on p. 41, 149).

[Kop+11b] O. Kopp et al. *An Event Model for WS-BPEL 2.0*. Tech. rep. 2011/07. University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2011 (Cited on p. 179, 189).

[Kop16] O. Kopp. “Partnerübergreifende Geschäftsprozesse und ihre Realisierung in BPEL.” Dissertation. University of Stuttgart, 2016 (Cited on p. 31, 70, 228).

[KPR12a] D. Knuplesch, R. Pryss, M. Reichert. “Data-aware interaction in distributed and collaborative workflows: Modeling, semantics, correctness.” In: *Proceedings of CollaborateCom’12*. Oct. 2012, pp. 223–232 (Cited on p. 16, 37).

[KPR12b] D. Knuplesch, R. Pryss, M. Reichert. “A formal framework for data-aware process interaction models.” In: *Open Access Repository of University Ulm* (2012) (Cited on p. 37).

[KV07] J. Koehler, J. Vanhatalo. “Process Anti-Patterns: How to Avoid the Common Traps of Business Process Modeling.” In: *IBM WebSphere Developer Technical Journal* 10.2 (2007), p. 4 (Cited on p. 82).

[Ley09] F. Leymann. “Cloud computing: The next revolution in IT.” In: *Proceedings of the 52nd Photogrammetric Week*. Wichmann Verlag, 2009 (Cited on p. 13).

[Lic22] T. Lichtenstein. “Preserving Data Consistency in Process Choreographies by Design.” In: *Proceedings of ZEUS’22*. Ed. by J. Manner, D. Lübke, S. Haarmann, S. Kolb, N. Herzberg, O. Kopp. CEUR-WS.org, 2022 (Cited on p. 57).

[LKP10] F. Leymann, D. Karastoyanova, M. Papazoglou. “Handbook on Business Process Management 1.” In: International Handbooks on Information Systems. Springer-Verlag, 2010. Chap. Business Process Management Standards (Cited on p. 13).

[LLW02] D. Liu, K. H. Law, G. Wiederhold. “Data-flow Distribution in FICAS Service Composition Infrastructure.” In: *Proceedings of ICPDCS’02*. Citeseer. 2002 (Cited on p. 16, 41).

[LN11] N. Lohmann, M. Nyolt. “Artifact-centric modeling using BPMN.” In: *Proceedings of ICSOC’11*. Springer. 2011, pp. 54–65 (Cited on p. 16, 47).

[LR00] F. Leymann, D. Roller. *Production Workflow - Concepts and Techniques*. PTR Prentice Hall, Jan. 2000, p. 479 (Cited on p. 14, 30, 40, 69, 80, 82, 84, 86–91, 98–100, 104, 106–109, 112, 113, 116, 118, 119, 122, 136, 144, 148, 151, 152, 163, 173, 257).

[LRS02] F. Leymann, D. Roller, M.-T. Schmidt. “Web services and business process management.” In: *IBM Systems Journal: Web services and business process management* (2002) (Cited on p. 13).

[LW10] N. Lohmann, K. Wolf. “Artifact-centric choreographies.” In: *Proceedings of ICSOC’10*. Springer. 2010, pp. 32–46 (Cited on p. 47).

[LWW19] J. Ladleif, M. Weske, I. Weber. “Modeling and Enforcing Blockchain-Based Choreographies.” In: *Proceedings of BPM’19*. Ed. by T. Hildebrandt, B. van Dongen, M. Röglinger, J. Mendling. Vol. 11675. Springer, Cham, 2019 (Cited on p. 48).

[Mas11] M. Masse. *REST API Design Rulebook*. O’Reilly Media, Inc., 2011 (Cited on p. 183).

[MMC+12] D. Molnar, R. Mukherjee, A. Choudhury, A. Mora, P. Binkele, M. Selzer, B. Nestler, S. Schmauder. “Multiscale Simulations on the Coarsening of Cu-rich Precipitates in a-Fe Using Kinetic Monte Carlo, Molecular Dynamics and Phase-field Simulations.” In: *Acta Materialia* 60.20 (2012), pp. 6961–6971 (Cited on p. 17).

