A Framework for Model-Driven Scientific Workflow Engineering

Dipl.-Inform. Guido Scherp

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2013

Kiel Computer Science Series (KCSS) 2013/2 v1.0 dated 2013-09-24

ISSN 2193-6781 (print version)
ISSN 2194-6639 (electronic version)

Electronic version, updates, errata available via https://www.informatik.uni-kiel.de/kcss

The author can be contacted via http://www.scherp.net

Published by the Department of Computer Science, Kiel University

Software Engineering Group

Please cite as:

▻ Guido Scherp. *A Framework for Model-Driven Scientific Workflow Engineering*. Number 2013/2 in Kiel Computer Science Series. Department of Computer Science, 2013. Dissertation, Faculty of Engineering, Kiel University.

```
@book{Scherp2013,
author = {Guido Scherp},
title = {A Framework for Model-Driven Scientific Workflow Engineering},
publisher = {Department of Computer Science, CAU Kiel},
year = {2013},
number = {2013/2},
isbn = {9783732279869},
series = {Kiel Computer Science Series},
note = {Dissertation, Faculty of Engineering, Kiel University.}
}
```

© 2013 by Guido Scherp

About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at Kiel University. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit `http://www.informatik.uni-kiel.de/kcss` for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

1. Gutachter:	Prof. Dr. Wilhelm Hasselbring Christian-Albrechts-Universität Kiel
2. Gutachter:	Prof. Dr. Odej Kao Technische Universität Berlin

Datum der mündlichen Prüfung: 25. Juni 2013

Zusammenfassung

So genannte *Scientific Workflows* werden zunehmend im Kontext datenintensiver Wissenschaften eingesetzt, um komplexe Verarbeitungen von Forschungsdaten effizient und zuverlässig in verteilten Infrastrukturen wie Grids auszuführen. *Scientific Workflow Management Systeme* (SWfMS) unterstützen Wissenschaftler in der Modellierung und Ausführung von Scientific Workflows, wobei zwischen der Modellierung durch einen Wissenschaftler auf der *domänenspezifischen Ebene* und der automatisierten Ausführung auf der *technischen Ebene* unterschieden werden kann. Erste SWfMS wurden von Grund auf neu entwickelt inklusive entsprechender Workflow-Technologien und -Sprachen. Bereits existierende und etablierte *Business Workflow*-Technologien aus dem betrieblichen Bereich wurden ursprünglich nicht genutzt, beispielsweise weil Scientific und Business Workflows unterschiedliche Lebenszyklen abbilden und auf Grund inkompatibler Schnittstellen und Kommunikationsprotokolle der jeweiligen Infrastrukturen.

Im Zuge der Etablierung von Service-orientierten Architekturen (SOAs) in betrieblichen IT-Infrastrukturen wurden zahlreiche Web Service-Standards und entsprechende Technologien entwickelt. Die *Web Services Business Process Execution Language* (BPEL) ist beispielsweise ein Standard für die Implementierung und Ausführung von Business Workflows in einer SOA. So genannte *Service Grids* haben das SOA-Architekturmuster für wissenschaftliche IT-Infrastrukturen übernommen und nutzen dabei die bereits existierenden Standards und Technologien. Somit ist BPEL generell auch für die Ausführung von Scientific Workflows auf der technischen Ebene geeignet, was bereits in zahlreichen Projekten und Publikationen gezeigt wurde. Allerdings ist BPEL eine Workflow-Sprache für IT-Experten und kann in der Form nicht zur Modellierung eines Scientific Workflows durch einen Wissenschaftler auf der domänenspezifischen Ebene genutzt werden. Es fehlen eine geeignete Abstraktion für BPEL, die speziell für den Einsatz auf der domänenspezifischen Ebene von Scientific Workflows zugeschnitten

ist, sowie eine passende Abbildung auf die technische Ebene.

Diese Herausforderungen der domänenspezifischen Abstraktion und der Abbildung werden in der vorliegenden Dissertation mit Hilfe der *Business Process Model and Notation* (BPMN) und Techniken aus der *modellgetriebenen Softwareentwicklung* adressiert. Dazu wird mit *MoDFlow* ein modellgetriebener Ansatz vorgestellt, um domänenspezifische Scientific Workflow-Modelle über eine auf BPMN basierende Zwischenschicht in eine technisch ausführbare Form zu überführen. Die Zwischenschicht basiert auf MoDFlow.BPMN, was eine Untermenge von BPMN definiert mit eigenen Erweiterungen für die wissenschaftliche Domäne. MoDFlow.BPMN2BPEL beschreibt drei aufeinanderfolgende Transformationsschritte zur Abbildung von MoDFlow.BPMN nach BPEL auf der technischen Ebene. Zudem werden in MoDFlow mehrere Möglichkeiten beschrieben, um MoDFlow.BPMN und MoDFlow.BPMN2BPEL zu nutzen und zu erweitern. Ein Schwerpunkt liegt dabei auf der Erstellung von so genannten domänenspezifischen Sprachen (DSLs) zur Modellierung von Scientific Workflows auf der domänenspezifischen Ebene. Mit dem *MoDFlow-Framework* wird eine Implementierung des Ansatzes bereitgestellt, die auf dem Eclipse Modeling Framework (EMF) aufsetzt.

Das MoDFlow-Framework wird in drei Anwendungsszenarien evaluiert, wobei unterschiedliche Nutzungsmöglichkeiten angewendet werden. Die ersten beiden Szenarien definieren Parameterraumexplorationen in Scientific Workflows und führen diese in einer Grid-Infrastruktur aus. Damit wird die technische Realisierbarkeit des Ansatzes gezeigt. Das dritte Szenario ist eine Kooperation mit dem Projekt PubFlow, in dem eine Infrastruktur zur Erstellung und Ausführung von Scientific Workflows für Datenpublikationen aufgebaut wird. Mit Hilfe des Frameworks Xtext wird eine textuelle DSL erstellt, die Entwickler beim Umgang mit Workflow-Technologien unterstützt. Dies Szenario zeigt die praktische Nutzbarkeit des MoDFlow-Frameworks. PubFlow plant im nächsten Schritt eine grafische DSL basierend auf der Notation von BPMN sowie einen entsprechenden Editor speziell für Wissenschaftler zu erstellen.

Abstract

So-called *scientific workflows* are one important means in the context of data-intensive science for reliable and efficient scientific data processing in distributed computing infrastructures such as Grids. *Scientific Workflow Management Systems* (SWfMS) help scientists model and run scientific workflows, whereas a domain-specific layer for workflow modeling by a scientist and a technical layer for automated workflow execution can be distinguished. Initially, many SWfMS were developed from scratch using custom workflow technologies languages without application of already existing and established business workflow technologies. Among the reasons were different life cycles for scientific and business workflows as well as incompatible interfaces and communication protocols of the respective execution infrastructures.

Meanwhile, several business IT infrastructures have evolved to service-oriented architectures (SOAs), for which many Web service standards and technologies have been developed. The *Web Services Business Process Execution Language* (BPEL), for example, is a well-accepted standard for the implementation and execution of business workflows in SOAs. The SOA architecture pattern has been adopted in scientific IT infrastructures by so-called *Service Grids* based on existing standards and technologies. Due to this development, BPEL is also suitable for the execution of scientific workflows at the technical layer, which has been elaborated on in many publications and projects. However, BPEL is a workflow language for IT experts and is originally not suited for scientific workflow modeling by a scientist at the domain-specific layer. A domain-specific abstraction of BPEL is therefore required that can be specifically tailored for scientific workflow modeling as well as a corresponding mapping to the technical layer.

These challenges of the domain-specific abstraction and the mapping are addressed in this thesis with the help of the *Business Process Model and Notation* (BPMN) standard and technologies from *Model-Driven Software*

Development (MDSD). Therefore, the *MoDFlow approach* for ***Mo**del-**D**riven Scientific Work**Flow** Engineering* is presented to map domain-specific scientific workflow models via a BPMN-based intermediate layer to an executable workflow model. The intermediate layer is specified by *MoDFlow.BPMN*, which is a BPMN metamodel subset with custom extensions for the scientific domain. *MoDFlow.BPMN2BPEL* defines three consecutive transformation steps to map MoDFlow.BPMN to BPEL for workflow execution. Furthermore, different methods to utilize and extend MoDFlow.BPMN and MoDFlow.BPMN2BPEL are described in the MoDFlow approach, in which the definition of so-called *domain-specific languages* (DSLs) for the modeling of scientific workflows at the domain-specific layer is focused. The *MoDFlow framework* is an implementation of the MoDFlow approach, which is based on the *Eclipse Modeling Framework* (EMF).

The MoDFlow framework is evaluated in *three application scenarios*, in which different utilization and extension mechanisms are examined. The first two application scenarios investigate the technical feasibility of the approach and support scientific workflows with parameter sweeps that are executed on a Grid infrastructure. The third application scenario has been conducted in collaboration with the PubFlow project, which aims to create an infrastructure to model and execute data publication workflows. Based on the Xtext framework, a textual DSL and a corresponding language infrastructure is defined for this purpose that supports developers in creating data publication workflows. This scenario aims to illustrate the practicability of the MoDFlow framework. PubFlow currently plans to implement an additional graphical DSL based on the BPMN notation and a corresponding workflow editor for scientists.

Preface

by Prof. Dr. Wilhelm Hasselbring

Work with scientific data is carried out in particular steps which as a whole constitute a workflow. These workflows may be executed manually (by scientists) or automatically via some workflow engine. In scientific computing, workflows are usually explorative whereby modeling and implementation of the workflows as well as the actual execution are performed by the same person, a researcher. Conversely, in industrial praxis these roles for modeling and implementation of the workflows are taken by different persons than roles for the actual workflow execution. To support these various roles, established and standardized middleware systems are available for automatic workflow execution.

The basic idea of the present Ph.D. thesis is to open up these established industrial technologies for scientific work. The approach is to provide domain-specific workflow modeling languages that are tailor-made for the scientific domain at hand. With model-driven techniques, these domain-specific specifications are then automatically transformed to industry-proven middleware platforms. This approach allows to effectively realize new role models in scientific practice. For instance, modeling and implementation of workflows may be done by IT staff, while the workflows are carried out by scientists and data managers.

Guido Scherp presents a new, innovative framework for model-driven engineering of scientific workflows on Grid computing platforms. The technical design and the implementation re-uses and integrates many software components and frameworks from various domains and sources. The re-use of such powerful components and frameworks relieves from building the respective functions, but imposes the challenge to check their fitness for purpose and to integrate diverse architectural styles into a coherent whole. The implementation as realized in this thesis constitutes a remarkable engineering achievement. Besides the conceptual and the technical design, this engineering thesis provides an extensive experimental evaluation with

partners from the excellence cluster Future Ocean and the PubFlow project.

If you are interested in model-driven workflow engineering, this is a recommended reading for you.

Wilhelm Hasselbring
Kiel, July 2013

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 The Fourth Research Paradigm and Scientific Workflows

In 2007, Jim Gray envisioned in a talk [Gray 2007] that data-explorative *e-Science* creates a fourth research paradigm besides empirical, theoretical, and computational science. Today, this is also known as *data-intensive science* [Hey et al. 2009]. It represents a data-centric scientific process that generally consists of data capture, data curation, and data analysis in order to gain scientific knowledge from research data. Besides the processing of research data in corresponding infrastructures, it also focuses on the interdisciplinary collaboration between scientists across organizational and national boundaries. Hey et al. [2012] recently emphasized the relevance of data-intensive science, especially for coping with current and future challenges of the so-called *data deluge* in research.

In order to enable the fast and scalable processing of often large amounts of research data, scientists need appropriate tools and infrastructures such as *Grids*. A Grid is a federated distributed computing infrastructure with numerous shared resources [Foster 2002], which can be allocated and used on demand for so-called *Grid computing* [Kesselman and Foster 1998]. Many Grid infrastructures have been built in national and transnational Grid initiatives such as TeraGrid (US)[1], OMII-UK (UK)[2], EGEE (EU)[3], and D-Grid[4].

[1] http://www.teragrid.org
[2] http://www.omii.ac.uk/
[3] http://public.eu-egee.org/
[4] http://www.d-grid.de

Due to the technical complexity of Grids and other distributed computing infrastructures, appropriate tools are needed to facilitate their use by scientists. *Scientific workflows* are one important means to support and automate scientific data processing in such infrastructures [Taylor et al. 2006; Goble and Roure 2009]. A scientific workflow is created by a scientist and it generally defines the execution order of multiple computational tasks (workflow activities) for data processing as process flow (workflow).

Ludäscher et al. [2009] describe the objectives of scientific workflows as:

> "The main goals of scientific workflows, then, are (i) to save 'human cycles' by enabling scientists to focus on domain-specific (science) aspects of their work, rather than dealing with complex data management and software issues; and (ii) to save machine cycles by optimizing workflow execution on available resources." [Ludäscher et al. 2009]

This description emphasizes two main aspects of scientific workflows: (1) the domain-specific design of scientific workflows by scientists and (2) their optimized technical execution in a suitable infrastructure. We thus distinguish between a domain-specific and a technical layer, see Figure 1.1.

The design of a scientific workflow by a scientist is assigned to the domain-specific layer. It provides an abstract and usually graphical notation for workflow modeling that is understood by scientists and often represents one scientific domain. Therefore, a corresponding workflow editor usually provides a repository of predefined and domain-specific workflow activities. Technical details regarding the execution infrastructure are usually hidden during workflow modeling.

After the design of a scientific workflow, it is (automatically) mapped to the technical layer for execution. This mapping includes, for example, an optimized resource selection or an enrichment with additional technical details for certain execution infrastructures such as Grids. The mapped scientific workflow is then executed by a workflow engine, which coordinates the execution of all workflow activities on the selected resources and all necessary data transfers. The scientist is informed about the workflow execution state and can intervene when necessary, e.g., if intermediate

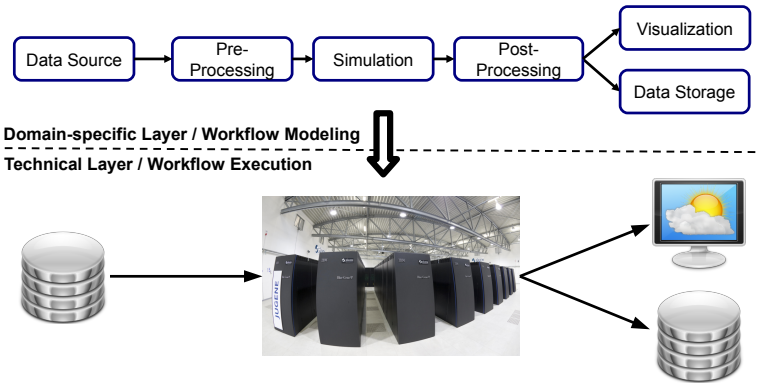

Figure 1.1. Layers for scientific workflows[a]

[a]Picture credits: `http://www.iconarchive.com/show/oxygen-icons-by-oxygen-icons.org/Places-server-database-icon.html` (database icon), `http://www.iconarchive.com/show/oxygen-icons-by-oxygen-icons.org/Apps-preferences-desktop-display-icon.html` (monitor icon), `http://www.iconarchive.com/show/oxygen-icons-by-oxygen-icons.org/Status-weather-clouds-icon.html` (weather icon), and `http://www.fz-juelich.de` (Jugene supercomputer image)

results do not look promising or in case of failures that cannot be handled automatically.

1.2 Business Workflow Technologies for Scientific Workflows

So-called *Scientific Workflow Management Systems* (SWfMS) [Lin et al. 2009] help scientists to create, run, and monitor scientific workflows and to analyze their results. When scientific workflows emerged in the context of data-intensive science, many SWfMS were developed from scratch including custom scientific workflow languages. The application of existing and established business workflow technologies, which had emerged approximately ten years before, was not considered for many years. Some reasons were the different life cycles and modeling approaches, but also incompatibilities between the execution infrastructures for business workflows (information

systems) and scientific workflows (Grid resources), e.g. regarding communication protocols. Thus, the effort was too high to utilize business workflow technologies in the scientific workflow domain.

Meanwhile, several business IT infrastructures have evolved to service-oriented architectures (SOA), for which many Web service standards[5,6] have been developed such as WSDL and SOAP. Such a standardization process has also been started for the business workflow domain, which is partly associated with SOA and corresponding standards. The *Web Services Business Process Execution Language* (BPEL) [OASIS 2007], for example, is a well accepted standard for the implementation and execution of business workflows in service-oriented execution infrastructures that are based on Web services. Business workflows that are implemented with BPEL are technically executed as *(Web) service orchestrations*.

Grid infrastructures have also adopted the SOA pattern [Foster et al. 2002] and a standardization process for Grids has been started that is driven by initiatives such as the *Open Grid Forum* (OGF)[7]. Many existing Web service standards has been reused and also the development of new standards has been fostered, which has led to a close collaboration between the Grid and Web service community. The *Web Service Resource Framework* (WSRF) [OASIS 2006] standard for stateful Web services, for example, is an important contribution by the OGF. WSRF is implemented by state-of-the-art Grid middlewares such as Globus Toolkit[8] and UNICORE[9].

Due to this development, business and scientific workflows are executed in service-oriented execution infrastructures that are based on Web services and use SOAP as communication protocol. Thus, business workflow languages such as BPEL are also suitable for the execution of scientific workflows from a technical point of view, which has been elaborated on in many publications and projects [Wang et al. 2005; Emmerich et al. 2005; Wassermann et al. 2007; Leymann 2006; Ezenwoye et al. 2007a; Tan et al. 2007; Dörnemann et al. 2007; Scherp et al. 2010; Görlach et al. 2011]. The utiliza-

[5] http://www.w3.org/

[6] http://www.oasis-open.org/

[7] http://www.ogf.org

[8] http://www.globus.org/toolkit/

[9] http://www.unicore.eu/

tion of existing and standardized business workflow technologies in the scientific workflow domain has generally many advantages, because several mature tools and workflow engines exist that can be reused.

Although the utilization of BPEL for scientific workflows is very reasonable, it has also some limitations [Görlach et al. 2011]. For example, BPEL is a workflow language for IT experts, but scientists usually have no comprehensive programing skills. They need a domain-specific abstraction for workflow modeling on an abstract level without any technical details and often prefer an appropriate graphical representation. Such an abstraction is not provided by the BPEL standard or corresponding tools and BPEL further does not define a graphical notation. In other words, BPEL is suited for its utilization at the technical layer of scientific workflows and not at the domain-specific layer. Thus, a domain-specific abstraction of BPEL is required that is specifically tailored for scientific workflows. Such an abstraction and differentiation between two layers further requires a mapping for scientific workflow models from the domain-specific layer to the technical layer. These challenges are addressed by this thesis.

1.3 Research Questions and Approach

The general objective of the thesis is to advance the current efforts utilizing business workflow technologies for scientific workflows. We aim at providing an extendable and customizable scientific workflow-specific abstraction for executable business workflow languages such as BPEL which itself is based on business workflow technologies. Therefore, we introduce the intermediate layer as additional layer between the domain-specific layer and technical layer. It represents a common layer for utilizing business workflow technologies in the scientific workflow domain, whereby any executable workflow language may be used at the technical layer. The domain-specific layer may be represented by existing and established scientific workflow modeling technologies. Two mappings are applied in order to transfer a created scientific workflow from the domain-specific layer to the technical layer. This starts with a mapping from the domain-specific to the intermediate layer, which is followed by a mapping from the intermediate layer to

the technical layer.

We do not intend to create a new scientific workflow language. Our focus is the representation of common scientific workflow concepts with existing and standardized business workflow technologies at the intermediate layer. Its introduction facilitates the combination of different workflow technologies and languages for workflow modeling, for example, from the scientific workflow domain, with different business workflow technologies and languages for workflow execution. Thus, the intermediate layer provides neither a graphical representation for workflow modeling nor an execution semantic for workflow enactment. We focus on the utilization of BPEL as executable business workflow language at the technical layer, but an extension to support similar workflow languages is possible. The utilization of a specific workflow language for the domain-specific layer is not planned.

To meet our objectives, we address the following research questions:

1. How can a scientific workflow-specific abstraction at the intermediate layer be realized for executable business workflow languages like BPEL?
2. How can the mappings be realized between the different layers and levels of abstraction?
3. How can the application of the scientific workflow-specific abstraction at the intermediate layer and corresponding mappings be fostered in the scientific domain?

The first question can be addressed by utilizing the *Business Process Model and Notation* (BPMN) [OMG 2011a] standard. BPMN is well-established in the business workflow domain and provides a graphical notation for the domain-specific representation of business processes and a basic mapping for a BPMN subset to BPEL. With the release of the current version 2.0, the standard was significantly extended, e.g. by a metamodel for standardized serialization and model exchange. This metamodel also provides a mechanism to define custom extension elements that can be added to any standard BPMN element. We use the BPMN metamodel to define a BPMN subset with custom metamodel extensions that represent the scientific workflow-specific abstraction at the intermediate layer. The subset

generally includes the definition of workflow activities as well as control dependencies and data dependencies between them. Thereby, we focus on compactness in order to cover common scientific workflow aspects with a few BPMN elements.

The second question can be addressed by utilizing transformation technologies from *Model-Driven Software Development* (MDSD) [Reussner and Hasselbring 2008]. Transformation languages provide means to encapsulate the domain knowledge for realizing such mappings as model transformations. They further support features like conventional programming languages, for example, to create libraries for common mapping aspects. We define a BPMN-to-BPEL mapping that maps the BPMN subset with custom extensions to BPEL within three steps. It is based on the BPEL mapping in the BPMN standard and a structure-identification algorithm for BPMN processes. We further implemented a transformation framework for the execution of transformation chains and single model transformations, which is used to realize the BPMN-to-BPEL mapping as transformation chain.

The third question can be addressed by appropriate mechanisms to utilize and extend the BPMN subset with custom extensions and the BPMN-to-BPEL mapping. Regarding utilization, we focus on two methods to integrate a scientific workflow language at the domain-specific layer. Both methods imply the creation of a corresponding mapping to the BPMN subset with custom extensions that can be realized with model transformations. The first method describes the adoption of existing scientific workflow languages. The second method describes the creation of new scientific workflow languages as *domain-specific language* (DSLs) [Fowler 2010], which are one important means in MDSD. We further describe several extension mechanisms that can be used for customizing the BPMN subset with custom extensions and the BPMN-to-BPEL transformation chain.

1.4 Contribution

In summary, the scientific contributions of this thesis are:

▻ *MoDFlow*, a conceptual approach for ***Mo**del-**D**riven Scientific Work**Flow***

Engineering. It distinguishes between a domain-specific workflow model (domain-specific layer), an intermediate workflow model (intermediate layer), an executable workflow model (technical layer), and corresponding mappings between these workflow models, see Figure 1.2. The focus is on the intermediate workflow model and its mapping to an executable workflow model. MoDFlow.BPMN defines a BPMN metamodel subset with custom extensions for representing intermediate workflow models. MoDFlow.BPMN2BPEL defines a BPMN-to-BPEL mapping that maps an intermediate workflow model based on MoDFlow.BPMN to an executable workflow model based on BPEL within three steps. The MoDFlow approach further describes different mechanisms to utilize and extend MoDFlow.BPMN and MoDFlow.BPMN2BPEL. Thereby, we focus on the definition of DSLs for creating domain-specific workflow models that are subsequently mapped to MoDFlow.BPMN.

- *MoDFlow framework*, an implementation for the MoDFlow approach. MoDFlow.BPMN is realized with Ecore models based on the Eclipse Modeling Framework (EMF)[10]. MoDFlow.BPMN2BPEL is realized as BPMN-to-BPEL transformation chain with corresponding model transformations. Therefore, a transformation framework is included in the MoDFlow framework that supports several transformation technologies for carrying out single model transformations and transformation chains on EMF models. All model transformations are implemented with Xtend[11]. The MoDFlow framework is published under the Eclipse Public License (EPL)[12] at `http://sourceforge.net/projects/bpmn2bpel/`.

- Three application scenarios that demonstrate the feasibility and practicability of MoDFlow, whereby particular utilization and extension mechanisms of MoDFlow are applied. In the first and second scenario, scientific workflows with parameter sweeps are executed in a Grid infrastructure. For the third scenario, an external textual DSL called *PubFlow.DSL* is

[10] `http://www.eclipse.org/modeling/emf/`
[11] `http://www.eclipse.org/xtend/`
[12] `http://www.eclipse.org/legal/epl-v10.html`

developed based on Xtext[13] within the PubFlow[14] project in order to support the development of data publication workflows. The *PubFlow.DSL* includes a corresponding editor and a model transformation for its mapping to MoDFlow.BPMN.

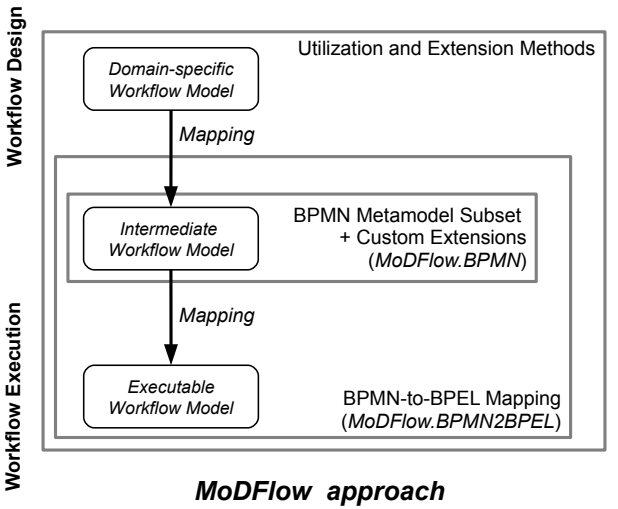

Figure 1.2. Overview of the MoDFlow approach

1.5 Structure of the Thesis

The thesis consists of four parts.

- Part I contains the foundations of the thesis, which includes the definition and introduction of relevant terms and concepts.
  - Chapter 2 describes the general concept of business workflows and introduces the business workflow languages BPMN and BPEL. It further presents different approaches for a BPMN-to-BPEL mapping.

[13] http://www.eclipse.org/Xtext/

[14] http://www.pubflow.de

- ▻ Chapter 3 describes the general concept of scientific workflows and discusses the differences between scientific and business workflows.
- ▻ Chapter 4 gives an overview on Model-Driven Software Development (MDSD) focused on model transformations and domain-specific languages (DSLs). It further introduces Xtend for implementing model transformations and Xtext for creating DSLs.
- ▻ Chapter 5 introduces the concept of Grid computing, the Grid middlewares UNICORE 6 and Globus Toolkit 4, and the Globus Security Infrastructure (GSI). It further describes the BIS-Grid Workflow Engine that provides the execution of BPEL processes in Grid infrastructures.

▻ Part II contains the description of our MoDFlow approach and the design of the MoDFlow framework.

- ▻ Chapter 6 motivates MoDFlow and gives an overview on its core components MoDFlow.BPMN and MoDFlow.BPMN2BPEL as well as its utilization and extensibility methods.
- ▻ Chapter 7 examines general requirements for SWfMS in order to define requirements for MoDFlow.
- ▻ Chapter 8 presents the definition of the BPMN subset with custom metamodel extensions for MoDFlow.BPMN.
- ▻ Chapter 9 describes the BPMN-to-BPEL mapping within three steps for MoDFlow.BPMN2BPEL.
- ▻ Chapter 10 describes different utilization and extensibility methods for MoDFlow. One main focus is the creation of DSLs for scientific workflows to utilize MoDFlow, whereby several extensibility mechanisms provided by MoDFlow.BPMN and MoDFlow.BPMN2BPEL can be applied.
- ▻ Chapter 11 defines requirements for the MoDFlow framework that consists of a transformation framework and a BPMN-to-BPEL transformation chain.
- ▻ Chapter 12 presents the design for the MoDFlow framework.

▻ Part III contains the implementation and evaluation of the MoDFlow framework.

- ▻ Chapter 13 presents the implementation of the MoDFlow framework.
- ▻ Chapter 14 presents three application scenarios that are realized with the MoDFlow framework. In the first two scenarios, scientific workflows with parameter sweeps are executed in a Grid infrastructure. In the third scenario, an external DSL is developed within the project PubFlow[15] that supports the creation of data publication workflows
- ▻ Chapter 15 discusses related work, which includes the general utilization of business workflow technologies for scientific workflows as well as approaches for a BPMN-to-BPEL mapping and transformation chaining.

▻ Part IV presents the conclusion of the thesis and gives an outlook for future work.

- ▻ Chapter 16 summarizes the thesis and discusses the central findings.
- ▻ Chapter 17 describes several possibilities to further enhance and utilize the MoDFlow approach and the MoDFlow framework in future.

[15] http://www.pubflow.uni-kiel.de/en

Part I

Foundations

Chapter 2

Business Workflows

2.1 Introduction and Basic Terminology

Workflows have their origin in the early 90's and were initially used to support and coordinate mainly human-centric *business processes* with information technology (IT). They provide a means to bridge the gap between the IT and the business domain.

Today, workflows play a central role in IT infrastructures of enterprises, especially in the context of *service-oriented architectures* (SOAs). Business processes are often completely automated as workflows and executed in SOAs that are built on Web service technology. Essential and business critical workflows in an enterprise are also called *production workflows* [Leymann and Roller 1999].

The Workflow Management Coalition (WfMC)[1] has developed a workflow reference model [Hollingsworth 1995] that defines major components and interfaces of a so-called *workflow management system* (WfMS). In addition to the workflow reference model, the WfMC also published a comprehensive document that contains a workflow terminology and glossary [WfMC 1999]. An overview of the most important terms and their relationships is given in Figure 2.1.

The WfMC defines a *business process* as follows:

> *Business Process*: "A set of one or more linked procedures or activities which collectively realise a business objective or policy goal, normally within the context of an organisational structure defining functional roles and relationships." [WfMC 1999]

[1] http://www.wfmc.org/

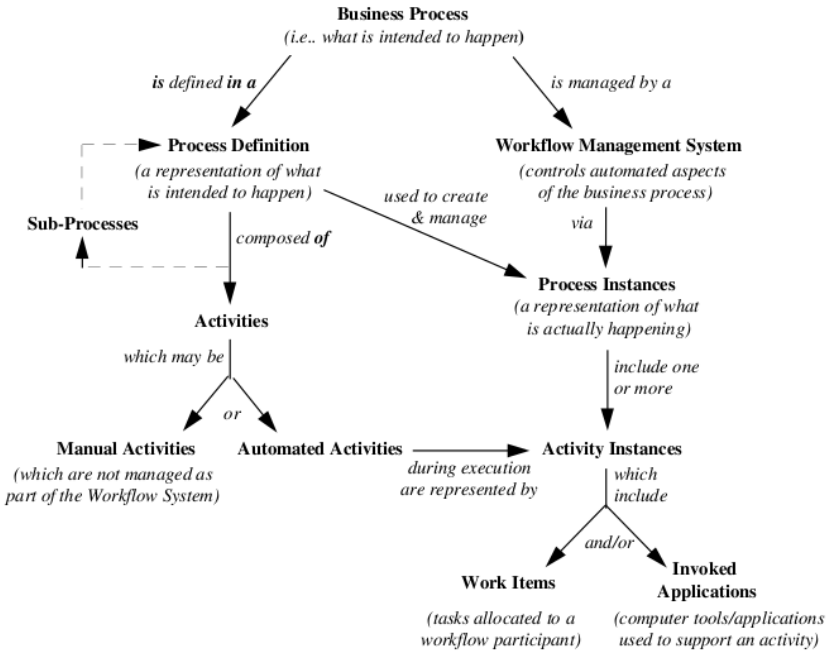

Figure 2.1. Basic WfMC terminology and its relationships (taken from [WfMC 1999])

A business process is usually modeled by business specialists that focus on domain-specific aspects. It is used as the basis for a technical realization by IT-specialists as *workflow*.

> *Workflow*: "The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules." [WfMC 1999]

The automation of a business process is defined with a *process definition*.

> *Process Definition*: "The representation of a business process in a form which supports automated manipulation, such as modelling, or enactment by a workflow management system.

> The process definition consists of a network of activities and their relationships, criteria to indicate the start and termination of the process, and information about the individual activities, such as participants, associated IT applications and data, etc." [WfMC 1999]

A process definition can be further structured with *sub-processes*.

> *Sub-Process*: "A process that is enacted or called from another (initiating) process (or sub process), and which forms part of the overall (initiating) process. Multiple levels of sub process may be supported." [WfMC 1999]

The process flow in a process definition is based on a composition of *activities*.

> *Activity*: "A description of a piece of work that forms one logical step within a process. An activity may be a manual activity, which does not support computer automation, or a workflow (automated) activity. A workflow activity requires human and/or machine resources(s) to support process execution; where human resource is required an activity is allocated to a workflow participant." [WfMC 1999]

Activities can be either *manual activities* or *automated activities*. We will focus on automated activities.

> *Automated Activity*: "An activity which is capable of computer automation using a workflow management system to manage the activity during execution of the business process of which it forms a part." [WfMC 1999]

A process definition and its activities are modeled and executed with a *workflow management system* (WfMS).

> *Workflow Management System*: "A system that defines, creates and manages the execution of workflows through the use of software, running on one or more workflow engines, which is

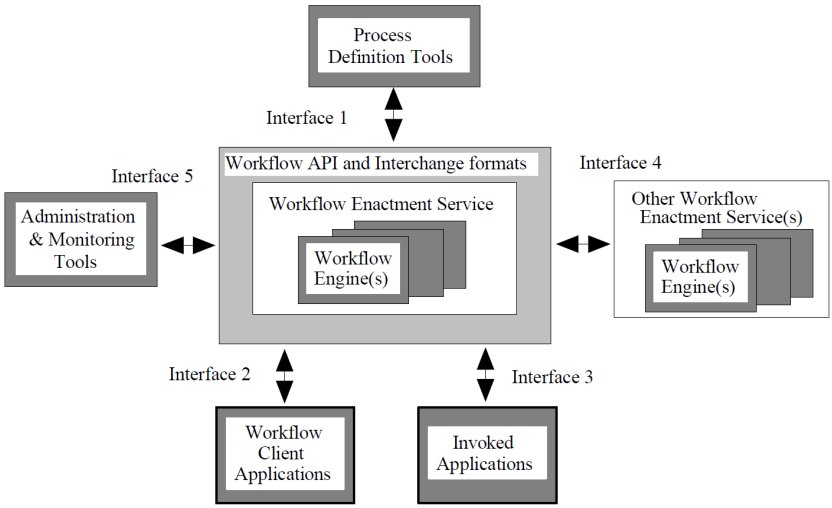

Figure 2.2. Major components and interfaces of WfMS (taken from [Hollingsworth 1995])

> able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applications." [WfMC 1999]

The major components and interfaces of a WfMS are defined within the workflow reference model [Hollingsworth 1995], see Figure 2.2. We will focus on *process definition tools*, *invoked applications*, and *workflow engines*.

A process definition tool is used to create a process definition, but the term is not further defined by the WfMC.

A workflow engine executes a process definition.

> *Workflow Engine*: "A software service or 'engine' that provides the run time execution environment for a process instance." [WfMC 1999]

The single execution of a process definition by a workflow engine is encapsulated in a *process instance*.

> *Process Instance*: "The representation of a single enactment of a process." [WfMC 1999]

An automated activity that is executed within a process instance is called *activity instance*.

> *Activity Instance*: "The representation of an activity within a (single) enactment of a process, i.e. within a process instance." [WfMC 1999]

Activity instances can include *work items* and *invoked applications*. We will focus on invoked applications.

> *Invoked Application*: "An invoked application is a workflow application that is invoked by the workflow management system to automate an activity, fully or in part, or to support a workflow participant in processing a workitem." [WfMC 1999]

We will focus on SOAs as execution infrastructures for workflows. Thus, each workflow application is invoked as a Web service. The process flow of a workflow is also called *orchestration*. Workflows whose invoked applications are all implemented as a Web service are also called *service compositions* or *service orchestrations*.

Workflows combine domain-specific aspects of business processes with technical aspects of process definitions. Thus, we distinguish between the domain-specific and the technical layer. We refer to the term *workflow model* as an IT-supported representation of a workflow. A workflow model that is created at the domain-specific layer is usually not executable and must be mapped to a corresponding executable workflow model at the technical layer by IT-specialists.

We further use the term *workflow language* to refer to a language for creating workflow models at the domain-specific or technical layer. Actual workflow languages such as the *Business Process Model and Notation* (BPMN) [OMG 2011a] and the *Web Services Business Process Execution Language* (BPEL) [OASIS 2007] are widely used standards in the business workflow domain. BPEL can be used for creating an executable workflow model at the technical layer. BPMN is originally used at the domain-specific layer, but since version

2.0, the standard also contains an execution semantics for a BPMN subset. The BPMN standard additionally defines a basic mapping from a BPMN subset to BPEL.

In the following, we will use the term *business workflow* instead of workflow. This allows a better differentiation from *scientific workflows*, which are introduced in Chapter 3. The term workflow refers to the general workflow concept independent from an application-specific domain such as business or research. We further introduce the following synonyms: *Workflow instance* for process instance, *workflow activity* for activity, *workflow application* for invoked application, and *workflow editor* for process definition tool. As we focus on SOAs, each workflow activity is executed by a workflow application that is represented by a corresponding Web service.

2.2 Business Process Model and Notation (BPMN)

The *Business Process Model and Notation* (BPMN) is a standard of the *Object Management Group* (OMG) for the control flow-centric, graphical modeling of business processes. It is well accepted in the business domain. The basis of BPMN is a set of graphical notation elements. It further provides a basic BPEL mapping for a BPMN subset.

BPMN was originally invented by Stephen A. White (IBM) in 2001 and published as *Business Process Modeling Notation* by the *Business Process Management Initiative* (BPMI) in 2004. Since BPMI merged with OMG in 2005, the further development of BPMN has been continued by the OMG. In 2006, BPMN 1.0 [OMG 2006a] was published as official OMG standard. This was followed by minor updates with BPMN 1.1 [OMG 2008] in 2008 and BPMN 1.2 [OMG 2009] in 2009. With Version 2.0, a major update for BPMN was released in 2011. The standard was extended, for example, by CMOF [OMG 2006b] metamodels and XML schemas for standardized model exchange and serialization, and an informal execution semantic for a BPMN subset. Thus, BPMN originally focused on the creation of workflow models at the domain-specific layer, but since version 2.0, it also supports executable workflow models at the technical layer.

Business processes can be modeled as *process diagrams*, *collaboration dia-*

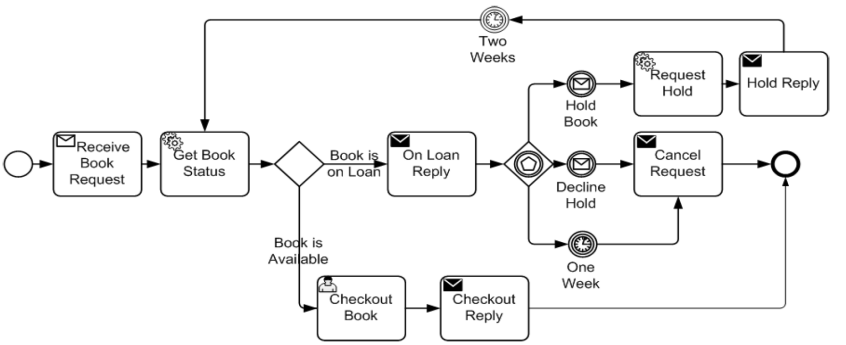

Figure 2.3. BPMN process example (taken from [OMG 2011a])

grams, and *conversation diagrams*. All diagram types are based on a specific subset of BPMN model elements. A process diagram defines the process flow for a *process*. A collaboration diagram specifies message interactions among participants in a *collaboration*, in which the behavior of each participant may be further specified with a process diagram. A conversation diagram groups the message exchange of collaborating participants as *conversations*. Each conversation can be further specified by the definition of a *correlation* that is used to create unique identifiers for message routing to process instances.

The process flow of a BPMN process is control flow-centric and graph-based. It is based on process model elements such as *activities*, *events*, *sequence flows*, and *gateways*. Some elements can access *data objects*, which can be regarded as data container like variables. An example BPMN process is shown in Figure 2.3.

Activities, see Figure 2.4, can be used to model workflow activities, and are often represented by *tasks*. Tasks can be further specified by one *task type* and multiple *task markers*, see Figure 2.5. Task types are, for example, the *service task* that represents the invocation of a service or the *script task* that represents the execution of a local script. Task markers are, for example, the *loop* marker for the iterative execution of a task and the *sub-process* marker for hierarchically structuring a process. The permissible combinations of all

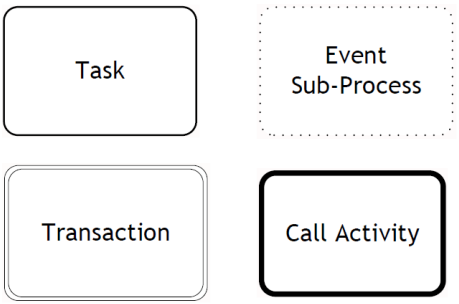

Figure 2.4. BPMN activities (taken from `http://bpmb.de/poster`)

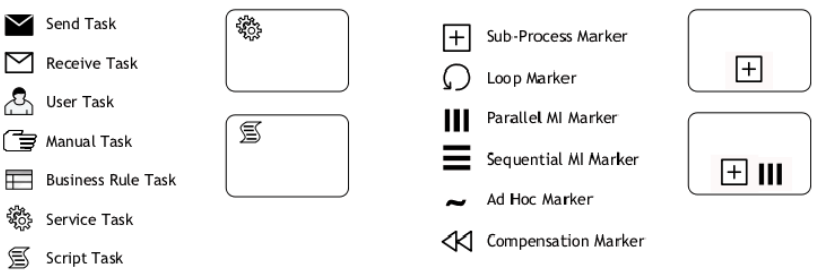

Figure 2.5. BPMN task types and markers (taken from `http://bpmb.de/poster`)

task markers are specified in the BPMN standard, e.g., the loop marker can be combined with the sub-process marker. Each task can have associations to data objects such as *data input* and *data output*.

Events can be used to model something that occurs during a process execution. The BPMN standard generally distinguishes between *start events*, *intermediate events* and *end events*. Start events indicate the beginning and end events the end of a process (or sub-process task). Intermediate events can be used anywhere between start and end events in the process flow as well as attached to activities, in some cases as so-called *intermediate boundary events*. The BPMN standard further distinguishes between *catch events* and *throw events*. Catch events consume and throw events produce internal or external events. Different *event types* can be used to further specify an event.

For example, *message events* receive (catch) or send (throw) messages from or to external participants and *error events* react on (catch) or create (throw) errors. The BPMN standard specifies the utilization for each event type, e.g., not all event types can be used as intermediate boundary events. All event types and their utilization are shown in Figure 2.6.

Sequence flows, see Figure 2.7, are control flow dependencies to connect two process elements such as activities, events and gateways. A basic control flow dependency is represented by a *normal flow* or *uncontrolled flow*. A *conditional flow* is a sequence flow with a Boolean condition. A *default flow* is a special sequence flow for gateways. Each process model element may have multiple ingoing and outgoing sequence flows. It is further permissible to create loop structures with sequence flows.

Gateways, see Figure 2.8, are used to split outgoing and join incoming sequence flows based on a corresponding split and join semantic. In *parallel gateways*, all paths between a splitting and joining parallel gateway are executed and regarded as concurrent. *Exclusive gateways* are used to create *if-then-elseif-else* structures in which only one path between a splitting and joining exclusive gateway is executed. A splitting exclusive gateway can have multiple outgoing conditional flows and optionally one default flow. A default flow is a sequence flow whose path is executed if no conditional flow is evaluated to `true`.

Each graphical model element is represented by a corresponding class in the BPMN metamodel. BPMN defines a metamodel extension mechanism to add individual information as extension attributes or extension elements to any existing class. The mechanism allows the integration of existing elements from other metamodels into the BPMN metamodel. Each extension defines a Boolean attribute `mustUnterstand` to indicate whether it must be supported by a BPMN tool (`true`) or may be ignored (`false`).