[MPB+13] A. Meyer, L. Pufahl, K. Batoulis, S. Kruse, T. Lindhauer, T. Stoff, D. Fahland, M. Weske. *Data Perspective in Process Choreographies: Modeling and Execution*. Tech. rep. Tech. Rep. BPM-13-29, BPMcenter. org, 2013 (Cited on p. 37, 48).

[MPB+15] A. Meyer, L. Pufahl, K. Batoulis, D. Fahland, M. Weske. “Automating Data Exchange in Process Choreographies.” In: *Information Systems* (2015) (Cited on p. 15, 16, 36, 37, 70).

[MSDC12] D. Miorandi, S. Sicari, F. De Pellegrini, I. Chlamtac. “Internet of things: Vision, applications and research challenges.” In: *Ad Hoc Networks* (2012) (Cited on p. 13).

[MSL12] G. Monsieur, M. Snoeck, W. Lemahieu. “Managing data dependencies in service compositions.” In: *Journal of Systems and Software* 85.11 (2012), pp. 2604–2628 (Cited on p. 43).

[MSMP11] S. Meyer, K. Sperner, C. Magerkurth, J. Pasquier. “Towards Modeling Real-world Aware Business Processes.” In: *Proceedings of WoT’11*. ACM, 2011, pp. 1–6 (Cited on p. 15, 46).

[MSW11] A. Meyer, S. Smirnov, M. Weske. *Data in Business Processes*. 50. Universitätsverlag Potsdam, 2011 (Cited on p. 55, 57).

[NC03] A. Nigam, N. S. Caswell. "Business artifacts: An approach to operational specification." In: *IBM Systems Journal* (2003) (Cited on p. 47).

[OAF+04] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, et al. "Taverna: a tool for the composition and enactment of bioinformatics workflows." In: *Bioinformatics* 20.17 (2004), pp. 3045–3054 (Cited on p. 15).

[Ott78] K. J. Ottenstein. "Data-flow Graphs As an Intermediate Program Form." PhD thesis. Purdue University, 1978 (Cited on p. 74).

[Pap03] M. P. Papazoglou. "Service-oriented computing: concepts, characteristics and directions." In: *Proceedings of WISE'03*. IEEE. 2003, pp. 3–12 (Cited on p. 13, 30).

[Pel03] C. Peltz. "Web services orchestration and choreography." In: *Computer* 36.10 (Oct. 2003), pp. 46–52 (Cited on p. 30).

[QLDG09] L. Qian, Z. Luo, Y. Du, L. Guo. "Cloud Computing: An Overview." In: *Cloud Computing*. Ed. by M. G. Jaatun, G. Zhao, C. Rong. Springer Berlin Heidelberg, 2009 (Cited on p. 13).

[Rei17] P. Reimann. "Data provisioning in simulation workflows." PhD thesis. University of Stuttgart, 2017 (Cited on p. 37, 260).

[RK+08] P. Reimann, O. Kopp, et al. *Generating WS-BPEL 2.0 Processes from a Grounded BPEL4Chor Choreography*. Tech. rep. 2008/07. University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2008 (Cited on p. 36, 73, 230, 248).

[RRS+11] P. Reimann, M. Reiter, H. Schwarz, D. Karastoyanova, F. Leymann. "SIMPL-A Framework for Accessing External Data in Simulation Workflows." In: *Proceedings of BTW'11*. Vol. 11. Citeseer. 2011, pp. 534–553 (Cited on p. 37, 260).

[SA14] O. Sukhoroslov, A. Afanasiev. "Everest: A Cloud Platform for Computational Web Services." In: *Proceedings of CLOSER'14*. 2014 (Cited on p. 50).

[SASL13] S. Strauch, V. Andrikopoulos, S. G. Sáez, F. Leymann. "ESB^{MT}: A Multi-tenant Aware Enterprise Service Bus." In: *International Journal of Next-Generation Computing* 4.3 (2013) (Cited on p. 249).

[SHK+11] M. Sonntag, S. Hotta, D. Karastoyanova, D. Molnar, S. Schmauder. "Using Services and Service Compositions to Enable the Distributed Execution of Legacy Simulation Applications." In: *Proceedings of ServiceWave'11*. Springer, 2011, pp. 1–12 (Cited on p. 18, 239).