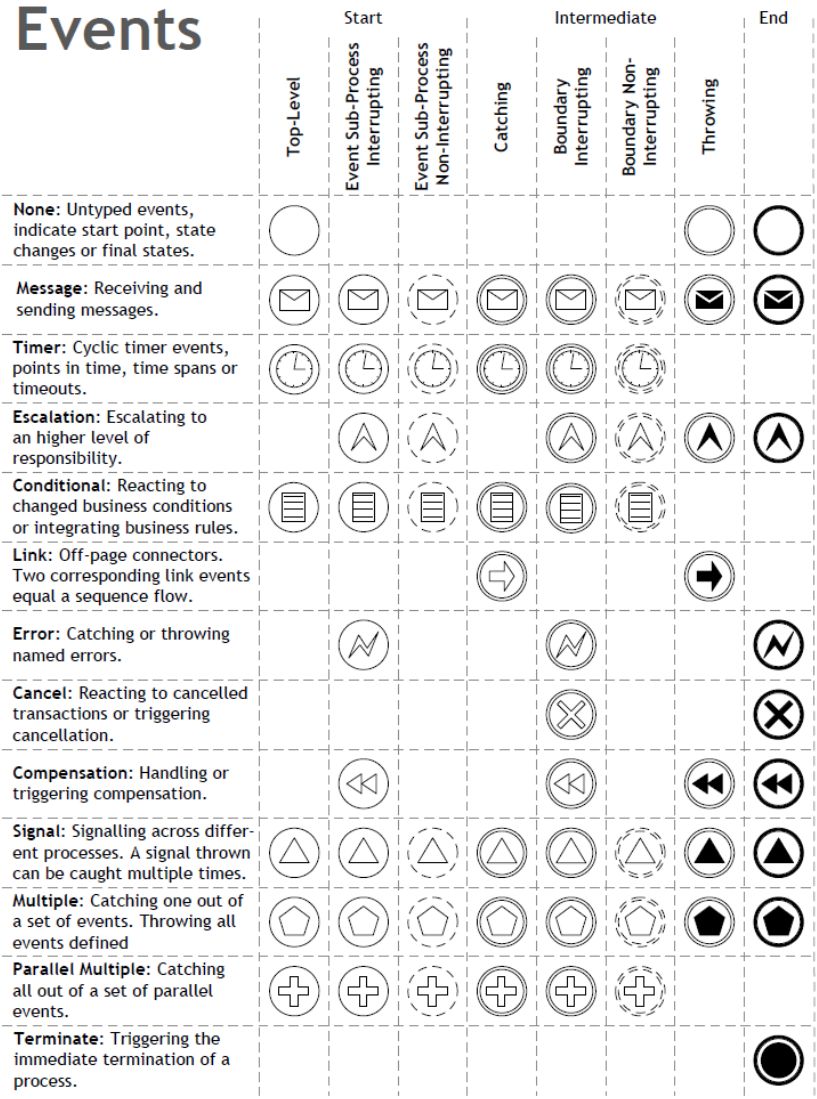

Figure 2.6. BPMN events (taken from `http://bpmb.de/poster`)

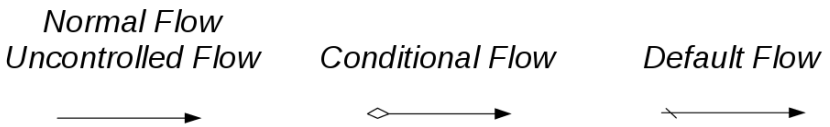

Figure 2.7. BPMN sequence flows (images taken from [OMG 2011a])

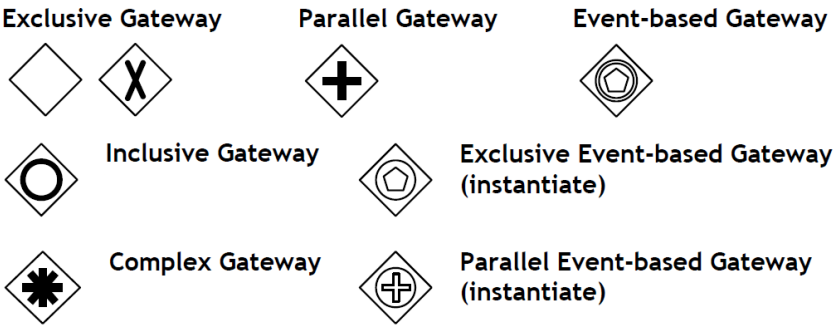

Figure 2.8. BPMN gateways (taken from `http://bpmb.de/poster`)

2.3 Web Services Business Process Execution Language (BPEL)

The *Web Services Business Process Execution Language* (WS-BPEL or just BPEL)[2] is a standard of the *Organization for the Advancement of Structured Information Standards* (OASIS) for implementing control-flow centric Web service orchestrations. It is well accepted in the business domain. The standard defined an XML-based workflow language for implementing service orchestrations as so-called *processes*. It does not specify any graphical representation for the XML syntax. Thus, workflow editors often use BPMN or a BPMN-like notation. A BPEL process itself is provided as a Web service that facilitates its integration in other BPEL processes.

Originally, the first version of BPEL was invented and published by BEA,

[2] BPEL is the prevalent abbreviation.

IBM, and Microsoft in 2002 as *Business Process Execution Language for Web Services* (BPEL4WS) [BEA et al. 2002]. The subsequent version 1.1 [BEA et al. 2003] was published in 2003 with the additional participation of SAP and Siebel Systems. Since then, BPEL4WS has been maintained by OASIS, and BPEL 2.0 [OASIS 2007] was published in 2007

The process flow of a BPEL process is based on *activities*. BPEL distinguishes between *basic activities* and *structured activities*. Basic activities are used to implement the behavior and data flow of workflow activities such as `receive`, `invoke`, `reply` for message exchange and `assign` for data manipulation. Variables can be used as data containers for messages or other XML-based content. Structured activities are used to create a block-based control flow structure for a BPEL process such as `sequence` for sequential, `repeatUntil` for iterative, and `flow` for concurrent execution. They can recursively contain other basic and structured activities. An excerpt of a BPEL process with a `flow` element as root structured activity is shown in Listing 2.1.

Listing 2.1. Excerpt of BPEL process (taken from [OASIS 2007])

```
1  ...
2  <flow>
3    <links>
4      <link name="receive-to-assess" />
5      <link name="receive-to-approval" />
6      <link name="approval-to-reply" />
7      <link name="assess-to-setMessage" />
8      <link name="setMessage-to-reply" />
9      <link name="assess-to-approval" />
10   </links>
11
12   <receive partnerLink="customer" portType="lns:loanServicePT"
13     operation="request" variable="request" createInstance="yes">
14     <sources>
15       <source linkName="receive-to-assess">
16         <transitionCondition>$request.amount &lt; 10000</transitionCondition>
17       </source>
18       <source linkName="receive-to-approval">
19         <transitionCondition>$request.amount >= 10000</transitionCondition>
20       </source>
21     </sources>
22   </receive>
23
24   <invoke partnerLink="assessor" portType="lns:riskAssessmentPT" operation="check"
25     inputVariable="request"  outputVariable="risk">
26     <targets>
```

```
27        <target linkName="receive-to-assess" />
28      </targets>
29      <sources>
30        <source linkName="assess-to-setMessage">
31          <transitionCondition>$risk.level='low'</transitionCondition>
32        </source>
33        <source linkName="assess-to-approval">
34          <transitionCondition>$risk.level!='low'</transitionCondition>
35        </source>
36      </sources>
37    </invoke>
38    ...
39  <flow>
40  ...
```

The structured activity `flow` provides the definition of an acyclic graph-based control flow. All basic and structured activities contained in a `flow` are regarded as concurrent. This concurrency can be synchronized by adding a control flow dependency as `link` between activities. Each `link` has exactly one target and one source activity. An activity may have multiple ingoing and outgoing `link` elements. The creation of loops with `link` elements is not permissible. Each activity in a `flow` can define a Boolean condition for all ingoing `link` elements (`joinCondition`) and for each outgoing `link` (`transitionCondition`). A `transitionCondition` sets an outgoing `link` state to either `true` or `false`. A `joinCondition` evaluates whether an activity is executed (`true`) or not (`false`) based on all incoming `link` states. The behavior of a `false joinCondition` can be further controlled with the Boolean attribute `supressJoinFailure`, which can be defined globally for the flow activity and locally for each contained activity. If it is set to `false`, a join failure is created. If it it set `true`, the activity is not executed and all outgoing `link` elements are set to `false`. The `false` link state is further propagated until an activity throws a join failure or a `joinCondition` is reached that evaluates to `true`. This mechanism is called *Dead-Path-Elimination* (DPE). It ensures that dead paths with `false link` elements are eliminated.

The message exchange between Web services can be described with a WSDL extension called *partner link type*. A partner link type defines roles for all participating Web services based on the WSDL port types from the respective WSDL interfaces. Each role is associated with one WSDL port type. Partner link types are instantiated by *partner links* within a BPEL process. A partner link references one or two roles from the corresponding

partner link type via the attributes `partnerRole` and `myRole`. A `partnerRole` indicates that the associated WSDL port type is used to send messages (e.g. via `invoke` and `reply`) and must be provided by the WSDL interface of an external Web service. A `myRole` indicates that the associated WSDL port type is used to receive messages (e.g. via `receive`) and must be provided by the WSDL interface of the BPEL process. The WS-Addressing [W3C 2006] endpoint for an invoked Web service is also assigned to the partner link. It is possible to modify this endpoint with the `assign` activity during process execution, which allows the dynamic invocation of Web services.

A BPEL process may be executed in different process instances. In order to route an incoming message to the correct process instance, BPEL provides a *correlation* mechanism. It basically utilizes unique identifiers in the content of exchanged messages, for example a customer number. Therefore, *properties* can be defined with a corresponding WSDL extension in order to reference values in WSDL messages. One or more properties can be used to define a *correlation set* within a BPEL process. Each correlation set has a unique name so that several correlation sets can be defined for a BPEL process. All properties of one correlation set must represent a unique identifier. The values for these properties are retrieved from the message that is currently being received or sent by a corresponding activity. A correlation set must be initialized once by a message exchange activity to associate the unique identifier with the process instance. Initialized correlation sets are immutable and can be used for message routing by all activities that receive messages. Therefore, a correlation set is determined for an incoming message and compared with the initialized correlation set of each process instance. The message is routed to the process instance with the matching initialized correlation set.

A BPEL process is deployed and executed by a compliant process engine such as Apache ODE (Orchestration Director Engine)[3]. Each BPEL process engine provides a vendor-specific deployment descriptor that usually contains binding information for all partner links.

BPEL provides a language extension mechanism, e.g., to define custom activities or to extend existing activities by adding additional information.

[3] `http://ode.apache.org/`

Extensions are identified by a namespace and specify an additional Boolean attribute `mustUnderstand` to indicate whether it is essential for workflow execution and therefore must be supported by a process engine (`true`) or may be ignored (`false`). Extensions can further be classified as *design time only extensions*, *design and runtime extensions*, and *runtime only extensions* [Kopp et al. 2011]. Design time extensions are modeling extensions and runtime extensions are execution extensions. Design time only extensions are transformed to standard BPEL elements before deployment and execution.

2.4 BPMN to BPEL Mapping Strategies

All existing mappings of a BPMN process to a BPEL process are restricted to a BPMN subset. A mapping generally concerns workflow activities and control flow. Many BPMN activities and events can be mapped directly to corresponding BPEL elements. For example, a BPMN service task can be mapped to BPEL `invoke` and a BPMN message start event to BPEL `receive`. A direct mapping to BPEL is not possible for all control flow constructs in BPMN because the definition of control flow in BPMN is more expressive in contrast to BPEL. BPMN provides cyclic graph-based control flow structures. BPEL provides block-based and acyclic graph-based control flow structures. One prominent BPMN-to-BPEL mapping problem regarding control flow is *arbitrary/untangled* cycles, which are loops with multiple entries and exits.

Mendling et al. [2008] describes different strategies for the mapping of control flow from graph-based to block-based workflow languages and vice-versa. BPEL is used as a block-based reference language whereby its capability to define acyclic graph-based structures is also taken into account. All of the described strategies can be applied for a BPMN to BPEL mapping or vice versa. The following is a description of all of the strategies regarding the mapping of graph-based to block-based workflow languages.

Element-Preservation:
The *element-preservation* strategy, see Figure 2.9, aims to preserve the graph-based control flow structure. Therefore, all nodes (BPMN activities, gateways, and events) in the graph are mapped to corresponding BPEL elements

and added to a single BPEL `flow` element. Gateways are mapped to BPEL `empty` elements, which can be regarded as no-op operations and are only used for control flow synchronization. All arcs (BPMN sequence flows) are mapped to BPEL `link` elements.

Different rules are applied to create `transitionCondition` and `joinCondition` elements for source and target BPEL elements of `link` elements, for example:

- All outgoing sequence flows for a splitting BPMN exclusive gateway (multiple conditional flows and one optional default flow) are mapped to corresponding BPEL `link` elements. Each condition of a BPMN conditional flow is mapped to a `transitionCondition` for the corresponding BPEL `link`. The `transitionCondition` for a BPMN default flow is the negation of the conjunction of all `transitionCondition` attributes that have been previously derived from BPMN conditional flows. Consequently, the `transitionCondition` for a BPMN default flow is `true` if all BPMN conditional flows are evaluated to `false`.

- The `joinCondition` for a joining BPMN exclusive gateway is the disjunction of all `transitionCondition` attributes from ingoing BPMN sequence flows or BPEL `link` elements. Consequently, it is `true` if one of the ingoing BPEL `link` elements is evaluated to `true`.

In order to prevent undesired join failures during the execution of a BPEL `flow`, some `supressJoinFailure` attributes are set to `true`. It is possible that `transitionCondition` attributes are correctly evaluated to `false` during execution, e.g. for BPEL `link` elements of BPMN conditional flows. Consequently, join failures may be created. These join failures are suppressed with the `supressJoinFailure` option and affected paths are eliminated by the BPEL DPE algorithm.

As we are not allowed to define cyclic `link` structures in a BPEL `flow`, the element-preservation strategy only supports acyclic BPMN workflows.

Element-Minimization:
The *element-minimization* strategy, see Figure 2.10, initially applies the element-preservation strategy, and then removes all BPEL `empty` elements.

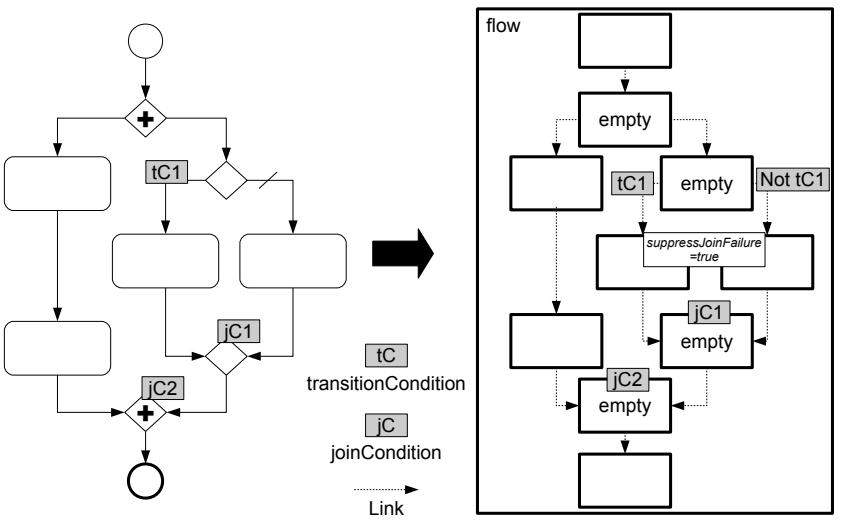

Figure 2.9. BPMN to BPEL mapping strategy Element-Preservation [Mendling et al. 2008]

To remove a BPEL `empty` element the following steps are executed:

1. Direct `link` elements are created between all source and target elements of the BPEL `link`.
2. Conjunction of the `transitionCondition` of each ingoing BPEL `link` with each `transitionCondition` of all outgoing BPEL `link` elements. These adjusted `transitionCondition` attributes are copied to the BPEL `link` elements created for them.
3. Conjunction of the `joinCondition` of each target element with the `joinCondition` of the BPEL `empty` element.
4. The BPEL `empty` element and its `link` elements are removed.

Structure-Identification:
The *structure-identification* strategy, see Figure 2.11, is based on structure

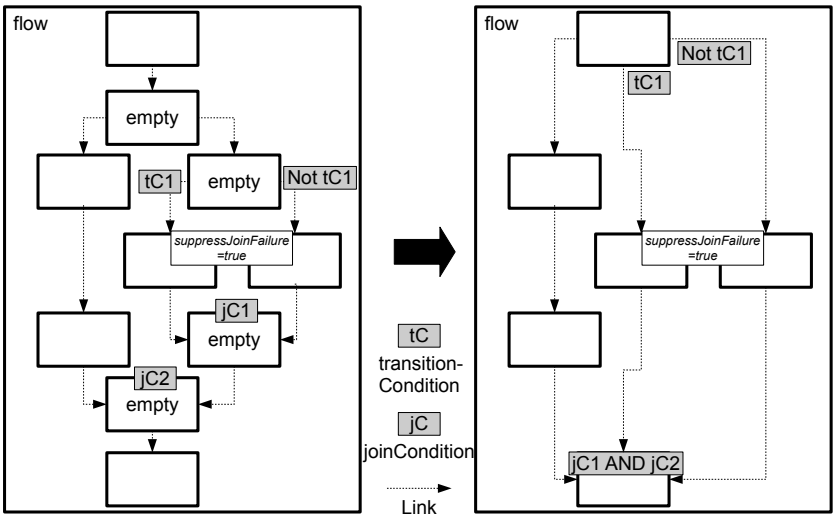

Figure 2.10. BPMN to BPEL mapping strategy Element-Minimization [Mendling et al. 2008]

pattern to identify *single entry single exit* (SESE) regions in a BPMN process. SESE regions are sequences or loops, for example. Each structure pattern defines a mapping to a BPEL activity structure, e.g. based on `sequence`, `while`, or `repeatUntil`. The BPEL mapping in the BPMN standard defines pattern for a structure-identification strategy.

The identification of structures is usually based on a folding strategy as described by Ouyang et al. [2006, 2009]. An identified structure is *folded* to one element in the workflow graph. This combination of identification and folding is applied until only one element is left or no more structures can be identified. In the first case, the workflow is structured according to the defined pattern and can thus be completely mapped.

The structure-identification strategy supports the mapping of cyclic structures in contrast to the element-preservation strategy. But arbitrary cycles are not supported, as they are not SESE regions. It is possible to *untangle* arbitrary loops as described by Zhao et al. [2006]. This will modify

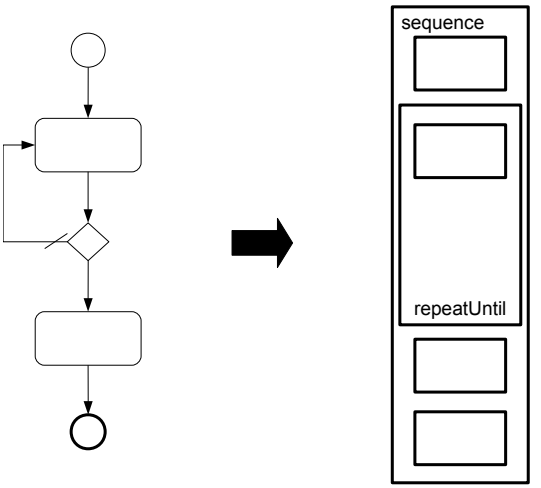

Figure 2.11. BPMN to BPEL mapping strategy Structure-Identification [Mendling et al. 2008]

the process flow so that an arbitrary loop is emulated with structured loops. Afterwards, the structure-identification strategy can be applied.

Structure-Maximization:
The *structure-maximization* strategy, see Figure 2.12, initially applies the structure-identification strategy. Afterwards, an element-preservation or element-minimization strategy is applied to map unmatched structures. The structure-maximization strategy only works if the folded graph returned by the structure-identification strategy is acyclic.

Event-Condition-Actions-Rules:
The *event-condition-action-rules* strategy, see Figure 2.13, is an extension of the structure-identification strategy. It is invented and comprehensively described by Ouyang et al. [2006, 2009] and is the only strategy that supports the mapping of arbitrary cycles. Therefore, the BPEL event handler mechanism is used. BPEL provides the definition of an event handler that

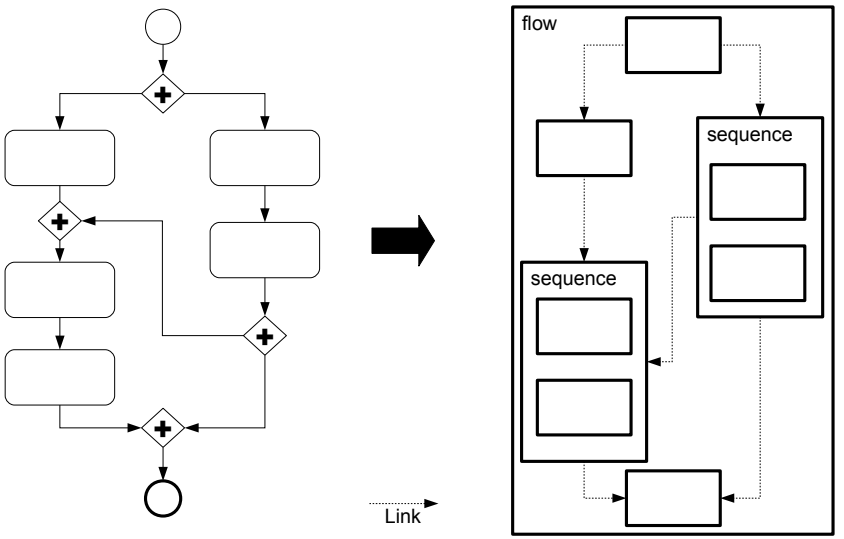

Figure 2.12. BPMN to BPEL mapping strategy Structure-Maximization [Mendling et al. 2008]

can be triggered by specified incoming message types. Each event handler is concurrent with the other activities or event handlers in a BPEL process.

First, the structure-identification strategy is applied based on the folding strategy described above. If the folded graph contains more than one element, it is unstructured. The unstructured parts are mapped based on *event-condition-action* rules. Thereby, each identified structure is mapped to BPEL and enclosed as *action* within an event handler. Each arc in the folded graph is mapped to a service invocation in which the source event handler triggers the target event handler by sending a corresponding message as an *event* to it. The target event handler receives this *event* and may check a *condition*, which is derived from a BPMN conditional flow or default flow, to determine if its contained *action* has to be executed or not. Technically, a BPEL process instance is sending messages to itself during its execution.

This strategy supports any control flow structure that can be defined with BPMN sequence flows. But a frequent internal message exchange will

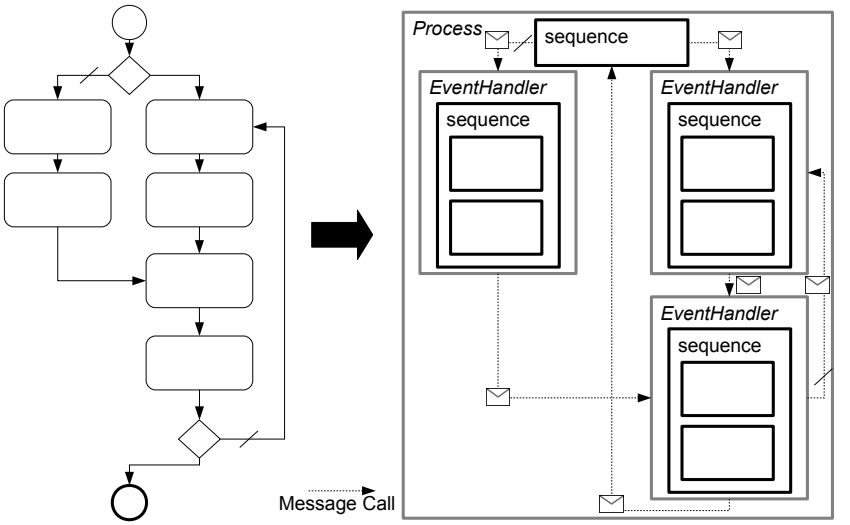

Figure 2.13. BPMN to BPEL mapping strategy Event-Condition-Action-Rules[a] [Mendling et al. 2008]

[a]Picture credits: `http://bpmb.de/poster` (message icon)

affect the performance of the process execution and it is hard to understand and to debug the BPEL process.

Chapter 3

Scientific Workflows

3.1 Introduction and Basic Terminology

Scientific workflows are one essential means to facilitate and automate the processing of high-volume scientific data in often large distributed computing infrastructures such as Grids [Goble and Roure 2009]. They provide a high-level abstraction from the underlying infrastructure so that scientists are able to create and run scientific workflows without comprehensive programing skills ("*where IT meets scientists*" [Hey et al. 2009]). Scientific workflows emerged in the context of e-Science [Taylor et al. 2006] in which the data-intensive science paradigm [Hey et al. 2009] was coined. The objectives of scientific workflows are clearly described by Ludäscher et al. [2009]:

> "The main goals of scientific workflows, then, are (i) to save 'human cycles' by enabling scientists to focus on domain-specific (science) aspects of their work, rather than dealing with complex data management and software issues; and (ii) to save machine cycles by optimizing workflow execution on available resources." [Ludäscher et al. 2009]

This description emphasizes domain-specific aspects regarding workflow creation and technical aspects regarding workflow execution. Thus, analogous to business workflows (see Chapter 2) we distinguish between a domain-specific and a technical layer for scientific workflows.

Lin et al. [2009] adopted the workflow terminology and concepts of the WfMC [WfMC 1999] for the definition of the terms scientific workflow and *Scientific Workflow Management System* (SWfMS).

> *Scientific Workflow*: "A scientific workflow is the computerized facilitation or automation of a scientific process, in whole or part, which usually streamlines a collection of scientific tasks with data channels and dataflow constructs to automate data computation and analysis to enable and accelerate scientific discovery." [Lin et al. 2009]
>
> *Scientific Workflow Management System*: "A scientific workflow management system (SWFMS) is a system that completely defines, modifies, manages, monitors, and executes scientific workflows through the execution of scientific tasks whose execution order is driven by a computerized representation of the workflow logic." [Lin et al. 2009]

Additionally, Lin et al. [2009] defined a scientific workflow reference model for *scientific workflow management systems* (SWfMS), see Figure 3.1, based on the workflow reference model of the WfMC [Hollingsworth 1995].

Scientists play the central role in scientific workflows. They create, run, re-run, and monitor scientific workflows as well as they analyze the results. Scientific workflows are data flow-centric and created with a corresponding workflow editor that usually provides a repository with predefined workflow activities. Such workflow activities are often domain-specific, i.e. they represent computational tasks for a particular scientific domain. Each workflow activity contains additional information about its technical execution, which is usually not visible for the scientist. In other words, a scientist focuses on the creation of a workflow model at the domain-specific layer based on predefined workflow activities from a workflow repository. Due to the additional technical information this workflow model can be either directly executed by a workflow engine or automatically be mapped to a corresponding executable workflow model.

Many SWfMS such as Kepler are based on one workflow language in which a created workflow model is directly used for execution. Therefore, each workflow activity references a software component, which contains the corresponding logic to execute the activity. Pegasus, however, uses a workflow language called DAX (Directed Acyclic Graph in XML) for workflow

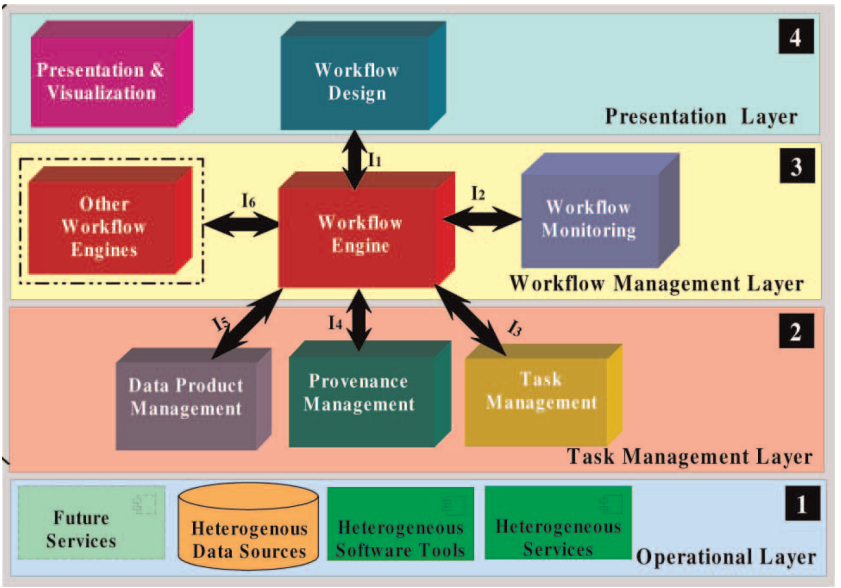

Figure 3.1. Major components and interfaces of a SWfMS (taken from [Lin et al. 2009])

modeling, which is mapped to a DAG-based (directed acyclic graph) workflow language for workflow execution with Condor DAGman (Directed Acyclic Graph Manager)[1]. The mapping includes resource planning and workflow optimization based on available resources.

3.2 Business Workflows vs. Scientific Workflows

Scientific workflows are often compared with the well-known business workflows (cf. [Wassermann et al. 2007; Barga and Gannon 2007; Ludäscher et al. 2009; Scherp et al. 2010; Görlach et al. 2011]). This helps to understand the core similarities between the two workflow technologies. Thus, we

[1] http://research.cs.wisc.edu/htcondor/dagman/dagman.html

will discuss and compare the essential aspects of business and scientific workflows in the following.

Domain and Objectives:
Business workflows support business processes and are continuously developed, monitored, and optimized within so-called *Business Process Management* (BPM), in which many stakeholders are involved. One major objective of business workflows is to implement production workflows [Leymann and Roller 1999].

Scientific workflows are used in an explorative trial-and-error manner and support experimental scientific data analysis and processing. They are continuously modified, executed, and monitored by scientists in order to study their results and to gain scientific knowledge. One major objective is that scientists can use workflow technologies without the need for comprehensive programming skills [Ludäscher et al. 2009].

A scientific workflow may be the basis for a business workflow. For example, an explorative optimized weather forecast scientific workflow can be used for commercial weather forecast products and services.

Life Cycle and Role Model:
A good comparison of the life cycles and roles for business and scientific workflows is depicted by Görlach et al. [2011], which is shown in Figure 3.2.

The business workflow life cycle is aligned to BPM and contains many roles that can be represented by different persons. Each role is associated with specific tasks and rights. Domain-specific parts are covered by *business specialists* and *business analysts* such as the workflow modeling of business processes and its monitoring and analysis for optimization. Technical aspects are covered by *IT-specialists* such as the implementation and deployment of business workflows as well as the monitoring of the execution infrastructure. Different *clients* such as *employees* or external partners can be involved during a workflow execution. After a business workflow has been initially developed and deployed, its execution is continuously monitored and analyzed. This may result in an optimization that requires a modification and redeployment of the business workflow. Business workflows

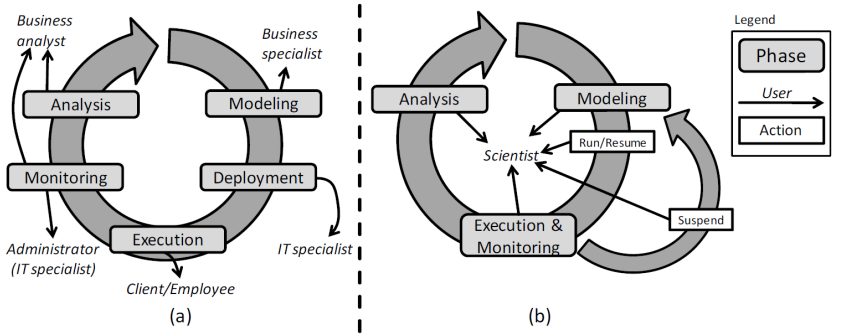

Figure 3.2. Life cycle of business and scientific workflows (taken from [Görlach et al. 2011])

that represent production workflows are usually rarely modified. They are deployed and used for a long period of time.

The scientific workflow life cycle is completely covered by the single role *scientist*. Scientists create, run, and monitor a scientific workflow and analyze the results. Due to the explorative nature of a scientific workflow, its life cycle is iterated frequently. After the initial creation and execution of a scientific workflow, it is often modified and restarted, based on the results of previous runs. Furthermore, a scientific workflow execution may be manually suspended, e.g., due to errors detected in intermediate results. The scientific workflow is then completely or partly modified and its execution is restarted. A scientist often uses a scientific workflow only for a short period of time, e.g., until its outcomes are useful. Afterwards, it may be shared with other scientists, for example, attached to a written publication of the scientific results or as an independent data publication.

Workflow Model and Execution:
Workflow languages for business workflows such as BPMN and BPEL are control flow-centric. The execution order of workflow activities is defined by control flow dependencies between them. Such a control flow dependency generally indicates that a target workflow activity is executed

after its source workflow activity is finished. Data flow is specified similar to common programming languages by read and write operations on data containers such as variables, e.g. within workflow activities. The data that is consumed and produced by invoked applications/services is usually small and contained in the exchanged messages. Thus, the complete data that is exchanged between interdependent service invocations is transferred through the workflow engine and often stored in workflow variables. It may be directly processed within a workflow activity that is executed by the workflow engine itself, e.g. for a mapping between different XML-based data structures.

Scientific workflows are usually data flow-centric with few control flow elements such as conditions and loops. The execution order in data flow-centric scientific workflows is defined by data flow dependencies between workflow activities. A data flow dependency between two workflow activities can be generally regarded as extended control flow dependency. The target workflow activity is executed after the source workflow activity is finished and consumes its produced data. Data that is processed by and exchanged between invoked applications/services is often very large. If different resources are used for interdependent processing steps, all required data transfers are executed directly between these resources as so-called *third party transfers*. These data transfers are initiated and monitored by the workflow engine (third party). Data is never processed by or transferred through a workflow engine. Thus, a scientific workflow execution just coordinates data processing and data transfers. The optimization of resource locations for data processing or the reuse of cached data can be significant performance drivers for scientific workflows.

Chapter 4

Model-driven Software Development (MDSD)

4.1 Introduction and Basic Terminology

Model-driven Software Development (MDSD) is a software development process based on *models* and *model transformations*. A (software) system is described with formally specified models that may represent different views of a software system and are usually abstracted from the underlying programming language and runtime environment. Model transformations are then applied on models for a stepwise refinement until the final source code is created.

MDSD aims to automate the mapping of models to source code as far as possible. Therefore, software developers encapsulate refinements and code generation in reusable model transformations. The objective is to improve quality and productivity in software development.

The rules for the creation of models are defined by a so-called *metamodel*. When a model *conforms* to a metamodel, it is an *(model) instance* of it. Model instances are modified or mapped to other model instances with model transformations. Model transformations and transformation languages are separately discussed in Section 4.2 in more detail.

We use the following central definitions regarding MDSD.

> *Model*: A model is a simplified and abstract view of a (real) system. [Reussner and Hasselbring 2008]

> *Metamodel*: A metamodel defines elements and rules for cre-

> ating models. It consists of an abstract syntax, at least one concrete syntax, and a static and dynamic semantic. [Reussner and Hasselbring 2008]
>
> *(Model) Instance*: A model is an instance of a metamodel, if it conforms to it. Conformance means that only defined elements of the metamodel are used and its rules are complied. [Reussner and Hasselbring 2008]
>
> *Model Transformation*: A model transformation is a computable mapping that transforms a set of input model instances to a set of output model instances. [Reussner and Hasselbring 2008]
>
> *Model-driven software development*: Model-driven software development (MDSD) refers to development processes that focus on models as independent development artifacts. [Reussner and Hasselbring 2008]

A metamodel focuses on four aspects. An *abstract syntax* defines allowed elements, attributes, and relations in model instances. One or more *concrete syntaxes* define the representation of a model instance that can be *textual*, *graphical* or *hybrid*. A *static semantic* defines rules to further restrict valid model instances, for example based on the *Object Constraint Language* (OCL) [OMG 2006c]. A *dynamic semantic* defines the behavior of a model instance and the way to interpret it.

A metamodel itself can be defined by a metamodel, which is called *meta-metamodel*. To limit the number of meta levels, a meta-metamodel is usually self-describing, that means it is its own metamodel. Metamodels and meta-metamodels are generally both models, but their name is based on their purpose. A fixed terminology like this with model, metamodel, and meta-metamodel is used by OMG, for example.

Favre and Nguyen [2005] utilize the set theory to define the term metamodel as a relation between models. They regard a metamodel as a model that defines a set of models. If a model is an element of such a set of models, it conforms to a corresponding defining model. The term metamodel is thus

regarded as the relation between two models. A meta-metamodel relation does not exist. It can be regarded as two successive metamodel relations.

A metamodel can be further regarded as a language for models. If the expressiveness of such a language focuses on a particular domain, it is also called a *domain-specific language* (DSL) [Fowler 2010]. DSLs are further discussed in Section 4.3.

> *Domain-specific language (DSL)*: A domain-specific language is a language defined by a metamodel that contains concepts for a specific domain. [Reussner and Hasselbring 2008]

The OMG have standardized MDSD with the *Meta Object Facility* (MOF) [OMG 2006b]. It introduces the four model layers *M3* (meta-metamodel), *M2* (metamodel), *M1* (model), and *M0* (real-world object). MOF specifies a meta-metamodel for the M3 layer in order to define metamodels at the M2 layer, which can be further used to create models at the M1 layer. The *Unified Modeling Language* (UML), for example, is defined as a metamodel that conforms to the MOF meta-metamodel. MOF also defines a serialization and exchange format (concrete syntax) for models based on *XML Metadata Interchange* (XMI) standard [OMG 2011c]. *Essential MOF* (EMOF) is a subset of MOF that facilitates the definition of metamodels without the need to understand the complete MOF.

The Eclipse Modeling Framework (EMF)[1] provides a framework and tools for MDSD in the Java domain. EMF consists of an *Ecore metamodel* (meta-metamodel) that is aligned to EMOF. The Ecore metamodel is used to define *Ecore models* (metamodels). An *EMF model* is a model that conforms to an Ecore model. EMF further utilizes XMI for serialization and model exchange.

If we use the term model in the following, we implicitly assume that a corresponding metamodel exists. The term model instance is then avoided.

[1] `http://www.eclipse.org/modeling/emf/`

4.2 Model Transformations

Model transformations or just transformations are used to modify or map models. They can be divided into *model-to-model transformations* (M2M) and *model-to-text transformations* (M2T) [Czarnecki and Helsen 2006]. M2M transformations define a mapping between the abstract syntaxes of source and target metamodels and can be applied to corresponding models (model instances). M2T transformations define a mapping of the abstract syntaxes of source metamodels to strings (textual concrete syntax). Thus, M2T are usually final model transformations in MDSD in order to generate source code. If a M2M transformation modifies existing models it is called *in-place transformation* or *endogenous transformation* [Czarnecki and Helsen 2006; Mens and Gorp 2006]. If a M2M transformation creates new models it is called *out-place transformation* or *exogenous transformation* [Czarnecki and Helsen 2006; Mens and Gorp 2006]. Model transformations can be further distinguished between *vertical transformation* and *horizontal transformation* [Mens and Gorp 2006; Reussner and Hasselbring 2008], see Figure 4.1. In a vertical transformation the level of abstraction is changed by adding or reducing details, e.g., for code generation whereby the general structure of the software system is not changed. In a horizontal transformation the internal structure of models is changed, e.g., for a refactoring whereby the level of abstraction is not changed.

A model transformation is defined with a transformation language that generally provides the definition of *transformation rules*. A transformation rule defines the mapping of particular source metamodel elements to target metamodel elements (M2M transformation) or text (M2T transformation). Transformation languages can be distinguished between *imperative/operational transformation languages* and *declarative/relational transformation languages* [Czarnecki and Helsen 2006; Mens and Gorp 2006]. A combination of both language paradigms is also called *hybrid transformation language* [Biehl 2010]. The major difference between these language paradigms is the way the transformation rules are applied. Imperative transformation languages can be compared with conventional programming languages such as Java, in which transformation rules are explicitly invoked. Declarative transformation languages can be compared to functional programming

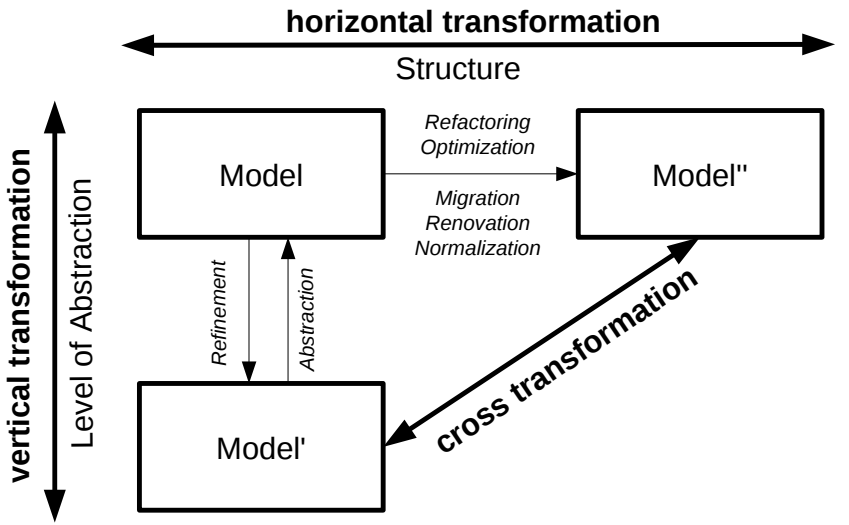

Figure 4.1. Classification of model transformations (based on [Reussner and Hasselbring 2008])

languages such as *XSL Transformations* (XSLT) [W3C 1999]. Transformation rules are applied based on a rule-matching algorithm. For a more detailed classification of model transformations and transformation languages please refer to [Czarnecki and Helsen 2006; Mens and Gorp 2006; Huber 2008; Biehl 2010].

Examples of M2M transformation languages are *Query/View/Transformation (QVT)* [OMG 2011b] and the *ATLAS Transformation Language* (ATL)[2]. QVT defines the transformation languages *Relations* (relative) and *Operational Mappings* (imperative) . It is part of MOF and the transformation languages are defined by corresponding MOF metamodels. ATL is a hybrid transformation language that unifies both the language concepts of QVT Relations and QVT Operational Mappings. It is declarative with particular imperative language constructs.

[2] http://www.eclipse.org/atl/

Xtend[3] is an object-functional programming language that provides comprehensive features to implement M2M and M2T transformations. It is not a dedicated transformation language in contrast to QVT and ATL. Xtend is further described in Section 4.4.

4.3 Domain-specific Languages (DSLs)

DSLs are custom programming languages that are designed for a specific application domain. They provide a language infrastructure with a usually compact language syntax, which allows developers to focus on essential aspects while abstracting from particular details of an underlying programming infrastructure. Transformation languages are DSLs, for example. One major objective of DSLs is to improve productivity in software development.

DSLs play an important role in MDSD. However, the introduction of DSLs (and MDSD) requires a significant initial effort. DSLs have to be designed and maintained, and the corresponding language infrastructure must be created. But the expected increase in productivity results in a better overall efficiency in software development in contrast to conventional software development processes.

DSLs are usually built upon existing programming infrastructures such as Java. They can be generally divided into *internal* and *external* DSLs [Fowler 2010]. An internal DSL utilizes an existing programming language as host language. It exploits only a subset of the language syntax and reuses its language infrastructure. An external DSL is realized as independent language, which is usually mapped to an existing programming language. This includes the definition of a metamodel and the creation of a corresponding language infrastructure. Based on the prevalent concrete syntax, DSLs can be further distinguished between *textual*, *graphical*, and *hybrid* DSLs.

Xtext provides an EMF-based framework for the creation of textual DSLs. It is further described in Section 4.5.

[3] http://www.eclipse.org/xtend/

4.4 Xtend

Xtend and its predecessor, which is part of the *openArchitectureWare* (oAW)[4] framework, are languages that are widely accepted in the model-driven community for implementing model transformations. *Xtend*[5] in the actual version 2 is generally an object-functional programming language for Java developers and part of Xtext (see Section 4.5). It provides a Java-like but less verbose syntax and uses the expression language Xbase [Efftinge et al. 2012]. Furthermore, Xtend provides many features[6] that are not provided by Java such as *Extension Methods*, *Lambda Expressions*, *Operator Overloading*, *Powerful Switch Expressions*, *Multiple Dispatch*, and *Template Expressions*. An Xtend class is *compiled* to standard Java code, which is then compiled with a standard Java compiler. Xtend can be generally compared with Groovy[7] and Scala[8].

Xtend is not a dedicated transformation language, but its comprehensive language features can be used to implement both M2M and M2T transformations. It is low-level in contrast to transformation languages such as ATL and QVT, e.g., the full Java API of EMF can be used to create, modify, and access EMF model elements. ATL and QVT are more restricted and abstract from concrete technologies such as EMF.

Extension Methods:
The main objective of extension methods is to externally define additional methods for any object type. An extension method is a normal method, which is syntactically used as if the method is provided by the corresponding object itself, see Listing 4.1. But in the generated Java code, the method is invoked with the object as the first parameter. If a syntactic object method invocation is an extension method or a normal object method, it is statically determined at design time. As useful side effect, extension methods provide code that is more readable, because nested method invocations can be

[4] http://www.openarchitectureware.org/
[5] http://www.eclipse.org/xtend/
[6] http://www.eclipse.org/xtend/documentation.html
[7] http://groovy.codehaus.org/
[8] http://www.scala-lang.org/

syntactically arranged as a chain.

Listing 4.1. Extension method invocation in Xtend (taken from `http://www.eclipse.org/xtend/documentation.html#extensionMethods`)

```
1 // calls toFirstUpper("hello")
2 "hello".toFirstUpper()
```

Methods that are locally defined within an Xtend class can also be used as an extension method, see Listing 4.2.

Listing 4.2. Local extension methods in Xtend (taken from `http://www.eclipse.org/xtend/documentation.html#extensionMethods`)

```
1  class MyClass {
2      def doSomething(Object obj) {
3          // do something with obj
4      }
5
6      def extensionCall(Object obj) {
7          // calls this.doSomething(obj)
8          obj.doSomething()
9      }
10 }
```

Xtend additionally provides many existing extension methods for different Java types such as `String` and `List`, see Listing 4.3. For example, a `map` extension method maps a list of input objects to a list of output objects. It generally invokes a specified method (*map function*) for each input object and adds each returned object to an output list. A similar operator is also provided by QVT and ATL.

Listing 4.3. Standard extension methods in Xtend (taken from `http://www.eclipse.org/xtend/documentation.html#extensionMethods`)

```
1 // calls StringExtensions.toFirstUpper(String)
2 "hello".toFirstUpper
3
4 // calls ListExtensions.<T, R>map(List<T> list , Function<? super T, ? extends R> mapFunction)
5 listOfStrings.map[ toFirstUpper ]
```

Multiple Dispatch:
Xtend utilizes the standard static binding mechanism of Java for overloaded methods, in which the static types of its parameter are analyzed at design

time. Additionally, Xtend provides dispatch methods, in which parameter types are analyzed at runtime and then *dispatched* to the corresponding method.

An example is shown in Listing 4.4. Corresponding methods are marked with the `dispatch` keyword. To understand the behavior of the multiple dispatch mechanism better, the generated Java code is shown in Listing 4.5. The basic idea is that the dispatch mechanism is realized with a generated if-elseif-else structure in Java.

Dispatch methods are especially useful for extension methods. For example, the `map` extension method can easily be used for a list of objects with different types. The map function for each object type is dispatched at runtime. A developer may alternatively implement a manual dispatch mechanism based on a switch or if-then-else structure. Thus, dispatch methods also reduce lines of codes. Dispatch methods can be used in model transformations to realize some kind of transformation rule matching similar to ATL and QVT.

Listing 4.4. Multiple dispatch in Xtend (taken from `http://www.eclipse.org/xtend/documentation.html#polymorphicDispatch`)

```
1  def dispatch printType(Number x) {
2      "it's a number"
3  }
4
5  def dispatch printType(Integer x) {
6      "it's an int"
7  }
```

Listing 4.5. Multiple dispatch compiled to Java (taken from `http://www.eclipse.org/xtend/documentation.html#polymorphicDispatch`)

```
1  protected String _printType(final Number x) {
2      return "it\'s a number";
3  }
4
5  protected String _printType(final Integer x) {
6      return "it\'s an int";
7  }
8
9  public String printType(final Number x) {
10     if (x instanceof Integer) {
11         return _printType((Integer)x);
12     } else if (x != null) {
```

```
13          return _printType(x);
14      } else {
15          throw new IllegalArgumentException("Unhandled parameter types: " +
16              Arrays.<Object>asList(x).toString());
17      }
18  }
```

Template Expressions:
Template expressions can be used for string literal generation and string concatenation. An example is shown in Listing 4.6. A template expression is encapsulated between two ''' literals. It may contain several expressions between the literals « and » to generate dynamic content. Template expressions provide a comprehensive basis to implement M2T transformations for code generation.

Listing 4.6. Template expression in Xtend (taken from `http://www.eclipse.org/xtend/documentation.html#templates`)

```
1  def someHTML(String content) '''
2      <html>
3          <body>
4              «content»
5          </body>
6      </html>
7  '''
```

Create Methods:

Create methods provide a caching mechanism for methods. All parameter values are used as unique identifier to cache the return object for the first method invocation. This cached object is returned, if the same parameter values are used in a subsequent method invocation. An example is shown in Listing 4.7.

Listing 4.7. Cached method in Xtend

```
1  def run() {
2    // A new list object is created for the parameters ("A","B") and cached
3    val list1 = createList("A","B")
4
5    // A new list object is created for the parameters ("B","C") and cached
```

```
6     val list2 = createList("B","C")
7
8     // The cached list object for the parameters ("B","C") is returned
9     val list3 = createList("B","C")
10  }
11
12
13  def create result: new ArrayList<String>()
14      createList(String string1, String string2) {
15
16      result.add(string1);
17      result.add(string2);
18  }
```

4.5 Xtext

Xtext[9] provides a model-driven framework for the creation of textual DSLs based on EMF. It contains a grammar language to define the grammar (concrete syntax) of a DSL. The grammar language generally provides the definition of language keywords as well as different rule types and relations between rules in order to specify the structure of the concrete syntax.

Based on an Xtext grammar, an initial language infrastructure can be generated. This includes the creation of a corresponding Ecore model (abstract syntax) for the defined grammar and a basic textual editor based on Eclipse. The generated language infrastructure has to be customized. Therefore, dedicated extension points are provided. For example, a model transformation has to be provided in Xtend (see Section 4.2) to map the defined language to an existing programming language. Furthermore, a validator can be implemented in Java to apply validation rules on EMF models that cannot be expressed with the grammar language of Xtext.

Based on its Ecore model (abstract syntax), an Xtext DSL can be further extended by a graphical concrete syntax with the *Graphical Modeling Framework* (GMF)[10] and *Graphiti*[11] from Eclipse.

A sample grammar defined with Xtext is shown in Listing 4.8. The elements `Domainmodel`, `Type`, `DataType`, `Entity`, and `Feature` are rules that

[9] `http://www.eclipse.org/Xtext/`

[10] `http://www.eclipse.org/modeling/gmp/`

[11] `http://www.eclipse.org/proposals/graphiti/`

form the basic structure of the language and the classes of the generated Ecore model (abstract syntax). Rules can reference other rules, which is represented as corresponding class hierarchy in the Ecore model. For example, the rule in lines 9 to 11 defines `Type` as super type for `DataType` and `Entity`. The `|` operator means that a `Type` element must be either represented by a `DataType` or an `Entity` element. Each rule can further define the concrete syntax by language keywords within single quotes ” and rules for variable contents and their types. For example, line 14 defines the language keyword 'datatype' followed by the variable content `name` of the type ID. The variable content `name` is represented by a corresponding attribute in the class `DataType` of the Ecore model. The type `ID` is a predefined type for unique identifiers. If an existing rule is referenced as type, a corresponding metamodel reference is created. For example, the reference of `Entity` in line 18 results in a `superType` attribute reference from the class `Feature` to the class `Entity` of the Ecore metamodel. A language example for the defined grammar is shown in Listing 4.9.

Listing 4.8. Example grammar in Xtext (taken from `http://www.eclipse.org/Xtext/documentation.html`)

```
1  grammar org.example.domainmodel.Domainmodel with org.eclipse.xtext.common.Terminals
2
3  generate domainmodel "http://www.example.org/domainmodel/Domainmodel"
4
5  Domainmodel :
6      elements += Type*
7  ;
8
9  Type:
10     DataType | Entity
11 ;
12
13 DataType:
14     'datatype' name = ID
15 ;
16
17 Entity:
18     'entity' name = ID ('extends' superType = [Entity])? '{'
19     features += Feature*
20     '}'
21 ;
22
23 Feature:
24     many?='many'? name = ID ':' type = [Type]
```

```
;
```

Listing 4.9. Language example of grammar in Xtext (taken from `http://www.eclipse.org/Xtext/documentation.html`)

```
datatype String

entity Blog {
  title: String
  many posts: Post
}

entity HasAuthor {
  author: String
}

entity Post extends HasAuthor {
  title: String
  content: String
  many comments: Comment
}

entity Comment extends HasAuthor {
  content: String
}
```

Chapter 5

Grid Computing

5.1 Introduction and Basic Terminology

A *Grid* is generally a geographically and organizationally distributed infrastructure for resource sharing. It originates from efforts in the 1990s to interlink US supercomputers with a high performance network for large-scale computing, e.g., in the project I-WAY (Information-Wide-Area-Year)[1]. Current national and transnational Grid initiatives in which Grid infrastructures have been built are TeraGrid (US)[2], OMII-UK (UK)[3], EGEE (EU)[4], and D-Grid[5].

Kesselman and Foster [1998] coined the term Grid in analogy to a power grid. It refers to Grid infrastructures that are based on computing resources, which are also called *Computational* or *Compute* Grid. Its utilization is called *Grid computing*.

> *Computational Grid*: "A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities." Kesselman and Foster [1998]

A Grid can generally consist of any type of resources such as computing and data resources as well as sensors and other scientific instruments. Foster et al. [2001] further specified the term Grid with the focus on resource

[1] http://www.nitrd.gov/pubs/bluebooks/1997/i-way.html
[2] http://www.teragrid.org
[3] http://www.omii.ac.uk/
[4] http://public.eu-egee.org/
[5] http://www.d-grid.de

sharing. It emphasizes a federated approach without centralized control, in which resource providers and users can negotiate individual *sharing rules* to form a so-called *virtual organization*.

> "... the Grid concept is coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations" [Foster et al. 2001]

Finally, Foster [2002] created a *three-point checklist* that summarizes the main characteristic of a Grid.

> "A Grid is a system that ...
> 1. coordinates resources that are not subject to centralized control ...
> 2. using standard, open, general-purpose protocols and interfaces ...
> 3. to deliver nontrivial qualities of service." [Foster 2002]

Foster et al. [2002] address the second point (standard, open, general-purpose protocols, and interfaces) by introducing the so-called Open Grid Services Architecture (OGSA). . OGSA is a basic architecture framework for service-oriented Grid infrastructures called *Service Grids*. It aims to foster a standardization process in the Grid community, whereby existing standards such as Web services are utilized. This standardization process is currently being pursued by the Open Grid Forum (OGF)[6] in several working groups. OGF has published several documents [7] including a specification for OGSA [Foster et al. 2006]. These efforts have finally led to a close collaboration between the Grid and Web service community.

One essential contribution of OGF is the participation in the development of the Web Services Resource Framework (WS-RF/WSRF) specification [OASIS 2006], which is an official OASIS standard. WSRF is a framework to create and access stateful *resources* with Web services. A resource generally consists of a unique identifier, a set of *resource properties*, and a life cycle. The current version 1.2 of the WSRF standard consists of the following specifications:

[6] http://www.ogf.org

[7] http://www.ogf.org/gf/docs/

- *WS-Resource*: Defines a *WS-Resource* as a combination of a Web service and a (stateful) resource. It further defines a *factory/instance pattern* to create and access resources via Web services. A WS-Resource and its associated resource can be referenced with a WS-Addressing [W3C 2006] endpoint, which contains the unique identifier of the resource.
- *WS-ResourceProperties (WSRF-RP)*: Defines a method to represent, access and modify resource properties via Web services. Resource properties usually represent the state of a WS-Resource.
- *WS-ResourceLifetime (WSRF-RL)*: Defines read-only WS-Resource properties that represent the lifetime of a WS-Resource based on the life cycle of the corresponding resource. It further defines methods to destroy a WS-Resource.
- *WS-ServiceGroup (WSRF-SG)*: Defines the grouping of multiple WS-Resources as a *service group* and operations that can be applied to it.
- *WS-BaseFaults (WSRF-BF)*: Defines common types for base faults and their utilization in WSRF-compliant Web services.

Each WSRF specification is associated with a corresponding WSDL definition. The WS-Resource concept is usually realized with two Web services, see Figure 5.1. A *factory service* is a standard Web service for creating resources. It returns a WS-Addressing endpoint that refers to the corresponding WS-Resource. This endpoint is used for invocations of an *instance service* in order to access the WS-Resource and its associated resource. An instance service is implemented as stateful, WSRF-compliant Web service that provides the corresponding WSDL interfaces defined in the WSRF specifications. The term *WSRF service* refers to the combination of a factory and an instance service. We further use the term *WSRF (service) instance* for a concrete WS-Resource. The unique identifier of a WS-Resource is called *WSRF (service) instance id*.

WSRF is implemented by the Grid middleware UNICORE[8] and Globus Toolkit[9], see Section 5.2. Many services in these Grid middlewares are

[8] http://www.unicore.eu/

[9] http://www.globus.org/toolkit/

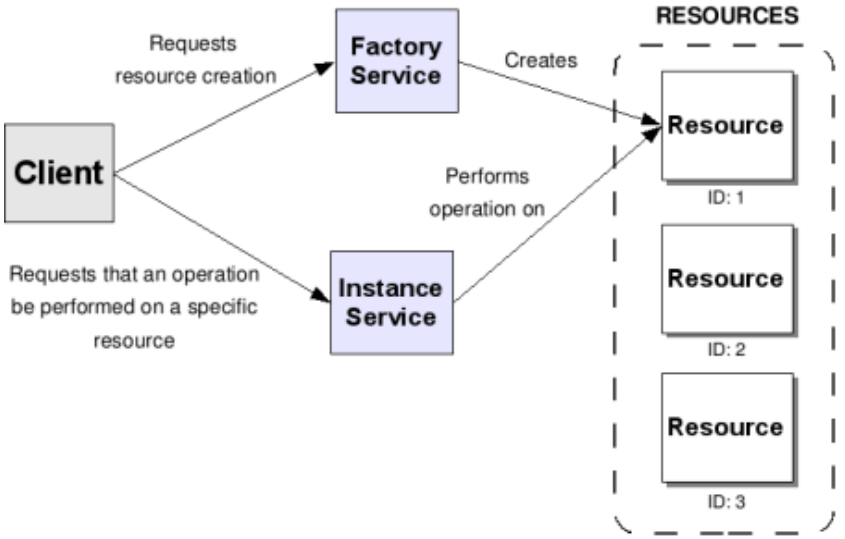

Figure 5.1. WS-Resource factory pattern (taken from `http://gdp.globus.org/gt4-tutorial/multiplehtml/ch05s01.html`)

realized as WSRF services, e.g., for the execution of a computation, which is described as a so-called *compute job*. Web services (and WSRF services) that are deployed in a Grid middleware are also called *Grid services* [Treadwell 2007].

5.2 Grid Middleware

A Grid middleware is generally a software for the common and standardized access to Grid resources. We focus on the Grid middleware UNICORE 6[10] and Globus Toolkit 4.0 (GT4)[11]. Both middlewares implement the WSRF standard in order built Service Grids.

[10] `http://www.unicore.eu/download/unicore6/`
[11] `http://www.globus.org/toolkit/downloads/4.0.8/`

5.2.1 Globus Toolkit 4 (GT4)

GT4 is developed by the Globus Alliance[12] and available as open source. It implements the WSRF standard[13]. GT4 provides a set of essential WSRF services such as the job execution service WS-GRAM (*Grid Resource Allocation and Management*)[14] and the data transfer service RFT (*Reliable File Transfer*)[15]. Furthermore, several command line clients are provided such as `globusrun-ws` for job submission. Even if the built-in functionality of GT4 is sufficient for a basic Grid site, the software is regarded as a toolkit to create custom clients, services, and further extensions.

The architecture of GT4 is shown in Figure 5.2. Besides service components based on Web or WSRF services, it consists of a couple of *non-WS* components such as GridFTP[16]. GridFTP is used to execute so-called *third party data transfers* between two GridFTP servers, initiated and controlled by an external client (third party). It is an extension for FTP (*File Transfer Protocol*) and provides several performance options, e.g., for utilizing parallel data channels. A GridFTP server is included in the GT4 installation package.

A compute job, which can be submitted via WS-GRAM, is described in a custom, XML-based job definition language. It generally specifies an executable that has to be invoked and additional parameters as well as file staging activities to transfer input and output files via GridFTP. Required data transfers for file staging are executed and coordinated with the RFT service. A submitted job is passed via a *job manager* to a local resource management/batch system such as TORQUE[17]. WS-GRAM provides an API for implementing custom job managers.

The security infrastructure in GT4 is based on GSI, which is described in Section 5.3.

[12] `http://www.globus.org/alliance/`
[13] A draft version of WSRF 1.2.
[14] `http://www.globus.org/toolkit/docs/4.0/execution/wsgram/`
[15] `http://www.globus.org/toolkit/docs/4.0/data/rft/`
[16] `http://www.globus.org/toolkit/docs/4.0/data/gridftp/`
[17] `http://www.adaptivecomputing.com/products/open-source/torque/`

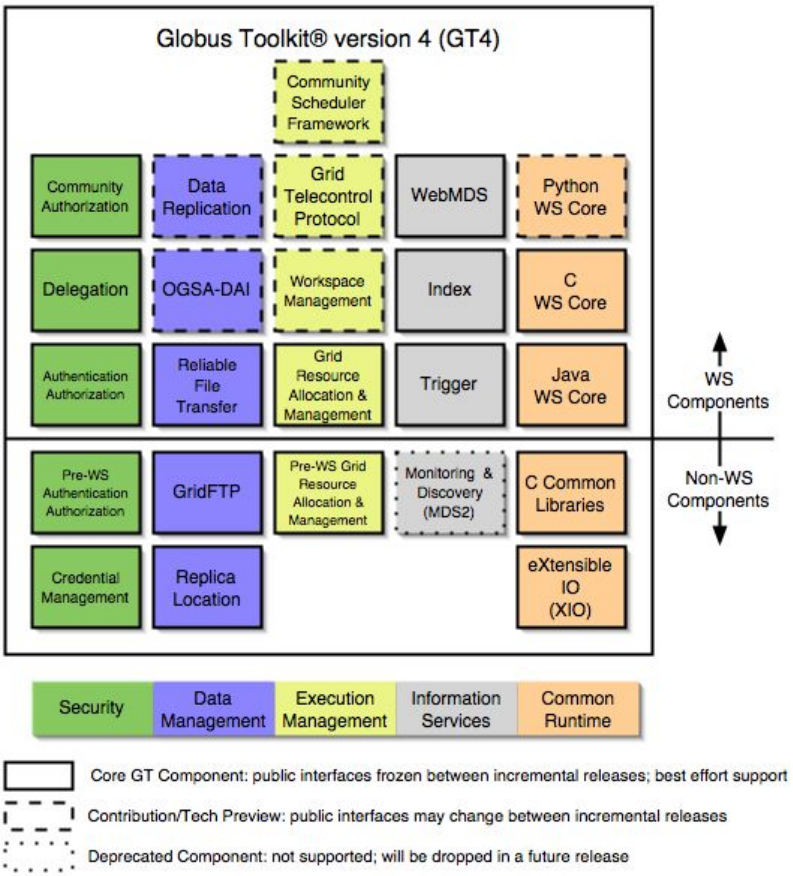

Figure 5.2. Globus Toolkit 4 architecture (taken from `http://gdp.globus.org/gt4-tutorial/multiplehtml/ch01s04.html`)

5.2.2 UNICORE 6

UNICORE 6 (**UN**iform **I**nterface to **CO**mputing **RE**sources) is a development of the UNICORE community and published as an open source project. The main objective of UNICORE 6 is to provide a uniform interface to access heterogeneous supercomputer. It consists of comprehensive client and server components. UNICORE 6 implements the WSRF standard and supports many further Web service and OGSA standards. It provides WSRF services for job execution as well as data transfers with different transport protocols including GridFTP. It further provides different graphical clients that can be individually extended via a plugin mechanism.

The architecture of UNICORE 6 consists of a *client layer*, *service layer*, and *system layer* and is shown in Figure 5.3. Central components are the *Gateway* and the service container *UNICORE/X* (UNICORE WS-RF hosting environment in Figure 5.3). The Gateway component is an external entry point for each component deployed at a UNICORE 6 site. It accepts only trusted and authenticated clients. The UNICORE/X component is the service container for all WSRF services and other Web services. It also contains the job execution engine called XNJS (*Network Job Supervisor*). UNICORE/X provides additional security mechanisms for authorization.

UNICORE 6 uses the *Job Submission Description Language* (JSDL)[18] to define compute jobs. JDSL is an OGF specification. A compute job is submitted to an XNJS via a corresponding WSRF service. The XNJS itself passes the job via a *Target System Interface* (TSI) to a local resource management/batch system. The supported resource management/batch systems of a TSI can be extended with an individual adapter.

Security in UNICORE 6 is generally based on a *public-key infrastructure* (PKI) with certificate authorities (CA) and X.509 certificates. The communication with a Gateway and between many UNICORE 6 components is encrypted with SSL/TLS (*Secure Sockets Layer/Transport Layer Security*). Authentication in the Gateway is based on a standard X.509 certificate. An X.509 certificate may contain additional *Security Assertion Markup Language* (SAML)[19] attributes that are used for authorization in UNICORE/X. Au-

[18] http://www.gridforum.org/documents/GFD.56.pdf

[19] https://www.oasis-open.org/committees/security/

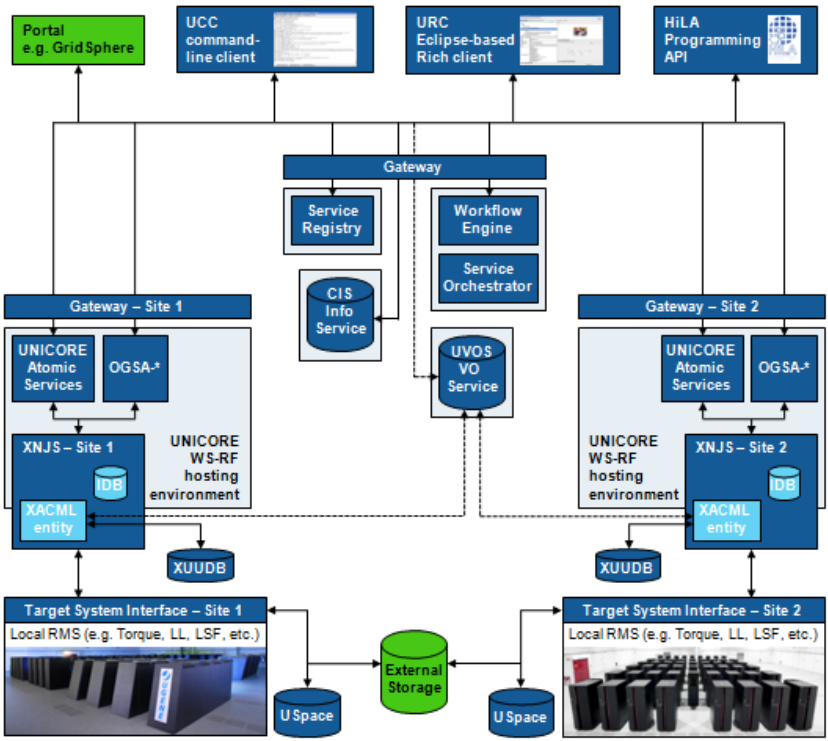

Figure 5.3. UNICORE 6 architecture (taken from `http://www.unicore.eu/unicore/architecture.php`)

thorization rules can be expressed with the *eXtensible Access Control Markup Language* (XACML)[20].

5.3 Grid Security Infrastructure (GSI)

The *Grid Security Infrastructure* (GSI) [Globus Security Team 2005] provides the security infrastructure for GT4 and defines three security mechanisms,

[20] `http://www.oasis-open.org/committees/xacml/`

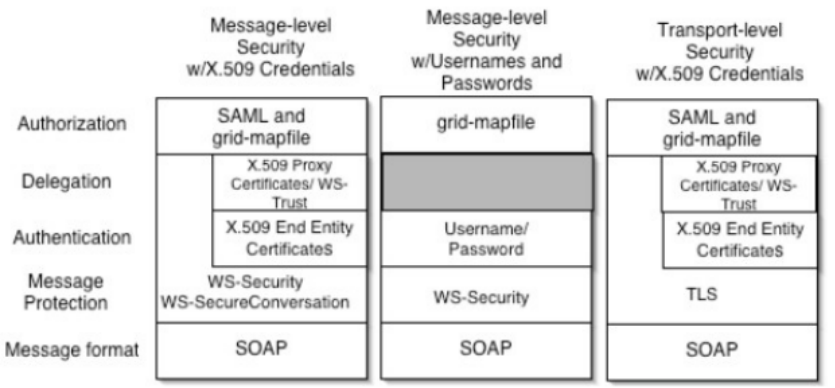

Figure 5.4. Overview of GSI (taken from [Globus Security Team 2005])

see Figure 5.4. Each security mechanism provides a particular method for message protection, authentication, delegation, and authorization. Therefore, GSI generally utilizes existing standards. Message protection is either based on standard SSL/TLS, WS-Security[21], or WS-SecureConversation[22]. Authentication is based on X.509 certificates (PKI with trusted CAs), which are also called *end entity certificates* (EEC). Authorization is based on either SAML or a so-called *grid-mapfile*. A grid-mapfile generally defines a mapping from a *distinguished name* (DN) of an EEC to a local UNIX user ID on a Grid site.

In a Grid infrastructure, it may be necessary that a Grid site has to act with the identity of a user. For example, file staging activities during the execution of a compute job usually require an authentication credential for a remote GridFTP server. Such a credential must be *delegated* previously. The EEC cannot be used for security reasons because the corresponding private key would be stored unencrypted at the GT4 site. Thus, GSI introduced a credential delegation mechanism based on so-called *proxy certificates*. A proxy certificate is a self-signed X.509 certificate (and corresponding private key) that is initially derived from an EEC. Subsequently, an arbitrary chain

[21] http://www.oasis-open.org/committees/wss/

[22] http://www.oasis-open.org/committees/ws-sx/

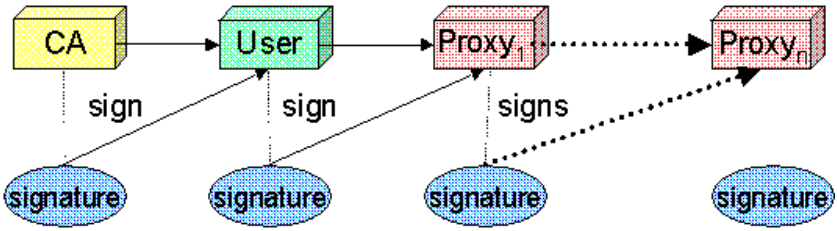

Figure 5.5. Chain of proxy certificates (taken from `http://www.globus.org/security/overview.html`)

of further proxy certificates may be created as shown in Figure 5.5. A proxy certificate allows the authentication with the identity of the EEC and can be delegated to a GT4 site via a corresponding WSRF-based delegation service. Whenever an authentication credential is required, a child proxy certificate is derived from the delegated proxy certificate.

The revocation and invalidation mechanism for X.509 certificates is not applicable for proxy certificates. Thus, a proxy certificate has usually a short lifetime to minimize misuse in case of loss.

5.4 BIS-Grid Workflow Engine

The BIS-Grid Workflow Engine [Höing et al. 2009; Gudenkauf et al. 2010a][23] provides a transparent Grid proxy for any BPEL 2.0 compliant process engine in order to execute workflows in Grid environments. It was developed in the D-Grid project BIS-Grid[24] [Hasselbring 2010]. The BIS-Grid Workflow Engine provides a plugin mechanism to support different BPEL process engines. A corresponding adapter for ActiveBPEL[25] of Active Endpoints[26] is included in the installation package. It further provides comprehensive security mechanisms to run secure workflows in distributed infrastructures [Höing 2010].

[23] `http://bis-grid.sourceforge.net/`
[24] `http://www.bisgrid.de`
[25] Version 5.0, which is no longer maintained as open source project.
[26] `http://www.activeendpoints.com/`

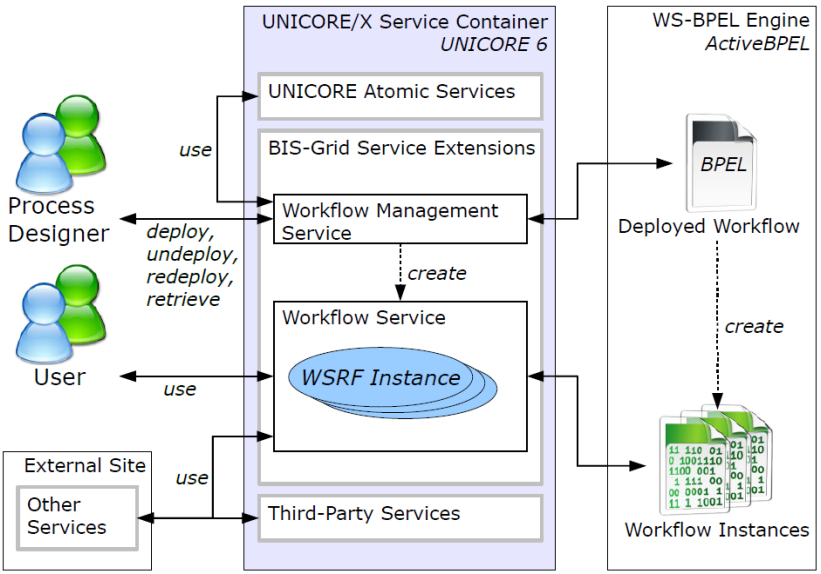

Figure 5.6. BIS-Grid Workflow Engine architecture (taken from `http://www.unicore.eu/summit/2009/presentations/05_Gudenkauf_BIS-Grid.pdf`)

The architecture of the BIS-Grid Workflow Engine is shown in Figure 5.6. Its core components are the *Workflow Management Service* and the *Workflow Service* that are deployed in the UNICORE/X service container of UNICORE 6.

The Workflow Management Service provides the deployment and undeployment of workflows. It uses a deployment package (ZIP archive) that contains all of the required BPEL process files and a BIS-Grid Workflow Engine-specific deployment descriptor. Besides common binding information, the deployment descriptor contains additional security configurations for external service invocations. A workflow deployment consists of two steps. First, the workflow is deployed to a BPEL workflow engine via the corresponding adapter. The adapter creates the engine-specific deploy-

ment descriptor and deployment package. Second, a *Workflow Service* is dynamically deployed in UNICORE/X. Each deployed workflow has its own Workflow Service. A workflow undeployment removes the Workflow Service and the BPEL process.

The Workflow Service handles incoming messages and executes external service invocations. Each WSRF instance of the Workflow Service has a corresponding workflow instance in the BPEL workflow engine. Thus, each workflow execution has exactly one WSRF instance and one workflow instance. Incoming service invocations for the workflow instance are handled by the WSRF instance and then sent to the BPEL workflow engine. This may require a previous authorization based on the standard UNICORE 6 security features. For external service invocations, we have extended the UNICORE/X service container with a standard HTTP(S) proxy that is used by the BPEL workflow engine[27]. If a workflow instance sends a message for an external service invocation, the proxy ensures that this message is routed to the corresponding WSRF instance. Based on the binding information and security configuration in the deployment descriptor, the WSRF instance selects the method to execute the service invocation. Currently, the invocation of standard Web services, UNICORE 6 services, and GT4 services is supported by the BIS-Grid Workflow Engine. The invocation of standard Web services and UNICORE 6 services is based on built-in mechanisms in UNICORE 6. For the invocation of GT4 services including the support of GSI we extended the service invocation mechanism of UNICORE 6 [Gudenkauf et al. 2010b]. Required credentials can be statically defined in the security configuration of the deployment descriptor or dynamically added to a WSRF instance. Proxy certificates used for GT4 service invocations are usually dynamically added.

In summary, the complete message communication of a BPEL process execution is handled by the BIS-Grid Workflow Engine. All security aspects required for the interaction in Grid infrastructures are provided by UNICORE 6 itself or custom extensions (GT4 support). The used BPEL process engine must be BPEL 2.0 compliant and must support the use of an HTTP(S) proxy.

[27] Usually every BPEL execution environment provides the utilization of HTTP(S) proxies.

Part II

Model-Driven Scientific Workflow Engineering with MoDFlow

Chapter 6

Introduction to MoDFlow

This chapter introduces MoDFlow, a flexible and extendable approach for ***Mo**del-**D**riven Scientific Work**Flow** Engineering*. Section 6.1 summarizes the basic assumptions for MoDFlow. Section 6.2 describes the general concept of MoDFlow and gives an overview about its central components.

6.1 Basic Assumptions

The basic assumptions for MoDFlow are:

- We focus on Service Grids (see Chapter 5) as execution infrastructures for scientific workflows. Thus, workflow activities are executed by Web or Grid services. We further focus on the Grid middlewares GT 4 and UNICORE 6.
- We use BPEL (see Chapter 2) as workflow language for the technical execution of scientific workflows in Service Grids [Scherp et al. 2010; Gudenkauf et al. 2010a]. To this end, we aim to exploit only standard language elements of BPEL so that different BPEL-compliant process engines can be used. The utilization of design and runtime extensions as well as runtime only extensions [Kopp et al. 2011] for BPEL is avoided.
- We use Apache ODE (Orchestration Director Engine)[1] for the execution of BPEL processes. The interoperability of BPEL process engines such as Apache ODE with Service Grids is based on the BIS-Grid Workflow Engine (see Chapter 5).

[1] http://ode.apache.org/

▻ We use the Eclipse Modeling Framework (EMF)[2] and corresponding tools as basis for model-driven software development (MDSD, see Chapter 4). Thus, each workflow model is represented by an EMF model that conforms to a corresponding Ecore model (metamodel).

6.2 Concept and Components of MoDFlow

MoDFlow is a continuation of recent efforts (see Chapter 15) in the scientific workflow domain to utilize the standardized and established business workflow language BPEL for scientific workflow execution in service-oriented execution infrastructures such as Service Grids. Its major objective is to provide a model-driven concept in order to bridge the gap between domain-specific modeling and technical execution of scientific workflows, whereby we introduce BPMN as common exchange format between both aspects [Scherp and Hasselbring 2010a,b]. As implication, we differentiate between a domain-specific and technical layer for scientific workflows (see Chapter 3).

Domain-specific modeling concerns the creation of a scientific workflow by a scientist with a workflow editor. It usually provides a simple graphical notation of a corresponding workflow language, which supports the definition of data dependencies between workflow activities as well as few control flow elements. Workflow activities that are specific for a particular scientific domain are often predefined in a repository, which can be used by a workflow editor. They are used by scientists for creating scientific workflows.

For the technical execution of a scientific workflow by a workflow engine, its workflow activities are executed based on the defined process flow (data and control flow dependencies). In service-oriented execution infrastructures such as Service Grids the execution of a workflow activity is usually based on one or multiple service calls. However, scientists want to focus on domain-specific aspects when modeling a workflow and they do not want or do not have the knowledge to deal with such technical details that are required for execution. Such details are hidden from the scientist,

[2] http://www.eclipse.org/modeling/emf/

for example, attached as additional information to each workflow activity of a repository.

The used workflow language in SWfMS such as Kepler references and configures internal software components to execute workflow activities so that the created scientific workflow is directly executed. In contrast, in the SWfMS Pegasus modeling and execution is separated by two workflow languages. The workflow language that is used for scientific workflow modeling is *compiled* to another, executable workflow language. Thereby, additional technical details are added during this mapping. Such a mapping can also be regarded as model transformation and be implemented with corresponding MDSD technologies (see Chapter 4).

We use BPEL for the execution of scientific workflows at the technical layer, which is a technical workflow language designed for IT-specialists and thus not suitable for scientists. Therefore, a domain-specific abstraction for executable business workflow languages such as BPEL is required for scientific workflows. For providing such an abstraction, we consider the following two objectives:

1. To allow the utilization of different workflow languages for the domain-specific and technical layer, which means BPEL may be replaced by any similar workflow language.

2. To facilitate the adoption of business workflow technologies in the scientific workflow domain, whereby we do not want to invent a complete new scientific workflow language.

To this end, we introduce the *intermediate layer* as additional layer between the domain-specific layer and technical layer. It is based on a BPMN metamodel subset with custom extensions and provides a common exchange format between the domain-specific and the technical layer. Thereby, an existing scientific workflow language may be used for the domain-specific layer and an executable business workflow language such as BPEL for the technical layer. This allows for a better adoption of new workflow languages technologies for both layers. A scientific workflow model created at the domain-specific layer is mapped to a scientific workflow model at the intermediate layer, which is subsequently mapped to a scientific workflow

model at the technical layer. Finally, we distinguish between the following three scientific workflow models and two model transformations:

- *Domain-specific Workflow Model (DWM)*: A DWM is associated with the domain-specific layer and is created by a scientist with a corresponding workflow editor. It can be based on any existing scientific workflow language, whereby MoDFlow does not focus on a certain language.
- *Intermediate Workflow Model (IWM)*: An IWM is associated with the intermediate layer and is the result of a *DWM2IWM mapping*. It is based on a BPMN metamodel subset with custom extensions.
- *Executable Workflow Model (EWM)*: An EWM is associated with the technical layer and is the result of an *IWM2EWM mapping*. It can be based on any existing business workflow language, which can be executed with a corresponding workflow engine. In MoDFlow, we focus on BPEL as executable workflow language for EWMs. Thus, the IWM2EWM mapping is realized as BPMN-to-BPEL model transformation.

As BPMN and BPEL (or similar business workflow languages) are focused on control flow, we consider an IWM and an EWM as such, too. Scientific workflows, however, are usually focused on data flow (see Chapter 3). Thus, within a DWM2IWM mapping the data flow of a DWM has to be mapped to corresponding control flow elements in an IWM. The foundations for such a mapping are discussed in Chapter 10.

Since version 2.0, the BPMN standard provides a metamodel for standardized serialization and model exchange. It also provides means for own extensions. We use this metamodel to define a BPMN subset with custom extensions that represents IWMs. The graphical notation of BPMN is not used as an IWM is only used as non-graphical exchange format.

The main reasons to use BPMN can be summarized as follows:

- BPMN is a widely accepted standard in the business domain. Thus, it plays a significant role in our efforts to utilize business workflow technologies in the scientific workflow domain. With advancement such as the metamodel and the defined execution semantics for a BPMN subset since version 2.0, we believe that BPMN has strengthened its position in the business domain and its utilization will increase in future.

- The capabilities of BPMN are sufficient to represent the process flow of workflow activities within a scientific workflow. The extension mechanism of the BPMN metamodel can be used to define additional information that cannot be expressed with existing BPMN elements.

- We can use existing approaches for BPMN-to-BPEL mappings, see Chapter 15. The BPMN standard already defines a basic BPEL mapping for a BPMN subset.

- The BPMN metamodel is formally specified by CMOF metamodels, which have been used to create corresponding Ecore models [Hille-Doering 2010]. This facilitates the technical realization of model transformations for DWM2IWM and IWM2EWM mappings with EMF.

- The utilization of BPMN can be expanded in future, which is discussed in Chapter 17. For example, the graphical notation of BPMN can be used for the domain-specific layer. Or the execution semantics of BPMN can be used for scientific workflow execution at the technical layer.

The concept of the IWM and the corresponding mappings as well as its utilization in the scientific workflow domain is specified in MoDFlow. It consists of the following components that are shown in Figure 6.1.

MoDFlow.BPMN:
MoDFlow.BPMN specifies a BPMN metamodel subset with custom extensions that represents IWMs. The basic definition of workflow activities and control flow dependencies is based on existing BPMN elements. These elements are individually extended by adding specific extension elements for two reasons. First, to represent information that is not covered by existing BPMN elements, e.g., the definition of parameter sweeps for workflow activities. Second, to represent technical information such as the configuration of Web service calls, which is required for a IWM2EWM mapping. All custom extensions are based on the metamodel extension mechanism of BPMN. MoDFlow.BPMN is further described in Chapter 8.

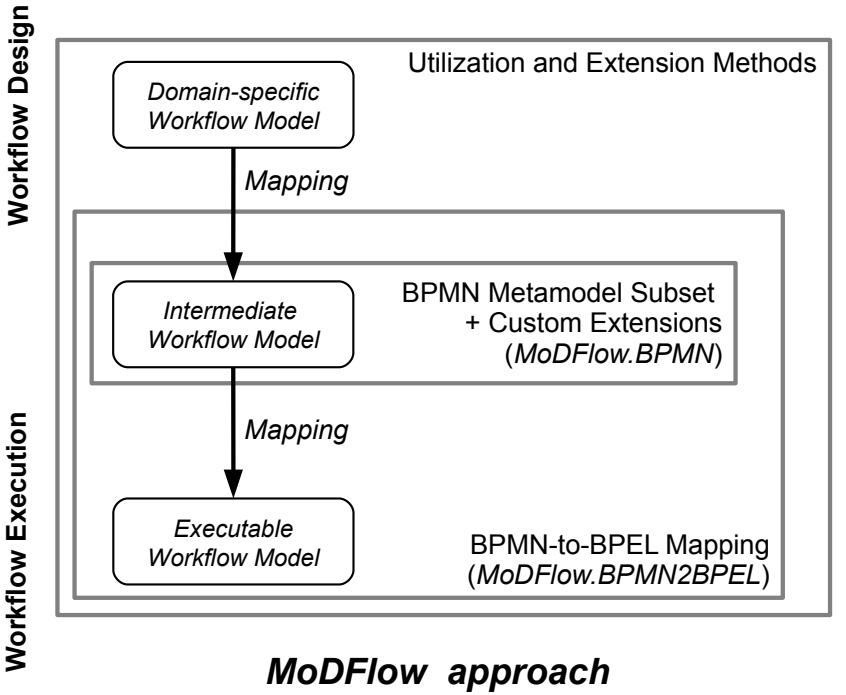

Figure 6.1. Overview of the MoDFlow approach

MoDFlow.BPMN2BPEL:
MoDFlow.BPMN2BPEL specifies an IWM2EWM mapping that maps MoDFlow.BPMN to BPEL within three single transformation steps. It is based on the BPEL mapping in the BPMN standard. MoDFlow.BPMN2BPEL generates BPEL code that executes a scientific workflow as defined in an IWM. Thereby, the structure of the resulting BPEL process is significantly more complex than the original BPMN process. For example, if the execution of a workflow activity consists of several Web service invocations with corresponding fault handling. The MoDFlow.BPMN2BPEL mapping is further described in Chapter 9.

Utilization and Extension Methods:
MoDFlow defines different methods how to use and how to extend MoDFlow.BPMN and MoDFlow.BPMN2BPEL. This concerns the representation of a DWM at the domain-specific layer and a corresponding DWM2IWM mapping. For representing DWMs, we focus on the creation of domain-specific languages (DSLs), which are one important means in the MDSD domain (see Chapter 4). We further discuss the exploitation of different mechanisms to customize and extend MoDFlow.BPMN and MoDFlow.BPMN2BPEL. All utilization and extension methods of MoDFlow are presented in Chapter 10.

Chapter 7

Scientific Workflow Requirements on MoDFlow

This chapter defines requirements on MoDFlow, which are derived from common requirements on SWfMSs in the scientific literature and own observations of existing SWfMSs such as GWES, Kepler, Pegasus, Taverna, Triana, and Trident. We thereby focus on aspects and components of scientific workflows and SWfMSs that are addressed by MoDFlow. Thus, the defined requirements are separated by workflow editor, DWM, DWM2IWM mapping, IWM, IWM2EWM mapping, EWM and workflow engine.

In Section 7.1, we discuss common requirements on scientific workflows and SWfMS as well as its relation to MoDFlow. Concrete requirements on MoDFlow are then defined in Section 7.2.

7.1 Common Requirements on Scientific Workflows and SWfMS

[Görlach et al. 2011] summarizes the main requirements for a SWfMS as *data-driven*, *advanced data handling*, *flexibility*, *monitoring*, *reproducibility*, *robustness*, *scalability*, and *domain-specific requirements*. We briefly discuss the meaning of each requirement and its relevance for a workflow editor, a DWM, a DWM2IWM mapping, an IWM, an IWM2EWM mapping, an EWM, and a workflow engine below. For a detailed description of each requirement, please refer to [Görlach et al. 2011]. The general term *workflow model* is used in the following, if a clear distinction between DWM, IWM, and EWM is not possible.

▻ *Data-driven*: Prevalent SWfMS such as Kepler, Taverna, Triana, and Trident provide graphical and data flow-centric workflow modeling with corresponding workflow languages. A workflow is represented as directed graph with nodes for workflow activities and edges for data dependencies between them. A scientist designs a workflow by dragging predefined and often domain-specific workflow activities from a repository to a work sheet. As workflow activities usually consume and produce data, each workflow activity defines input and output parameters that are usually visualized graphically such as in Kepler. The execution order of workflow activities is defined by data dependencies between output and input parameters, which are usually visualized as lines with an ending arrow to indicate the direction. Thereby, data dependencies can be used to define a sequential or a parallel execution of workflow activities. Certain control flow elements are often supported to define conditional paths and loops.

A graph-based and data flow-centric modeling approach is seen as the most intuitive for scientists from data-intensive research domains [Gil et al. 2007]. Thus, data-driven respectively data flow-centric modeling is a common requirement for SWfMS, which implies additional mechanisms for *advanced data handling* [Görlach et al. 2011]. A scientist generally needs data management support, for example to search and select data for workflow processing. To efficiently cope with huge amounts of data, workflow activities are usually executed on an external computing resource with data from external data sources. This requires a mechanism to use data references for external data sources in workflows activities that can be used to execute so-called third party data transfers directly between data locations and computing resources.

The representation and creation of a data flow-centric workflow model concerns the workflow editor and consequently the workflow language used for the DWM. BPMN (IWM) and BPEL (EWM), however, are both control flow-centric[1]. Thus, data dependencies in a DWM model must be translated to the appropriate control dependencies during a DWM2IWM mapping. An approach for such a mapping is presented in Chapter 10.

[1] Business workflow languages are usually control-flow centric

Therefore, to preserve all data dependencies between input and output parameter of workflow activities a data link extension for BPMN is needed.

A concrete representation of data references mainly depends on the kinds of data in a particular scientific domain and its technical interpretation often requires particular solutions. Thus, we do not provide a generic approach to express data references. We regard data references as special data type that can be interpreted within the model transformations to generate an EWM or during its execution. For example, a workflow activity may consume a data reference as input parameter and itself initiates the needed data staging before its execution.

Finally, data management support must be provided by the workflow editor.

- *Usability*: A SWfMS must be tailored for the use by scientists especially regarding workflow modeling and workflow execution. It must be considered that scientists usually do not have comprehensive programming skills and do not want to deal with technical details during workflow modeling. Thus, usability is an important issue for a SWfMS.

  Usability concerns the workflow editor. For example, it should provide a repository with basic predefined workflow activities that are commonly used in the scientist's domain.

- *Flexibility*: Flexibility is the ability to react on expected and unexpected changes in the execution environment, for example, if a resource fails. This can be realized by automatic or manual workflow model modifications during runtime. Or by carefully designed exception handling for expected errors to avoid the modification of the workflow model (*avoid change* [Görlach et al. 2011]).

  Manual modifications are applied to the DWM by scientists with a workflow editor. Automatic modifications are applied to the EWM by a workflow engine. The use of special constructs to react on known changes concerns the execution infrastructure and thus an EWM. A scientist usually does not want to define such constructs in the DWM

manually. Thus, these constructs must be created during an IWM2EWM mapping.

- *Monitoring*: A scientist needs information about a workflow execution and inspects intermediate result. Thus, a SWfMS must provide appropriate monitoring functions. Besides general status information about the workflow execution, monitoring data also include provenance information about processed data. Special constructs within the workflow model may be used to collect monitoring data within each workflow instance, for example for data provenance.

  Usually, technical monitoring data is collected by a workflow engine during a workflow execution and must be visualized in a workflow editor. Similar to flexibility, special workflow constructs to collect monitoring data within a single workflow instance concerns the EWM. These constructs must be created during an IWM2EWM mapping.

- *Reproducibility*: The reproducibility of scientific results is an important requirement for good scientific practice. It must be possible to re-run an existing workflow in order to reproduce the data output, which usually includes the utilization of provenance data of previous runs. Thus, monitoring and data provenance is a prerequisite for reproducibility.

  Reproducibility concerns the workflow editor. To re-run a workflow the corresponding DWM together with provenance data must be loaded by a workflow editor and then executed. The provenance data can be used to find the used input data to the re-run as well as to compare the created output data with previous runs. Monitoring as prerequisite for reproducibility is addressed above and thus not mentioned here again.

- *Robustness*: As scientific workflows executions are often long-running, they must be robust in case of failures. This includes that the instance of a workflow execution is persisted. Failures in the execution infrastructure that kills the execution of workflow activities or even a crash of the workflow engine itself must not lead to an uncontrolled and complete abortion of the workflow execution. It must be restarted after the erroneous situation is resolved based on the last stable state. Furthermore,

flexibility mechanisms by special constructs in a workflow model are a way of increasing the robustness of a SWfMS.