[SK10] M. Sonntag, D. Karastoyanova. "Next Generation Interactive Scientific Experimenting Based On The Workflow Technology." In: *Proceedings of MS'10*. Ed. by R. S. Alhajj, V. C. M. Leung, M. Saif, R. Thring. ACTA Press, July 2010 (Cited on p. 15).

[SK11] M. Sonntag, D. Karastoyanova. "Enforcing the Repeated Execution of Logic in Workflows." In: *Proceedings of BUSTECH'11*. IARIA, Sept. 2011, pp. 1–6 (Cited on p. 15).

[SK13] M. Sonntag, D. Karastoyanova. "Model-as-you-go: An Approach for an Advanced Infrastructure for Scientific Workflows." In: *Journal of Grid Computing* 11.3 (Sept. 2013), pp. 553–583 (Cited on p. 15).

[SKD10] M. Sonntag, D. Karastoyanova, E. Deelman. "Bridging The Gap Between Business And Scientific Workflows." In: *Proceedings of e-Science'10*. IEEE Computer Society, Dec. 2010, pp. 206–213 (Cited on p. 15).

[Slo07] A. Slominski. "Workflows for e-Science: Scientific Workflows for Grids." In: Springer London, 2007. Chap. Adapting BPEL to Scientific Workflows, pp. 208–226 (Cited on p. 15).

[SMM+14] R. Schmidt, M. Möhring, S. Maier, J. Pietsch, R.-C. Härting. "Big Data as Strategic Enabler - Insights from Central European Enterprises." In: *Business Information Systems*. Vol. 176. Lecture Notes in Business Information Processing. Springer International Publishing, 2014, pp. 50–60 (Cited on p. 15).

[Sne06] H. M. Sneed. "Integrating legacy software into a service oriented architecture." In: *Proceedings of CSMR'06*. 2006 (Cited on p. 50).

[Son16] M. Sonntag. “Model-as-you-go - ein Ansatz zur flexiblen Entwicklung von wissenschaftlichen Workflows.” Dissertation. University of Stuttgart, 2016 (Cited on p. 228).

[SSJS20] C. Sturm, J. Szalanczi, S. Jablonski, S. Schönig. “Proceedings of PoEM’20.” In: *The Practice of Enterprise Modeling*. Springer International Publishing, 2020, pp. 261–276 (Cited on p. 48).

[Ste08] T. Steinmetz. “Ein Event-Modell für WS-BPEL 2.0 und dessen Realisierung in Apache ODE.” MA thesis. University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2008 (Cited on p. 189).

[SW22] F. Stiehle, I. Weber. *Blockchain for Business Process Enactment: A Taxonomy and Systematic Literature Review*. 2022. URL: https://arxiv.org/abs/2206.03237 (Cited on p. 49).

[TDGS07] I. J. Taylor, E. Deelman, D. B. Gannon, M. Shields, eds. *Workflows for e-Science: Scientific Workflows for Grids*. Springer London, 2007 (Cited on p. 13).

[TOSCA] OASIS. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*. OASIS Standard. Nov. 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html (Cited on p. 202, 233).

[TSWH07] I. Taylor, M. Shields, I. Wang, A. Harrison. “The Triana Workflow Environment: Architecture and Applications.” In: *Workflows for e-Science*. Springer, 2007, pp. 320–339 (Cited on p. 15).

[UML] Object Management Group (OMG). *Unified Modeling Language (UML) Version 2.5.1*. Object Management Group, Inc. Dec. 2017. URL: https://www.omg.org/spec/UML/ (Cited on p. 179).

[Vuk+16] K. Vukojevic-Haupt et al. “On-demand provisioning of workflow middleware and services into the cloud: an overview.” In: *Computing* (Oct. 2016) (Cited on p. 78).

[WAG+13] A. Weiß, V. Andrikopoulos, S. Gómez Sáez, D. Karastoyanova, K. Vukojevic-Haupt. *Modeling Choreographies using the BPEL4Chor Designer: an Evaluation Based on Case Studies*. Tech. rep. 2013/03. University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2013, p. 23 (Cited on p. 229).

[WAHK15a] A. Weiß, V. Andrikopoulos, M. Hahn, D. Karastoyanova. "Fostering Reuse in Choreography Modeling Through Choreography Fragments." In: *Proceedings of ICEIS'15*. SciTePress, Apr. 2015, pp. 28–36 (Cited on p. 64, 69).

[WAHK15b] A. Weiß, V. Andrikopoulos, M. Hahn, D. Karastoyanova. "Rewinding and Repeating Scientific Choreographies." In: *Proceedings of OTM 2015 Conferences*. Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2015, pp. 337–347 (Cited on p. 15).

[WAHK17] A. Weiß, V. Andrikopoulos, M. Hahn, D. Karastoyanova. "Model-as-you-go for Choreographies: Rewinding and Repeating Scientific Choreographies." In: *IEEE Transactions on Services Computing* PP.99 (2017) (Cited on p. 84).

[WEB+07] B. Wassermann, W. Emmerich, B. Butchart, N. Cameron, L. Chen, J. Patel. "Workflows for e-Science: Scientific Workflows for Grids." In: Springer London, 2007. Chap. Sedna: A BPEL-Based Environment for Visual Scientific Workflow Modeling, pp. 428–449 (Cited on p. 15).

[Wei+16] A. Weiß et al. "ChorSystem: A Message-Based System for the Life Cycle Management of Choreographies." In: *Proceedings of OTM 2016 Conferences*. Ed. by C. Debruyne, H. Panetto, R. Meersman, T. Dillon, E. Kühn, D. O'Sullivan, C. A. Ardagna. Springer International Publishing, Oct. 2016, pp. 503–521 (Cited on p. 260).

[Wei18] A. Weiß. "Flexible modeling and execution of choreographies." Dissertation. University of Stuttgart, 2018 (Cited on p. 84, 228, 260).

[Wes12] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer Science & Business Media, 2012 (Cited on p. 13, 26, 53, 67, 256).

[Wet+15] J. Wettinger et al. “Streamlining APIfication by Generating APIs for Diverse Executables Using Any2API.” In: *Proceedings of CLOSER'15*. 2015 (Cited on p. 51).

[WK14a] A. Weiß, D. Karastoyanova. “A Life Cycle for Coupled Multi-scale, Multi-field Experiments Realized through Choreographies.” In: *Proceedings of EDOC'14*. Sept. 2014, pp. 234–241 (Cited on p. 15, 67).

[WK14b] A. Weiß, D. Karastoyanova. “Enabling Coupled Multi-scale, Multi-field Experiments Through Choreographies of Data-Driven Scientific Simulations.” In: *Computing* (2014), pp. 1–29 (Cited on p. 15, 20).

[WKK+10] B. Wetzstein, D. Karastoyanova, O. Kopp, F. Leymann, D. Zwink. “Cross-Organizational Process Monitoring Based on Service Choreographies.” In: *Proceedings of SAC'10*. ACM, 2010, pp. 2485–2490 (Cited on p. 81).

[WKMS14] A. Weiß, D. Karastoyanova, D. Molnar, S. Schmauder. “Coupling of Existing Simulations using Bottom-up Modeling of Choreographies.” In: *Proceedings of SimTech@GI workshop at INFORMATIK'14*. Gesellschaft für Informatik e.V. (GI), Sept. 2014, pp. 101–112 (Cited on p. 15, 17).

[WS-CDL] World Wide Web Consortium (W3C). *Web Services Choreography Description Language (WS-CDL) Version 1.0*. W3C Candidate Recommendation. Nov. 2005. URL: https://www.w3.org/TR/ws-cdl-10/ (Cited on p. 33, 70).

[WSDL] World Wide Web Consortium (W3C). *W3C Web Services Description Language (WSDL) 1.1*. W3C Recommendation. Apr. 2001. URL: https://www.w3.org/TR/2001/NOTE-wsdl-20010315 (Cited on p. 129).

[XML] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Online. 2008. URL: https://www.w3.org/TR/xml/ (Cited on p. 176, 207).

[XML-NS] World Wide Web Consortium (W3C). *Namespaces in XML 1.0 (Third Edition)*. Online. 2009. URL: https://www.w3.org/TR/xml-names/ (Cited on p. 174).

[XPath] World Wide Web Consortium (W3C). *XML Path Language (XPath) 3.1*. Online. 2017. URL: https://www.w3.org/TR/xpath-31/ (Cited on p. 49, 175).