Robustness concerns the workflow engine that, for example, provides a persistence mechanism. Flexibility can be achieved by special constructs in an EWM as addressed above.

- *Scalability*: A SWfMS must scale, for example, by providing distributed execution of workflows.

  Scalability mainly concerns the workflow engine. It often depends on an optimized resource selection.

- *Domain-specific requirements*: Each scientific domain may have domain-specific requirements, for example, Görlach et al. [2011] focus on simulation workflows. Our domain-specific requirements are derived from our application scenarios (see Chapter 14) and not specific for a certain scientific domain. This includes the definition of parameter sweeps for a workflow activity and the support for the invocation of services deployed in the Grid Middleware Globus Toolkit 4 and UNICORE 6, e.g., a job submission service to execute workflow activities.

  A parameter sweep must be defined with a workflow editor and represented in the DWM. Its information must be preserved during the DWM2IWM mapping to an IWM, which must provide corresponding language elements. The support of Globus Toolkit 4 and UNICORE 6 concerns the workflow engine, which must support the Grid middleware-specific security mechanisms.

The discussion of the main requirements for SWfMSs and their relation to MoDFlow is summarized in Table 7.1.

7.2 Definition of Requirements on MoDFlow

Based on the findings in Section 7.1 we define common requirements for a workflow editor, a DWM, a DWM2IWM mapping, an IWM, an IWM2EWM mapping, an EWM, and a workflow engine, which have to be addressed by

Table 7.1. Main requirements for SWfMSs and their relation to MoDFlow

	Workflow Editor	DWM	DWM2IWM Mapping	IWM	IWM2EWM Mapping	EWM	Workflow Engine
Data-driven	+	+	+	+	+		
Usability	+						
Flexibility: Change of Workflow Model	+						+
Flexibility: Workflow Language Constructs					+	+	
Monitoring	+				+	+	+
Reproducibility	+				$+^a$	$+^a$	$+^a$
Robustness					$+^b$	$+^b$	+
Scalability							+
Parameter Sweep	+	+	+	+	+		
Grid Middleware							+

[a] Derived from *Monitoring*.
[b] Derived from *Flexibility: Workflow Language Constructs*

MoDFlow. Please note that these requirements are not all-encompassing for scientific workflows and SWfMSs. MoDFlow focuses on central and basic aspects for the used workflow models (DWM, IWM, EWM) and mappings (DWM2IWM mapping, IWM2EWM mapping). However, it must provide a certain flexibility and extensibility for individual customization. A workflow editor must generally be able to create a corresponding DWM and a workflow engine to execute a corresponding EWM, Special topics, e.g. data provenance and resource optimization, are out of scope and omitted.

Requirements on a workflow editor:

- **RQ_WF-ED_01** *Data Driven Workflow Modeling*: The workflow editor should provide the data flow-centric creation of a DWM based on a visual, graph-based representation. Workflow activities and data dependencies between them as well as input and output parameter should be visualized suitably.
- **RQ_WF-ED_02** *Repository*: The workflow editor should provide a repository for predefined common and domain-specific workflow activities.

Requirements on a DWM:

- **RQ_DWM_01** *BPMN 2.0 Restrictions*: A complete mapping of the used workflow language to the BPMN metamodel subset with custom extensions must be possible. Otherwise, the definition of a DWM must be restricted to workflow language elements that can be mapped.
- **RQ_DWM_02** *Workflow Activity Representation*: The used workflow language must support the representation of workflow activities with input and output parameters.
- **RQ_DWM_03** *Data Flow-centric Representation*: The used workflow language should be data flow-centric with optional control flow-elements such as sequential, parallel, conditional, and looped execution.
- **RQ_DWM_04** *Parameter Sweep Representation*: The used workflow language must support language constructs to define parameter sweeps for workflow activities.

Requirements on a DWM2IWM mapping:

- **RQ_DWM2IWM_01** *BPMN 2.0 Mapping*: A mapping must be provided so that a DWM is mapped to an IWM based on the used BPMN metamodel elements with custom extensions. If a DWM is data flow-centric, the data flow dependencies must be replaced by corresponding control flow dependencies.

Requirements on an IWM:

- **RQ_IWM_01** *BPMN 2.0 Compliance*: An IWM must be based on BPMN process elements and only standard compliant extensions are allowed. Each IWM must be a valid process model with respect to the BPMN standard.
- **RQ_IWM_02** *Control Flow Elements*: The definition of sequential, parallel, conditional, and looped execution of workflow activities creates a minimal set of control flow elements that must be supported.
- **RQ_IWM_03** *Workflow Activities*: The definition of workflow activities with input and output parameters must be supported.
- **RQ_IWM_04** *Data Dependencies*: The definition of data dependencies must be supported between input and output parameters of workflow activities.
- **RQ_IWM_05** *Parameter Sweeps*: The definition of parameter sweeps for workflow activities must be supported.
- **RQ_IWM_06** *Extensibility*: An extension mechanism must be provided to define individual customizations.

Requirements on an IWM2EWM mapping

- **RQ_IWM2EWM_01** *Executable Workflow Language Mapping*: A mapping must be provided to map all BPMN metamodel elements and custom extensions of an IWM to an EWM based on the workflow language used for EWMs.

- ▻ **RQ_IWM2EWM_02** *Workflow Engine Deployment Descriptor*: A deployment descriptor must be created for at least one workflow language that supports the workflow language used for EWMs.
- ▻ **RQ_IWM2EWM_03** *Flexibility Constructs*: Additional language constructs may be generated to increase flexibility and robustness of workflow execution.
- ▻ **RQ_IWM2EWM_04** *Monitoring Constructs*: Additional language constructs may be generated for the collection of monitoring and provenance data.
- ▻ **RQ_IWM2EWM_05** *Extensibility*: An extension mechanism must be provided to define individual customizations.

Requirements on an EWM

- ▻ **RQ_EWM_01** *IWM2EWM Mapping*: A mapping of the allowed BPMN metamodel elements with custom extensions for an IWM to the used workflow language must be possible.
- ▻ **RQ_EWM_01** *Executability*: An EWM is executable per definition. Thus, the used workflow language must be executable, too, and it must be supported by at least one workflow engine.

Requirements on a workflow engine

- ▻ **RQ_WF-EN_01** *Workflow Execution*: The engine must support the workflow execution based on the workflow language used for EWMs.
- ▻ **RQ_WF-EN_02** *Grid Middleware Support*: The workflow engine must support the invocation of services deployed in the Grid middlewares Globus Toolkit 4 and UNICORE 6. Therefore, respective security mechanisms must be supported.

Chapter 8

Scientific Workflow Model Representation with MoDFlow.BPMN

This chapter defines MoDFlow.BPMN, which is a BPMN metamodel subset with custom extension for IWMs at the intermediate layer (see Chapter 6). One central aspect of MoDFlow.BPMN is the representation of workflow activities and data dependencies between them, for which several custom metamodel extensions are defined based on the extension mechanism in the BPMN standard. All custom extensions are specified by a corresponding metamodel definition. MoDFlow.BPMN fulfills all requirements for an IWM (RQ_IWM_*) that are defined in Chapter 7.

General considerations for defining MoDFlow.BPMN are presented in Section 8.1. The basic representation of workflow activities is discussed in Section 8.2 and in Section 8.3 describes the BPMN metamodel subset for MoDFlow.BPMN. The metamodel that represents our custom extensions is described in Section 8.4.

8.1 Basic Design Considerations

The BPMN metamodel is linguistically described in the specification document and formalized with the two CMOF metamodels `BPMN20.cmof` and `BPMNDI.cmof`. The first metamodel contains information about properties and associations for all BPMN elements. The second metamodel contains layout information for BPMN elements that are arranged within a diagram.

As an IWM has no defined graphical representation, the BPMN metamodel subset of MoDFlow.BPMN is based on the `BPMN20.cmof` metamodel. The graphical notation of BPMN is used for illustration purposes only. All custom metamodel extensions are based on the standard extension mechanism for the BPMN metamodel. They are defined in a separate metamodel.

MoDFlow.BPMN generally considers all requirements for an IWM (RQ_-IWM_*) that are defined in Chapter 7. Furthermore, the definition of the BPMN metamodel subset focuses on compactness, which means that we aim to use as few standard BPMN elements as possible to represent common scientific workflow aspects. This reduces the complexity of an IWM and simplifies the creation of a DWM2IWM mapping. For example, all information for a workflow activity that is required to invoke a Web service is encapsulated within a single BPMN `ServiceTask` element. The interaction with a Web service in BPMN is usually represented by an additional BPMN collaboration with BPMN participants, BPMN conversations, BPMN interfaces, and so on. All required information that cannot be expressed with the existing features of a `ServiceTask` element is added via a corresponding metamodel extension.

Generally, a custom extension is defined due to one or multiple of the following reasons:

- To reduce the complexity of MoDFlow.BPMN. If certain information can be expressed with standard BPMN elements, but these elements result in a complex construct, the relevant information is covered by a compact metamodel extension. One example is the technical information about a Web service interaction mentioned above. The objective is to avoid the distribution of required information across many elements and complex structures. However, it must be ensured that the available information is sufficient to create executable workflow code.

- To express information, which cannot be expressed with standard BPMN elements, e.g., to define parameter sweeps.

- To provide additional information required for an automatic mapping to executable workflow code. For example, to create data flow elements such as variables and assignment operations based on the definition of

input and output parameters for workflow activities and data dependencies between them.

The definition of a workflow activity is a central aspect for scientific workflows. As we focus on SOAs as execution infrastructures for scientific workflows, the execution of a workflow activity is technically initiated via one or multiple Web service calls. We provide a basic representation of workflow activities in the context of SOAs, which is further described in Section 8.2.

Some executable business workflow languages such as BPEL require a WSDL definition that represents the service interface for a BPEL process. MoDFlow.BPMN provides a WSDL definition that can be used as basic service interface for defining the interaction of a workflow client and a scientific workflow instance. The complete WSDL definition can be found in Appendix A It provides the following methods to start, monitor and end a scientific workflow execution:

- `String workflowid , String message startWorkflow(String workflowid)`: This method starts a scientific workflow execution and creates a corresponding scientific workflow instance. The parameter `workflowid` is a unique identifier for the corresponding workflow instance[1]. The method returns the `workflowid` and a `message` that contains additional information about the workflow instance creation.

- `String state fetchWorkflowState(String workflowid)`: This method can be invoked arbitrarily to fetch the current state of a scientific workflow execution. The returned `state` is currently either 'Running', 'Faulted' or 'Done'.

- `String message endWorkflow(String workflowid)`: Aborts the scientific workflow execution and destroys the workflow instance. It returns a `message` as acknowledgment for the client and may contain information about the workflow instance destruction.

[1] As the creation of unique IDs within BPEL is not supported, a corresponding ID must be currently provided by the client

Finally, the requirements for an IWM (RQ_IWM_*) are recognized by MoDFlow.BPMN as follows:

- **RQ_IWM_01** *BPMN 2.0 Compliance*: The BPMN subset defined by MoDFlow.BPMN uses only standard BPMN elements. Extensions are based on the standard metamodel extension mechanism of BPMN. As an IWM is to be transformed to an EWM and we selected BPEL as executable workflow language, the BPMN subset is further limited to those elements for which a BPEL mapping has already been specified by the BPMN standard or can be realized. The BPMN process of an IWM must fulfill the constraints defined in the BPEL mapping, e.g., it must not contain *deadlocks* and *lacks of synchronization*. Furthermore, only those BPMN elements with execution semantics defined by the BPMN standard are allowed in order to enable the utilization of executable BPMN for scientific workflow execution at a later stage.
- **RQ_IWM_02** *Control Flow Elements*: The required control flow constructs are supported by the corresponding BPMN gateways, loops, and sequence flows.
- **RQ_IWM_03** *Workflow Activities*: A workflow activity with input and output parameters is represented by a BPMN *Task* or *Service Task*, for which we defined appropriate custom metamodel extensions. We also defined a basic representation of workflow activities in the context of SOAs (see Section 8.2).
- **RQ_IWM_04** *Data Dependencies*: We support data dependencies with a custom metamodel extension that provides referencing of an output parameter of workflow activities or of a defined sweep parameter as input parameter.
- **RQ_IWM_05** *Parameter Sweeps*: A parameter sweep is represented by a BPMN *Parallel Multiple Instance Loop* in combination with a custom metamodel extension.
- **RQ_IWM_06** *Extensibility*: The metamodel for our custom metamodel extensions provides the definition of individual configuration parameters for several extension elements.

8.2 Representation of Workflow Activities

A scientific workflow generally consists of workflow activities and data dependencies between them (see Chapter 3). Each workflow activity can be regarded as computational step that consumes and produces data. Thereby, only metadata information about the data and not the data itself is passed between two consecutive workflow activities within a scientific workflow instance. Required data transfer activities can either be added during the mapping to an EWM or they are executed by the workflow engine on demand.

As we focus on the execution of scientific workflows in SOA environments, the execution of a workflow activity is based on the invocation of one or more Web services. Thus, the execution of a scientific workflow is technically a service orchestration that coordinates the exchange of SOAP messages. These SOAP messages also contain the metadata information about data that have to be processed and that have been created.

As workflow activities are central for scientific workflow, we created a basic schema for their representation, see Figure 8.1. It focuses on workflow activities that represent one Web service invocation. Such a workflow activity can define multiple input parameters that are copied to the request message and multiple output parameters that are fetched from the response message. Each output parameter can be used by other workflow activities as input parameter via a corresponding data dependency.

This basic schema is applied in MoDFlow.BPMN and is supported by MoDFlow.BPMN2BPEL so that a basic service orchestration can be fully defined in an IWM. It is further extended by MoDFlow.BPMN2BPEL within expansions to support workflow activities that require multiple service invocations, see Chapter 9.

8.3 BPMN Metamodel Subset

The description of the BPMN metamodel subset of MoDFlow.BPMN is analogous to the linguistically described metamodel in the specification document of the BPMN standard. We describe all used metamodel classes

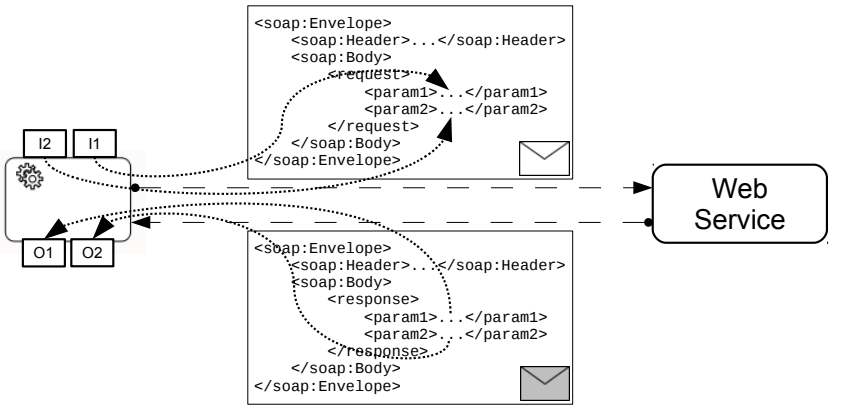

Figure 8.1. Basic schema for workflow activities[a]

[a]Picture credits: `http://bpmb.de/poster` (BPMN service task image, message icons)

and their used attributes and model associations. If necessary, we give an additional explanation for our decision. We further implicitly assume that all (metamodel) constraints described in the BPMN standard are valid for our subset unless we explicitly modify or extend them.

Figure 8.2 shows a graphical representation of the selected BPMN elements[2]. The BPMN *Task* element and its combination with the task type *Service Task* are used to represent workflow activities. The BPMN events *Start Event* and *End Event* define the start and the end of a workflow or sub-process, which is required for valid BPMN process models. A BPMN process can further be structured with the BPMN *Sub-Process* element. BPMN activity markers, gateways, and sequence flows are used to define control flow constructs for the sequential, parallel, conditional, and looped execution of workflow activities.

The metamodel for the BPMN subset is shown in Figure 8.3. Note that each graphical BPMN element presented in Figure 8.2 has a corresponding class in the metamodel.

[2]Note that the graphical representation is only used for illustration purposes.

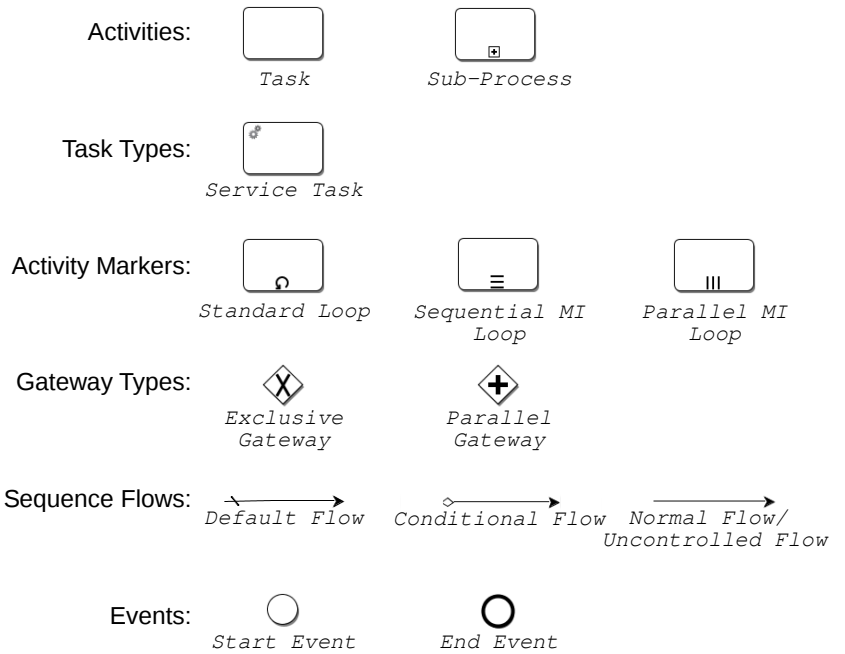

Figure 8.2. BPMN metamodel subset for MoDFlow.BPMN[a]]

[a]BPMN element images created with Yaoqiang BPMN Editor (`http://sourceforge.net/projects/bpmn`)

Used Basic BPMN Metamodel Classes and Attributes:
Some classes in the BPMN metamodel are basic classes such as abstract classes or containers for other metamodel classes. This includes all metamodel classes that represent the metamodel extension mechanism of BPMN. These basic classes have no defined graphical representation.

▻ `Documentation`: The class `Documentation` represents human-readable documentation information. The used attributes and model associations of `Documentation` are shown in Table 8.2.

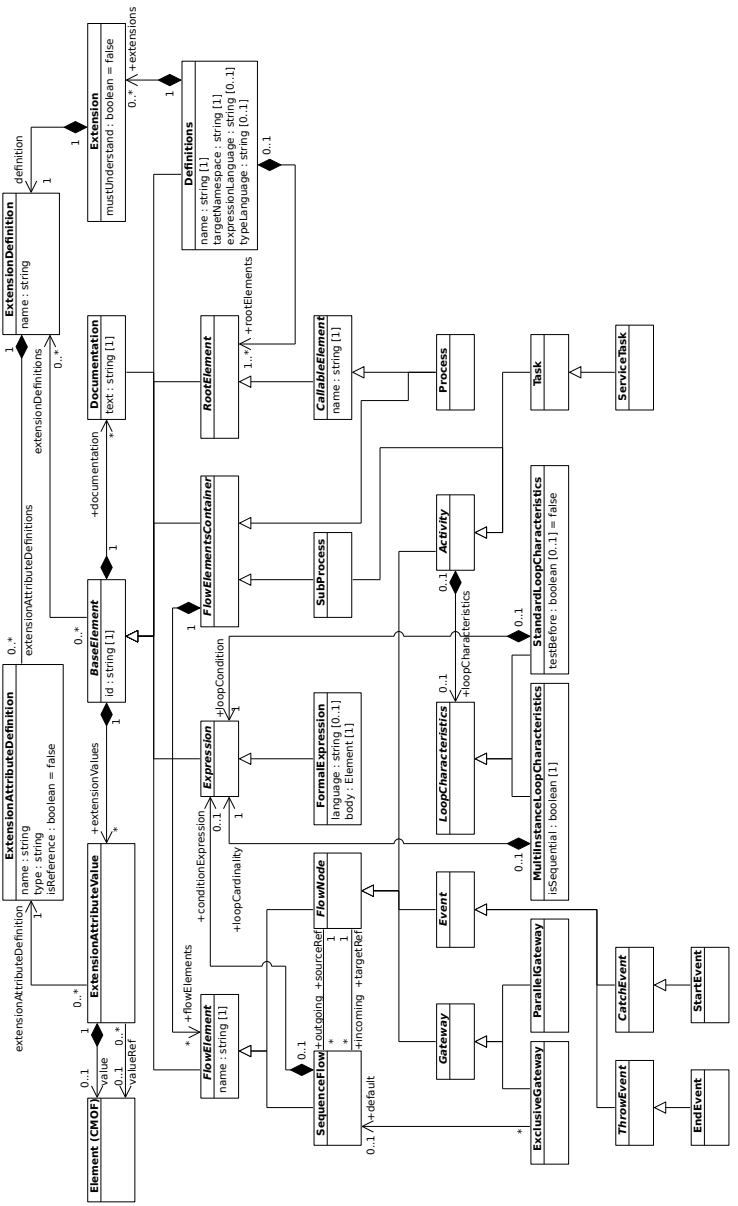

Figure 8.3. Metamodel for BPMN subset

Table 8.1. Used attributes and model associations of `Documentation`

Attribute Name	Description/Usage
text : string [1]	Human-readable description in plain text.

Table 8.2. Used attributes and model associations of `BaseElement`

Attribute Name	Description/Usage
id : string [1]	A unique identifier.
documentation : Documentation [0..*]	A list of additional documentation elements based on the class `Documentation`.
extensionValues : ExtensionAttributeValue [0..*]	A list of extension elements based on the class `ExtensionAttributeValue` of the metamodel extension mechanism (see below).

- `BaseElement`: `BaseElement` is the abstract super class for all BPMN classes that need a unique identifier and that must support extension elements. The used attributes and model associations of `BaseElement` are shown in Table 8.2.

- `RootElement`: The abstract class `RootElement` is the super class for all BPMN classes that represent central BPMN artifacts such as BPMN processes. It is derived from the abstract class `BaseElement` and has no own attributes and model associations.

- `Definitions`: The class `Definitions` represents a BPMN definition, which is the top-level container for all BPMN artifacts such as BPMN processes. Each contained element must derive the abstract class `RootElement`. The used attributes and model associations of `Definitions` are shown in Table 8.3.

- `Extension`: The class `Extension` describes that a certain metamodel extension is supported and can be utilized within a BPMN model. The used attributes and model associations of `Extension` are shown in Table 8.4.

- `ExtensionDefinition`: The class `ExtensionDefinition` defines the structure of a metamodel extension. The used attributes and model associations of `ExtensionDefinition` are shown in Table 8.5.

Table 8.3. Used attributes and model associations of `Definitions`

Attribute Name	Description/Usage
name : string [1]	A label for the BPMN definition.
targetNamespace : string [1]	A target namespace for the BPMN definition.
expressionLanguage : string [0..1]	A URI identifier for the used expression language within the BPMN definition. In contrast to the BPMN standard, which allows using arbitrary expression languages, we only allow the use of XPATH. The default value is `http://www.w3.org/1999/XPath` for XPATH version 1.0.
typeLanguage : string [0..1]	A URI identifier for the used type language. In contrast to the BPMN standard, which allows using arbitrary type languages, we only allow XML Schema. Thus, the default and only allowed value is `http://www.w3.org/2001/XMLSchema`.
rootElements : RootElement [1..*]	A list of contained root elements. In contrast to the BPMN standard, which makes no further restrictions, the list must only contain one BPMN process based on the class `Process` (see below). This BPMN process represents the scientific workflow.
extensions : Extension [0..*]	A list of supported extensions within a BPMN definition. A supported extension is represented by the class `Extension` (see below).

Table 8.4. Used attributes and model associations of `Extension`

Attribute Name	Description/Usage
mustUnderstand : Boolean [0..1] = false	If `true`, the metamodel extension must be supported by any BPMN tool, otherwise it may be ignored. The default value is `false`. Our custom metamodel extensions (see Section 8.4) are marked as `false`.
definition : ExtensionDefinition [1]	An `ExtensionDefinition` element (see below) that further specifies the metamodel extension.

Table 8.5. Used attributes and model associations of `ExtensionDefinition`

Attribute Name	Description/Usage
name : String [1]	A name for the metamodel extension.
extensionAttribute-Definitions : ExtensionAttributeDefinition [0..*]	A list of extension elements that specifies extension values. Each extension element is represented by the class `ExtensionAttributeDefinition` (see below).

Table 8.6. Used attributes and model associations of `ExtensionAttributeDefinition`

Attribute Name	Description/Usage
name : String [1]	A name for the metamodel extension element.
type : String [1]	A type of the metamodel extension element.
isReference : Boolean [0..1] = false	Defines if a concrete metamodel extension element is contained (`false`) or referenced (`true`). Contained means that the element is regarded and serialized as part of the associated BPMN element. Referenced means that the element is regarded as external element of the associated BPMN element and it can be serialized at different locations in the same BPMN model or in any external model. The default value is `false`. All custom metamodel extension values are defined as contained.

Table 8.7. Used attributes and model associations of `ExtensionAttributeValue`

Attribute Name	Description/Usage
value : Element [0..1]	A contained element as metamodel extension value. It is represented by an arbitrary subclass of the CMOF class `Element`[a].
valueRef : Element [0..1]	A referenced element as metamodel extension value. It is represented by an arbitrary subclass of the CMOF class `Element`[a].
extensionAttribute-Definition : ExtensionAttributeDefinition [1]	An `ExtensionAttributeDefinition` for which this class represents corresponding extension values.

[a]Note that the Ecore metamodel class `EFeatureMapEntry` is used in the BPMN Ecore model instead of the corresponding Ecore metamodel class `EObject`.

▷ `ExtensionAttributeDefinition`: The class `ExtensionAttributeDefinition` represents a value container for a concrete metamodel extension element. The used attributes and model associations of `ExtensionAttribute-Definition` are shown in Table 8.6.

▷ `ExtensionAttributeValue`: The class `ExtensionAttributeValue` represents a concrete metamodel extension element. We use it to encapsulate values for our custom metamodel extensions, as detailed in Section 8.4. The used attributes and model associations of `ExtensionAttributeValue` are shown in Table 8.7.

Table 8.8. Used attributes and model associations of `FormalExpression`

Attribute Name	Description/Usage
language : string [0..1]	A URI identifier for the used expression language. Per default, the used expression language is derived from the attribute `expressionLanguage` of the class `Definitions`. In contrast to the BPMN standard, which allows any expression language, we only allow the use of XPATH.
body : Element [1]	An expression based on the specified expression language. It is represented by an arbitrary subclass of the CMOF class `Element`[a]. As the allowed expression languages are restricted to XPATH, the attribute value must always be an XPATH expression.

[a]Note that the Ecore metamodel class `EString` is used in the BPMN Ecore model instead of the corresponding Ecore metamodel class `EObject`.

Table 8.9. Used attributes and model associations of `FlowElement`

Attribute Name	Description/Usage
name : string [1]	A name of the element. In contrast to the BPMN standard, which defines no further restrictions, we require that the attribute value is unique. We use the attribute value to unambiguously identify workflow activities, which are represented as process flow elements. This is relevant for the BPMN-to-BPEL mapping (IWM2EWM mapping) described in Chapter 9. The unique attribute `id` derived from the class `BaseElement` is only used as internal technical identifier.

▷ `Expression`: The abstract class `Expression` is the super class for all expressions. Expressions are used, for example, as conditions in loops. The class `Expression` has no own attributes and model associations.

▷ `FormalExpression`: The class `FormalExpression` is derived from the abstract class `Expression`. It is the only class in the BPMN standard that represents a concrete expression. We use it to define XPATH expressions. The used attributes and model associations of `FormalExpression` are shown in Table 8.8.

▷ `FlowElement`: `FlowElement` is the abstract super class for elements that can be used within a BPMN process such as BPMN activities, gateways, events, and sequence flows. The used attributes and model associations of `FlowElement` are shown in Table 8.9.

Table 8.10. Used attributes and model associations of `FlowElementsContainer`

Attribute Name	Description/Usage
flowElements : FlowElement [0..*]	A list of contained process flow elements. Each process flow element is derived from the abstract class `FlowElement` (see below).

Table 8.11. Used attributes and model associations of `FlowNode`

Attribute Name	Description/Usage
incoming : SequenceFlow [0..*]	A list of incoming sequence flows. Each sequence flow is represented by the class `SequenceFlow` (see below).
outgoing : SequenceFlow [0..*]	A list of outgoing sequence flows. Each sequence flow is represented by the class `SequenceFlow` (see below).

- `FlowElementsContainer`: The abstract class `FlowElementsContainer` serves as a super class for all BPMN metamodel classes such as `Process` and `SubProcess` (see below) that may contain BPMN process elements. The used attributes and model associations of `FlowElementsContainer` are shown in Table 8.10.

- `FlowNode`: The abstract class `FlowNode` is the super class for all process elements such as BPMN activities, events, and gateways that can be connected with sequence flows. It is derived from the abstract class `FlowElement`. The used attributes and model associations of `FlowNode` are shown in Table 8.11.

- `Activity`: The abstract class `Activity` is the super class for all BPMN activities. It is derived from the abstract class `FlowNode`. The used attributes and model associations of `Activity` are shown in Table 8.12.

- `LoopCharacteristics`: The abstract class `LoopCharacteristics` is used as super class for loop definitions. It has no own attributes or model

Table 8.12. Used attributes and model associations of `Activity`

Attribute Name	Description/Usage
loopCharacteristics : LoopCharacteristics [0..1]	A loop definition for the activity. It is derived from the abstract class `LoopCharacteristics` (see below).

Table 8.13. Used attributes and model associations of `SequenceFlow`

Attribute Name	Description/Usage
sourceRef : FlowNode [1]	The source process flow element.
targetRef : FlowNode [1]	The target process flow element.
conditionExpression : Expression [0..1]	A `boolean` expression that is used as condition. The corresponding expression element must derive the abstract class `Expression`. In our case, we use the class `FormalExpression` to define XPATH expressions.

associations. The used loop types are described below.

- `Event`: The abstract class `Event` is the super class for all BPMN events. It is derived from the abstract class `FlowNode`. No attributes or model associations of `Event` are used.

- `CatchEvent`: The abstract class `CatchEvent` is the super class for all catching events in BPMN. It is derived from the abstract class `Event`. No attributes and model associations of `CatchEvent` are used.

- `ThrowEvent`: The abstract class `ThrowEvent` is the super class for all throwing events in BPMN. It is derived from the abstract class `Event`. No attributes and model associations of `ThrowEvent` are used.

- `Gateway`: The abstract class `Gateway` is the super class for all BPMN gateways. It is derived from the abstract class `FlowNode`. No attributes and model associations of `Gateway` are used.

- `SequenceFlow`: The class `SequenceFlow` represents all sequence flows in BPMN. Each sequence flow connects two process flow elements, which derive the abstract class `FlowNode`. The used sequence flow types are described below. The used attributes and model associations are shown in Table 8.13.

- `CallableElement`: The abstract class `CallableElement` is the super class for all BPMN metamodel classes such as `Process` (see below) that can be invoked externally. The used attributes and model associations of `CallableElement` are shown in Table 8.14.

Table 8.14. Used attributes and model associations of `CallableElement`

Attribute Name	Description/Usage
name : string [1]	A name for the element. In contrast to the BPMN standard, which defines the attribute as optional, we require a mandatory value. This ensures that a BPMN process represented by the class `Process` (see below) has always a name, which is relevant for the BPMN-to-BPEL mapping (IWM2EWM mapping) described in Chapter 9.

▻ `Process`: The class `Process` represents a BPMN process. It is derived from the abstract classes `CallableElement` and `FlowElementsContainer`. In contrast to the BPMN standard, which allows to define multiple start and end events in the process flow, we require that a BPMN process must have exactly one start and end event. This is relevant for the BPMN-to-BPEL mapping (IWM2EWM mapping) described in Chapter 9. No attributes and model associations of `Process` are used.

Used BPMN Activities:
The used BPMN activities are the *Task* and the *Sub-Process*. We further use the task type *Service Task* and the loop markers *Standard Loop* and *Multiple Instance Loop*.

▻ *Task*: The BPMN activity *Task* represents a local workflow activity that is executed by the workflow engine itself, for example, to modify or create input parameter for other workflow activities based on XSL Transformations. The corresponding class `Task` is derived from the abstract class `Activity`. No attributes and model associations of `Task` are used.

▻ *Service Task*: The task type *Service Task* represents an external workflow activity that has to be executed by one or more external Web service invocations. The corresponding class `ServiceTask` is derived from the class `Task`. No attributes and model associations of `ServiceTask` are supported.

▻ *Sub-Process*: The BPMN activity *Sub-Process* is used to structure a workflow. The corresponding class `SubProcess` is derived from the abstract classes `Activity` and `FlowElementsContainer`. No attributes and model

Table 8.15. Used attributes and model associations of `StandardLoopCharacteristics`

Attribute Name	Description/Usage
testBefore : boolean [0..1] = false	If `true`, the attribute `loopCondition` is checked before each loop execution (*while-do* loop), otherwise after it (*do-while*). The default value is `false`.
loopCondition : Expression [1]	A loop condition as `boolean` XPATH expression, which must be defined as `FormalExpression`. The loop execution continues until the expression evaluates to `true`. In contrast to the BPMN standard, which allows to omit a loop condition and to underspecify the loop behavior, the attribute must be defined.

associations of `SubProcess` are used. In contrast to the BPMN standard, which allows to define multiple start and end events in the process flow, we require that a *Sub-Process* must have exactly one start and end event. This is relevant for the BPMN-to-BPEL mapping (IWM2EWM mapping) described in Chapter 9.

- *Standard Loop*: The loop marker *`Standard Loop`* represents common loop types such as *while-do* and *do-while*. It is defined with the class `StandardLoopCharacteristics`, which derives the abstract class `LoopCharacteristics`. The used attributes and model associations of `StandardLoopCharacteristics` are shown in Table 8.15.

- *Multiple Instance Loop*: The loop marker *Multiple Instance Loop* can either represent a *Sequential Multiple Instance Loop* or a *Parallel Multiple Instance Loop*, in which each loop execution has its own activity instance. These activity instances are executed sequentially in a *Sequential Multiple Instance Loop*, while their execution order in a *`Parallel Multiple Instance`* is concurrent. Both loop types are used to define common loops. We additionally use the *Parallel Multiple Instance Loop* in combination with a custom metamodel extension to define parameter sweeps, see Section 8.4. A *Multiple Instance Loop* is defined by the class `MultiInstanceLoopCharacteristics`. The used attributes and model associations of `MultiInstanceLoopCharacteristics` are shown in Table 8.16.

Used BPMN Gateways:
The used BPMN gateways are the `Exclusive Gateway` and the `Parallel`

Table 8.16. Used attributes and model associations of `MultiInstanceLoopCharacteristics`

Attribute Name	Description/Usage
isSequential : boolean [1]	The value is `true` for a *Sequential Multiple Instance Loop* and `false` for a *Parallel Multiple Instance Loop*. The default value is `false`.
loopCardinality : Expression [1]	An `integer` XPATH expression to determine the number of loop activity instances, which must be defined as `FormalExpression`. In contrast to the BPMN standard, which allows to omit a loop cardinality and to underspecify the loop behavior, the attribute must be defined.

Table 8.17. Used attributes and model associations of `ExclusiveGateway`

Attribute Name	Description/Usage
default : SequenceFlow [0..1]	A default outgoing *Sequence Flow*, which is used if no conditional *Sequence Flow* from the derived attribute `outgoing` (class `FlowNode`) evaluates to `true`.

`Gateway`.

- *Exclusive Gateway*: The *Exclusive Gateway* is used to define *if-then-elseif-else* control flow constructs that start with a conditional split gateway and end with a corresponding join gateway. It is defined by the class `ExclusiveGateway` that is derived from the abstract class `Gateway`. The used attributes and model associations of `ExclusiveGateway` are shown in Table 8.17.

- *Parallel Gateway* The *Parallel Gateway* is used to define concurrent control flow constructs that start with a parallel split gateway and end with a corresponding join gateway. It is defined by the class `ParallelGateway` that is derived from the abstract class `Gateway`. No attributes and model associations of `ParallelGateway` are used.

Used BPMN Sequence Flows:
The used *Sequence Flows* are the *Normal Flow*, the *Uncontrolled Flow*, the *Conditional Flow*, and the *Default Flow*. All sequence flow types are represented by the same class `SequenceFlow` (see above). The type of a *Sequence*

Flows depends on the BPMN elements it connects and whether the attribute *conditionExpression* of the class `SequenceFlow` is defined or not.

- ▻ *Normal Flow/Uncontrolled Flow*: A *Normal Flow/Uncontrolled Flow* represents a common control flow dependency between BPMN process flow elements such as activities, gateways, and events. The attribute `conditionExpression` of the class `SequenceFlow` must be undefined.
- ▻ *Conditional Flow*: A `Conditional Flow` defines an XPATH expression as `FormalExpression` via the attribute `conditionExpression` of `SequenceFlow`. In contrast to the BPMN standard, which allows *Conditional Flows* at different process flow elements, we only allow *Conditional Flows* for splitting *Exclusive Gateways*.
- ▻ *Default Flow*: A `Default Flow` is used as default outgoing *Sequence Flow* for a splitting *Exclusive Gateway*. The attribute `conditionExpression` must be undefined.

Used BPMN Events:
The used BPMN events are the *blank Start Event* and *End Event*. These events are required to create valid BPMN processes or sub-processes. A corresponding workflow editor should add these elements automatically.

- ▻ *Start Event*: A *Start Event* is defined by the class `StartEvent` and represents the start of a BPMN process or sub-process. The class `StartEvent` is derived from the abstract class `CatchEvent`. No attributes and model associations of `StartEvent` are used.
- ▻ *End Event*: An *End Event* is defined by the class `EndEvent` and represents the end of a BPMN process or sub-process. The class `EndEvent` is derived from the abstract class `ThrowEvent`. No attributes and model associations of `EndEvent` are used.

8.4 BPMN Metamodel Extensions

All custom extensions for the BPMN metamodel subset described in Section 8.3 are defined within one metamodel that is shown in Figure 8.4. Each

metamodel extension is considered as design time only [Kopp et al. 2011], which means corresponding extension elements have to be replaced by standard workflow language elements within the BPMN-to-BPEL mapping defined by MoDFlow.BPMN2BPEL (see Chapter 9).

The relevant classes to define a standard BPMN metamodel extension are described in Section 8.3. We always set the attribute `mustUnderstand` of the class `Extension` to `false` so that our metamodel extension can be ignored by existing BPMNN tools. Values for extensions elements are represented by the class `ExtensionAttributeValue` that can be attached to extendable BPMN elements.

An overview about possible combinations of BPMN elements and classes of the BPMN subset and the defined custom metamodel extensions is given in Table 8.18.

In the following, we present all custom metamodel extension and describe their purpose.

ProcessConfiguration:
A `ProcessConfiguration` contains configuration options for a service interface of a BPMN process and can be attached to the class `Process`. It is mandatory if a Web service interface is used for the workflow invocation by a client. The exchanged of messages via this interface can be specified by BPMN events (see `EventConfiguration`). The provided attributes and model associations of `ProcessConfiguration` are shown in Table 8.19.

ActivityConfiguration:
An `ActivityConfiguration` contains configuration options for workflow activities and is mandatory for the classes `Task` and `ServiceTask`. The provided attributes and model associations of `ActivityConfiguration` are shown in Table 8.20.

ServiceTaskConfiguration:
A `ServiceTaskConfiguration` contains configuration options for the invocation of a Web service and is attached to the class `ServiceTask`. It is mandatory if the `activityType` (`ActivityConfiguration`) for the correspond-

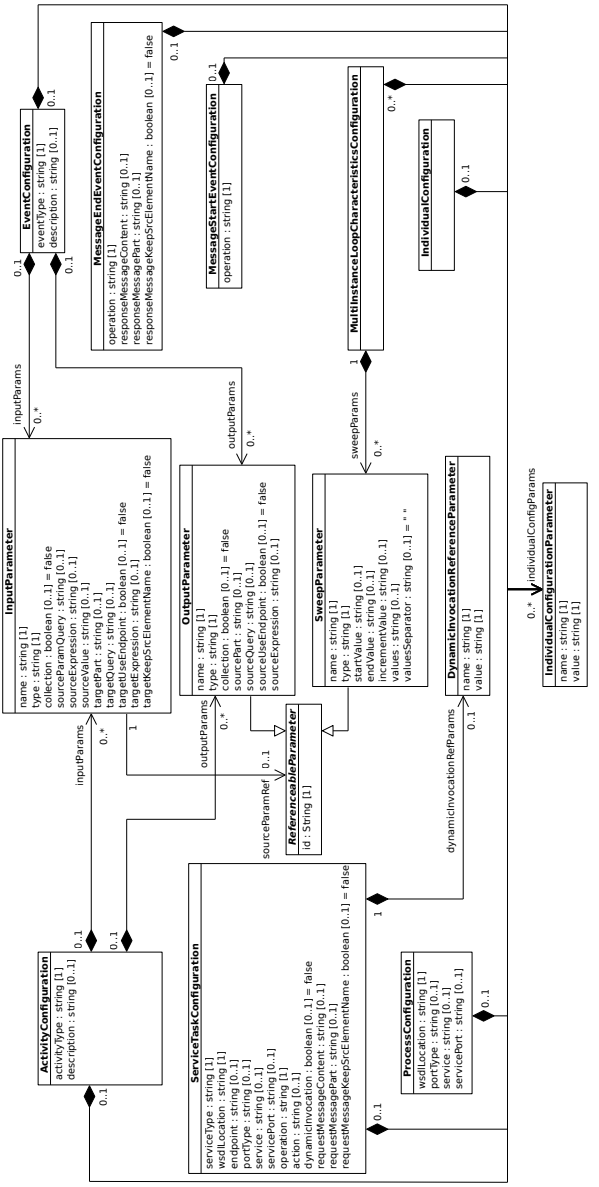

Figure 8.4. Metamodel for custom BPMN metamodel extensions

Table 8.18. Metamodel extensions for BPMN subset

	Process	Task[a]	SubProcess[a]	ServiceTask	*Gateways*	*Sequence Flows*	StartEvent	EndEvent
Process-Configuration	+							
Activity-Configuration		+	+	+				
ServiceTask-Configuration				+				
Event-Configuration							+	+
MessageStartEvent-Configuration							+	
MessageEndEvent-Configuration								+
MultiInstance-Loop-Characteristics-Configuration	-	+[b]	+[b]	+[b]	-	-	-	-
Individual-Parameter-Configuration	+	+	+	+	+	+	+	+

[a] Including with attached activity markers *Standard Loop* and *Parallel Multiple Instance Loop*
[b] Added to required activity marker *Parallel Multiple Instance Loop*

Table 8.19. Attributes and model associations of `ProcessConfiguration`

Attribute Name	Description/Usage
wsdlLocation : string [1]	An URI to a local WSDL definition file that defines the used Web service interface for the workflow.
portType : string [0..1]	The qualified name of the used WSDL port type in the format "{<namespace>}<name>". Per default, the first defined WSDL port type in the WSDL definition is used.
service : string [0..1]	The qualified name of the selected WSDL service in the format "{<namespace>}<name>". Per default, the first defined WSDL service is used that contains the selected `servicePort`.
servicePort : string [0..1]	The name of the selected WSDL port. Per default, the first WSDL port is used that references a WSDL binding for `portType`.
individualConfigParams : IndividualConfigurationParameter [0..*]	A list of additional and individual configuration parameters that are represented by the class `IndividualConfigurationParameter`.

Table 8.20. Attributes and model associations of `ActivityConfiguration`

Attribute Name	Description/Usage
activityType : string [1]	A unique string literal for the type of the workflow activity, whereby the following types are predefined: ▷ "tf.activity.xslt": Represents the invocation of a XSL transformation in a `Task`. ▷ "tf.activity.xslt": Represents the invocation of a Web service in a `ServiceTask`.
description : string [0..1]	A human-readable description of the workflow activity.
inputParams : InputParameter [0..*]	A list of input parameters whose utilization depend on the `activityType`. In case of "tf.activity.webservice", all input parameters are used to create the request message for a Web service invocation.
outputParams : OutputParameter [0..*]	A list of output parameters whose utilization depends on the `activityType`. In case of "tf.activity.webservice", all output parameters are fetched from the response message of a Web service invocation.
individualConfigParams : IndividualConfigurationParameter [0..*]	A list of additional and individual configuration parameters that are represented by the class `IndividualConfigurationParameter`.