[XQuery] World Wide Web Consortium (W3C). *XQuery 3.1: An XML Query Language*. Online. 2017. URL: https://www.w3.org/TR/xquery-31/ (Cited on p. 37, 49, 192).

[XSD1] World Wide Web Consortium (W3C). *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. W3C Recommendation. Apr. 2012. URL: https://www.w3.org/TR/xmlschema11-1/ (Cited on p. 60, 176).

[XSD2] World Wide Web Consortium (W3C). *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. W3C Recommendation. Apr. 2012. URL: https://www.w3.org/TR/xmlschema11-2/ (Cited on p. 98).

[XSLT] World Wide Web Consortium (W3C). *XSL Transformations (XSLT) Version 3.0*. Online. 2017. URL: https://www.w3.org/TR/xslt-30/ (Cited on p. 49, 192).

[YAML] *YAML: YAML Ain't Markup Language*. Online. URL: https://yaml.org/ (Cited on p. 177).

[YB05] J. Yu, R. Buyya. "A taxonomy of scientific workflow systems for grid computing." In: *ACM Sigmod Record* 34.3 (2005), pp. 44–49 (Cited on p. 15).

[ZBDH06] J. M. Zaha, A. Barros, M. Dumas, A. ter Hofstede. "Let's Dance: A Language for Service Behavior Modeling." In: *Proceedings of OTM 2006 Conferences*. Vol. 4275. LNCS. Springer Berlin Heidelberg, 2006, pp. 145–162 (Cited on p. 33, 70).

[ZDH+08] J. M. Zaha, M. Dumas, A. H. M. ter Hofstede, A. Barros, G. Decker. "Bridging Global and Local Models of Service-Oriented Systems." In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews* 38.3 (2008), pp. 302–318 (Cited on p. 33).

[Zdu02] U. Zdun. "Reengineering to the web: A reference architecture." In: *Proceedings of CSMR'02*. 2002 (Cited on p. 50).

[Zim16] O. Zimmermann. “Microservices tenets.” In: *Computer Science - Research and Development* (2016), pp. 1–10 (Cited on p. 13).

All links were last followed on 22nd February 2023.

List of Figures

List of Definitions

List of Algorithms

List of Symbols

Ψ	Map of an activity's implementation, see Definition 4.9.
$\overrightarrow{\Delta_C}$	Map of choreography data connectors, see Definition 4.18.
$G_{C_{DDG}}$	Choreography data dependency graph (CDDG), see Definition 4.26.
ι_C	Input data container in a choreography, see Definition 4.3.
C_{DM}	Choreography data model (CDM), see Definition 4.23.
o_C	Output data container in a choreography, see Definition 4.3.
$\overrightarrow{\Delta_{\mathscr{M}}}$	Map of choreography message connectors, see Definition 4.14.
$\mathscr{C}$	Choreography model.
G_C	CM-Graph.
$\mathfrak{C}$	Set of all choreography models.
E	Set of control connectors, see Definition 4.12.
Δ_{CS}	Map of correlations between a communication activity and correlation sets, see Definition 4.21.
Δ_{CP}	Map of correlation properties of a correlation set to a data element of a data object or message, see Definition 4.20.
$CS(V)$	Set of all correlation sets, see Definition 4.19.

Δ_C	Map of data connectors between two model elements, see Definition 4.17.
Δ	Map of data connectors within PM-Graphs, see Definition 4.29.
$\rho_{\mathscr{D}}$	Map of deletion strategies of a data object, see Definition 4.7.
$\mu_{\mathscr{D}}$	Map of multiplicity of a data object, see Definition 4.6.
$\mathscr{D}(V)$	Set of all data objects, see Definition 4.5.
Ψ_τ	Map of a data transformation's implementation, see Definition 6.2.
$\mathcal{E}_\tau$	Set of all data transformation implementations, see Definition 6.1.
χ_τ	Map of the input parameters of a data transformation, see Definition 6.3.
τ	Set of data transformations, see Definition 6.1.
$\Delta_{\mathscr{M}}$	Map of message connectors between two communication activities, see Definition 4.13.
$\mathscr{M}(V)$	Set of all messages, see Definition 4.4.
$\mathscr{P}(N)$	Set of all participants, see Definition 4.11.
$\overrightarrow{\Delta}$	Map of process data connectors within PM-Graphs, see Definition 4.29.
ι	Input data container in a process, see Definition 4.29.
o	Output data container in a process, see Definition 4.29.
G	Process model graph (PM-Graph), see Definition 4.29.
N	Set of activities.
$\mathcal{H}_C$	Set of all activities, participants and data-aware choreography models, see Section 4.2.1.3.
$\mathcal{E}$	Set of all activity implementations, see Definition 4.9.
N_{com}	Set of communication activities, see Definition 4.10.
$\mathcal{C}$	Set of all conditions.