Table 8.21. Attributes and model associations of `ServiceTaskConfiguration`

Attribute Name	Description/Usage
serviceType : string [1]	A unique string literal for the Web service type.
wsdlLocation : string [1]	An URI to a local WSDL definition file that defines the interface for the used Web service.
endpoint : string [0..1]	An URL for the Web service endpoint. If this attribute is not set, the endpoint URL is derived from `wsdlLocation`.
portType : string [0..1]	The qualified name of the used WSDL port type in the format "{<namespace>}<name>". Per default, the first defined WSDL port type in the WSDL definition is used.
service : string [0..1]	The qualified name of the selected WSDL service in the format "{<namespace}<name>". Per default, the first defined WSDL service is used that contains the selected `servicePort`.
servicePort : string [0..1]	The name of the selected WSDL port. Per default, the first WSDL port is used that references a WSDL binding for `portType`.
operation : string [1]	The name of the used WSDL operation. It must be contained in the selected `portType`.
action : string [0..1]	A SOAP action for the Web service invocation. Per default, the SOAP action is derived from the corresponding WSDL binding operation for the selected `operation`.
dynamicInvocation : boolean [0..1] = false	If set to `true`, the Web service invocation is regarded as dynamic. The default value is `false` that represents a static Web service invocation.
dynamicInvocation-RefParams : DynamicInvocation-ReferenceParameter [0..*]	A list of reference parameters for the WS-Addressing endpoint of a dynamic Web service invocation. It is only recognized if the attribute `dynamicInvocation` is `true`.
requestMessageContent : string [0..1]	A plain string or XML literal that initializes or completely represents the request message (SOAP body) for the Web service invocation.
requestMessagePart : string [0..1]	The name of the WSDL message part that is used for request message initialization. Per default, the first WSDL message part of the corresponding WSDL message type of the `operation` is used.
requestMessageKeepSrc-ElementName : boolean [0..1] = false	If set to `true`, the complete root XML element of an XML literal-based attribute `requestMessageContent` is used for request message initialization. The default value is `false`, which means only the content of the root XML element is used.
individualConfigParams : IndividualConfiguration-Parameter [0..*]	A list of additional and individual configuration parameters that are represented by the class `IndividualConfigurationParameter`.

ing element is "tf.activity.webservice". The provided attributes and model associations of `ServiceTaskConfiguration` are shown in Table 8.21.

Table 8.22. Attributes and model associations of `EventConfiguration`

Attribute Name	Description/Usage
eventType : string [1]	A unique string literal for the type of the event. The type "tf.event.message" is predefined and represents a message event that sends or receives SOAP messages.
description : string [0..1]	A human-readable description of the event.
inputParams : InputParameter [0..*]	A list of input parameters whose utilization depends on the `eventType`. In case of "tf.event.message" for an `EndEvent`, all input parameters are used to create the output message.
outputParams : OutputParameter [0..*]	A list of output parameters whose utilization depends on the `eventType`. In case of "tf.event.message" for an `StartEvent`, all output parameters are fetched from the input message

Table 8.23. Attributes and model associations of `MessageStartEventConfiguration`

Attribute Name	Description/Usage
operation : string [0..1]	The name of the used WSDL operation of the workflow interface (see `ProcessConfiguration`) to receive request messages.

EventConfiguration:
An `EventConfiguration` contains common configuration options for BPMN events and is mandatory for the classes `StartEvent` and `EndEvent`. The provided attributes and model associations of `EventConfiguration` are shown in Table 8.22.

MessageStartEventConfiguration:
A `MessageStartEventConfiguration` contains special configuration options for a `StartEvent` to receive (catch) messages. It is mandatory if the attribute `eventType` (`EventConfiguration`) for the corresponding element is "tf.event.message". The provided attributes and model associations of `MessageStartEventConfiguration` are shown in Table 8.23.

MessageEndEventConfiguration:
A `MessageEndEventConfiguration` contains special configuration options for an `EndEvent` to send (throw) messages. It is mandatory if the `eventType` (`EventConfiguration`) for the corresponding element is "tf.event.message". The provided attributes and model associations of `MessageEndEventConfigu-`

Table 8.24. Attributes and model associations of `MessageEndEventConfiguration`

Attribute Name	Description/Usage
operation : string [0..1]	The name of the used WSDL operation of the workflow interface (see `ProcessConfiguration`) to send response messages.
responseMessageContent : string [0..1]	A plain string or XML literal that initializes or completely represents the response message (SOAP body) for the defined operation.
responseMessagePart : string [0..1]	The name of the WSDL message part used for response message initialization. Per default, the first WSDL message part of the corresponding WSDL message type of the `operation` is used.
responseMessageKeepSrcElementName : boolean [0..1] = false	If set to `true`, the complete root XML element of an XML literal-based attribute `responseMessageContent` is used for response message initialization. The default value is `false`, which means only the content of the root XML element is used.
individualConfigParams : IndividualConfigurationParameter [0..*]	A list of additional and individual configuration parameters that are represented by the class `IndividualConfigurationParameter`.

Table 8.25. Attributes and model associations of `MultiInstanceLoopCharacteristicsConfiguration`

Attribute Name	Description/Usage
sweepParams : SweepParameter [0..*]	A list of sweep parameters that represent the range for a parameter sweep.
individualConfigParams : IndividualConfigurationParameter [0..*]	A list of additional and individual configuration parameters that are represented by the class `IndividualConfigurationParameter`.

`ration` are shown in Table 8.24.

MultiInstanceLoopCharacteristicsConfiguration:
A `MultiInstanceLoopCharacteristicsConfiguration` is used to define a parameter sweep for a *Parallel Multiple Instance loop* that is represented by the class `MultiInstanceLoopCharacteristics`. The provided attributes and model associations of `MultiInstanceLoopCharacteristicsConfiguration` are shown in Table 8.25.

IndividualConfiguration:
An `IndividualConfiguration` can contain any number of `IndividualCon-`

Table 8.26. Attributes and model associations of `IndividualConfiguration`

Attribute Name	Description/Usage
name : String [1]	A name for an `IndividualConfiguration`. It must be unique within all `IndividualConfiguration` elements that are defined for a single BPMN element.
individualConfigParams : IndividualConfigurationParameter [0..*]	A list of additional and individual configuration parameters that are represented by the class `IndividualConfigurationParameter`.

Table 8.27. Attributes and model associations of `ReferenceableParameter`

Attribute Name	Description/Usage
id : string [1]	A unique identifier.

`figurationParameter` for all elements of the BPMN subset. It can be added several times to a single BPMN element. The provided attributes and model associations of `IndividualConfiguration` are shown in Table 8.26.

ReferencecableParameter:
A `ReferenceableParameter` is the abstract super class for all parameters such as `OutputParameter` and `SweepParameter` that can be referenced as input parameter. The provided attributes and model associations of `ReferenceableParameter` are shown in Table 8.27.

InputParameter:
An `InputParameter` defines a single input parameter for a workflow activity (`Task` or `ServiceTask`) or a BPMN event (`StartEvent` or `EndEvent`). The provided attributes and model associations of `InputParameter` are shown in Table 8.28.

OutputParameter:
An `OutputParameter` defines a single output parameter for a workflow activity(`Task` or `ServiceTask`) or a BPMN event (`StartEvent` or `EndEvent`). It is derived from the abstract class `ReferenceableParameter`. The provided attributes and model associations of `OutputParameter` are shown in Table 8.29.

Table 8.28. Attributes and model references of `InputParameter`

Attribute Name	Description/Usage
name : string [1]	A name for the input parameter that must be unique within a workflow activity or a BPMN event.
type : string [1]	A qualified name of the XML type in the format "{<namespace>}<name>".
collection : boolean [0..1] = false	If set to `true`, the input parameter is regarded as a collection of `type`. The default value is `false`.
sourceParamRef : ReferencecableParameter [0..1]	A data dependency to a `ReferencecableParameter` that is used as source for the input parameter.
sourceParamQuery : String [0..1]	An XPATH expression to select (query) the input parameter value from `sourceParamRef`.
sourceExpression : string [0..1]	An XPATH expression to determine the input parameter value. It can be alternatively used for `sourceParamQuery` and is only recognized if `sourceParamRef` is undefined. Each output parameter of a workflow activity can be used within via: `$ActivityConfiguration.Name_OuputParameter.Name`.
sourceValue : string [0..1]	A string or XML literal used as input value. It is recognized if `sourceParamRef` and `sourceExpression` are undefined.
targetPart : string [0..1]	The name of the WSDL message part used as based location for an input parameter in an outgoing message. Per default, the first WSDL message part of the corresponding WSDL message type of the `operation` is used. The attribute is only recognized if the workflow activity or BPMN event is configured to send messages (`activityType`="tf.activity.webservice" or `eventType`="tf.event.message").
targetQuery : string [0..1]	An XPATH expression to determine the target location for an input parameter value in an outgoing message. It is recognized if the corresponding workflow activity or event is configured to send messages (`activityType`="tf.activity.webservice" or `eventType`="tf.event.message").
targetUseEndpoint : boolean [0..1] = false	If set to `true`, the input parameter value is copied to the WS-Addressing endpoint used for Web service invocation instead of to the outgoing message It is recognized for a `ServiceTask` that is configured for a Web service invocation (`activityType`="tf.activity.webservice"). If defined, the attribute `targetPart` is ignored. The default value is `false`.
targetExpression : string [0..1]	An XPATH expression to determine the target location for an input parameter value in an outgoing message. It is recognized if the corresponding workflow activity or BPMN event is configured to send messages (`activityType`="tf.activity.webservice" or `eventType`="tf.event.message"). If defined, the attributes `targetPart`, `targetQuery`, and `targetUseEndpoint` are ignored.
targetKeepSrcElementName : string [0..1]	If set to `true`, the complete root XML element of an XML-structured input parameter value is copied to the target location in the outgoing message or WS-Addressing endpoint. The default value is `false`, which means only the content of the root XML element is used. It is only recognized if the corresponding workflow activity or event is configured to send messages (`activityType`="tf.activity.webservice" or `eventType`="tf.event.message").

Table 8.29. Attributes and model references of `OutputParameter`

Attribute Name	Description/Usage
name : string [1]	A name for the output parameter that must be unique within a workflow activity or a BPMN event.
type : string [1]	A qualified name of the XML type in the format "{<namespace>}<name>".
collection : boolean [0..1] = false	If set to `true`, the output parameter is regarded as a collection of `type`. The default value is `false`.
sourcePart : string [0..1]	The name of the WSDL message part used as base location of an output parameter in an ingoing message. Per default, the first WSDL message part of the corresponding WSDL message type is used. The attribute is only recognized if the corresponding workflow activity or event is configured to receive messages (`activityType`="tf.activity.webservice" or `eventType`="tf.event.message").
sourceQuery : string [0..1]	An XPATH expression to determine the source location for an input parameter value in an ingoing message. It is only recognized if the corresponding workflow activity or event is configured to receive messages (`activityType`="tf.activity.webservice" or `eventType`="tf.event.message").
sourceUseEndpoint : boolean [0..1] = false	If the values is `true`, the output parameter value is fetched from the used WS-Addressing endpoint a Web service invocation. The default value is `false`, which means the output parameter value is fetched from an ingoing message. The attribute is only recognized for a `ServiceTask`, which is configured for a Web service invocation (`activityType`="tf.activity.webservice").
sourceExpression : string [0..1]	An XPATH expression to determine the source location for an output parameter value in an ingoing message. It can be alternatively used for `sourceQuery` and is only recognized if the corresponding workflow activity or BPMN event is configured to send messages (`activityType`="tf.activity.webservice" or `eventType`="tf.event.message"). If defined, the attributes `sourcePart`, `sourceQuery`, and `sourceUseEndpoint` are ignored.

SweepParameter:
A `SweepParameter` defines a single sweep parameter for a workflow activity. It is derived from the abstract class `ReferenceableParameter`, but it can only be referenced internally by an `InputParameter` of the corresponding workflow activity. The provided attributes and model associations of `SweepParameter` are shown in Table 8.30.

DynamicInvocationReferenceParameter:
A `DynamicInvocationReferenceParameter` defines a reference parameter for a WS-Addressing endpoint. The provided attributes and model associations

Table 8.30. Attributes and model references of `SweepParameter`

Attribute Name	Description/Usage
name : string [1]	A name for the sweep parameter that must be unique within a workflow activity.
type : string [1]	A qualified name of the XML type in the format "{<namespace>}<name>".
startValue : string [0..1]	A start value for the sweep parameter, which requires that the `type` is either `int` or `float`.
endValue : string [0..1]	An end value for the sweep parameter, which requires that the `type` is either `int` or `float`.
incrementValue : string [0..1]	An increment value for the sweep parameter, which requires that the `type` is either `int` or `float`. The increment value is added to the `startValue` until it is greater than the `endValue`.
values : string [0..1]	A string literal that contains all values for a sweep parameter, which are separated by the `valuesSeparator`. If this attribute is defined, the attributes `startValue`, `endValue`, and `incrementValue` are ignored.
valuesSeparator : string [0..1] = " "	A value separator for `values`. The default value is the space character " ".

Table 8.31. Attributes and model references for `DynamicInvocationReferenceParameter`

Attribute Name	Description/Usage
name : string [1]	A name for the reference parameter element that is also represents the name of the corresponding XML element.
value : string [1]	The content of the reference parameter element as plain string or XML literal.

of `DynamicInvocationReferenceParameter` are shown in Table 8.31.

IndividualConfigurationParameter:
A `IndividualConfigurationParameter` is used to define custom configuration options that cannot be expressed with existing metamodel extension elements. The provided attributes and model associations of `IndividualConfigurationParameter` are shown in Table 8.32.

Table 8.32. Attributes and model references of `IndividualConfigurationParameter`

Attribute Name	Description/Usage
name : String [1]	A name for the individual configuration parameter that must be unique within an `IndividualConfiguration`.
value : String [1]	A value for the individual configuration parameter.

Chapter 9

Scientific Workflow Model Mapping with MoDFlow.BPMN2BPEL

This chapter describes MoDFlow.BPMN2BPEL that represents an IWM2EWM mapping to transform an IWM based on MoDFlow.BPMN (see Chapter 8) to an EWM based on BPEL. BPEL fulfills the respective requirements on an EWM (RQ_EWM) defined in Chapter 7. The corresponding BPMN-to-BPEL model transformation consists of three single transformation steps.

Basic considerations on the design of MoDFlow.BPMN2BPEL are presented in Section 9.1. Additional metamodel extensions that are used within the transformation are defined in Section 9.2. Each transformation step of MoDFlow.BPMN2BPEL is described separately in the Sections 9.3, 9.4, and 9.5.

9.1 Basic Design Considerations

MoDFlow.BPMN2BPEL considers all requirements for an IWM2EWM mapping (RQ_IWM2EWM_*) that are defined in Chapter 7. We assume that WSDL definition files exist for the interface of the workflow and for all invoked external Web services. Due to the complexity of a BPMN-to-BPEL mapping, it is split up into the following three single mapping steps, see Figure 9.1:

- *Step 1: BPMN Process Expansion*: The elements in the BPMN process model of an IWM are expanded, e.g., to refine the process flow for a

workflow activity that invokes several Web service. The objective is to represent as much information as possible with BPMN and to reduce the complexity of the mapping to an executable business workflow language. The *BPMN Process Expansion* step is described in Section 9.3 in more detail.

- *Step 2: BPMN Mapping*: The expanded BPMN process model is mapped to an EWM based on a corresponding executable business workflow language. MoDFlow.BPMN2BPEL includes a mapping to BPEL that is based on the BPEL mapping in the BPMN standard. It creates a BPEL process model and a corresponding WSDL Extensions model that contains WSDL extension elements of BPEL. The *BPMN Mapping* step is described in Section 9.4 in more detail.
- *Step 3: Workflow Engine Adaptation*: Everything is prepared so that the created EWM can be executed by a workflow engine. This includes the generation of a corresponding deployment descriptor and the EWM may be slightly modified, e.g., to select another version of the XPATH expression language. MoDFlow.BPMN2BPEL includes the creation of a deployment descriptor for Apache ODE. The *Workflow Engine Adaptation* step is described in Section 9.5 in more detail.

Finally, the requirements for a IWM2EWM mapping (RQ_IWM2EWM_*) are recognized by MoDFlow.BPMN2BPEL as follows:

- **RQ_IWM2EWM_01** *Executable Workflow Language Mapping*: The mapping is realized within three steps in MoDFlow.BPMN2BPEL. The first step *BPMN Process Expansion* is independent of the used executable workflow language. Thereby, only those BPMN process elements are allowed for which an execution semantic is defined in the BPMN standard and a mapping to BPEL is possible. The second and third step, *BPMN Mapping* and *Workflow Engine Adaptation*, create an executable BPEL process. Thereby, the mapping to BPEL is based on the mapping in the BPMN standard.
- **RQ_IWM2EWM_02** *Workflow Engine Deployment Descriptor*: The generation of a deployment descriptor for Apache ODE is included in the *Workflow Engine Adaptation* step of MoDFlow.BPMN2BPEL.

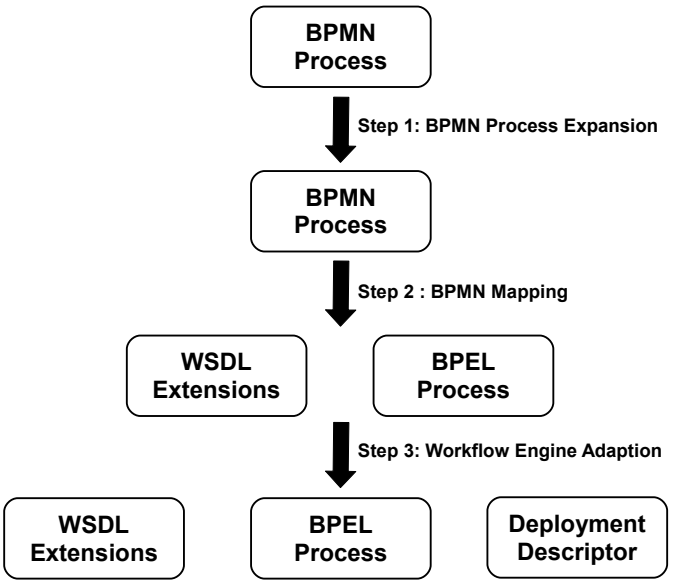

Figure 9.1. BPMN-to-BPEL transformation steps and models

- **RQ_IWM2EWM_03** *Flexibility Constructs*: The generation of flexibility constructs should be applied within the *BPMN Process Expansion* step. This ensures that such constructs are created independently from the used executable workflow language.

- **RQ_IWM2EWM_04** *Monitoring Constructs*: The generation of monitoring constructs should be applied within the *BPMN Process Expansion* step. This ensures that such constructs are created independently from the used executable workflow language.

- **RQ_IWM2EWM_05** *Extensibility*: MoDFlow.BPMN2BPEL provides several extension methods that are individual for each mapping step. They are separately described in Chapter 10.

Table 9.1. Attributes and model associations of `InterfaceConfiguration`

Attribute Name	Description/Usage
wsdlLocation : string [1]	An URI to a local WSDL definition file that defines the Web service interface.
namespace : string [1]	The target namespace used in the WSDL definition of `wsdlLocation`.
portType : string [1]	The qualified name of the used WSDL port type in the format "{<namespace>}<name>". Per default, the first defined WSDL port type in the WSDL definition is used.
service : string [0..1]	The qualified name of the selected WSDL service in the format "{<namespace>}<name>". Per default, the first defined WSDL service is used that contains the selected `servicePort`.
servicePort : string [0..1]	The name of the selected WSDL port. Per default, the first WSDL port is used that references a WSDL binding for `portType`.

9.2 IWM2EWM Mapping Extensions

The BPEL mapping in the BPMN standard is just a basic mapping. The described patterns do not create complete BPEL elements so that the BPEL process is not executable. This is often because certain information, which is required for a mapping to executable BPEL code, cannot be expressed with BPMN or the standard is too unspecific. For this reason, we defined the following BPMN metamodel extensions that fills these information gaps. They supplement the metamodel extensions of MoDFlow.BPMN (see Chapter 8) and are exclusively used in the *BPMN Process Expansion* step of MoDFlow.BPMN2BPEL.

InterfaceConfiguration:
An `InterfaceConfiguration` defines additional technical information for a BPMN interface represented by the metamodel class `Interface` and is mandatory. We assume that each BPMN interface represents a Web service interface, either of the BPMN process itself or of an invoked Web service. The provided attributes of `InterfaceConfiguration` are shown in Table 9.1.

OperationConfiguration:
An `OperationConfiguration` defines additional technical information for a BPMN operation represented by the metamodel class `Operation`, whereby

Table 9.2. Attributes and model associations of `OperationConfiguration`

Attribute Name	Description/Usage
action : string [0..1]	A SOAP action that is used in the SOAP request to invoke the corresponding Web service method.

Table 9.3. Attributes and model associations of `DataTypeConfiguration`

Attribute Name	Description/Usage
type : string [1]	A qualified name of the XML type in the format "{<namespace>}<name>".
collection : boolean [0..1] = false	If set to `true`, the input parameter is regarded as a collection of `type`. The default value is `false`.

each BPMN operation represents a Web service method of a BPMN interface. It is mandatory if the BPMN operation is used for the invocation of a Web service method within a BPMN process. The provided attributes of `OperationConfiguration` are shown in Table 9.2.

DataTypeConfiguration:
A `DataTypeConfiguration` defines additional technical information for a BPMN item definition represented by the metamodel class `ItemDefinition` and is mandatory. We use BPMN item definitions to represent simple and complex XML types, e.g., for WSDL messages and for input and output parameters of workflow activities. The provided attributes of `DataTypeConfiguration` are shown in Table 9.3.

9.3 Step 1: BPMN Process Expansion

The first transformation step *BPMN Process Expansion* expands the BPMN process model of an IWM to represent as much information as possible with BPMN. MoDFlow.BPMN provides a compact BPMN metamodel subset with custom extensions so that only few BPMN elements are required to define a scientific workflow. These elements are expanded in this step, mainly to refine the process flow and the interaction with Web services. Thereby, one BPMN elements may be replaced by a complex and detailed

process flow structure. After the *BPMN Process Expansion* step, the expanded IWM must remain valid with respect to the BPMN standard. Expansions can use the following BPMN elements:

- All BPMN elements and metamodel extensions of MoDFlow.BPMN
- All additional BPMN metamodel extensions that are defined in Section 9.2.
- All BPMN process elements for which an execution semantic is defined in the BPMN standard and a mapping to BPEL mapping possible, e.g., based on the mapping in the BPMN standard. The restriction to the execution semantic ensures that executable BPMN may be used in future instead of BPEL. The restriction to a mapping to BPEL ensures that an executable BPEL process can be created.

The objective to create an expanded IWM is to reduce the complexity of the mapping to BPEL in the *BPMN Mapping* step. It allows providing a mostly direct mapping to BPEL based on the mapping in the BPEL standard and facilitates the adoption of new executable workflow languages, especially regarding a future utilization of executable BPMN. Thereby, we exploit the full potential of BPMN to express a fine-grained process flow of a scientific workflow and its workflow activities.

Expansions cover common and domain-specific aspects. Common expansions often create additional BPMN elements that are required to apply a BPEL mapping of the BPEL standard afterward. Domain-specific expansions usually provide a mapping of a particular workflow activity (BPMN service task element) to the process flow that is required to invoke the corresponding Web services. In the following, we exemplary present two common expansions. As no specific scientific domain is addressed by MoDFlow.BPMN2BPEL, domain-specific expansions, e.g., for a job submission workflow activity, are presented with the application scenarios in Chapter 14.

BPMN process expansion:
The first example expands a BPMN process to represent the communication

between a workflow client and the BPMN process as BPMN collaboration including BPMN participants and conversations. These BPMN elements are the basis of the BPEL mapping in the BPMN standard to generate BPEL partner links and corresponding partner link types. The expansion requires a present `ProcessConfiguration` for the BPMN process element. It is shown in Figure 9.2 and consists of the following steps:

1. A BPMN collaboration is created and added to the BPMN definition via its collection attribute `rootElements`.

2. A reference from the BPMN process to the BPMN collaboration is established via its attribute `definitionalCollaborationRef`.

3. A BPMN participant is created representing the workflow client and added BPMN collaboration via its collection attribute `participants`.

4. A BPMN participant is created representing the BPMN process and added to the BPMN collaboration via its collection attribute `participants`.

5. A reference from the last created BPMN participant to the BPMN process is established via its attribute `processRef`.

6. An expansion is invoked to create a BPMN interface with an `InterfaceConfiguration` for the WSDL definition, that is specified in the attribute `wsdlLocation` of the `ProcessConfiguration`.

7. A reference from the last created BPMN participant to the BPMN interface is established via its collection attribute `interfaceRefs`.

8. A BPMN conversation is created and added to the BPMN collaboration via its collection attribute `conversations`.

9. Two references from the BPMN conversation element to both BPMN participant elements are established via its collection attribute `participantRefs`.

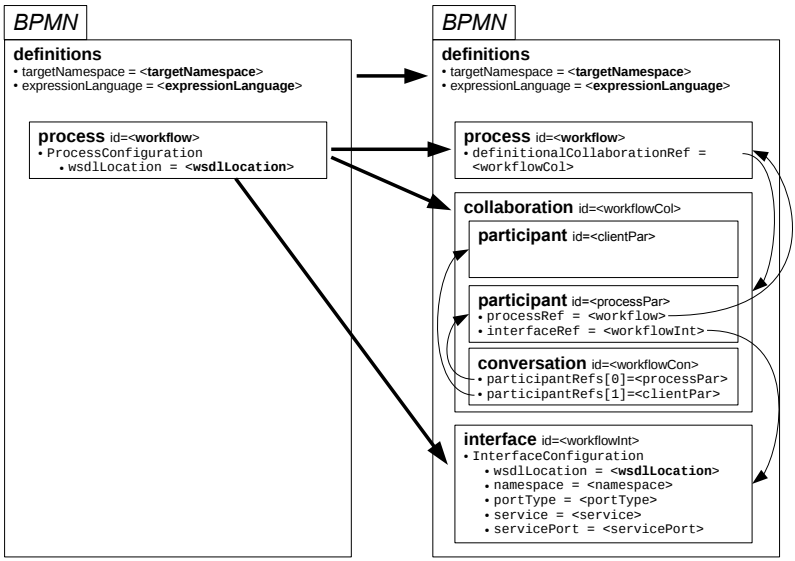

Figure 9.2. BPMN process expansion

BPMN service task fault handling expansion:
The second example is the expansion of a BPMN service task for fault handling. It is applied if the attribute `activityType` of the `ActivityConfiguration` of a BPMN service task is set to "tf.activity.webservice". It further required a present `ServiceTaskConfiguration` whose attributes `operation`, `portType`, and `wsdlLocation` are analyzed to fetch the WSDL faults for the used WSDL operation. A BPMN error event (catching) is created for each WSDL fault and attached as BPMN intermediate boundary event (interrupting) to the BPMN service task. Afterwards, a default BPMN error event is added. The expansion mechanism to map one WSDL fault to corresponding BPMN elements is shown in Figure 9.3 and consists of the following steps:

1. A BPMN item definition is created and added to the BPMN definition.

2. A `DataTypeConfiguration` is created and added to the BPMN item defini-

tion. The attribute `type` of the `DataTypeConfiguration` is set to "{<namespace>}<name>" in which "<namespace>" is the target namespace of the corresponding WSDL definition and "<name>" is the value of the attribute `name` of the WSDL fault.

3. A BPMN error that represents the WSDL fault is created and added to the BPMN definition. The BPMN item definition is referenced by the BPMN error via its attribute `structureRef`.

4. A BPMN boundary event is created and added to the BPMN process. The corresponding BPMN service task is referenced by the BPMN boundary event via its attribute `attachedToRef`.

5. A BPMN error event definition is created and added to the BPMN boundary event. The BPMN error is referenced by the BPMN error event definition via its attribute `errorRef`.

6. A BPMN end event is created and added to the BPMN process.

7. A BPMN sequence flow is created that connects the BPMN boundary event with the BPMN end event and is added to the BPMN process.

8. A BPMN terminate event definition is created and added to the BPMN end event.

Per default, all BPMN error events lead to the termination of the BPMN process via a terminate end event. This default fault handling behavior should be replaced, e.g., for a specific Web service type.

9.4 Step 2: BPMN Mapping

In the second transformation step *BPMN mapping*, the expanded IWM is mapped to an EWM based on BPEL. The result is a BPEL model and a corresponding WSDL Extensions model that contains WSDL extensions elements of BPEL such as partner link types. Thereby, all elements of the expanded IWM are mapped to standard BPEL elements.

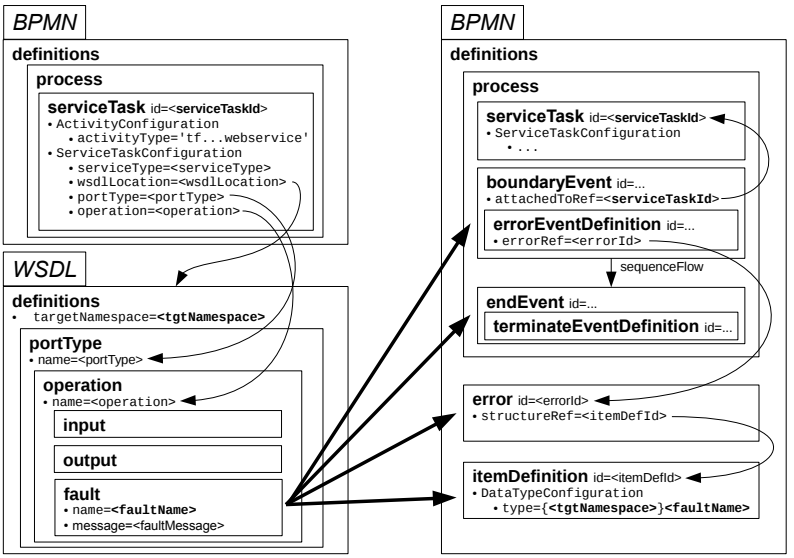

Figure 9.3. Expansions for WSDL fault

Our BPMN-to-BPEL mapping approach is based on a *structure-identification* strategy (see Chapter 2). This strategy generally consists of an algorithm to identify control flow structures (*single entry single exit* (SESE) regions) within a (process) graph, and a mapping for defined control flow patterns such as sequences and loops to the target language. An algorithm for identifying structures in a BPMN process graph was designed and implemented within a diploma thesis [Kippscholl 2012]. The mapping of the identified structures to BPEL is based on patterns that are described in the BPMN standard.

The mapping to BPEL in the BPMN standard is incomplete, because it is defined for a BPMN subset and does not recognize all attributes of source BPMN elements and target BPEL elements. For example, the mapping of a BPMN service task to a BPEL invoke does not create the required attributes `inputVariable` and `outputVariable`. Or it is not clearly specified how information is derived from the BPMN model such as in the mapping

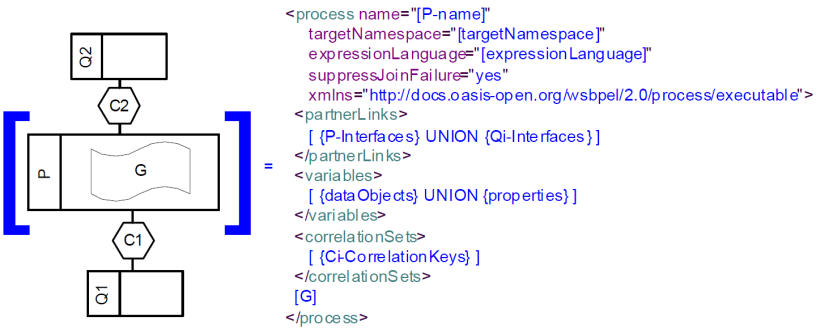

Figure 9.4. Mapping of a BPMN process to BPEL (taken from [OMG 2011a])

of BPMN interfaces to BPEL partner links (see below). Consequently, the created BPEL process is not executable, which an essential requirement for an IWM2EWM mapping is. We thus extended the BPEL mapping so that executable BPEL code is generated. In the following, we give some sample mapping examples.

Mapping of a BPMN process to a BPEL process:
Section 14.1.1 of the BPMN standard defines the mapping of a BPMN process to a BPEL process as follows, see Figure 9.4.

> "The following figure (Figure 9.4[1]) describes the mapping of a Process, represented by its defining Collaboration, to WS-BPEL. The process itself is described by a contained graph G of flow elements to WS-BPEL. The Process interacts with Participants Q1...Qn via Conversations C1...Cm:
>
> The partner links of the corresponding WS-BPEL process are derived from the set of interfaces associated with each participant. Each interface of the Participant containing the Process P itself is mapped to a WS-BPEL partner link with a 'myRole' specification, each interface of each other Participant Qi is mapped

[1]Remark of the authors.

> to a WS-BPEL partner link with a 'partnerRole' specification.
>
> The variables of the corresponding WS-BPEL process are derived from the set 'dataObjects' of all Data Objects occurring within G, united with the set 'properties' of all properties occurring within G, without Data Objects or properties contained in nested Sub-Processes. See 'Handling Data' on page 465 for more details of this mapping.
>
> The correlation sets of the corresponding WS-BPEL process are derived from the correlation keys of the set of Conversations C1...Cn. See page 450 for more details of this mapping." [OMG 2011a]

In the following, we focus on the creation of a BPEL partner link with a 'myRole' specification and corresponding BPEL partner link type for a BPMN process. A BPEL partner link has the attributes `name`, `partnerLinkType`, `myRole`, and `partnerRole`. However, the mapping rule above only defines when a 'myRole' or 'partnerRole' specification is needed and not how it has to be created. The creation of a BPEL partner link type is not addressed.

We extended this mapping so that the required BPEL partner link and BPEL partner link type are created. Thereby, we assume that a BPEL process was previously created as defined in the BPMN standard[2]. We further assume that a WSDL Extension model exists, which is imported by the BPEL process. The steps described below are applied for a BPMN participant whose attribute `processRef` references the BPMN process and that is included in the BPMN collaboration, which itself is referenced by the BPMN process via its attribute `definitionalCollaborationRef`. In other words, the selected BPMN participant contains the BPMN interface that is used for the BPMN process.

1. The first BPMN interface[3] referenced by the collection attribute `interfaceRefs` of the BPMN participant is mapped to a WSDL import that is added to the WSDL Extensions model. The attributes `wsdlLocation`

[2] In the BPMN standard the attribute `name` for a BPEL process element is derived from the attribute `name` of the BPMN process element. We use the attribute `id` of the BPMN process element instead.

[3] In our case, a BPMN process has only one interface.

and `namespace` of the created WSDL import are derived from the corresponding attributes of the `InterfaceConfiguration` of the BPMN interface element.

2. The BPMN conversation in which the BPMN participant element is contained via the collection attribute participantRefs is mapped to a BPEL partner link type that is added to the WSDL Extensions model. The attribute `name` of the BPEL partner link type is derived from the attribute `id` of the corresponding BPMN conversation.

3. The BPMN interface above is mapped to a BPEL role that is added to the created BPEL partner link type. The attribute `name` of the BPEL role is derived from the attribute `id` of the BPMN interface. The attribute `portType` is derived from the attribute `portType` of the `InterfaceConfiguration` of the BPMN interface[4].

4. A BPEL partner link is created and added to the collection attribute `partnerLinks` of the BPEL process in the BPEL model. The attribute `name` of the BPEL partner link is derived from the attribute `id` of the BPMN participant. The attribute `myRole` references the BPEL role via its attribute `name`. The attribute `partnerLinkType` references the BPEL partner link type via is attribute `name`[5].

Mapping of a workflow activity with a parameter sweep to BPEL:
In the following, we describe our basic approach to map a workflow activity to BPEL that is configured as parameter sweep, see Figure 9.6. It requires that the corresponding BPMN service task is configured as *Parallel Multiple Instance Loop* and a `MultiInstanceLoopCharacteristicsConfiguration` (see Chapter 8) is present.

[4]Please note that the namespace prefix is usually determined automatically from the WSDL Extension model implementation and a corresponding namespace definition is added to the WSDL definition element.

[5]Please note that the namespace prefix is usually determined automatically from the BPEL model implementation and a corresponding namespace definition is added to the BPEL process.

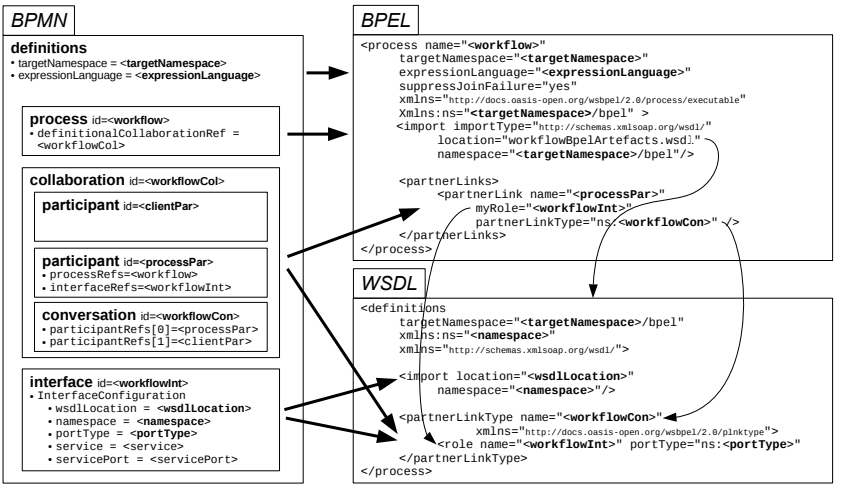

Figure 9.5. Mapping of BPMN process (simplified)

Our approach is that an XML structure is generated and stored in a corresponding BPEL variable that contains all tuples of a parameter sweep, see Listing 9.1. A configurable number of concurrent loop iterations in a BPEL `forEach` element (`parallel=yes`) iterates over this structure and executes corresponding workflow activities using the respective tuples as input.

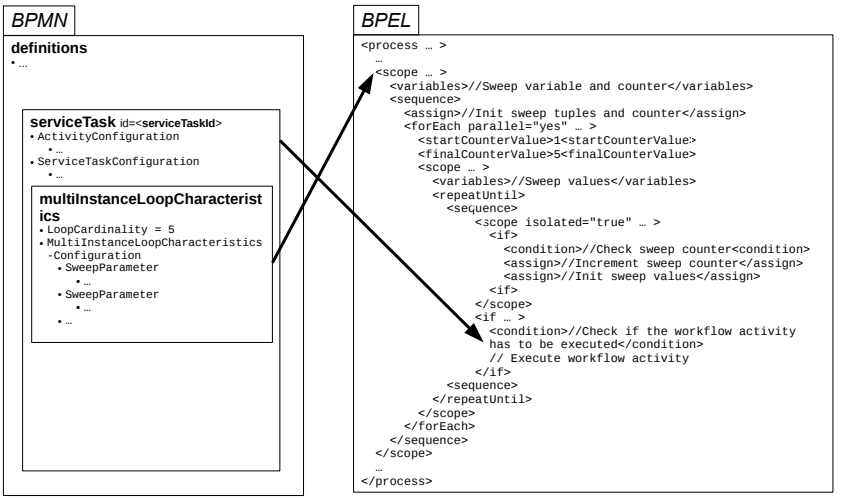

Figure 9.6. Mapping of a workflow activity with a parameter sweep (simplified)

Listing 9.1. BPEL literal structure for parameter sweep tuples

```
1  <sweep>
2      <tuple>
3          <sweepParam1>value1<sweepParam1>
4          ...
5          <sweepParamN>valueN<sweepParamN>
6      </tuple>
7      <tuple>
8          <sweepParam1>value1<sweepParam1>
9          ...
10         <sweepParamN>valueN<sweepParamN>
11     </tuple>
12     ...
13 </sweep>
```

A loop iteration consists of a BPEL `repeatUntil` element that is executed until all tuples are processed. Therefore, a sweep counter is incremented within an isolated BPEL scope by all concurrent loop iterations. The `repeatUntil` element is finished when the sweep counter is greater than the total number of tuples.

9.5 Step 3: Workflow Engine Adaptation

The objective of this mapping step is to prepare everything so that the created EWM can be executed by a workflow engine. Therefore, a corresponding deployment descriptor must be generated, whereby each workflow engine usually provides its own deployment descriptor. We focus on Apache ODE as BPEL workflow engine. Furthermore, an EWM may be slightly modified in this step, e.g., to select another version of the XPATH expression language.

A BPEL process is usually deployed via a deployment package. After this step, all artifacts are ready to be packaged and deployed to a workflow engine. This includes the BPEL model, the WSDL Extensions model and the deployment descriptor as well as the WSDL definition for the BPEL process interface, WSDL definitions for external Web services, and imported XML Schemas and XSLT files.

A deployment descriptor for Apache ODE contains, for example, the binding of all BPEL partner links in the BPEL process to concrete Web services. Thereby, a defined "myRole" in a BPEL partner link refers to a Web service that represents the interface of the BPEL process itself. A defined "partnerRole" in a BPEL partner link refers to an external Web service.

Mapping of a "myRole" BPEL partner link:
The general mapping of a "myRole" BPEL partner link to an Apache ODE deployment descriptor (deploy.xml) is depicted in Figure 9.7 and consists of the following steps:

1. A `provide` element is created and added to the `process` element of the deployment descriptor. The attribute `partnerLink` of the created element is derived from the `name` attribute of the BPEL partner link.

2. A `service` element is created and added to the `provide` element. The attribute `name` of the created element is derived from the referenced WSDL definition. The attribute `portType` is derived from the `role` attribute of the corresponding BPEL partner link type.

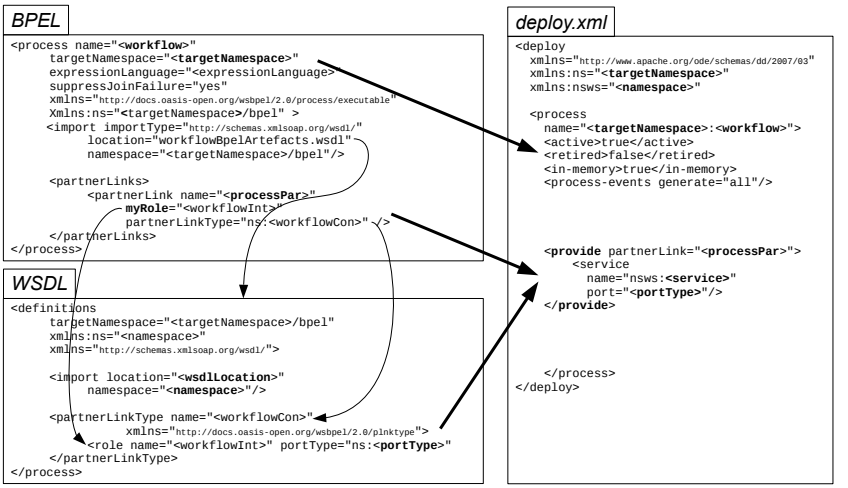

Figure 9.7. Generation of Apache ODE deployment descriptor (simplified)

Chapter 10

Utilization and Extension of MoDFlow

This chapter describes different mechanism to utilize and to extend MoDFlow.BPMN (see Chapter 8) and MoDFlow.BPMN2BPEL (see Chapter 9). They are used for realizing the application scenarios in Chapter 14.

In Section 10.1, we describe different mechanisms to realize a DWM and a DWM2IWM mapping. Mechanisms to extend MoDFlow.BPMN and MoDFlow.BPMN2BPEL are described in Section 10.2.

10.1 Realization of a DWM and a DWM2IWM mapping

The realization of a DWM and a corresponding DWM2IWM mapping should fulfill the requirements (RQ_DWM_* and RQ_IWM2DWM_*) that are described in Chapter 7. A workflow editor for a DWM should support the respective requirements (RQ_WF-ED_*), too.

One mechanism to realize a DWM is the creation of domain-specific languages (DSLs) in the context of model-driven software development (MDSD), which can be distinguished between internal and external DSLs (see Chapter 4). Another mechanism is the adoption of an existing scientific workflow language.

Both approaches require a DWM2IWM mapping to map the used language elements to MoDFlow.BPMN. If a DWM is based on a data flow-centric workflow language, the DWM2IWM mapping must provide a mapping from data flow to control flow as MoDFlow.BPMN is control

flow-centric.

To realize a DWM and a DWM2IWM mapping, different mechanisms to extend MoDFlow.BPMN and MoDFlow.BPMN2BPEL may be applied, see Section 10.2.

10.1.1 Creation of DSLs

The creation of a DSL implies the definition of a new workflow language. A DSL generally consists of the following parts:

- An *abstract syntax* (metamodel).
- At least one *concrete syntax* (textual, graphical or hybrid).
- *Semantics* for the metamodel (behavior, meaning).

In the following, we distinguish between the creation of an internal and an external DSL.

Creation of an internal DSL based on MoDFlow.BPMN:
The characteristic of an internal DSL is that an existing language serves as host language. MoDFlow.BPMN can already be regarded as internal DSL, whereby the host language is BPMN. The abstract syntax is defined by the BPMN metamodel subset with custom extensions. Concrete syntaxes are the textual serialization formats XML and XMI[1] and the graphical BPMN notation. The semantics are based on the BPMN standard and the definition of the custom metamodel extensions.

However, the existing concrete syntaxes are not applicable for the creation of DWMs. The textual concrete syntaxes XML and XMI are exchange formats whose syntax elements should not be used as programming language. The BPMN notation is a graphical concrete syntax that is easy to understand and to use, but it is designed for the business workflow domain and cannot be directly used for scientific workflow modeling. Thus, a graphical concrete syntax for MoDFlow.BPMN is missing that is tailored for the use by scientists. It requires extensions for the BPMN notation,

[1] XMI is a special XML-based format.

e.g., to represent workflow activities and data dependencies between them within a BPMN process diagram. Therefore, the BPMN metamodel for diagram interchange (BPMN DI) can be reused. A corresponding workflow editor can be built on existing tools to visualize and graphically edit BPMN process diagrams.

In summary, the BPMN language infrastructure can be reused to a great extent for creating an appropriate graphical concrete syntax for MoDFlow.BPMN and to provide an internal DSL for scientific workflow modeling. A corresponding workflow editor is directly working on IWMs, whereby the concept of a DWM is realized by the graphical concrete syntax and a DWM2IWM mapping is not required. However, it has to be considered that an IWM is control flow-centric. In order to allow data flow-centric modeling of scientific workflows, a mapping to corresponding control flow elements is required as described in Section 10.1.3.

Creation of an external DSL with a mapping to MoDFlow.BPMN:
The characteristic of an external DSL is that an independent language is created, which is usually mapped to an existing target language. Thus, an abstract syntax, at least one concrete syntax, and a mapping to a target language must be provided to create an external DSL. The semantics is usually given by the mapping to the target language.

In contrast to an internal DSL, the creation of an external DSL has a higher effort, but it provides more flexibility for language design and is not limited by a host language. An external DSL can realize a complete new scientific workflow language for DWMs with a corresponding DWM2IWM mapping. Thereby, it must be ensured that the new workflow language can be mapped to MoDFlow.BPMN. If an external DSL is data flow-centric, a mapping to corresponding control flow constructs is required as described in Section 10.1.3.

The creation of external DSLs is supported by frameworks such as Xtext (see Chapter 4). Xtext allows the definition of textual DSLs based on EMF, whereby a basic language infrastructure including an Eclipse-based editor is automatically generated. It requires that a mapping of the defined DSL to a target language is provided. With further frameworks and tools such

as the Graphical Modeling Framework (GMF)[2] and Graphiti[3], the textual DSL can be supplemented with a graphical concrete syntax. Xtext is used in the PubFlow project and provides the basis for an application scenario that is presented in Chapter 14.

10.1.2 Adoption of Existing Scientific Workflow Languages

A DWM can be generally represented by an existing scientific workflow language, which has the advantage that existing tools such as workflow editors can be reused. However, this approach may cause great efforts for realizing the required DWM2IWM mapping. It may be possible that only a subset of the scientific workflow language can be mapped to MoDFlow.BPMN, e.g., when workflow engine-specific functions are addressed. Certain features of a scientific workflow language may only be covered by utilizing mechanisms to extend MoDFlow.BPMN and MoDFlow.BPMN2BPEL as described in Section 10.2. In the following, we discuss some important aspects to realize a DWM2IWM mapping for an existing scientific workflow language:

- *Mapping of workflow activities*: MoDFlow assumes that the execution of a workflow activity is initiated and controlled by Web service calls. Therefore, a corresponding concept to represent workflow activities is defined in MoDFlow.BPMN. Only those workflow activities defined with a scientific workflow language can be supported, which are representable by a corresponding process flow of Web service invocations. To support a certain workflow activity type or service type, new values for the attributes `activityType` or `serviceType` (see Chapter 8) may be defined, which requires the extension MoDFlow.BPMN2BPEL. In SfWMSs such as Kepler, internal software components are used to execute workflow activities, which starts local processing steps or use external Web services. Thus, it may be necessary that the process flow for a workflow activity has to be extracted from the source code of a software component.
- *Mapping of data flow constructs*: As an IWM is control flow-centric, data flow constructs supported by a scientific workflow language such as

[2] `http://www.eclipse.org/modeling/gmp/`

[3] `http://www.eclipse.org/proposals/graphiti/`

data dependencies between workflow activities must be mapped to corresponding control flow constructs. A basic approach is presented in Section 10.1.3. We assume that required data transfers between workflow activities can be initiated and controlled by corresponding data transfer services. For executing data transfers, MoDFlow.BPMN2BPEL may be extended to add data transfer activities between workflow activities when needed.

- *Mapping of control flow constructs*: Only those control flow constructs of a scientific workflow language can be supported that can be mapped to corresponding elements of MoDFlow.BPMN such as gateways and sequence flows.

- *Mapping of data types*: All data types in MoDFlow.BPMN are based on simple XML types or custom XML schema types. Thus, the supported data types of a scientific workflow language must be mapped to corresponding XML-based types.

As a DWM2IWM mapping should be realized as model transformation, it is an advantage if a corresponding Ecore model (metamodel) already exists for a scientific workflow language. Otherwise, an Ecore model can be derived from an XML schema, which is usually provided by all scientific workflow languages that are based on XML.

10.1.3 Mapping of Data Flow-centric to Control Flow-centric Workflow Languages

If a data dependency is defined between two workflow activities A and B, it means that B consumes data from A and is thus to be executed after A. As this implies an order of execution, it can be considered as a control dependency as well. Vice versa, a control dependency between two workflow activities does not automatically represent a data dependency. Data flow is usually expressed explicitly in control flow-centric languages, e.g., by read and write operations on variables.

Based on the fact that a data dependency is always a control dependency, certain data flow constructs in data flow-centric workflow languages can

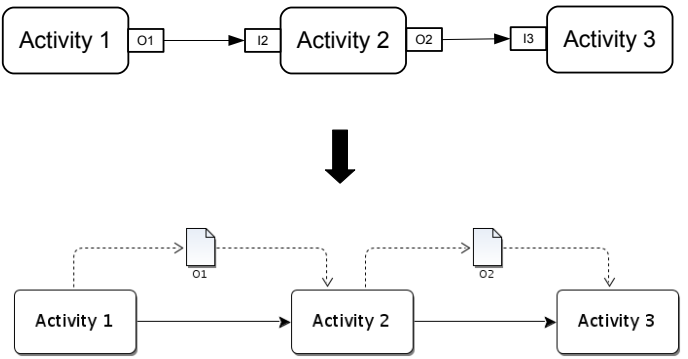

Figure 10.1. Mapping of sequential data flow to sequential control flow[a]

[a]Lower BPMN process created with Yaoqiang BPMN Editor (`http://sourceforge.net/projects/bpmn`)

always be mapped to corresponding control flow constructs in control flow-centric workflow languages. Thereby, the execution of the data flow must be ensured by appropriate elements of the target workflow language. In the following, we present two basic patterns in order to map data flow constructs to control flow constructs, whereby the control flow-centric workflow language is illustrated with BPMN.

The first example is shown in Figure 10.1 that contains a sequence of three workflow activities (Activity 1-3) with corresponding data flow dependencies between input and output parameters (O1->I2, O2->I3). This construct is mapped to a sequence of three workflow activities in BPMN with sequence flows as control flow dependencies. The data flow is represented as BPMN data objects with corresponding read and write data references. Please note, that BPMN data objects are created solely for the output parameters (O1 and O2) of workflow activities.

The second example is shown in 10.2. Output O1 is consumed by Activity 2 and Activity 3, which can be executed concurrently. As Activity 4 consumes the output of both aforementioned activities, the concurrency synchronizes at this point. This construct can be mapped to BPMN by using the BPMN parallel gateway to split and join control flow.

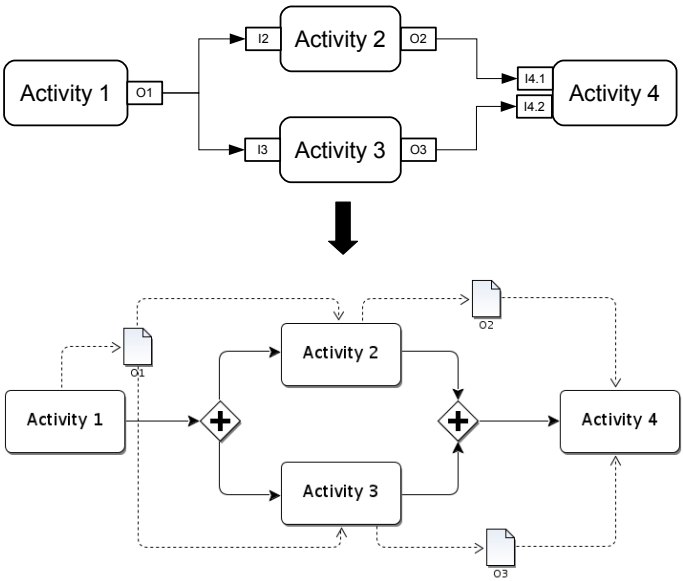

Figure 10.2. Mapping of concurrent data flow to concurrent control flow[a]

[a]Lower BPMN process created with Yaoqiang BPMN Editor (`http://sourceforge.net/projects/bpmn`)

The data that is exchanged between workflow activities within a workflow engine only contains information about the data locations, e.g. data references, and not the data itself. Thus, required data transfers between different locations for workflow activity executions are either identified and executed by the workflow engine itself or MoDFlow.BPMN2BPEL is extended so that special data transfer workflow activities are added to the workflow model when needed.

10.2 Extension of MoDFlow.BPMN and MoDFlow.BPMN2BPEL

MoDFlow can be extended in different ways that are presented in the following.

10.2.1 General Extension of MoDFlow.BPMN

MoDFlow.BPMN provides the definition of own values for the attributes `activityType`, `eventType`, and `serviceType` (see Chapter 8). In addition, many custom metamodel extensions allow for defining individual configuration parameter to express additional information.

Furthermore, the BPMN metamodel subset and custom extensions of MoDFlow.BPMN itself could be extended. Thereby, it should be considered to only use BPMN elements for which an execution semantics is defined in the BPMN standard and a BPEL mapping is possible. Another option is to utilize the metamodel extension mechanism of BPMN.

All approaches require the extension of MoDFlow.BPMN2BPEL to support the new customizations. In most cases, the *BPMN Process Expansion* mapping step should be extended.

10.2.2 General Extension of MoDFlow.BPMN2BPEL

MoDFlow.BPMN2BPEL separates the BPMN-to-BPEL mapping into three steps. Each step of this transformation chain can provide special methods to extend existing or to provide own mappings, which is recognized in the design and implementation on MoDFlow.BPMN2BPEL (see Chapter 12 and 13). Furthermore, a complete mapping step may be exchanged or an additional mapping step may be added. For the addition, the required input and output models of previous or succeeding transformation steps must be recognized.

As BPEL already covers common requirements for the execution of scientific workflows and to avoid the modification of an existing BPEL workflow engine, we focus on the utilization of standard BPEL language elements. However, for the creation of a complete SWfMS and to cover

specific aspects of scientific workflows, certain BPEL runtime extension can be used as depicted by Görlach et al. [2011]. MoDFlow.BPMN2BPEL can be extended to support such extensions, however, this also requires the modification of the Apache ODE workflow engine.

10.2.3 Definition of Workflow Activities

One essential extension method is the definition of own workflow activities that can be identified via custom values for the attribute `activityType` of the BPMN metamodel extension class `ActivityConfiguration` (see Chapter 8). Certain values for `activityType` are usually interpreted in the *BPMN Process Expansion* mapping step of MoDFlow.BPMN2BPEL.

For the further definition of a workflow activity, existing elements of MoDFlow.BPMN can be reused. e.g., to define one single Web service invocation based on the `ServiceTaskConfiguration` metamodel extension. In this case, MoDFlow.BPMN2BPEL must not be changed. If the process flow for a workflow activity is more complex, the *BPMN Process Expansion* mapping step of MoDFlow.BPMN2BPEL must be extended by a corresponding expansion for the `activityType` value.

10.2.4 Integration of Third Party Software

One challenge for the execution of scientific workflows in SOAs is the integration of third party software as workflow activity, because each used software must be executable via Web service invocations. We thereby distinguish the following solutions:

- In the best case the software already provides a Web service interface that can be used.
- The software is manually *wrapped* with a Web service, which must be implemented and deployed to a corresponding service container.
- Workflow activities can be executed on a Grid site. Therefore, common Grid middlewares such as Globus Toolkit 4 and UNICORE 6 provide job submission services to execute a local executable as *job*. A job submission

service can be used to execute a third party software if it can be installed on a Grid site and if it can be invoked via command line.

For each integrated software, a workflow activity and corresponding expansion may be defined as described above. A job submission workflow activity, for example, consists of several Web service invocations for job creation, job monitoring, and cleanup.

10.2.5 Adoption of other BPEL Process Engines

The most important point for the adoption of a new BPEL process engine is the creation of the engine-specific deployment descriptor. Therefore, the *Workflow Engine Adaptation* mapping step of MoDFlow.BPMN2BPEL must be replaced. It may be further required that the corresponding EWM is slightly modified, e.g., to select a certain XPATH version.

10.2.6 Adoption of other Executable Workflow Languages

For the adoption of a new executable workflow language for EWMs, the *BPMN Mapping* and *Workflow Engine Adaptation* mapping steps of MoDFlow.BPMN must be replaced.

Chapter 11

Requirements on the MoDFlow Framework

The MoDFlow framework refers to the technical realization of the MoDFlow approach (see Chapter 6). It requires an implementation for MoDFlow.BPMN (see Chapter 8) and MoDFlow.BPMN2BPEL (see Chapter 9).

For the implementation of the BPMN subset defined in MoDFlow.BPMN the existing Ecore model of [Hille-Doering 2010][1] can be reused. The custom metamodel extensions of MoDFlow.BPMN can be realized as own Ecore model based on the definition in Chapter 8.

An implementation of MoDFlow.BPMN2BPEL is based on a transformation framework and a BPMN-to-BPEL transformation chain. The transformation framework provides the execution of single model transformations or a sequence of them as transformation chains. It is used for creating the BPMN-to-BPEL transformation chain as defined in MoDFlow.BPMN2BPEL (see Chapter 8).

First of all, we classify the BPMN-to-BPEL mapping defined MoDFlow.BPMN2BPEL in Section 11.1 in order to characterize the transformation problem and to identify implications on usable transformation technologies. Based on the results, we define requirements for a transformation framework in Section 11.2 and a BPMN-to-BPEL transformation chain in Section 11.3.

[1] Published at http://www.eclipse.org/modeling/mdt/?project=bpmn2

11.1 Classification of BPMN-to-BPEL mapping

Different approaches exist in the scientific literature in order to classify transformation problems and transformation technologies [Czarnecki and Helsen 2006; Mens and Gorp 2006; Huber 2008; Biehl 2010]. A compact classification scheme that is based on other approaches is presented by Biehl [2010]. It differentiates between a *Classification Scheme for Model Transformation Problems* and a *Classification Scheme for Model Transformation Languages*.

In the following, we apply both classification schemes to the BPMN-to-BPEL mapping defined by MoDFlow.BPMN2BPEL. We use the *Classification Scheme for Model Transformation Problems* to characterize the transformation problem for a BPMN-to-BPEL mapping on a general level. We use the *Classification Scheme for Model Transformation Languages* to identify implications on transformation technologies, which help us formulate requirements for the transformation framework as well as for the BPMN-to-BPEL transformation chain. Each relevant aspect in both classification schemes is introduced briefly before we proceed with our findings. For a detailed description of each aspect please refer to [Biehl 2010] and its references. All terms reused from [Biehl 2010] are highlighted as *italic*.

11.1.1 Classification Scheme for Model Transformation Problems

The *Classification Scheme for Model Transformation Problems* is used to classify and characterize a transformation problem on a general level. It consists of the following aspects:

- **Change of Abstraction**: Transformations on models can be distinguished between *vertical transformations* and *horizontal transformations*, cf. [Reussner and Hasselbring 2008]. A *vertical transformation* adds information (*refinement transformation*) or removes information (*abstraction transformation*) and therefore changes a model's level of detail. A *horizontal transformation* changes the representation of a model only, e.g. for refactoring, whereby no refinement or abstraction is conducted.

In this sense, the overall mapping from BPMN to BPEL falls into the category of a *vertical/refinement transformation*, whereby each mapping step has its own classification. The mapping steps *BPMN Process Expansion* and *BPMN Mapping* are *horizontal transformations*, because they are reducing the level of abstraction of the workflow by adding details. The *Workflow Engine Adaptation* mapping step creates an additional Deployment Descriptor model without changing the level of abstraction of the workflow, and is thus classified as *horizontal transformation*.

- **Change of Metamodels**: Each source and target model of a transformation has a corresponding metamodel. In an *endogenous transformation*, the metamodels of all source and target models are identical. In an *exogenous transformation*, the metamodels are different. An *endogenous transformation* can modify all source models instead of creating new target models. The modified source models are then used as target models, which is called an *in-place transformation* [Mens and Gorp 2006]. A so-called *out-place transformation* [Mens and Gorp 2006] maps information from source to new and therefore empty target models. *Exogenous transformations* are always *out-place transformations*. One transformation may also be a combination of *in-place transformations* and *out-place transformations* on source models.

  The BPMN-to-BPEL mapping consists of both *in-place transformations* and *out-place transformations*:

  - The *BPMN Process Expansion* step (BPMN model -> BPMN model) is an *in-place transformation*.
  - The *BPMN Mapping* step (BPMN model -> BPEL model and WSDL Extensions model) transformation step is an *out-place transformation*.
  - The *Workflow Engine Adaptation* step (BPEL model and WSDL Extensions model -> BPEL, WSDL and Deployment Descriptor model) is a combination of *in-place transformations* (BPEL model -> BPEL model, WSDL Extensions model -> WSDL Extensions model) and one *out-place transformation* (BPEL model and WSDL Extensions model -> Deployment Descriptor model).

- **Supported Technical spaces**: *Technical spaces* can be used to characterize and group artifacts and to define boundaries of model-driven technologies and concepts [Bézivin and Kurtev 2005]. Examples for *technical spaces* in MDSD are the OMG technical space and the EMF technical space. Thereby, the EMF *technical space* can be viewed as a link between the OMG technical space and the Java technical space.

  As the use of EMF is a prerequisite (see Chapter (6), the BPMN-to-BPEL mapping depends on the EMF technical space.

- **Supported Number of Models**: Transformations can have multiple source and target models. The source and target models of an *in-place transformations* are identical and counted once. Thus, an *in-place transformation* may be executed on one model only.

  The BPMN-to-BPEL mapping steps involve the following number of models:

  - The *BPMN Process Expansion* step involves one model (BPMN model) (*in-place transformation*).
  - The *BPMN Mapping* step involves three models (BPMN model, BPEL model, and WSDL Extensions model) (*out-place transformation*).
  - The *Workflow Engine Adaptation* step involves three models (BPEL model, WSDL Extensions, and Deployment Descriptor model) (*in-place transformation* and *out-place transformation*).

- **Supported Target Type**: A target type is either *model* or *text* and refers to the type of a transformation output. If an input model is mapped to an output model, the transformation is a *model-to-model (M2M) transformation*. *M2M transformations* are defined on the metamodels of models. If an input model is mapped to plain text (string), e.g. for code generation, the transformation is a *model-to-text (M2T) transformation*. A *M2T* transformation uses the metamodels of source models only.

  Each step of the BPMN-to-BPEL mapping is a *M2M transformation*. We assume that for all models (BPMN model, BPEL model, WSDL Extensions model, and Deployment Descriptor model) a corresponding metamodel exists on which *M2M transformations* can be defined.

- **Preservation of Properties**: The preservation of certain properties during a transformation is classified as *semantics-preserving*, *behavior-preserving* and *syntax-preserving*. *Semantics-preserving* is fulfilled when both source and target model have the same computational output after a transformation. It is used for performance improvements, for example. *Behavior-preserving* is part of *semantics-preserving* and is fulfilled if constraints (implicit or explicit) on a source model are maintained in the target model while the computational output differs slightly, for example, in code generation with a *M2T transformation*. A transformation is *syntax-preserving* if the abstract syntax of the target model does not change, which is commonly the case in *in-place horizontal transformations*, e.g., where changes are applied to the layout of a graphical concrete syntax only.

  The application of this aspect to the BPMN-to-BPEL mapping is difficult, e.g., because we do not have a computational output for the BPMN models. However, the *BPMN Process Expansion* and *BPMN Mapping* steps may be classified as *behavior-preserving* as the general behavior of a workflow is not changed. The *Workflow Engine Adaptation* transformation step may be classified as *semantics-preserving* as the computational output of the BPEL workflow is not changed.

Table 11.1 summarizes the results obtained from the *Classification Scheme for Model Transformation Problems*.

Change of Abstraction	*Refinement transformation* based on the combination of prevalent *vertical/refinement transformations* and *horizontal transformations*
Change of Metamodels	Combination of *in-place* and *out-place transformations*
Supported Technical Spaces	EMF technical space
Supported Number of Models	From one to multiple models

Supported Target Type	*Model-to-Model (M2M) transformations*
Preservation of Properties	Prevalent *behavior-preserving transformations* and one *semantic-preserving transformation*

Table 11.1. Classification of BPMN-to-BPEL mapping problem

11.1.2 Classification Scheme for Model Transformation Languages

The *Classification Scheme for Model Transformation Languages* is used to classify and characterize transformation languages on a general level. We use it to determine applicable transformation technologies for the BPMN-to-BPEL mapping, in which we recognize the results of the *Classification Scheme for Model Transformation Problems* (see above). It consists of the following aspects:

- **Paradigm**: Transformation languages can be divided into *imperative (operational) transformation languages*, *declarative (relational) transformation languages* and their combination as *hybrid languages*. *Imperative transformation languages* can be compared with general purpose programming languages (such as Java) in which the control flow of a transformation is explicitly defined. *Declarative transformation languages* can be compared with XSLT in which transformations are defined as mapping rules whose order of execution (control flow) is not explicitly specified. *Graph languages* are often used as formal foundation for *declarative transformation languages*. *Template-based transformation languages* are used for code generation in a *M2T transformation*. *Direct manipulation* refers to the utilization of general purpose programming languages such as Java to implement a transformation.

  In principle, any type of transformation language can be used to implement the BPMN-to-BPEL mapping. Each mapping step may be

implemented using a different transformation language. One common observation is that *declarative transformation languages* are more suitable for small transformation problems and *imperative transformation languages* are more suitable for complex transformation problems [Huber 2008]. We regard the overall BPMN-to-BPEL mapping as a complex transformation problem in which each mapping step may differ in complexity. Thus, we consider *imperative transformation languages* more applicable.

- **Rule Application Control**: Transformation languages provide different means for defining the execution order of transformation rules. *Implicit control* means that the execution order cannot be controlled directly. *Explicit control* means that the execution order is defined together with the transformation rules. *External control* means that the execution order is defined separately from the transformation rules. *Rule application scoping* means that the application of transformations rules only affects parts of the source or target model.

  The BPMN-to-BPEL mapping has no specific implications for any type of *rule application control* in a transformation language.

- **Rule Scheduling**: *Rule scheduling* concerns means for *rule application controlling* during transformation execution. *Rule selection* uses particular matching algorithms to produce transformation sequences, which may be *deterministic* or *non-deterministic*. *Rule-iteration* is based on recursion, looping or fixed point operations for transformation rule scheduling. *Phasing* means that some transformation rules are allowed in a predefined transformation execution phase only.

  The BPMN-to-BPEL mapping has no specific implications for any type of *rule scheduling* in a transformation language.

- **Rule Organization**: *Rule Organization* addresses the modularization of transformations. It allows, for example, the reuse of transformation in transformation compositions. *Internal compositions* are supported by a transformation language itself to internally organize transformation rules, e.g. by inheritance. *External compositions* provide the composition of many transformations as transformation chain that may be implemented with different transformation languages.

Due to the complexity of the BPMN-to-BPEL mapping and its organization as transformation chain, both composition approaches (*internal* and *external*) are needed. *Internal composition* is needed to modularize complex transformations and to allow extensions. *External composition* is needed to define transformation chains.

- **Traceability**: *Traces* are used as transformation execution log. They provide provenance information that allows tracking the transformation of source model elements to target model elements and the applied transformation rules. *Traces* can be used, for example, to debug transformations.

  To implement the BPMN-to-BPEL mapping a debug mechanism is essential. However, it is not mandatory that this mechanism is based on explicit *trace* support.

- **Directionality**: *Unidirectional transformation languages* allow *unidirectional transformations* from source to target models. *Multi-directional transformation languages* allow to define transformations in several directions, which are typically *bidirectional transformations* between one source and one target model.

  Each step of the BPMN-to-BPEL mapping is a *unidirectional transformation*.

- **Incremental Model Transformation**: The execution of a transformation can be either *non-incremental* or *incremental*. A *non-incremental transformation* always creates a new target model and is usually supported by any transformation language. An *incremental transformation* updates an existing target model based on changes in the source model. This can be done by just updating the target model without complete recreation (*target-incremental*) or by minimizing the number of rechecked elements in a source model (*source-incremental*). Thus, *incremental transformations* execute the same transformation several times to propagate changes from source models to target models. Manual updates that are applied meanwhile to the target model may be recognized and preserved. The support of *incremental transformations* is usually based on *traces*.

  The BPMN-to-BPEL mapping is *non-incremental*, because it is intended that each step always creates new output models. Thus, the support of *incremental transformations* is not needed.

▻ **Representation of the Transformation**: Transformation languages may be based on manually created plain text files or they are themselves created with model-driven technologies. The latter requires a metamodel with a corresponding abstract syntax and concrete syntax. That allows the execution of so-called *high-order transformations*, which are transformations on transformation models.

The BPMN-to-BPEL mapping has no implications for the representation of a transformation. Thus, the used transformation language does not necessarily have to provide a metamodel.

Table 11.1 summarizes the results obtained from the *Classification Scheme for Model Transformation Languages* and its implications for applicable transformation languages.

Paradigm	Depends on the complexity of each transformation step but *imperative transformation languages* appear more applicable
Rule Application Control	No implications
Rule Scheduling	No implications
Rule Organization	*Internal* and *external compositions*
Traceability	Not mandatory but valuable for debugging purposes
Directionality	*Unidirectional transformations*
Incremental Model Transformation	Not needed as the BPMN-to-BPEL transformation is *non-incremental*
Representation of the Transformation	No implications

Table 11.2. Implications of BPMN-to-BPEL mapping for transformation languages

11.2 Requirements for a Transformation Framework

Based on the classification in Section 11.1, we define the following requirements for a transformation framework:

- **RQ-TF-01** *EMF Support*: The transformation framework must be based on EMF. All metamodels for models must be defined by a corresponding Ecore model. All used transformation technologies must support EMF as well. Input and output models of single transformations and transformation chains must always be EMF models.

- **RQ-TF-02** *Transformation Execution*: The transformation framework must provide the execution of single transformations on EMF models based on arbitrary transformation technologies/languages, which must support *unidirectional transformations* and *M2M transformations*. Each used transformation technology must further support *in-place transformations*, the stand-alone invocation of a single transformation within Java code, and *internal composition* features such as inheritance. At least one transformation technology must be imperative.

- **RQ-TF-03** *Transformation Chain Execution*: The transformation framework must provide the execution of sequential transformation chains (*external composition*) on EMF models based on arbitrary transformation technologies. Each used transformation technology must provide the stand-alone invocation of transformation chains within Java code. All transformation executions in a transformation chain must fulfill RQ-TF-02.

- **RQ-TF-04** *Tooling*: An appropriated tooling should be supported for each supported transformation technology to define single transformation and transformation chains. Thereby, an Eclipse-based tooling is preferred due to the utilization of EMF.

- **RQ-TF-05** *Extensibility* : The transformation framework must provide practical means for integrating other transformation technologies, which again must fulfill RQ-TF-02 and RQ-TF-03.

11.3 Requirements for a BPMN-to-BPEL Transformation Chain

Based on the classification in Section 11.1, we define the following requirements for a BPMN-to-BPEL transformation chain:

- **RQ-B2B-01** *Ecore Models*: Each model (BPMN, BPEL, WSDL and Deployment Descriptor model) included in the transformation chain must have a corresponding Ecore model as metamodel.

- **RQ-B2B-02** *BPMN Subset and Extensions Ecore Model*: The BPMN subset of MoDFlow.BPMN must be represented by the existing Ecore model for BPMN [Hille-Doering 2010]. The custom metamodel extensions of MoDFlow.BPMN and MoDFlow.BPMN2BPEL must be represented by an own Ecore model. Additionally, a validation mechanism must be provided to ensure that a BPMN process model is valid according to MoDFlow.BPMN as defined in Chapter 8.

- **RQ-B2B-03** *BPMN-to-BPEL Mapping Steps*: Each mapping step of MoDFlow.BPMN2BPEL must be realized as single M2M transformation.

- **RQ-B2B-04** *Transformation Framework Utilization*: The BPMN-to-BPEL transformation chain and each contained model transformation must be executed with the transformation framework. It is considerable to use only one transformation language for all model transformations so that functions can be reused.

- **RQ-B2B-05** *Structure-Identification Algorithm*: An algorithm must be provided for the identification of structures in BPMN process models. It is needed for the *BPMN Mapping* step of MoDFlow.BPMN2BPEL.

- **RQ-B2B-06** *Apache ODE Support*: Apache ODE must be supported as default BPEL workflow engine. This includes the generation of a corresponding deployment descriptor in the *Workflow Engine Adaptation* step of MoDFlow.BPMN2BPEL.

▻ **RQ-B2B-07** *Extensibility*: Each single model transformation should provide individual extension mechanisms. Extension mechanisms should utilize the inheritance support of the transformation language.

Chapter 12

Design of the MoDFlow Framework

This Chapter presents the design for the MoDFlow framework, which includes a basic architecture of the transformation framework in Section 12.2 and a conceptual design of the BPMN-to-BPEL transformation chain in Section 12.2.

12.1 Transformation Framework

The basic architecture for the transformation framework consists of three layers represented as single components, see Figure 12.1. Each component has some basic classes to better illustrate the interaction between the different layers. They are described from bottom-up in the following.

Transformation Executor Layer:
The *Transformation Executor Layer* provides a simple plugin mechanism based on the factory design pattern to encapsulate different transformation technologies as *Transformation Executor*. Therefore, each *Transformation Executor* implements the common interface `ITransformationExecutor`. A *Transformation Executor* executes one single model transformation based on a list of input EMF models and a transformation artifact that is defined with the used transformation technology. Optionally, individual properties can be passed within each method invocation. The method `runTransformation(...)` can be used for out-place transformations and the method `runInPlaceTransformation(...)` for in-place transformations.

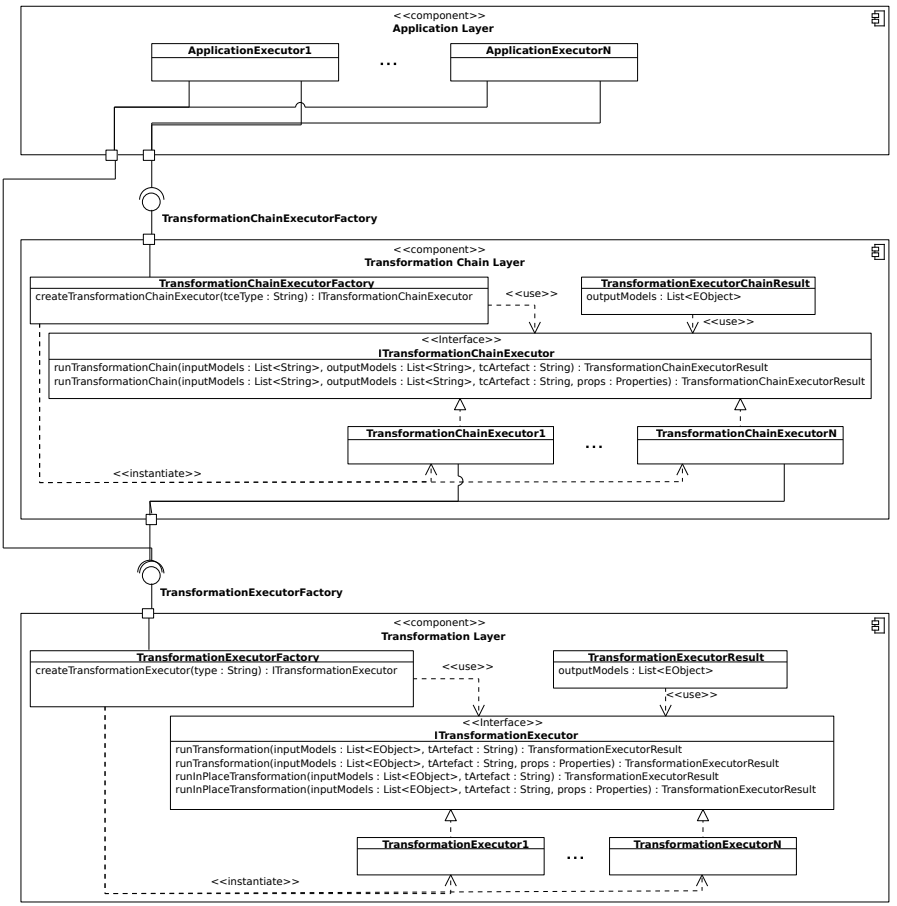

Figure 12.1. Basic architecture of transformation framework

The result of a model transformation is a `TransformationExecutorResult` that contains the output EMF models. A *Transformation Executor* can be instantiated based on its type (unique String identifier) via the factory `TransformationExecutorFactory`.

Transformation Chain Layer:
The *Transformation Chain Layer* provides a simple plugin mechanism based on the factory design pattern to encapsulate different transformation chain technologies as *Transformation Chain Executor*. Therefore, each *Transformation Chain Executor* implements the common interface `ITransformationChainExecutor`. A *Transformation Chain Executor* executes one single transformation chain via the method `runTransformationChain(...)` based on lists of input and output EMF models paths, and a transformation chain artifact, which is defined with the used transformation chain technology. Optionally, individual properties can the passed within each method invocation. The result of a transformation chain is a `TransformationChainExecutorResult` that contains the output EMF models. *Transformation Chain Executor* can be instantiated based on their type (unique String identifier) via the factory `TransformationChainExecutorFactory`.

Application Layer:
The *Application Layer* contains *Application Executor*, which represent applications that utilize *Transformation Chain Executor* and *Transformation Executor*. An *Application Executor* could be, for example, a command-line client to test transformation chains and single transformations or an Apache ODE client to deploy generated BPEL processes.

Figure 12.2 shows a sample interaction sequence between all components in which the *Application Executor* (ApplicationExecutor1) invokes a transformation chain that consists of two single transformations. At first, the system instantiates the *Transformation Chain Executor* `TransformationChainExecutorN` via `TransformationChainExecutorFactory` and invokes (`runTransformationChain(...)`). The corresponding transformation chain artifact defines a sequence of two sequential transformation steps (in-place and out-place transformation) based on the *Transformation Executor* `TransformationExecutor1`

and `TransformationExecutorN`. Thus, for each transformation step the corresponding *Transformation Executor* is created via the factory `TransformationExecutorFactory` and invoked by `runInPlaceTransformation(...)` or `runTransformation(...)`.

12.2 BPMN-to-BPEL Transformation Chain

The conceptual design for the BPMN-to-BPEL transformation chain is shown in Figure 12.3. Each step is realized as single model transformation. The execution of the transformation chain is based on the transformation framework, see Section 12.1. We assume that the used transformation language provides inheritance.

We provide the BPMN-to-BPEL transformation chain as specified by MoDFlow.BPMN2BPEL (see Chapter 9) as standard transformation chain. The input BPMN model must be compliant to the BPMN metamodel subset and custom extensions as defined by MoDFlow.BPMN (see Chapter 8). It creates all models (BPEL model, WSDL Extensions model, and Deployment Descriptor model) so that a BPEL process can be deployed to an Apache ODE engine. The BPEL model and WSDL Extensions model comply with the BPEL 2.0 standard. Finally, the standard transformation chain consists of the following transformation artifacts:

- *MoDFlow Expansions*: This transformation artifact realizes the default *BPMN Process Expansions* step. It creates an expanded BPMN model based on the corresponding expansion rules. Only the predefined values are supported for the attributes `activityType` ("tf.activity.webservice") and `eventType` ("tf.event.message") of `ActivityConfiguration` and `EventConfiguration`.

- *MoDFlow Mapping*: This transformation artifact realizes the default *BPMN Mapping* transformation step. It creates a BPEL model and a WSDL Extensions model based on the corresponding mapping rules. The mapping of BPMN to BPEL as defined in the BPMN standard is encapsulated in a single transformation artifact *BPMN 2.0 Mapping*. This

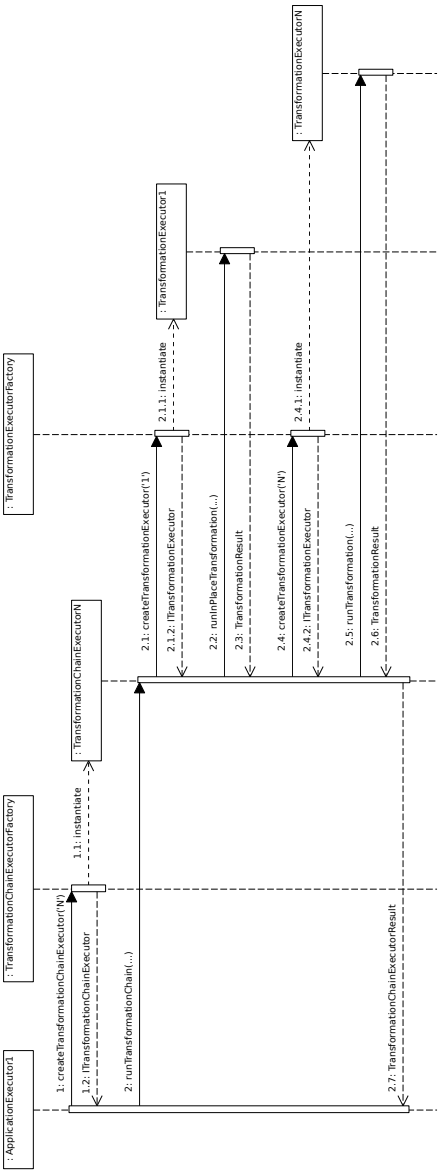

Figure 12.2. Interaction of components in the transformation framework

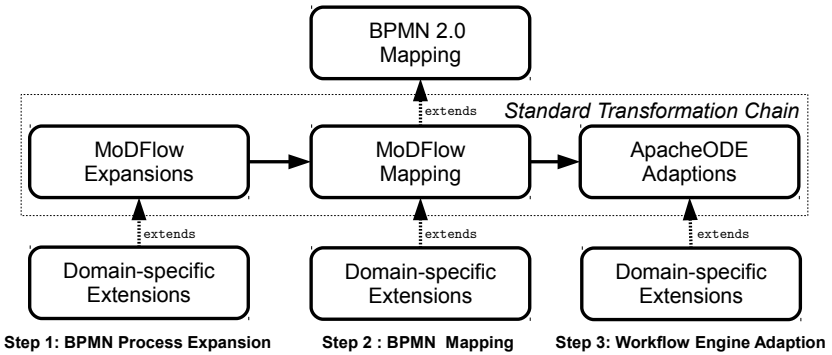

Figure 12.3. Conceptual design of BPMN-to-BPEL transformation chain

artifact is extended by *MoDFlow Mapping* with all extensions we defined in MoDFlow.BPMN2BPEL (see Chapter 9).

- *Apache ODE Adaptions*: This transformation artifact realizes the default *Workflow Engine Adaptation* transformation step. It creates a deployment descriptor for Apache ODE.

Individual extensions for the standard transformation chain should be applied as follows:

- Insert a new transformation artifact at any position in the transformation chain.
- Extend a transformation artifact with domain-specific extensions.
- Replace a transformation artifact with a new one.

Part III

Evaluation

Chapter 13

Implementation of the MoDFlow Framework

This chapter presents the implementation of the MoDFlow framework, which includes the transformation framework in Section 13.1 and the BPMN-to-BPEL transformation chain in Section 13.2. All software artifacts and examples are published under the Eclipse Public License at `http://sourceforge.net/projects/bpmn2bpel/` as an Apache Maven[1] project.

13.1 Transformation Framework

The implementation of the transformation framework is based on Java and realizes the design as introduced in Chapter 12. It provides the execution of single model transformations and transformation chains on EMF models. A simple plugin mechanism enables the use of different technologies for both model transformation execution and transformation chaining. The corresponding Apache Maven project contains the following Maven artifacts:

- `tf.common`: Common Java interfaces and classes
- `tf.xtend`: Xtend classes
- `tf.transformation`: *Transformation Layer*
- `tf.chaining`: *Transformation Chain Layer*
- `tf.application`: *Application Layer*

[1] `http://maven.apache.org/`

13.1.1 Implementation Decisions

The description of our implementation decisions regarding the transformation framework and the technologies used is structured according to the requirements defined in Chapter 11.

RQ-TF-01 ***EMF Support*****:**
The entire transformation framework is designed on the basis of EMF, which is also reflected in the method signatures of its interfaces and classes. All technologies used to execute single model transformations (ATL, QVT Operational Mappings, and Xtend) and transformation chains (MWE2) support EMF. EMF models are internally represented by the Ecore class `EObject`. The reading and writing of EMF models is encapsulated by the utility class `ModelUtils` located in `tf.common`.

RQ-TF-02 ***Transformation Execution*****:**
Based on this requirement as well as RQ-TF-01 and RQ-TF-04, an optimal transformation language covers the following features:

- ▻ EMF support (RQ-TF-01)
- ▻ Eclipse Tooling (RQ-TF-04)
- ▻ Imperative language (RQ-TF-02)
- ▻ Support of M2M transformations (RQ-TF-02)
- ▻ Support of in-place transformation (RQ-TF-02)
- ▻ Support of internal composition as inheritance (RQ-TF-02)
- ▻ Support of standalone invocation from within Java (RQ-TF-02)
- ▻ Support of external Java code invocation from within a transformation (RQ-TF-02)

We used RQ-TF-01 (EMF support) and RQ-TF-04 (Eclipse Tooling) to initially filter potential candidates and selected the following transformation languages with corresponding implementations for further investigations:

- The hybrid (declarative and imperative) transformation language ATL with the tooling from the Eclipse M2M project[2].
- The imperative transformation language QVT Operational Mappings with the tooling from Eclipse M2M project[2].
- The declarative transformation language QVT Relations with the tooling from mediniQVT[3].
- The imperative transformation language Xtend with the tooling from the corresponding Eclipse project[4].

ATL and QVT Relations/Operational Mappings are domain-specific languages (DSLs, see Chapter 4) for defining model transformations. Xtend is a Java-like programming language. It originates from the model-driven community and is used to implement model transformations as well. To evaluate these transformation languages, we implemented two simple transformation examples (in-place and out-place transformation) in each language. The results can be summarized as:

- All Eclipse tools satisfy the requirements except for mediniQVT that does not provide adequate features for transformation validation and debugging.
- All languages support M2M transformations. Xtend additionally supports M2T transformations based on so-called template expressions. Template Expressions can also be used in M2M transformations for string generation. In addition, QVT Relations supports bi-directional transformations.
- All languages support in-place transformations.
- All languages support (slightly different) internal composition features such as inheritance of transformations, package transformations into libraries, or definition of auxiliary functions.

[2] http://www.eclipse.org/m2m/
[3] http://projects.ikv.de/qvt
[4] http://www.eclipse.org/xtend/

- We successfully realized standalone invocation of a transformation from within Java code except for QVT Relations. mediniQVT appears to have too many dependencies with the Eclipse runtime environment, so that standalone invocation becomes impractical.

- Model transformations based on Xtend are mapped to Java code and executed as Java programs. The implementations of ATL, QVT Operational Mappings, and QVT Relations provide transformation execution engines (written in Java).

- As Xtend is Java-like and compiled into Java, external Java code can easily be integrated within a model transformation. ATL provides no comparable mechanism. The QVT specification defines a so-called black-box mechanism to invoke external program code. This feature is supported by the QVT Operational Mappings implementation of Eclipse M2M for Java code invocation. However, it only works if the transformation is executed within an Eclipse runtime environment. The black-box feature cannot be used in standalone invocation. The mediniQVT implementation for QVT Relations does not provide an implementation for the black-box feature.

We found that ATL, QVT Operational Mappings, and Xtend are suitable languages for implementing model transformation. Respective *Transformation Executors* are located in `tf.transformation`. The implementation from mediniQVT for QVT Relations was dropped, mainly because of the shortcomings in the tooling and the tight coupling to the Eclipse runtime environment that hampers standalone invocation from within Java. Furthermore, QVT Relations is a complex, pure declarative language that is rather suited for simpler transformations and to define bi-directional transformations [Huber 2008].

Finally, we made Xtend the transformation language of our choice mainly due to the following reasons:

- It provides the best Java integration.

- The support of template expressions can be used to generate complex strings within M2M transformations.

- ▻ The syntax of Xtend is based on Java and thus easy to learn by Java developers. Existing tooling such as the Java debugger can be reused. We believe that Xtend will be more accepted in our Java-dominated environment than ATL and QVT Operational Mappings. ATL and QVT Operational Mappings introduce a complete new syntax, which requires additional efforts for learning the language.

- ▻ Xtend provides language constructs (see Chapter 4) that are similar to transformation languages such as ATL and QVT Operational Mappings. For example, the *multiple dispatch* mechanism can be used to apply a simple rule matching to dynamically invoke transformation methods. It further provides a *map* operator in order to map from a list of input objects to a list of output objects. Xtend also provides a caching mechanism for method results that facilitates referencing of output objects from previous mappings if needed.

RQ-TF-03 ***Transformation Chain Execution*****:**
Based on this requirement as well as RQ-TF-01 and RQ-TF-04 an optimal transformation chain language covers the following:

- ▻ EMF support (RQ-TF-01)

- ▻ Eclipse Tooling (RQ-TF-04)

- ▻ Support of standalone invocation within Java (RQ-TF-03)

As with RQ-TF-02 we initially used RQ-TF-01 (EMF support) and RQ-TF-04 (Eclipse Tooling) to find potential candidates. We identified the Modeling Workflow Engine 2 (MWE2) in our search as the only suitable technology for transformation chaining, which is part of the Eclipse Modeling Framework Technology (EMFT)[5]. MWE2 is the successor of MWE[6] and provides the definition of a workflow *module* for the sequential invocation of different *components*, which are not limited to but usually Eclipse modeling components. The language used for creating modules is defined by a

[5] http://www.eclipse.org/modeling/emft/

[6] http://wiki.eclipse.org/Modeling_Workflow_Engine_(MWE)

corresponding Ecore model. Xtext uses MWE2, for example, to generate the language infrastructure for an Xtext grammar (see Chapter 4). A corresponding editor for Eclipse is available as well as the standalone invocation of MWE2 modules from within Java is possible. Finally, we utilize MWE2 to define and execute transformation chains. A respective *Transformation Chain Executor* is given in `tf.chaining`.

RQ-TF-04 *Tooling*:
Eclipse-based tooling is available for ATL, QVT Operational Mappings, Xtend and MWE2.

RQ-TF-05 *Extensibility*:
The transformation framework provides a simple plugin mechanism based on the factory design pattern in order to integrate different technologies to execute single model transformations and transformation chains. Respective Java interfaces and classes are located in `tf.transformation` and `tf.chaining`.