$\boldsymbol{E}_{\Delta_C}$	Set of all data connectors of a choreography model, see Definition 4.17.
V	Set of all data elements, see Definition 4.1.
$\boldsymbol{E}_{\Delta_{\mathcal{M}}}$	Set of all message connectors of a choreography model, see Definition 4.13.
$\mathcal{E}_{receive}$	Set of all message receiving activity implementations, see Definition 4.10.
$\mathcal{E}_{send}$	Set of all message sending activity implementations, see Definition 4.10.
$N_{receive}$	Set of receiving communication activities, see Definition 4.10.
N_{send}	Set of sending communication activities, see Definition 4.10.
N'	Set of choreography start activities, see Definition 4.15.
Π	Set of data consumers and producers, see Definition 4.24.
$E_{C_{DDG}}$	Set of data dependency edges, see Definition 4.25.
$\hat{V}$	Set of all data elements associated to a data dependency node, see Definition 4.24.
$\mathcal{N}$	Set of data dependency nodes, see Definition 4.24.
V_{Π}	Set of all data elements being referenced as source or target of a data connector map between an input or output data container of a data processor and a cross-partner data object, see Definition 4.24.
$V_{\mathcal{D}}$	Set of all data elements used as part of a data object definition, see Definition 4.23.
$\mathbf{P}$	Set of data object deletion strategies, see Definition 4.7.
$\mathbf{P}_M$	Set of data object deletion strategy names, see Definition 4.7.
$\mathbf{P}_V$	Set of data object deletion strategy values, see Definition 4.7.
F	Set of fault handling strategies, see Definition 4.27.
M	Set of all names.
$N'(R)$	Set of participant start activities, see Definition 4.16.
Σ	Set of all staging elements, see Definition 4.27.
Σ_{Ψ}	Set of staging methods, see Definition 4.27.
S	Set of all structures.

$\mathcal{C}_T$	Set of trigger conditions, see Definition 4.27.
σ	Map of staging elements, see Definition 4.28.
$DOM(v)$	Domain of well-formed values of a data element, see Definition 4.2.

Acronyms

API	Application Programming Interface.
BPE	Business Process Engine.
BPEL	Business Process Execution Language.
BPM	Business Process Management.
BPMN	Business Process Model and Notation.
CDDG	Choreography Data Dependency Graph.
CDM	Choreography Data Model.
CLI	Command Line Interface.
CM-Graph	Choreography Model Graph.
DT	Data Transformation.
ESB	Enterprise Service Bus.
ETL	Extract Transform Load.
HTTP	Hypertext Transfer Protocol.
IoT	Internet of Things.
JSON	JavaScript Object Notation.

KMC	Kinetic Monte Carlo.
KPI	Key Performance Indicator.
ODE	Apache Orchestration Director Engine.
OPAL	Ostwald ripening of Precipitates on an Atomic Lattice.
OS	Operating System.
PM-Graph	Process Model Graph.
QName	Unique fully-qualified name.
REST	Representational State Transfer.
SMT	Satisfiability Modulo Theories.
SOA	Service-oriented Architectures.
sWfMS	Scientific Workflow Management System.
TDT	TraDE Data Transformation.
TQL	TraDE Query Language.
TraDE	Transparent Data Exchange.
UML	Unified Modeling Language.
URI	Uniform Resource Identifier.
URL	Uniform Resource Locator.
WS-CDL	Web Service Choreography Description Language.
WSDL	Web Service Description Language.
XML	eXtensible Markup Language.
XPath	XML Path Language.
XQuery	XML Query Language.
XSD	XML Schema Definition.
XSLT	Extensible Stylesheet Language Transformations.
YAML	YAML Ain't Markup Language.