13.1.2 Implementation

An overview of the most important classes of the transformation framework is given in Figure 13.1. Configuration options for the transformation framework can be defined in the property file `tf.properties`. A default property file is located in `tf.common`.

tf.common:
This Maven artifact contains the following common Java interfaces and classes:

- `AbstractExecutorResult`: This abstract class represents a generic result from a model transformation or a transformation chain execution based on the attributes `successful`, `statusCode`, `statusMessage`, and `outputModels`. If `successful` is true, the corresponding output models are located in `outputModels`. The values for `statusCode` and `statusMessage`

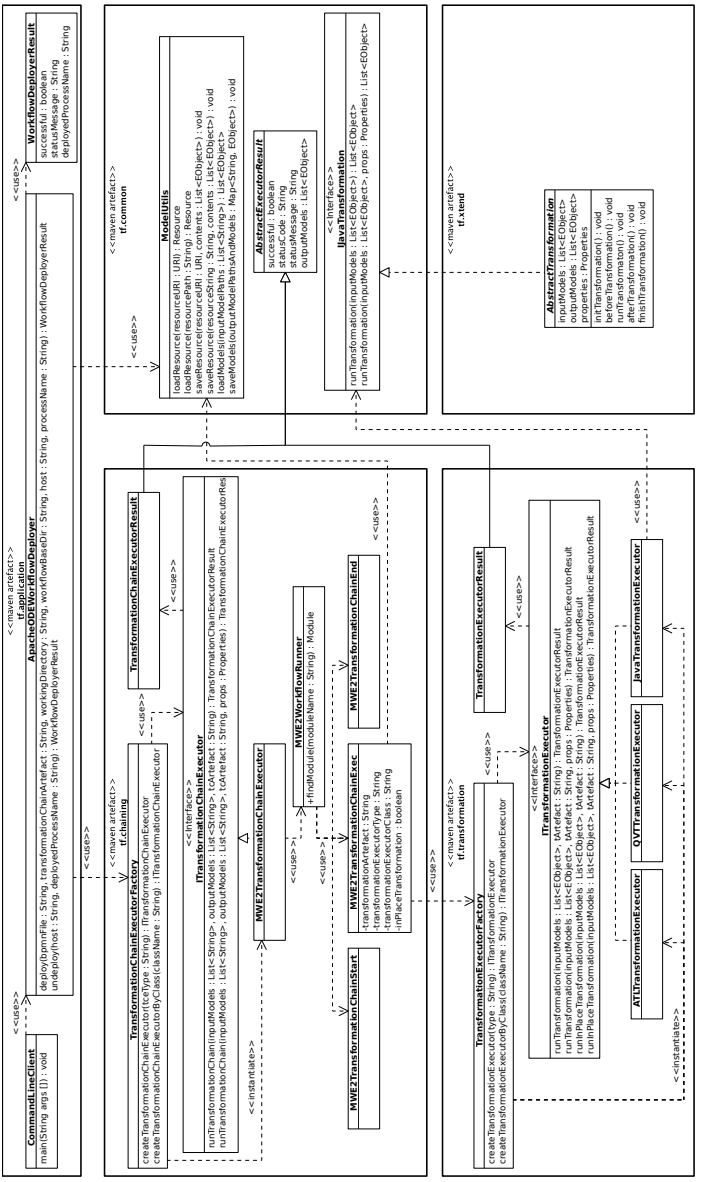

Figure 13.1. Central classes of the transformation framework

depend on the used *Transformation Executor* or *Transformation Chain Executor*. The class is extended by the specific classes `TransformationExecutorResult` (`tf.transformation`) and `TransformationChainExecutorResult` (`tf.chaining`).

- `IJavaTransformation`: This interface must be implemented by transformations written in Java, Xtend or other transformation languages that are compiled to Java. It is used by the class `JavaTransformationExecutor` (`tf.transformation`) in order to execute Java-based transformations. A distinction between in-place and out-place transformations is not necessary, because the creation of new output models or the modification of input models cannot be defined by language elements and is solely controlled by the transformation developer. Thus, the interface only provides the method `runTransformation(...)`. The Xtend class `AbstractTransformation` (`tf.xtend`) extends `IJavaTransformation` in order to provide a specific abstraction for transformations in Xtend.

- `ModelUtils`: This utility class contains different methods to read and write local EMF models. It is used, for example, by the classes `ApacheODEWorkflowDeployer` (`tf.application`) as well as `MWE2TransformationChainExec` (`tf.chaining`).

tf.xtend:
We integrated Xtend as preferred transformation language in a separate Maven artifact. All classes for the BPMN-to-BPEL transformation chain (see Section 13.2) are located here. Xtend provides a Maven plugin that compiles Xtend classes to Java code.

The Xtend class `AbstractTransformation`[7] implements the interface `IJavaTransformation` (`tf.common`). It provides attributes for `inputModels` and (`outputModels` as well as for transformation `properties`. Furthermore, it executes the following sequence of method invocations to organize and extend a transformation in Xtend better:

[7] Please note that Xtend does not provide the concept of abstract classes, but Xtend classes used as abstract Java classes are named accordingly.

1. `initTransformation`: This method should be used to fetch the needed input models and properties from the attributes `inputModels` and `properties`, and to store them in appropriate variables.

2. `beforeTransformation`: This method should be used to prepare the transformation, e.g., to initialize output models.

3. `runTransformation`: This method should be used to execute the actual transformation.

4. `afterTransformation`: This method should be used to complete the transformation, e.g., to create the list of output models.

5. `finishTransformation`: This method should be used to finish the transformation, e.g., to set the attribute `outputModels`. It returns all input objects per default.

tf.transformation:
This Maven artifact contains all Java interfaces and classes for the *Transformation Layer*, which are described below. It provides a simple plugin mechanism based on the factory design pattern in order to support different technologies for the execution of single model transformation within a *Transformation Executor*.

▻ `TransformationExecutorFactory`: The `TransformationExecutorFactory` creates a *Transformation Executor* either via a string representing a predefined type or a Java class name. Types can be configured via the property file `tf.properties` based on the pattern:
`tf.transformationexecutor.<typeName>=<className>`[8]
Currently, the types 'ATL', 'QVTOM', and 'JAVA' are supported.

▻ `ITransformationExecutor`: Each *Transformation Executor* must implement the interface `ITransformationExecutor`. It provides methods to execute an out-place transformation (`runTransformation(...)`) or in-place transformation (`runInPlaceTransformation(...)`).

[8] For example `tf.transformationexecutor.JAVA =net.scherp.tf.transformation.impl.JavaTransformationExecutor`

- `TransformationExecutorResult`: A `TransformationExecutorResult` represents the result of a single model transformation and extends the abstract class `AbstractExecutorResult` (`tf.common`).

The following *Transformation Executor* types are currently provided:

- `ATLTransformationExecutor`: It executes an ATL transformation based on the implementation of Eclipse M2M. The list of input models must be ordered with respect to the order of defined input models of an ATL transformation. The binding of transformation libraries is currently not supported.

- `QVTTransformationExecutor`: It executes an QVT transformation based on the QVT Operational Mappings implementation of Eclipse M2M. The black-box mechanism to invoke standard Java code is currently not supported.

- `JavaTransformationExecutor`: It executes transformations based on Java or transformation languages such as Xtend that are compiled to Java code. Each Java-based transformation must implement the interface `IJavaTransformation` (`tf.common`). For Xtend transformations the class `AbstractTransformation` (`tf.xtend`) can be used, which extends `IJavaTransformation`.

tf.chaining:
This maven artifact contains all Java interfaces and classes for the *Transformation Chain Layer*. It includes *Transformation Chain Executor* for transformation chain execution. Each contained model transformation is executed by a corresponding *Transformation Executor* (`tf.transformation`). A simple plugin mechanism based on the factory design pattern provides the utilization of different technologies for transformation chaining.

- `TransformationChainExecutorFactory`: The `TransformationChainExecutorFactory` creates a *Transformation Executor* either via a string representing a predefined type or a Java class name. Types can be configured via the property file `tf.properties` based on the pattern:

`tf.transformationchainexecutor.<typeName>=<className>`[9]
Only the type 'MWE2' is supported currently.

- `ITransformationChainExecutor`: Each *Transformation Chain Executor* must implement the interface `ITransformationChainExecutor`. It provides methods for transformation chain execution (`runTransformationChain(...)`).

- `TransformationChainExecutorResult`: Represents the result of a transformation chain and extends the abstract class `AbstractExecutorResult` (`tf.common`).

Only MWE2 is supported as *Transformation Chain Executor* currently. A transformation chain in MWE2 is represented by a MWE2 module (see Chapter 4), which defines a sequence of transformation components. Each component of a MWE2 module must extend the abstract class `AbstractWorkflowComponent`. It can provide additional attributes as well whose values can be set from within a MWE2 module. So-called slots allow for passing data between components, whereby each slot has a unique string name. We predefined the following slot names and corresponding data objects:

- 'SLOT_INPUT_MODEL_PATHS' (`List<String>`): A list of paths to load all input models from.

- 'SLOT_OUTPUT_MODEL_PATHS'(`List<String>`): A list of paths to save all output models to.

- 'SLOT_WORKING_DIRECTORY' (`String`): A working directory that is used to store the output models of each transformation step.

- 'SLOT_PROPERTIES' (`Properties`): Java properties that are passed to the used *Transformation Chain Executor*.

- 'SLOT_MODELS' (`List<EObject>`): A list of EMF models used to pass models between two model transformations.

- 'SLOT_COUNTER' (`int`) : A transformation step counter.

[9] For example `tf.transformationchainexecutor.MWE2 =net.scherp.tf.chaining.impl.MWE2TransformationChainExecutor`

The *Transformation Chain Executor* for MWE2 consists of the following classes.

- `MWE2TransformationChainExecutor`: This class implements the interface `ITransformationChainExecutor` and uses the class `MWE2WorkflowRunner` from MWE2 to execute a MWE2 module. Based on the parameter of `runTransformationChain(...)` the slots 'SLOT_INPUT_MODEL_PATHS', 'SLOT_OUTPUT_MODEL_PATHS', 'SLOT_WORKING_DIRECTORY' and 'SLOT_PROPERTIES' are initialized. The value for the slot 'SLOT_WORKING_DIRECTORY' is derived from the property 'PROP_WORKING_DIRECTORY'. Per default the value './tmp' is used.

- `MWE2WorkflowRunner`: This class extends the class `Mwe2Runner` from MWE2 that executes a MWE2 module and overrides the method `findModule(...)` to properly locate an MWE2 module in the local file system or a JAR archive.

- `MWE2TransformationChainStart`: The class `MWE2TransformationChainStart` is a transformation component and extends the abstract class `AbstractWorkflowComponent` of MWE2. It represents the initialization of a transformation chain and is thus used as first component in a corresponding MWE2 module. All input models are loaded during its execution based on the slot 'SLOT_INPUT_MODEL_PATHS' and stored in the slot 'SLOT_MODELS'.

- `MWE2TransformationChainExec`: The class `MWE2TransformationChainExec` is a transformation component and extends the abstract class `AbstractWorkflowComponent` of MWE2. It represents the invocation of a *Transformation Executor* and can thus be used several times after a `MWE2TransformationChainStart` component in a corresponding MWE2 module. The following attributes are provided.

  - `String transformationArtefact`: The artifact that defines the model transformation, e.g. a location of a QVT file or a Java class name.

  - `String transformationExecutorType`: The type of the corresponding *Transformation Executor*.

- ▻ `String transformationExecutorClass`: The Java class name of the corresponding *Transformation Executor*. It is ignored if `transformationExecutorType` is defined.
- ▻ `boolean inPlaceTransformation`: To execute an in-place transformation the value must be `true`. The default value is `false`.

The attribute `transformationExecutorType` or `transformationExecutorClass` is used to create a *Transformation Executor* via the `TransformationExecutorFactory` (`tf.transformation`) and to execute the model transformation. The input models are derived from the slot 'SLOT_MODELS'. When the model transformation has finished, its output models are stored in the slot 'SLOT_MODELS', which overrides the previously used input models. Furthermore, all output models are saved to the working directory that is defined in the slot 'SLOT_WORKING_DIRECTORY' based on its type, e.g. BPMN model or BPEL model. The file names contain the current transformation count of the slot 'SLOT_COUNTER' and the list position of the model, e.g. "TransformationRun-2_-_Model-1.bpel"'. At the end, the slot 'SLOT_COUNTER' is incremented by one.

- ▻ `MWE2TransformationChainEnd`: The class `MWE2TransformationChainEnd` is a transformation component and extends the abstract class `AbstractWorkflowComponent` from MWE2. It represents the finalization of a transformation chain and is thus the last component in a corresponding MWE2 module. All models of the slot 'SLOT_MODELS' are saved during its execution based on the output model paths defined in the slot 'SLOT_OUTPUT_MODEL_PATHS'.

An example MWE2 module that executes a transformation chain with two transformation steps is shown in Listing 13.1. It defines a QVT Operational Mappings transformation (lines 7 to 11) that is followed by a Java transformation based on Xtend (lines 16 to 17). Please note, that input and output models are not defined within the module and passed via the corresponding slots.

Listing 13.1. Example transformation chain as MW2E module

```
1  module tf.mwe2.MWE2Workflow
2
3  Workflow {
4
5    component = net.scherp.tf.chaining.mwe2.MWE2TransformationChainStart {}
6
7    component = net.scherp.tf.chaining.mwe2.MWE2TransformationChainExec {
8      transformationArtefact = "tf/qvt/QVTTransformation.qvto"
9      transformationExecutorType = "QVTOM"
10     inPlaceTransformation = "true"
11   }
12
13   component = net.scherp.tf.chaining.mwe2.MWE2TransformationChainExec {
14     transformationArtefact = "tf.xtend.XtendTransformation"
15     transformationExecutorClass
16       = "net.scherp.tf.transformation.impl.JavaTransformationExecutor"
17   }
18
19   component = net.scherp.tf.chaining.mwe2.MWE2TransformationChainEnd {}
20 }
```

tf.application:
This Maven repository represents the *Application Layer*. It contains the following Java classes:

- `ApacheODEWorkflowDeployer`: This class supports the generation of a BPEL process and its deployment and undeployment by an Apache ODE workflow engine and is designed for the integration in other applications. Its method `deploy` must be invoked with the following parameters and returns a `WorkflowDeployerResult`:
  - `String bpmnFile`: The location of the BPMN file that defines the workflow based on MoDFlow.BPMN (see Chapter 8).
  - `String mweModule`: The name of the MWE2 module that defines the used BPMN-to-BPEL transformation chain.
  - `String workingDirectory`: A working directory in which also the created deployment archive is stored.
  - `String workflowBaseDir`: The directory that contains all used WSDL definitions, XML schemas, and XSLT files.

- ▻ `String host`: The host of the Apache ODE workflow engine.
- ▻ `String processName`: The name of the workflow used for deployment, for example “myworkflow”.

The method `undeploy` must be invoked with the following parameters and returns a `WorkflowDeployerResult`:

- ▻ `String host`: The host of the Apache ODE workflow engine.
- ▻ `String deployedProcessName`: The deployment name of the workflow, which is a combination of the process name and the deployed version, for example “myworkflow-1”.

A `WorkflowDeployerResult` provides the following attributes:

- ▻ `boolean successful`: The value is `true` if a deployment or undeployment was successful, otherwise `false`.
- ▻ `String statusMessage`: The response message of a deployment or undeployment operation, which is provided by Apache ODE in our case.
- ▻ `String deployedProcessName`: The deployment name of the deployed or undeployed workflow, which is in Apache ODE a combination of the process name and the deployed version, for example “myworkflow-1”.

▻ `CommandLineClient`: It provides a command-line interface for the deployment and undeployment of workflows and currently uses the class `ApacheODEWorkflowDeployer`. The first parameter must be either “deploy” or “undeploy”. All succeeding parameters are identical with the corresponding method signatures in `ApacheODEWorkflowDeployer`.

13.2 BPMN-to-BPEL Transformation Chain

The three steps of the BPMN-to-BPEL transformation chain (see Chapter 12) are realized as MWE2 module (see Chapter 4) with corresponding transformation components. It requires a directory path passed by the property

'WORKFLOW_BASE_DIR' in which all WSDL definitions, XML schemas, and XSLT files used are located. There must be one WSDL definition file available in this directory that specifies the interface of the generated BPEL process.

13.2.1 Implementation Decisions

The description of our implementation decisions regarding the BPMN-to-BPEL transformation chain and technologies used is structured according to the requirements defined in Chapter 11.

RQ-B2B-01 ***Ecore Models*****:**
The following Ecore models that are available in the Eclipse context are reused:

- The BPMN 2.0 Ecore model from the Eclipse modeling development tools (MDT)[10].
- The BPEL Ecore model from the Eclipse BPEL project[11].
- The WSDL Ecore model from the Eclipse Web Tools Platform (WTP) project[12].

We created additional Ecore models for the custom BPMN metamodel extensions of MoDFlow.BPMN (see below) and the Apache ODE deployment descriptor. The Ecore model for the Apache ODE deployment descriptor is derived from the corresponding XML schema file (`dd.xsd`) and located in the Maven artifact `tf.common`.

RQ-B2B-02 ***BPMN Subset and Extensions Ecore Model*****:**
The existing BPMN Ecore model is used unmodified for the BPMN subset of MoDFlow.BPMN. We created an additional Ecore model for the custom BPMN metamodel extensions that are defined by MoDFlow.BPMN and

[10] `http://www.eclipse.org/modeling/mdt/?project=bpmn2`
[11] `http://www.eclipse.org/bpel/`
[12] `http://www.eclipse.org/webtools/`

MoDFlow.BPMN2BPEL. It is located in the Maven artifact `tf.common`. An IWM based on MoDFlow.BPMN is validated with the Ecore validation framework. Therefore, we implemented an `IntermediateWorkflowModelValidator` that is located in the Maven artifact `tf.common`.

RQ-B2B-03 ***BPMN-to-BPEL Transformation Steps*****:**
The MWE2 module for the BPMN-to-BPEL transformation chain consists of three sequential model transformations based on Xtend. Each model transformation is represented by a Java transformation component, which is configured to invoke the corresponding Xtend class.

RQ-B2B-04 ***Transformation Framework Utilization*****:**
The BPMN-to-BPEL transformation chain is represented by a corresponding MWE2 module with three Xtend transformations. Besides the argumentation given in Section 13.1, there are further reasons for using Xtend:

- ▻ Utility libraries that are used by many Xtend classes can be provided easily.
- ▻ Referenced WSDL files must be loaded dynamically during a transformation, which can be realized easily in Xtend. A corresponding realization with ATL or QVT Operational Mappings is not possible in that way.
- ▻ The template expressions of Xtend provide a simple way to generate XML literals for BPEL, for example, endpoint references based on WS-Addressing as shown in Listing 13.2.

Listing 13.2. Xtend template expression to generate WS-Addressing literals.

```
1  '''<sref:service-ref>
2      <wsa:EndpointReference>
3          <wsa:Address>«endpoint»</wsa:Address>
4          <wsa:ServiceName xmlns:s="«serviceNamespace»"
5              PortName="«servicePort»"
6          >s:«service»</wsa:ServiceName>
7          <wsa:ReferenceParameters>
8              <wsa:To>«endpoint»</wsa:To>
9              <wsa:Action>«serviceAction»</wsa:Action>
10             «FOR List<String> rP : referenceParameters»
```

```
11                  <«rP.get(0)»:«rP.get(1)»
12                  >«rP.get(2)»</«rP.get(0)»:«rP.get(1)»>
13              «ENDFOR»
14          </wsa:ReferenceParameters>
15      </wsa:EndpointReference>
16  </sref:service_ref>'''.toString
```

RQ-B2B-05 ***Structure-Identification Algorithm*:**
The mapping of BPMN to BPEL is based on a structure-identification algorithm, which was implemented in Xtend within a diploma thesis [Kippscholl 2012]. It is based on the token flow analysis introduced by Götz et al. [2008]. The algorithm returns a hierarchical structure tree for a BPMN process, which can be used for a top-down transformation to BPEL.

Each supported structure is based on a pattern described in the BPEL mapping of the BPMN standard. Currently, the structure-identification algorithm supports the structure pattern described below, see Figure 13.2. Structures that cannot be matched are represented as *Unknown Structure*.

- *Event*: Simple structure that consists of one BPMN event.
- *Activity*: Structure that consists of one BPMN element, which is derived from the BPMN metamodel class `Activity` such as `Task` and `ServiceTask`. Sub-graphs for interrupting and non-interrupting boundary events that never join the main sequence flow are referenced as independent structures.
- *Sequence*: A sub-graph that represents a sequence of BPMN elements or other structures.
- *Activity With Merging Events*: Same as *Activity* but at least one sub-graph for an interrupting boundary event exists that joins the main sequence flow. The corresponding structure comprises a sub-graph from the activity itself until the last BPMN element (usually BPMN exclusive gateway) that joins the sub-graph of an interrupting boundary event.
- *Pick*: A sub-graph from a splitting BPMN event-based gateway to a corresponding joining BPMN exclusive gateway.

- ▻ *Flow*: A sub-graph from a splitting BPMN parallel gateway to a corresponding joining BPMN parallel gateway.
- ▻ *Switch*: A sub-graph from a splitting BPMN exclusive gateway to corresponding joining BPMN exclusive gateway.
- ▻ *Repeat Until Loop*: A sub-graph that represents a repeat-until loop.
- ▻ *While*: A sub-graph that represents a do-while loop.
- ▻ *Repeat While Loop*: A sub-graph that represents a combination of a do-while and a repeat-until loop.

Each structure is represented by an Xtend class that provides methods to access the contained BPMN elements and structures. For example, the class `SequenceStructure` provides the method `getChilds` to get all contained structures. Figure 13.3 shows all Xtend structure classes.

RQ-B2B-06 ***Apache ODE Support*****:**
We defined a standard BPMN-to-BPEL transformation chain in which Apache ODE is supported as default BPEL workflow engine. Unfortunately, a runtime extension of Apache ODE must be supported so that individual WS-Addressing reference parameters can be added to the SOAP header of a request message within a dynamic service invocation.

A WS-Addressing endpoint such as shown in Listing 13.3 is assigned to a BPEL partner link in dynamic service invocations. Individual reference parameters such as `<ResourceID>` (line 8) are usually added to the SOAP header of the corresponding request message. This mechanism is required to invoke WSRF services in GT4 within BPEL.

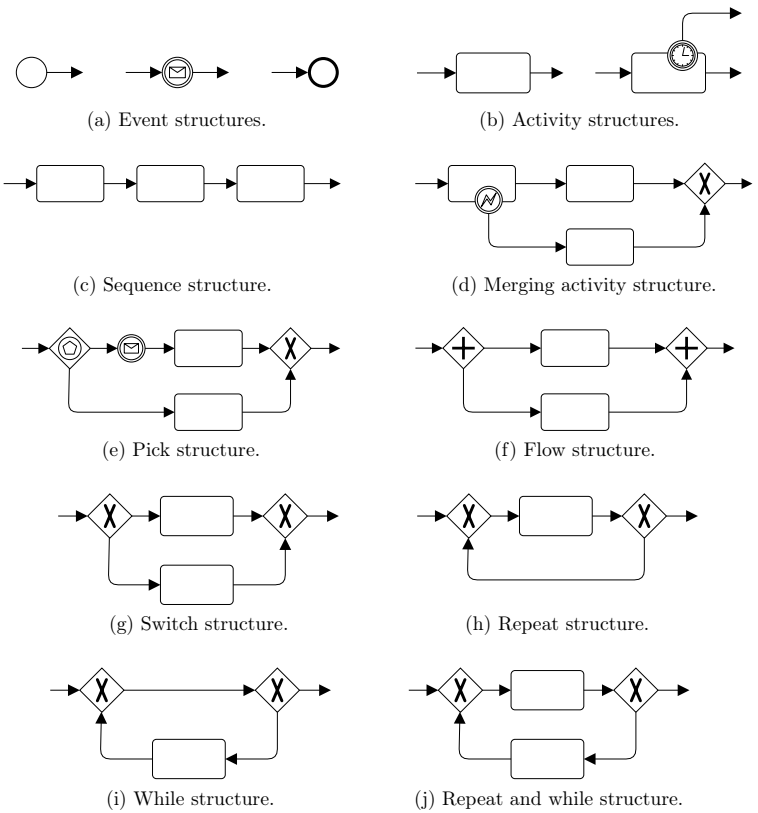

Figure 13.2. Supported structures in BPMN workflows (taken from [Kippscholl 2012])

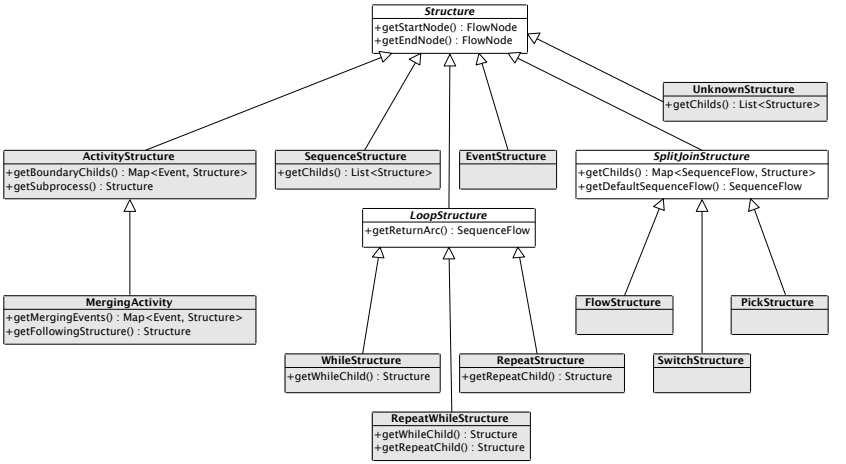

Figure 13.3. Class diagram for structures (taken from [Kippscholl 2012])

Listing 13.3. Individual reference parameter in WS-Addressing endpoints

```
1  <sref:service-ref>
2    <wsa:EndpointReference>
3      <wsa:Address>...</wsa:Address>
4      <wsa:ServiceName xmlns:s="..." PortName="...">s:...</wsa:ServiceName>
5      <wsa:ReferenceParameters>
6        <wsa:To>...</wsa:To>
7        <wsa:Action>...</wsa:Action>
8        <ResourceID>ID</ResourceID>
9      </wsa:ReferenceParameters>
10   </wsa:EndpointReference>
11 </sref:service-ref>
```

This is not supported by Apache ODE. Instead, it provides a runtime extension for BPEL assign operations to add or access individual parameters in a SOAP header, see Listing 13.4. It is based on a `header` extension element for the BPEL elements `from` (line 12) and `to` (line 5). We consequently have to use a BPEL runtime extension of Apache ODE for the dynamic service invocation of WSRF services deployed in GT4. Corresponding transformation code that utilizes this runtime extension is separated in an additional Xtend class.

Listing 13.4. Addition of individual reference parameter to SOAP message header in Apache ODE

```
1  <!-- Add parameter to SOAP header -->
2  <bpel:assign>
3      <bpel:copy>
4          <bpel:from ... >...<bpel:from/>
5          <bpel:to header="ID" variable="requestMessage"/>
6       </bpel:copy>
7  </bpel:assign>
8
9  <!-- Access parameter from SOAP header -->
10 <bpel:assign>
11     <bpel:copy>
12         <bpel:from header="ID" variable="requestMessage"/>
13         <bpel:to ... >...<bpel:to/>
14      </bpel:copy>
15 </bpel:assign>
```

The additional attribute `header` for the BPEL elements `from` and `to` is defined as BPEL extension attribute. The BPEL standard allows adding arbitrary attributes or elements to existing BPEL elements based on the WSDL extensibility mechanism. Therefore, the `javax.wsdl` package provides the abstract interfaces `AttributeExtensible` and `ElementExtensible`, which are both extended via the abstract interface `ExtensibleElement` of the `org.eclipse.wst.wsdl` package. The mechanism of extensibility attributes is currently not supported by the corresponding implementation class `ExtensibleElementImpl` in the used version 1.2 of the `org.eclipse.wsdl.wsdl` package . Thus, we decided to extend the `From` and `To` metamodel classes in the BPEL Ecore model with the `header` attribute. The corresponding extended BPEL library is integrated in the Maven project.

RQ-B2B-07 *Extensibility*:
One main Xtend class exists for each transformation step. It provides an individual internal structure, which can be used for custom extensions. Finally, the following extension mechanisms are provided:

- An existing Xtend class that represents a transformation step can be replaced by an own class.
- An existing Xtend class that represents a transformation step can be extended by utilizing its individual extension mechanism.

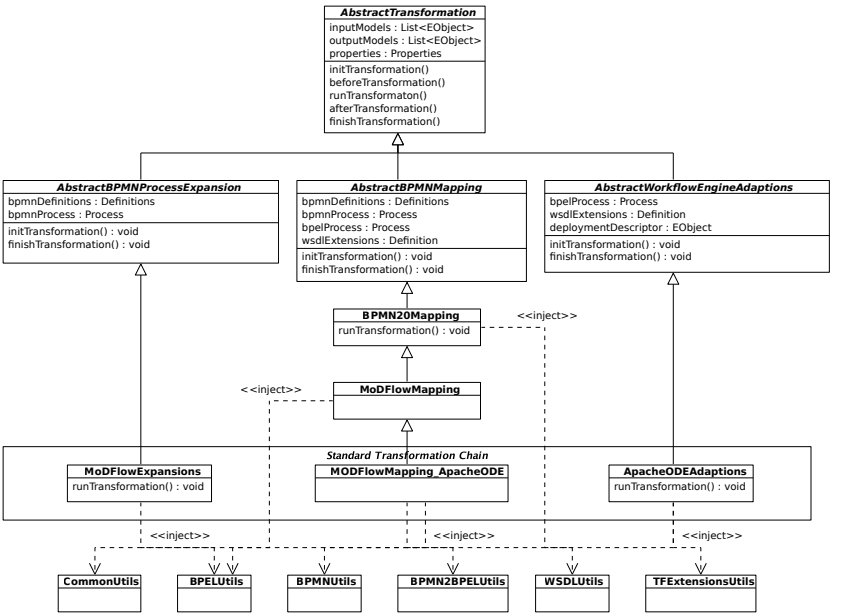

Figure 13.4. Xtend classes for standard BPMN-to-BPEL transformation chain

▻ An additional Xtend class can be implemented to add a new transformation step to the corresponding MWE2 module.

13.2.2 Implementation

All Xtend classes of the BPMN-to-BPEL transformation chain are shown in Figure 13.4. They are located in the Maven artifact `tf.xtend`.

For each transformation step an (abstract) Xtend class is provided that extends the Xtend class `AbstractTransformation` from `tf.xtend` and overrides its methods `initTransformation` and `finishTransformation`. These classes are:

▻ `AbstractBPMNExpansions`: The attributes `bpmnDefinition` (BPMN definition) and `bpmnProcess` (BPMN process) are derived from the BPMN

model in `inputModels` within the method `initTransformation()`. The BPMN process element is contained in the BPMN definition element, but both elements are stored in separate attributes as they are frequently used within the transformation. Within the method `finishTransformation()` the attribute `bpmnDefinition` must contain the expanded BPMN model that is added to `outputModels`.

- `AbstractBPMNMapping`: Within the method `initTransformation()` the attributes `bpmnDefinition` (BPMN definition) and `bpmnProcess` (BPMN process) are derived from the BPMN model in `inputModels` . Within the method `finishTransformation()` the attributes `bpelProcess` and `wsdlExtensions` must contain the created BPEL process model and WSDL definition model, which are added to `outputModels`.

- `AbstractWorkflowEngineAdaptions`: The attributes `bpelProcess` (BPEL process) and `wsdlExtensions` (WSDL definition) are derived from the corresponding models in `inputModels` within the method `initTransformation()`. Within the method `finishTransformation()` the attributes `bpelProcess` and `wsdlExtensions` as well as the attribute `deploymentDescriptor`, which contains the created Deployment Descriptor model, are added to `outputModels`.

Xtend classes that extend one of the (abstract) Xtend classes should override the method `runTransformation()`. They can use different utility Xtend classes such as `BPMNUtils` and `BPELUtils` that provide methods for creating and accessing BPMN and BPEL elements. Utility classes are injected via the dependency injection framework Google Guice[13], so that they can be extended and exchanged easily.

we describe all main Xtend classes that implement a transformation step in the following:

MoDFlowExpansions:
This Xtend class implements the *BPMN Process Expansion* step of the BPMN-to-BPEL transformation chain. It extends `AbstractBPMNExpansions` and overrides the method `runTransformation()`. The transformation step expands

[13] `http://code.google.com/p/google-guice/`

certain elements of a BPMN process model (see Chapter 9). The basic structure of `MoDFlowExpansions` is shown in Listing 13.5.

`MoDFlowExpansions` provides a mechanism to define expansions for all BPMN elements that are derived from the abstract class `FlowNode` of the BPMN metamodel. `FlowNode` is the super class for all BPMN elements that can be used within a BPMN process. The following methods must be implemented to provide an expansion for a BPMN flow node element:

1. A *dispatch method* that invokes the expansion method below:
   `def dispatch void dispatchExpansion(org.eclipse.bpmn2.<Class> element)`

2. An *expansion method*:
   `def org.eclipse.bpmn2.FlowNode expand(org.eclipse.bpmn2.<Class> element)`

Listing 13.5 shows the basic structure of expansions for the BPMN classes `StartEvent` (lines 12 to 22) and `ServiceTask` (lines 25 to 44).

The dispatch method uses the multiple dispatch mechanism of Xtend and is used to invoke the corresponding expansion method only. It has no return value (`void`). Its purpose is to facilitate the invocation of expansion methods for a list of `FlowNode` elements in Xtend. New BPMN element types can be supported by just defining the respective dispatch method.

The expansion method contains the actual expansion implementation. It may further analyze the BPMN element in order to apply different expansions. The return value is a `FlowNode` element, which must represent the root or start element of the created graph structure. Such a structure may consist of one element only.

This mechanism, based on two methods, facilitates flexible extensions. A common scenario to extend an existing expansion is to initially execute the original expansion and then to apply the custom extensions However, the invocation of a dispatch method by an overriding dispatch method via `super.<method>()` causes an infinite loop in the generated Java code. Thus, an expansion method is invoked that is defined as common method without the limitations mentioned above. Extending Xtend classes can always invoke the original expansion methods via `super.<method>()`. A default expansion method always returns the original BPMN element (see lines 49 to 55).

We defined individual *support methods* for certain BPMN elements that provide additional extensibility options. For example, a default error handling (lines 60 to 64) and one specific error handling (lines 65 to 69) for each particular WSDL fault declared within the operations is generated when expanding a BPMN service task. They are referenced by the BPMN service task via boundary BPMN events. Therefore, the corresponding methods `createDefaultErrorHandler(...)` and `createErrorHandler(...)` are invoked, which can be extended individually to provide own error handling mechanisms.

Listing 13.5. Basic structure of `MoDFlowExpansions`

```
1  ....
2  class MoDFlowExpansions extends AbstractBPMNExpansions {
3
4      override void runTransformation() {
5          ...
6          // Run expansion for each flow node of BPMN process
7          bpmnProcess.flowElements.filter(typeof(org.eclipse.bpmn2.FlowNode))
8          .toList.forEach[it.dispatchExpansion]
9      }
10
11     // Expansion for BPMN start events
12     def dispatch void dispatchExpansion(org.eclipse.bpmn2.StartEvent startEvent) {
13         // Invocation of expansion method
14         startEvent.expand()
15     }
16     def org.eclipse.bpmn2.BaseElement expand(org.eclipse.bpmn2.StartEvent startEvent) {
17         // Expansion code for start event
18         ...
19         // Either the start event itself or the root element or start element
20         // of the expanded BPMN structure is returned
21         ...
22     }
23
24     // Expansion for BPMN service tasks
25     def dispatch void dispatchExpansion(org.eclipse.bpmn2.ServiceTask serviceTask) {
26         // Invocation of expansion method
27         serviceTask.expand()
28     }
29     def org.eclipse.bpmn2.BaseElement expand(org.eclipse.bpmn2.ServiceTask serviceTask) {
30         // Expansion code for service task
31         ...
32         // Creation of boundary event handler
33         ...
34         val defaultBoundaryEventErrorHandler = createDefaultErrorHandler(serviceTask)
35         ...
36         serviceTask.operationRef.errorRefs.forEach[
```

```
37              ...
38              val boundaryEventErrorHandler = createErrorHandler(serviceTask, it);
39              ...
40          ]
41          // Either the service task itself or the root element or start element
42          // of the expanded BPMN structure is returned
43          ...
44      }
45
46      ...
47
48      // Default expansions
49      def dispatch void dispatchExpansion(org.eclipse.bpmn2.FlowNode flowNode) {
50          flowNode.expand
51      }
52      def org.eclipse.bpmn2.BaseElement expand(org.eclipse.bpmn2.FlowNode flowNode) {
53          // per default do nothing and just return the original element
54          flowNode
55      }
56
57      ...
58
59      // Support methods
60      def org.eclipse.bpmn2.Activity createDefaultErrorHandler(
61              org.eclipse.bpmn2.ServiceTask serviceTask) {
62          // create default error handling activities
63          ...
64      }
65      def org.eclipse.bpmn2.Activity createErrorHandler(org.eclipse.bpmn2.ServiceTask serviceTask,
66              org.eclipse.bpmn2.Error error) {
67          // create specific error handling activities
68          ...
69      }
70
71      ...
72
73  }
```

One of the following mechanisms may be used to extend `MoDFlowExpansions` by a new inheriting Xtend class, see Listing 13.6:

- ▻ *Mechanism 1 - Definition of an expansion for an unsupported BPMN element type*: A corresponding dispatch and expand method must be implemented.
- ▻ *Mechanism 2 - Replacement of an existing expansion*: The new Xtend class must override the corresponding expand method of `MoDFlowExpansions`.
- ▻ *Mechanism 3 - Extension of an existing expansion*: The new Xtend class

must override the corresponding expand method of `MoDFlowExpansions`. First, the original expansion method of `MoDFlowExpansions` is invoked. Via its return value the corresponding BPMN element or graph structure can be used to apply further expansions.

- *Mechanism 4 - Conditional replacement or extension of an existing expansion*: If the condition is fulfilled mechanism 2 or 3 is applied, otherwise the expand method of the super class is invoked.
- *Mechanism 5: (Conditional) Replacement or Extension of support method*: Mechanism 2, 3 or 4 for expansions may be applied for support methods, too.

Listing 13.6. Mechanisms to extend `MoDFlowExpansions`

```
1  ...
2
3  class MyExpansions extends AbstractBPMNExpansions {
4
5      // Mechanism 1: Support new flow node type.
6      def dispatch void dispatchExpansion(org.eclipse.bpmn2.CallActivity callActivity) {
7          // Invocation of expansion method
8          callActivity.expand()
9      }
10     def org.eclipse.bpmn2.FlowNode expand(org.eclipse.bpmn2.CallActivity callActivity) {
11         // Expansion code for new flow node type
12         ...
13     }
14
15     // Mechanism 2: Replace existing expansion for some cases
16     override org.eclipse.bpmn2.BaseElement expand(org.eclipse.bpmn2.EndEven endEvent) {
17         // New expansion code
18         ...
19     }
20
21     // Mechanism 3: Extend existing expansion
22     override org.eclipse.bpmn2.BaseElement expand(org.eclipse.bpmn2.StartEvent startEvent) {
23         // The invocation of super expand method
24         val flowNode = super.expand(startEvent)
25         //  Additional expansion code
26         ....
27     }
28
29     // Mechanism 4: Conditional application of szenario 2 or 3
30     override org.eclipse.bpmn2.BaseElement expand(org.eclipse.bpmn2.EndEvent endEvent) {
31         if ( <condition> ) {
32             // Application of scenario 2 or 3
```

```
33              ...
34          } else {
35              // Per default call the orginal expansion
36              super.expand(endEvent)
37          }
38      }
39
40      // Mechanism 5: Conditional extension of support method
41      def org.eclipse.bpmn2.Activity createErrorHandler(org.eclipse.bpmn2.ServiceTask serviceTask,
42              org.eclipse.bpmn2.Error error) {
43          if (error.type.equals(...)  ) {
44               // Create specific error handling
45               ...
46          } else {
47              // Per default call super method
48              super.createErrorHandler(serviceTask, error);
49          }
50      }
51
52  }
```

BPMN20Mapping:
This Xtend class implements the *BPMN Mapping* step of the BPMN-to-BPEL transformation chain. It extends the Xtend class `AbstractBPMNExpansions` and overrides the methods `beforeTransformation()` and `runTransformation()`. A BPEL process model and a corresponding WSDL Extensions model are created within the execution of `BPMN20Mapping`. All applied mappings are derived from the BPEL mapping in the BPMN standard.

`BPMN20Mapping` utilizes the implementation of the structure-identification algorithm described in [Kippscholl 2012]. This algorithm analyzes the BPMN process model for known structures based on defined patterns and creates a corresponding structure tree. Each structure is represented by a corresponding Xtend class that provides methods to access all contained BPMN elements and further structures. A top-down transformation to BPEL is executed based on the structure tree, in which BPEL transformations are applied to all structures and single BPMN elements. The basic structure of `BPMN20Mapping` is shown in Listing 13.7.

The mechanism to define transformations is identical to the expansion mechanism in `MoDFlowExpansions`. It is based on the following type of *dispatch methods* and *transformation methods*, which are implemented for

structure and BPMN classes:

```
▷ def dispatch org.eclipse.bpel.model.Activity
  dispatchTransformation(<Class> sequence)

▷ def org.eclipse.bpel.model.Activity
  transform(<Class> sequence)

▷ def dispatch org.eclipse.bpel.model.Activity
  dispatchTransformation(org.eclipse.bpmn2.<Class> element)

▷ def org.eclipse.bpel.model.Activity
  transform(org.eclipse.bpmn2.<Class> element
```

Listing 13.7 shows the basic structure of mappings for the structure class `SequenceStructure` (lines 22 to 33) BPMN classes `StartEvent` (lines 49 to 57) and `ServiceTask` (lines 60 to 67).

A structure is mapped either to a single BPEL element or to a hierarchical BPEL structure with one root element. All BPEL elements that can be used as activity within a BPEL process are derived from the class `Activity` of the BPEL metamodel. Thus, all dispatch and transformation methods return a BPEL activity element. The final BPEL process is the result of the top-down combination of all BPEL transformations. A default transformation maps unknown structures (lines 38 to 45) and unsupported BPEL activity elements (lines 71 to 79) to BPEL empty elements.

`BPMN20Mapping` also contain *support methods*, for example, to map BPMN data objects to BPEL variables (lines 83 to 94). Since BPMN data objects are usually referenced by many BPMN elements, these methods are defined as *cached methods* in Xtend (see Chapter 4), which means the transformation result of a certain BPMN data object to a BPEL variable will be cached based on the first method invocation. A second invocation of the method with the same BPMN data object will return the previously created BPEL variable from an internal cache instead of re-running the transformation code. This prevents the generation of multiple BPEL variables for one BPMN data object.

Listing 13.7. Code snippet of `BPMN20Mapping`

```
...

class BPMN20Mapping extends AbstractBPMNMapping {

    override void beforeTransformation() {
        // Creation of empty BPEL process model and WSDL Extensions model
        ...
    }

    override void runTransformation() {
        // Creation of basic BPEL process model
        ...

        // Structure—identification algorithm
        val structureTree = ...

        // Mapping of structure tree to corresponding BPEL activity structure
        bpelProcess.activity = structureTree.dispatchTransformation
    }

    // Dispatch methods for sequence structures
    def dispatch org.eclipse.bpel.model.Activity dispatchTransformation(
            structureIdentification.structure.SequenceStructure sequence) {
        sequence.transform
    }
    def org.eclipse.bpel.model.Activity transform(
            structureIdentification.structure.SequenceStructure sequence) {
        val bpelSequence = bpelUtils.createSequence()
        sequence.childs.forEach[
            bpelSequence.activities.add(it.dispatchTransformation)
        ]
        bpelSequence
    }

    ...

    // Default mapping for unknown structures
    def dispatch org.eclipse.bpel.model.Activity dispatchTransformation(
            UnknownStructure unknown) {
        unknown.transform
    }
    def org.eclipse.bpel.model.Activity transform(UnknownStructure unknown)
        // Per default an BPEL empty element is created
        ...
    }


    // Dispatch methods for BPMN start events
    def dispatch org.eclipse.bpel.model.Activity dispatchTransformation(
        org.eclipse.bpmn2.StartEvent startEvent) {
```

```
51          startEvent.transform
52      }
53      def org.eclipse.bpel.model.Activity transform(org.eclipse.bpmn2.StartEvent startEvent) {
54          // Transformation code for start event to BPEL receive (message event)
55          // or BPEL empty (otherwise)
56          ...
57      }
58
59      // Dispatch methods for BPMN service tasks
60      def dispatch org.eclipse.bpel.model.Activity dispatchTransformation(
61              org.eclipse.bpmn2.ServiceTask serviceTask) {
62          serviceTask.transform
63      }
64      def org.eclipse.bpel.model.Activity transform(org.eclipse.bpmn2.ServiceTask serviceTask) {
65          // Transformation code for service task to BPEL invoke
66          ...
67      }
68
69                  ....
70
71      // Default mapping for all BPMN activity elements
72      def dispatch org.eclipse.bpel.model.Activity dispatchTransformation(
73              org.eclipse.bpmn2.Activity bpmnActivity) {
74          bpmnActivity.transform
75      }
76      def org.eclipse.bpel.model.Activity transform(org.eclipse.bpmn2.Activity bpmnActivity)
77          // Per default an BPEL empty element is created
78          ...
79      }
80
81
82      // Support methods
83      def create bpelVariable : BPELFactory::eINSTANCE.createVariable
84          createGlobalBPELVariableCached(org.eclipse.bpmn2.DataOject dataObject) {
85          // Add global BPEL variable to BPEL process
86          ...
87      }
88
89      def create bpelVariable : bpelUtils.createVariable
90          createLocalBPELVariableCached(org.eclipse.bpel.model.Scope scope,
91              org.eclipse.bpmn2.DataOject dataObject) {
92          // Add global BPEL variable to BPEL scope
93          ...
94      }
95
96      ...
97
98  }
```

The same scenarios can be applied as presented for `MoDFlowExpansions` to extend `BPMN20Mapping`.

MoDFlowMapping:
This Xtend class extends `BPMN20Mapping`. It contains our extensions for the BPEL mappings of the BPMN standard realized in `BPMN20Mapping`. Each extension for the existing mapping is applied with one of the extension scenarios described for `MoDFlowExpansions`.

The basic structure of `MoDFlowMapping` is shown in Listing 13.8. For example, it overrides the `transform` method for the BPMN class `ServiceTask` (lines 16 to 57). It also defines new *support methods*, for example, to generate BPEL assign operations for dynamic service invocations (lines 61 to 75).

Listing 13.8. Basic structure of `MoDFlowMapping`

```
1  ...
2  class MoDFlowMapping extends BPMN20Mapping {
3
4
5      override void runTransformation() {
6          // Create WSDL Extensions model
7          ...
8          // Invoke BPEL mapping
9          super.runTransformation
10     }
11
12
13     ...
14
15     // Extension of BPMN service task transformation
16     override org.eclipse.bpel.model.Activity transform(
17             org.eclipse.bpmn2.ServiceTask serviceTask) {
18         // Create scope with sequence
19         val scope = bpelUtils.createScope
20         ...
21         // Add BPEL copy operation to sequence in order to initalize request message
22         ...
23         // If service task represents dynamic invocation
24         if (serviceTask.dynamicInvocation) {
25             // Add BPEL copy operations to sequence in order to initialize endpoint variable
26             ...
27             val assigns = createEndpointInitialization(serviceTask, scope, endpointVariable)
28             ...
29         }
30         // Add BPEL copy operations to sequence to copy each input parameter
31         // to the request message or endpoint variable
32         ...
33         // If service task represents dynamic invocation
34         if (serviceTask.dynamicInvocation) {
35             // Add BPEL copy operations to sequence in order to copy endpoint variable
36             // to corresponding partner link
```

```
37              ...
38              val assigns = createEndpointCopyBeforeInvoke(serviceTask, scope, endpointVariable)
39              ...
40          }
41          // Get BPEL invoke element super method
42          val invoke = super.transform(serviceTask) as org.eclipse.bpel.model.Invoke
43          // Modify and add invoke to sequence
44          ...
45          // Add BPEL copy operations to sequence to initialize output parameter variables
46          // from response message
47          ....forEach[
48              ...
49              val variable = createGlobalBPELVariableCached(dataObject)
50              ...
51          ]
52          ...
53         // Add fault handlers to scope
54          ...
55          // Return scope
56          scope
57      }
58
59
60      // Support methods
61      def List<org.eclipse.bpel.model.Assign> createEndpointInitialization(
62              org.eclipse.bpmn2.ServiceTask serviceTask,
63          org.eclipse.bpel.model.Scope scope, org.eclipse.bpel.model.Variable endpointVariable) {
64          // Create BPEL assign operations to initialize endpoint variable
65          // including dynamic invocation parameter
66          ...
67      }
68
69      def List<org.eclipse.bpel.model.Assign> createEndpointCopyBeforeInvoke(
70              org.eclipse.bpmn2.ServiceTask serviceTask,
71          org.eclipse.bpel.model.Scope scope, org.eclipse.bpel.model.Variable endpointVariable) {
72          // Create BPEL assign operations to copy endpoint variable
73          // to the corresponding partner link for the service task
74          ...
75      }
76  }
```

MoDFlowMapping_ApacheODE:
This Xtend class extends `MoDFlowMapping` and overrides a support method for dynamic service invocations so that BPEL assign elements with the header extension for Apache ODE are generated in order to copy individual reference parameter to the SOAP header of a request message.

The basic structure of `MoDFlowMapping_ApacheODE` is shown in Listing 13.9.

For example, it overrides the support method `createEndpointCopyBeforeInvoke` (lines 5 to 12).

Listing 13.9. Basic structure of `MoDFlowMapping_ApacheODE`

```
1  ...
2  class MoDFlowMapping_ApacheODE extends MoDFlowMapping {
3      ...
4
5      override List<org.eclipse.bpel.model.Assign> createEndpointCopyBeforeInvoke(
6              org.eclipse.bpmn2.ServiceTask serviceTask,
7          org.eclipse.bpel.model.Scope scope, org.eclipse.bpel.model.Variable endpointVariable) {
8          val assigns = super.createEndpointCopyBeforeInvoke(serviceTask, scope, endpointVariable)
9          // Add BPEL copy operations to assigns in order to copy dynamic invocation parameter
10         // to request message via the header extension of Apache ODE
11         ...
12     }
13
14 }
```

ApacheODEAdaptions:
The Xtend class *ApacheODEAdaptions* implements the *Workflow Engine Adaptation* step of the BPMN-to-BPEL transformation chain. It extends the Xtend class `AbstractWorkflowEngineAdaptions` and overrides the method `runTransformation()` to create a deployment descriptor for Apache ODE during execution.

The basic structure of `ApacheODEAdaptions` is shown in Listing 13.10. As the generation of an Apache ODE deployment descriptor is quite simple, it provides no special method structure for custom extensions.

Listing 13.10. Basic structure of `ApacheODEAdaptions`

```
1  class ApacheODEAdaptions extends AbstractWorkflowEngineAdaptions {
2
3
4      override void runTransformation() {
5          // Create Apache ODE deployment descriptor
6          ...
7      }
8
9      ...
10
11 }
```

Standard Transformation Chain:
Finally, the standard transformation chain with Apache ODE support consists of the Xtend classes `MoDFlowExpansions`, `MoDFlowMapping_ApacheODE`, and `ApacheODEAdaptions`. The corresponding MWE module is shown in Listing 13.11.

Listing 13.11. MWE2 module for standard transformation chain

```
1  module net.scherp.tf.chaining.mwe.StandardChain
2
3  Workflow {
4
5    component = net.scherp.tf.chaining.mwe2.MWE2TransformationChainStart {}
6
7    component = net.scherp.tf.chaining.mwe2.MWE2TransformationChainExec {
8      transformationArtefact
9        = "net.scherp.tf.transformations.xtend.MoDFlowExpansions"
10     transformationExecutorType = "JAVA"
11   }
12
13   component = net.scherp.tf.chaining.mwe2.MWE2TransformationChainExec {
14     transformationArtefact
15       = "net.scherp.tf.transformations.xtend.MoDFlowMapping_ApacheODE"
16     transformationExecutorType = "JAVA"
17   }
18
19   component = net.scherp.tf.chaining.mwe2.MWE2TransformationChainExec {
20     transformationArtefact
21       = "net.scherp.tf.transformations.xtend.ApacheODEAdaptions"
22     transformationExecutorType = "JAVA"
23   }
24
25   component = net.scherp.tf.chaining.mwe2.MWE2TransformationChainEnd {}
26
27 }
```

Chapter 14

Application Scenarios

This chapter describes how the MoDFlow framework has been used in three different application scenarios. Here, several utilization and extension methods of MoDFlow described in Chapter 10 have been applied.

In Section 14.1, we describe the applied validation approach in software engineering. General preparations to execute scientific workflows in Grid environments are presented in Section 14.2. The three application scenarios are described in Section 14.3, 14.4, and 14.5.

14.1 Validation in Software Engineering

In software engineering, there are different approaches for validating methods and tools. We pursue an approach, in which the following three different types of validation are considered [Eusgeld et al. 2008; Koziolek 2008; Bärisch 2010]:

- *Type I - Feasibility*: This validation type addresses the technical feasibility of a method or tool. In our case, we examine whether the MoDFlow approach can in general be implemented and used to execute scientific workflows. The feasibility has been partly shown by the implementation of the MoDFlow framework (see Chapter 13). We further used the MoDFlow framework for realizing two application scenarios, in which scientific workflows with parameter sweeps are executed in a Grid infrastructure. The corresponding application scenarios are described in Section 14.3 and 14.4.
- *Type II - Practicability*: This validation type focuses on the practicability of a method or tool when applied or used by other users. In our case, we

examine the introduction of MoDFlow in the PubFlow project. Together with the developers, we created a textual DSL called PubFlow.DSL using Xtext, which facilitates the examination of workflow technologies for data publication workflows and the definition of domain-specific workflow activities. Individual positive feedback from the developers let us conclude that the MoDFlow framework was in fact a helpful means. The corresponding application scenario is described in Section 14.5.

▻ *Type III - Cost-Benefit*: This validation type examines the relation between the required costs and the gained benefits, which is typically based on controlled experiments and comparison with related methods and tools. As applying this validation type is generally high in effort, we postpone this study to future work.

14.2 Preparations for Scientific Workflow Execution in Grid Environments

In the following, we describe our preparations to support the execution of scientific workflows in Grid environments (application scenario I and II). This includes extensions for the BIS-Grid workflow engine and the creation of a custom BPMN-to-BPEL transformation chain for Grid-based scientific workflows. We used one Grid site with an installed GT4 middleware for all scientific workflow scenarios.

14.2.1 Workflow Interface for Scientific Workflows

Chapter 8 defines a basic Web service interface for scientific workflows based on the methods `startWorkflow`, `fetchWorkflowState`, and `endWorkflow`. In all scientific workflow examples, however, the start and the end event of the corresponding BPMN process are configured as message events for the operation `startWorkflow`. Thus, the standard BPMN-to-BPEL transformation generates a BPEL process that only provides the (synchronous) method `startWorkflow` to start a workflow instance, in which the method execution is finished with the end of workflow execution. This is implemented

by corresponding BPEL `receive` and `reply` elements. The methods `fetchWorkflowState` and `endWorkflow` are ignored. To support these methods, we defined the following expansions for the Xtend class `MoDFlowExpansions` that are located in the Xtend class `ScientificWorkflowExpansions`:

- The BPMN start event is expanded by a succeeding BPMN task and a BPMN intermediate throw event. The BPMN task represents a workflow activity that sets the current state of a scientific workflow execution to 'Running'[1]. The BPMN intermediate throw event is configured as a message event for the operation `startWorkflow`. BPMN intermediate throw events configured as message events are mapped to BPEL `reply` elements.

- The operation of the BPMN end event is changed to `endWorkflow`. The BPMN end event is further expanded by a preceding BPMN intermediate catch event and a BPMN task. The BPMN intermediate catch event is configured as message event for the operation `endWorkflow`. BPMN intermediate catch events configured as message events are mapped to BPEL `receive` elements. The BPMN task represents a workflow activity that sets the current state of a scientific workflow execution to 'Done'.

- A BPMN event sub-process is added to the BPMN process with a BPMN start event and a succeeding BPMN intermediate throw event that is configured as message event for the operation `fetchWorkflowState`. BPMN even sub-processes are mapped to global BPEL event handler. The BPMN intermediate throw event returns the current state of a scientific workflow execution.

- The error handling creation in the methods `createDefaultErrorHandler` and `createErrorHandler` is replaced by a BPMN task that represents a workflow activity to set the current state of a scientific workflow execution to 'Failed'.

- A BPMN correlation key is generated for the common method parameter `workflowid` and added to the BPMN conversation that is associated to the

[1] A corresponding global variable for the workflow state is created, too.

communication between the workflow client and BPMN process. BPMN correlation keys are mapped to BPEL correlation sets in the BPEL model and corresponding `property` and `propertyAlias` elements in the WSDL Extensions model. Consequently, the created correlation key is used for all BPMN (message) events concerning the communication with the workflow client.

14.2.2 Support for BIS-Grid Workflow Engine

We use the BIS-Grid Workflow Engine for the scientific workflow execution in Grid environments, because it fulfills the respective requirements (RQ_-WF-EN_*) defined in Chapter 7. The BIS-Grid Workflow Engine originally provides an adapter for the open source BPEL workflow engine ActiveBPEL only (see Chapter 5). As we use Apache ODE (version 1.3.5), an additional adapter was implemented for it.

BPEL partner links can be initialized with a `service-ref` element that should be returned when the partner link is read. This is not the case in the Apache ODE workflow engine, which returns only the endpoint URL of the Web service defined in the respective `service-ref` element. Thus, we extended the handling of service endpoints in the source code (Maven artifact `ode-bpel-epr`). The patched library is part of the Maven project for the MoDFlow framework.

Furthermore, the required deployment descriptor for the BIS-Grid workflow engine is generated by the Xtend class `BISGridWorkflowEngine-Adaptions` that extends `AbstractWorkflowEngineAdaptions`. The used security credentials for the invocation UNICORE 6 and Gobus Toolkit 4 services are part of the deployment descriptor.

We additionally provide the Java class `BISGridWorkflowExecutor` that encapsulates the generation, deployment and execution of scientific workflows with the BIS-Grid Workflow Engine.

14.2.3 Support of Globus Toolkit 4 Delegation Service

GT4 provides a delegation service for basic credential management. It allows the delegation of a proxy certificate to a GT4 site that can be used, e.g.,

as credential for succeeding data staging activities during a job execution.

The delegation service is implemented as a (stateful) WSRF service. Each delegated proxy certificate is attached to a WSRF instance, which is identifiable by a unique resource key. This resource key can be used to reference a proxy certificate as credential in a GT4 job description. If the WSRF instance is destroyed, the corresponding proxy certificate is deleted as well.

To use the GT4 delegation service within a scientific workflow, we initially examined the mechanism to delegate a proxy certificate as credential via the GT4 delegation service. Therefore, we executed a job submission with file staging via the GT4 command line tool `globusrun-ws` with enabled debug mode and analyzed the recorded SOAP messages. Furthermore, we inspected the source code of `globusrun-ws`. A proxy certificate was previously generated with the GT4 command line tool `grid-proxy-init` based on a corresponding user certificate. Finally, we identified the following credential delegation mechanism for job submission, see Figure 14.1:

1. The WSRF resource property `delegationFactoryEndpoint` is fetched from the `ManagedJobFactoryService` (GT4 job submission service/WS-GRAM) to get the endpoint for the associated `DelegationFactoryService`.

2. The endpoint for the `DelegationFactoryService` is used to get the WSRF resource property `CertificateChain`. In our case, the host certificate of the GT4 site is returned.

3. The proxy certificate of the user is used to generate a new proxy certificate, which is signed with the public key of the host certificate.

4. The new proxy certificate is delegated to the GT4 site via the `DelegationFactoryService`. It returns the resource key `DelegationKey`, which identifies the corresponding WSRF instance for the delegated proxy certificate. The `DelegationKey` is automatically added to the job description as credential reference, before the job is submitted via the `ManagedJobFactoryService`. A resource key is returned for the WSRF instance that represents the job execution.

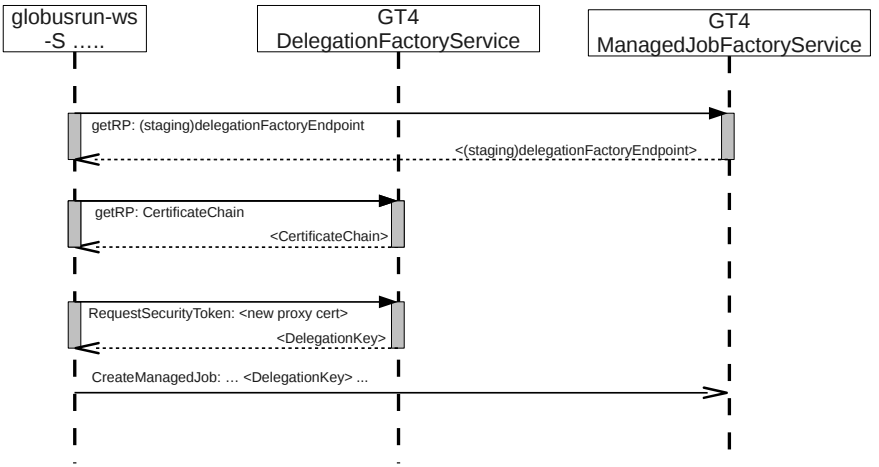

Figure 14.1. Utilization of the GT4 delegation service in a job submission.

5. After the job execution is finished the WSRF instances for the job submission and the delegated proxy certificate are destroyed, which also deletes the delegated proxy certificate.

Each BPEL workflow engine is capable to create and handle the SOAP messages that are exchanged in this scenario. But to the best of our knowledge no existing BPEL workflow engine such as Apache ODE supports the generation of proxy certificates as described above in step three. Thus, we developed a solution in which the proxy generation is transparently supported by the GT4 adapter of the BIS-Grid workflow engine. We assume that a proxy certificate were previously attached to the corresponding WSRF instance of the workflow execution in the BIS-Grid workflow engine (UNICORE/X service container in the UNICORE 6 middleware). Based on the BPEL workflow perspective, we apply the following mechanism to use the GT4 delegation service for job submission [Scherp and Hasselbring 2011]:

1. The BPEL workflow invokes the `ManagedJobFactoryService` to fetch the WSRF resource property `delegationFactoryEndpoint`.

2. The BPEL workflow uses the `delegationFactoryEndpoint` to invoke the corresponding `DelegationFactoryService` in order to fetch the WSRF resource property `CertificateChain` that contains the host certificate.
3. The BPEL workflow invokes the `DelegationFactoryService` with the host certificate as credential.
4. The GT4 handler chain of the BIS-Grid Workflow Engine detects the invocation of a GT4 delegation service. It then generates a new proxy certificate based on the proxy certificate of the WSRF instance and signs it with the public key of the host certificate in the SOAP message. Afterwards, the host certificate in the SOAP message is replaced with the new proxy certificate as credential.
5. The GT4 handler chain uses the modified SOAP message for the invocation of the `DelegationFactoryService` to get the `DelegationKey`, which is returned to the WS-BPEL workflow.
6. The BPEL workflow adds the `DelegationKey` to a job description as credential reference and submits the job to the `ManagedJobFactoryService`, which return a WSRF instance for job execution.
7. The BPEL workflow waits until the job execution is finished or failed, e.g., by periodically fetching the current job execution state from the WSF instance.
8. The BPEL workflow destroys the WSRF instance of the job execution.
9. The BPEL workflow destroys the WSRF instance of the `DelegationKey` to delete the delegated credential.

This approach allows a flexible management of delegated proxy certificates for different Grid sites within a BPEL workflow. The invocation of the GT4 delegation service can be located at any location in the process flow.

14.2.4 Workflow Activity for GT4 Job Submissions

We have defined the additional value “gt4.jobsubmission” for the attribute `activityType` of `ActivityConfiguration`. It is used for BPMN service tasks

(see Chapter 8) that represent a job submission with credential delegation to a GT4 site, which is based on a respective job submission pattern [Gudenkauf et al. 2008, 2009]. Therefore, the following individual configuration parameter (*key* : value) for the corresponding `ActivityConfiguration` are supported to further configure a job submission:

- *server*: Defines the used GT4 host.
- *credentialDelegation*: The value must be either 'true' or 'false'. If 'true', the delegation service of GT4 is invoked before job submission in order to delegate the proxy certificate of the user as credential. Otherwise, the credential delegation is omitted. The default value is 'false'.
- *jobTemplate*: A CDATA element that contains a job definition based on the GT4 job definition syntax. It is used as a template to initialize the request message for the job submission service. The template may be complete so that it could be used directly for job submission. More usually is that input parameter of the corresponding workflow activity are used to complete the job definition, e.g. as arguments for the defined executable in the job definition.
- *resourceID*: Defines the batch system that should be used for the job submission, e.g. PBS. The possible values depend on the used GT4 site.

A sample workflow activity for a GT4 job submission based on the XML syntax of BPMN is schematically shown in Listing 14.1.

Listing 14.1. Definition of Workflow Activity for GT4 Job Submission

```
1  <bpmn2:serviceTask id="submitGT4Job" name="submitGT4Job">
2    <bpmn2:extensionElements>
3      <tf-ext:tfExtensions>
4        <tf-ext:activityConfig>
5          <tf-ext:activityType>gt4.jobsubmission</tf-ext:activityType>
6          <inputParam ... >
7          ...
8          <outputParam ... >
9          ...
10         <tf-ext:invidualConfigParam
11           name="server">srvgrid01.offis.uni-oldenburg.de</tf-ext:invidual...>
12         <tf-ext:invidualConfigParam name="credentialDelegation">true</tf-ext:invidualConfigParam>
13         <tf-ext:invidualConfigParam
```

```
14              name="jobTemplate"><![CDATA[
15              <des:job xmlns:job="http://www.globus.org/..." >
16                ...
17              </des:job>]]>
18            </tf-ext:invidualConfigParam>
19            <tf-ext:invidualConfigParam name="resourceID">PBS</tf-ext:invidualConfigParam>
20          </tf-ext:activityConfig>
21        </tf-ext:tfExtensions>
22      </bpmn2:extensionElements>
23    </bpmn2:serviceTask>
```

During the *BPMN Process Expansion* step, the job submission workflow activity is expanded to a BPMN sub-process that contains all activities required to invoke the job submission service, see Figure 14.2. Note that the delegation activities are used only if the individual configuration property `credentialDelegation` is set to `true`. All workflow activities required for a job submission are represented by BPMN service tasks of the activity type "tf.activity.webservice". Thus, the expansion of a job submission workflow activity only uses existing elements of MoDFlow.BPMN in order to generate corresponding service invocations in BPEL.

All input and output parameter defined for the job submission workflow activity are copied during the expansion to the BPMN service task that represents the actual submission of the job. If a parameter sweep is defined for a job submission workflow activity, it is attached to the BPMN service task mentioned above.

All BPMN expansions associated with scientific workflows are implemented in the Xtend class `ScientificWorkflowExpansions` that extends `MODWEExpansions` (see Chapter 13).

14.2.5 Scientific Workflow Transformation Chain

Based on the descriptions above, the MWE2 transformation chain for the scientific workflow application scenarios consists of the following Xtend classes:

▻ `ScientificWorkflowExpansions`

▻ `MODWEMapping_ApacheODE`

▻ `BISGridWorkflowEngineAdaptions`

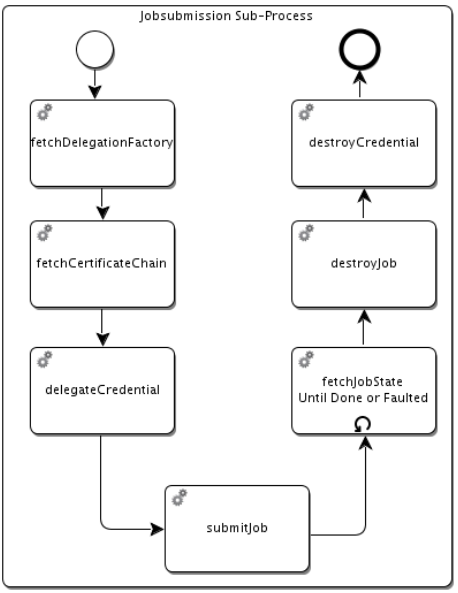

Figure 14.2. Expansion Template for job submission[a]

[a]Created with Yaoqiang BPMN Editor (`http://sourceforge.net/projects/bpmn`)

14.3 Scenario I: Optimization of 3D-Images

The first scenario tests basic parameter sweeps for the GT4 job submission activity. It examines configuration options for the software COSIMA[2] that generates 3D images (anaglyph) based on two input images (left and right). COSIMA was previously installed on a GT4 site and an appropriate wrapper script was provided in order to run the software as a Grid job. The workflow consists of the following two workflow activities, see Figure 14.3:

1. `imageService`: The workflow activity `imageService` invokes an image service that pushes two images (left and right) from corresponding cameras via GridFTP to a GT4 site. The image service returns the GridFTP loca-

[2]`http://www.cosima-3d.de/`

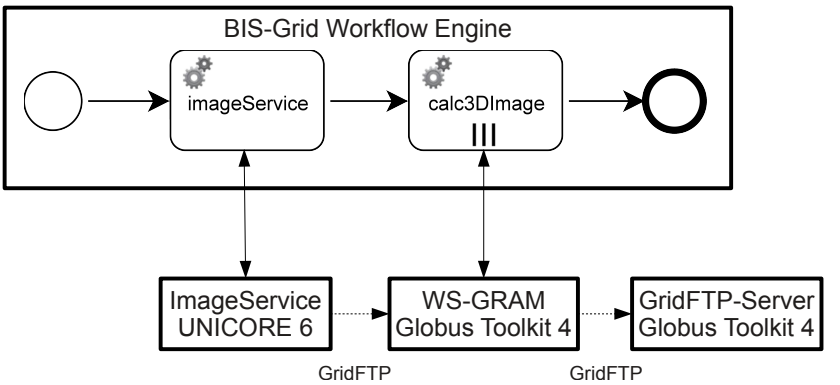

Figure 14.3. Workflow for 3D image creation[a]

[a]BPMN process created with Yaoqiang BPMN Editor (`http://sourceforge.net/projects/bpmn`)

tions and the dates of the two images via the output parameter `leftImage`, `rightImage`, and `date`. The workflow activity is represented by a BPMN service task that is configured as a standard Web service invocation.

2. `calc3DImage`: The workflow activity `calc3DImage` is a GT4 job submission activity that is configured to submit a Grid job to a GT4 site in order to compute a 3D image with COSIMA. It provides the input parameters `leftImage`, `rightImage`, `date`, `AT`, `BN`, and `DN`. The input parameters `leftImage`, `rightImage`, and `date` are data dependencies to the corresponding output parameters of `imageService`. The parameter sweep is defined for `AT`, `BN`, and `DN`, which are configuration options for COSIMA (AT=anaglyph type, BN=brightness, DN=density).

The configuration of the parameter sweep for `calc3DImage` is shown in Listing 14.2. In summary, 18 ($2 * 3 * 3 = 18$) jobs are submitted within two concurrent loop iterations (`loopCardinality`=2). The complete BPMN workflow for this scenario is shown in Appendix B. We implemented a simple Web-based viewer to inspect the results of the parameter sweep.

Listing 14.2. Sweep definition for `calc3DImage`

```
<bpmn2:multiInstanceLoopCharacteristics>
  <bpmn2:extensionElements>
    <tf-ext:tfExtensions>
      <tf-ext:multiInstanceLoopCharacteristicsConfig>
        <tf-ext:sweepParam id="sweep_AT" name="AT"
            type="{http://www.w3.org/2001/XMLSchema}int"
            startValue="4" endValue="5" incrementValue="1" />
        <tf-ext:sweepParam id="sweep_BN" name="BN"
            type="{http://www.w3.org/2001/XMLSchema}string"
            values ="0.8;1.0;1.2" valuesSeparator=";" />
        <tf-ext:sweepParam id="sweep_DN" name="DN"
            type="{http://www.w3.org/2001/XMLSchema}string"
            values ="0.8;1.0;1.2" valuesSeparator=";" />
      </tf-ext:multiInstanceLoopCharacteristicsConfig>
    </tf-ext:tfExtensions>
  </bpmn2:extensionElements>
  <bpmn2:loopCardinality><![CDATA[2]]></bpmn2:loopCardinality>
</bpmn2:multiInstanceLoopCharacteristics>
```

During different executions of the generated BPEL process, non-deterministic errors occurred due to a bug with correlation handling in Apache ODE when the methods `startWorkflow`, `fetchWorkflowState` and `endWorkflow` are invoked. Thus, we decided to use a simple workflow version without the need for correlation, in which only the method `startWorkflow` is invoked and the client does not further interact with the workflow instance. This is realized in the Xtend class `ScientificWorkflowExpansions_Simple`, which is a simplified version of the Xtend class `ScientificWorkflowExpansions`. To avoid code redundancy, `ScientificWorkflowExpansions` extends `ScientificWorkflowExpansions_Simple`.

By using either `ScientificWorkflowExpansions_Simple` or `ScientificWorkflowExpansions` in the corresponding MWE2 transformation chain, we can easily modify the BPEL process generation so that either the simple or the correlation-based invocation method can be applied. Only the workflow client must be configured to support the different invocation mechanisms. The input BPMN workflow model itself is unchanged. To avoid timeouts in the simple version, the workflow client was configured with an extended time limit.

14.4 Scenario II: Fishery Simulation

The second scenario tests larger parameter sweeps and was conducted together with the Working Group for Environmental, Resource and Ecological Economics[3] at the university of Kiel. Within an interdisciplinary cooperation between biologists and economists in the context of the excellence cluster Future Ocean[4], they develop an ecological-economical optimization model (simulation) based on various MATLAB[5] programs in order to examine new concepts of sustainable fisheries management. For some simulation scenarios, the scientists have the need to run parameter sweeps on corresponding MATLAB programs.

One of these MATLAB simulation models optionally applies random variations on two values. To examine the implications of these random variations for the simulation outcome, the scientists have to run the same MATLAB program up to 10,000 times. Each of both random functions can be individually activated or deactivated by a corresponding parameter. We implemented the parameter sweep as a workflow that consists of the following GT4 job submission activity, see Figure 14.4:

1. `runCodSim`: The workflow activity `runCodSim` is based on the GT4 job submission activity that is configured to submit a Grid job to a GT4 site in order to run a MATLAB program. We used the MATLAB compiler toolkit[6] to build packages for a MATLAB program that can be deployed and used on external machines without an installed MATLAB license. Such a MATLAB package was created for the MATLAB simulation program and installed on a Grid site. It can be invoked as a Grid job with the help of a wrapper script. The workflow activity `runCodSim` requires the following four input parameters:

   - `randomBRicker`: If set to '1', the first random function is activated. Otherwise, it is deactivated.

[3] `http://www.eree.uni-kiel.de/en`

[4] `http://www.futureocean.org/english/`

[5] `http://www.mathworks.com`

[6] `http://www.mathworks.de/products/compiler/`

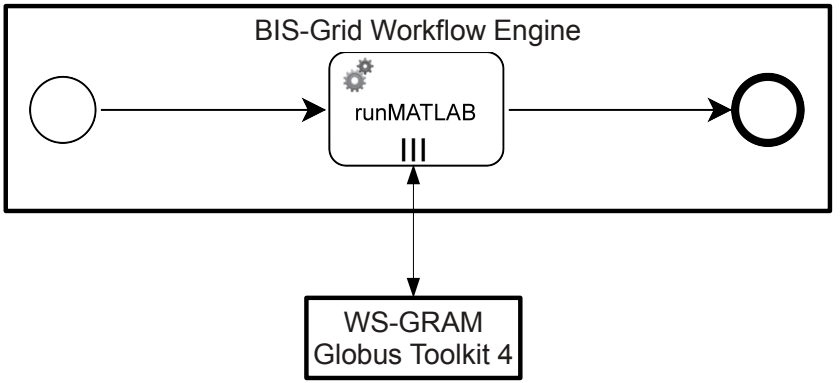

Figure 14.4. Workflow for fishery simulation[a]

[a] BPMN process created with Yaoqiang BPMN Editor (`http://sourceforge.net/projects/bpmn`)

- ▷ `randomW`: If set to '1', the second random function is activated. Otherwise, it is deactivated.
- ▷ `outputdir`: An output directory where output files are stored.
- ▷ `invocationCount`: The number of a simulation run.

Further input parameters are defined with fixed values.

The configuration of the parameter sweep is shown in Listing 14.3. The complete BPMN workflow for this scenario is shown in Appendix C.

Listing 14.3. Sweep definition for `runSensitivity`

```
1  <bpmn2:multiInstanceLoopCharacteristics >
2    <bpmn2:extensionElements>
3      <tf-ext:tfExtensions>
4        <tf-ext:multiInstanceLoopCharacteristicsConfig>
5          <tf-ext:sweepParam id="sweep_invocationCount" name="invocationCount"
6              type="{http://www.w3.org/2001/XMLSchema}int"
7              startValue="1" endValue="10000" incrementValue="1" />
8        </tf-ext:multiInstanceLoopCharacteristicsConfig>
9      </tf-ext:tfExtensions>
10   </bpmn2:extensionElements>
11   <bpmn2:loopCardinality><![CDATA[5]]></bpmn2:loopCardinality>
12 </bpmn2:multiInstanceLoopCharacteristics>
```

We started to execute three scientific workflows, whereby the parameter sweep is executed with five concurrent loop iterations to submit the required GT4 jobs (`loopCardinality`=5). In the first and second run, only one of both random functions were activated respectively with a parameter sweep of 1.000 runs. In the third run, both random functions were activated at the same time with a parameter sweep of 10.000 runs.

Unfortunately, the GT4 middleware is quite resource consuming so that the massive job submission in five concurrent loop iterations to one single GT4 site could not be processed by the machine (Quad-Core, 16 GB Ram). Thus, timeout errors occurred after one minute. To overcome this problem, we reduced the number of concurrent loop iterations to two and modified the BPEL mapping in `MODWEMapping_ApacheODE` so that a configurable BPEL wait operation is executed after each GT4 job submission.

Afterwards, we executed further parameter sweeps with other MATLAB programs and also implemented simple web-based viewers to inspect the results. Finally, each parameter sweep was conducted with the following approach:

- Create a MATLAB package for the MATALB program and deploy it to the Grid resource.

- Provide a wrapper script for the MATLAB program.

- Create a job template for the GT4 job submission workflow activity that invokes the wrapper script.

- Define suitable input parameters for the GT4 job submission activity and mappings to the job template.

- Define a parameter sweep for certain input parameter of the GT4 job submission activity.

Figure 14.5. Data conversion workflow in PubFlow[a]

[a]Created with Yaoqiang BPMN Editor (`http://sourceforge.net/projects/bpmn`)

14.5 Scenario III: Publication Workflows in PubFlow

The PubFlow project[7] aims to develop a platform that provides workflow-based modeling and execution of data publication processes for scientists. Initially, marine sciences are focused, but the plan is to apply the approach in different research domains.

The first application scenario of PubFlow is a data conversion workflow in the context of marine sciences, see Figure 14.5. It transforms data from an institutional data repository into the data format of PANGAEA[8], which is a digital data library system deployed at many world data centers such as WDC-MARE[9]. The workflow consists of three workflow activities, each representing the invocation of one dedicated Web service:

1. `Load Data`: Selected data is loaded from an institutional data repository.
2. `Map Data`: The data is mapped to the PANGAEA data format.
3. `Export Data`: The mapped data is exported to a special data file for its upload to a PANGAEA archive.

In the current stage of the project, the test data sample is small and is thus directly exchanged via SOAP messages with the Web service, which is rather unconventional for scientific workflows (see Chapter 3). However, this solution can be implemented easily and is sufficient for testing. PubFlow

[7] `http://www.pubflow.uni-kiel.de/en`
[8] `http://www.pangaea.de/`
[9] `http://www.wdc-mare.org/`

plans to build a data infrastructure so that big input and output data can be referenced by workflow activities instead of transferring it through the workflow engine.

This scenario serves as an initial test for applying the MoDFlow approach and the MoDFlow framework in PubFlow. Therefore, an external textual DSL called PubFlow.DSL with a mapping to MoDFlow.BPMN was defined with Xtext. PubFlow.DSL can be regarded as a language for defining DWMs. It is designed for developers in order to facilitate the analysis of workflow technologies for data publication processes and to define specific workflow activities. Based on the results, a graphical DSL for MoDFlow.BPMN based on the BPMN notation will be developed that is designed for scientists. However, PubFlow does not intend to create a complete SWfMS such as Kepler.

Xtext can be used to define a textual DSL with a mapping to an existing target language such as Java. The creation of a DSL in Xtext starts with the definition of a new grammar based on the Xtext grammar language, which generally provides rules and literals for defining the concrete syntax of a DSL. For a detailed description of the Xtext grammar language, please refer to the Xtext website[10]. The defined grammar is used to generate a basic language infrastructure, which includes an Ecore model (metamodel) for the abstract syntax and a corresponding Eclipse-based editor.

PubFlow.DSL was designed together with the PubFlow developers and generally defines a compact block-based workflow language (similar to BPEL) for MoDFlow.BPMN with a less verbose syntax. It can be regarded as subset of MoDFlow.BPMN, whereby certain aspects are represented by new elements and attributes that are mapped accordingly to MoDFlow.BPMN. Thus, MoDFlow.BPMN was not extended. All elements of the PubFlow.DSL metamodel and their attributes and model associations are summarized below. Newly defined elements and attributes that have no direct counterpart in MoDFlow.BPMN are additionally explained. For details about MoDFlow.BPMN, please refer to Chapter 8.

▻ **`Process`**: Represents a workflow that is directly mapped to a *Process* element with a `ProcessConfiguration` in MoDFlow.BPMN. The attributes

[10] `http://www.eclipse.org/Xtext/`

`id`, `targetNamespace`, `wsdlLocation`, `portType`, `service`, and `servicePort` of the metamodel classes `Process` and `ProcessConfiguration` of MoDFlow.BPMN are reused. The following additional attributes and model associations are supported:

- ▻ **`workflowBaseDir`** : `string [0..1]`: Defines a local directory in which all WSDL and XML schema files used are located.
- ▻ **`startEvent`** : `StartEvent [1]`: Refers to the `StartEvent` element of the workflow. The attribute is mandatory.
- ▻ **`processElementRoot`** : `ProcessElementRoot [1]`: A reference to the `ProcessElementRoot` element that contains all process flow elements of the workflow. The attribute is mandatory.
- ▻ **`endEvent`** : `EndEvent [1]`: Refers to the `EndEvent` element of the workflow. The attribute is mandatory.

▻ **`ProcessElementRoot`**: Represents the super type for each `ProcessElement` that can be used as a root element for a `Process` or `SubProcess`. It has no attributes and model associations and is not mapped to MoDFlow.BPMN. A `ProcessElementRoot` is either a `Sequence` or a Flow.

▻ **`ProcessElement`**: Represents the super type for all elements that can be used within a `Process`. It has no attributes and model associations, and is not mapped to MoDFlow.BPMN. A `ProcessElement` is a `ServiceTask`, `Task`, `SubProcess`, `Sequence`, `Flow`, or `Switch`.

▻ **`StartEvent`**: Represents a start event that is directly mapped to a `StartEvent` element with an `EventConfiguration` in MoDFlow.BPMN. The attributes and model references `id`, `eventType`, `operation`, and `outputParam` of the metamodel classes *StartEvent* and `EventConfiguration` of MoDFlow.BPMN are reused. It has no further attributes and model associations.

▻ **`EndEvent`**: Represents an end event that is directly mapped to an `EndEvent` element with an `EventConfiguration` in MoDFlow.BPMN. The attributes and model associations `id`, `eventType`, `operation`, `responseMessageContent`,

`responseMessagePart`, `responseMessageKeepSrcElementName`, and `inputParam` of the metamodel classes `EndEvent` and `EventConfiguration` of MoDFlow.BPMN are reused. It has no further attributes and model associations.

- **`Sequence`**: Represents a block-based sequence structure that is mapped to a corresponding graph-based structure in BPMN based on *Sequence Flows*. It reuses the attribute `id` of the metamodel class `Sequence` of MoDFlow.BPMN. The following additional model association is supported:
  - **`processElements`** : `ProcessElement [1..*]`: Defines a list of process elements that are contained in the sequence. The list must contain at least one process element. A process element is represented by `ProcessElement`.
- **`Flow`**: Represents a block-based concurrent structure that is mapped a corresponding split-join construct of *Parallel Gateways* in MoDFlow.BPMN. It reuses the attribute `id` of the metamodel class `ParallelGateway` of MoDFlow.BPMN. The following additional model association is supported:
  - **`processElements`** : `ProcessElement [1..*]`: Defines a list of concurrent process elements that are contained in the `Flow` element. The list must contain at least one process element. A process element is represented by `ProcessElement`.
- **`SubProcess`**: Represents a sub process that is directly mapped to a `SubProcess` element with an `ActivityConfiguration` in MoDFlow.BPMN. The attributes `id` and `activityType` of the metamodel classes `SubProcess` and `ActivityConfiguration` of MoDFlow.BPMN are reused. The following additional model association is supported:
  - **`processElementRoot`** : `ProcessElementRoot [1]`: Refers to the `ProcessElementRoot` element that contains all process flow elements of the sub process. The attribute is mandatory.
- **`Switch`**: Represents a block-based switch structure that is mapped to a corresponding split-join construct of `Exclusive Gateways` in MoDFlow.BPMN.

It reuses the attribute `id` of the metamodel class *ExclusiveGateway* of MoDFlow.BPMN. The following additional model associations are supported:

- ▻ **`case`** : `Case [1..*]`: A list with one or multiple `Case` elements.
- ▻ **`default`** : `Default [0]`: Refers to an optional `Default` element.

▻ **`Case`**: Represents a conditional path for a switch structure that is mapped to a conditional *Sequence Flow* represented by the metamodel class `SequenceFlow` in MoDFlow.BPMN. The following attribute and model association are supported:

- ▻ **`condition`** : `boolean [1]`: Represents a Boolean expression based on XPATH.
- ▻ **`processElement`** : `ProcessElement [1]`: Refers to a contained `ProcessElement`. The attribute is mandatory.

▻ **`Default`**: Represents a default path for a switch structure that is mapped to a default *Sequence Flow* represented by the metamodel class `SequenceFlow` in MoDFlow.BPMN. The following model associations is supported:

- ▻ **`processElement`** : `ProcessElement [1]`: Refers to a contained `ProcessElement`. The attribute is mandatory.

▻ **`ServiceTask`**: Represents a Web service invocation that is directly mapped to a `ServiceTask` element with an `ActivityConfiguration` and `ServiceTaskConfiguration` in MoDFlow.BPMN. The attributes and model associations `id`, `activityType`, `serviceType`, `wsdlLocation`, `portType`, `service`, `servicePort`, `operation`, `requestMessageContent`, `requestMessagePart`, `requestMessageKeepSrcElementName`, `inputParam`, and `outputParam` of `ServiceTask`, `ActivityConfiguration`, and `ServiceTaskConfiguration` of MoDFlow.BPMN are reused. It has no further attributes and model associations.

▻ **`Task`**: Represents a blank workflow activity that is directly mapped to a `Task` element in MoDFlow.BPMN. The attributes and model associations `id`, `activityType`, `inputParam`, and `outputParam` of the metamodel classes `Task` and `ActivityConfiguration` of MoDFlow.BPMN are reused. It has no further attributes and model associations.

- **`InputParameter`**: Represents an input parameter that is directly mapped to an `InputParameter` element in MoDFlow.BPMN. The attributes and model associations `name`, `type`, `collection`, `sourceParamRef`, `sourceParamQuery`, `sourceExpression`, `sourceValue`, `targetPart`, `targetQuery`, `targetExpression`, and `targetKeepSrcElementName` of the metamodel classes `InputParameter` of MoDFlow.BPMN are reused. It has no further attributes and model associations.

- **`OutputParameter`**: Represents an output parameter that is directly mapped to an `OutputParameter` element in MoDFlow.BPMN. The attributes and model associations `id`, `name`, `type`, `collection`, `sourcePart`, `sourceQuery`, and `sourceExpression` of the metamodel classes `OutputParameter` of MoDFlow.BPMN are reused. It has no further attributes and model associations.

An excerpt of the corresponding grammar for the PubFlow.DSL is shown in Listing 14.4. For example, the rule for the `Process` element as well as its attributes and model references is given in lines 2 to 25. The complete grammar is shown in Appendix D.

Listing 14.4. Extract of PubFlow.DSL

```
1  // Rule that defines the model element Process
2  Process :
3      // Literal 'process'
4      'process'
5      // Literal '{'
6      '{'
7          // Defines the optional attribute workflowBaseDir
8          // with corresponding literals.
9          // It calls the STRING keyword rule.
10         ('workflowBaseDir' '=' workflowBaseDir = STRING)?
11         // Defines the mandatory attribute id
12         'id' '=' id = STRING
13         'targetNamespace' '=' targetNamespace = STRING
14         'wsdlLocation' '=' wsdlLocation = STRING
15         ('portType' '=' portType = STRING)?
16         ('service' '=' service = STRING)?
17         ('servicePort' '=' servicePort = STRING)?
18         startEvent = StartEvent
19         // Defines the mandatory attribute processElementRoot
20         // without literals.
21         // If calls the ProcessElementRoot rule.
22         processElementRoot = ProcessElementRoot
```

```
23          endEvent = EndEvent
24      '}'
25  ;
26
27  // Abstract rule that may be represented by the
28  // model elements Sequence or Flow.
29  ProcessElementRoot:
30      Sequence | Flow
31  ;
32
33  ProcessElement:
34      ServiceTask | Task | SubProcess | Sequence | Flow | Switch
35  ;
36
37  Sequence : {Sequence}
38      'sequence'
39      '{'
40          'id' '=' id = STRING
41          // Defines the list attribute processElements
42          // with cardinality [1..*].
43          processElements += ProcessElement+
44      '}'
45  ;
46
47  StartEvent :
48      'startEvent'
49      '{'
50          'id' '=' id = STRING
51          'eventType' '=' eventType = STRING
52          'operation' '=' operation = STRING
53          // Defines the list attribute outputParam
54          // with cardinality [0..*].
55          outputParam += OutputParameter*
56      '}'
57  ;
58
59
60  // Keyword rule to define Boolean literals
61  terminal BOOLEAN returns ecore::EBoolean:
62      'true' | 'false' | 'yes' | 'no'
63  ;
64
65
66  ...
```

After the definition of the DSL grammar for PubFlow.DSL, the language infrastructure can be generated. The generated editor can be used immediately for workflow creation. It also provides basic validation and content assist features. A basic code example based on the PubFlow.DSL is presented in Listing 14.5. For example, it contains the required start event

(lines 7 to 13) and end event (lines 32 to 44) as well as one Web service invocation (lines 18 to 28) within a sequence (lines 15 to 30).

Listing 14.5. Simple code example of PubFlow.DSL

```
1  process {
2      workflowBaseDir = "/tmp"
3      id = "Process"
4      targetNamespace = "process.org"
5      wsdlLocation = "workflow.wsdl"
6
7    startEvent {
8        id = "start"
9        eventType = "tf.event.message"
10       operation = "process"
11       outputParam { id="start_out" name="request" type="{...}..."
12         sourcePart="request" }
13   }
14
15   sequence {
16       id = "workflowSequence"
17
18     serviceTask {
19         id = "service"
20         activityType = "tf.activity.webservice"
21         serviceType = "service.sample"
22         wsdlLocation = "service.wsdl"
23         operation = "method"
24         inputParam { name="request" type="{...}..."
25           sourceParamRef="start_out" targetPart="request" }
26         outputParam { id="service_out" name="response" type="{...}..."
27           sourcePart="response" }
28     }
29
30   }
31
32   endEvent {
33       id = "end"
34       eventType = "tf.event.message"
35       operation = "process"
36       responseMessageContent
37         = "<![CDATA[<tns:OCN_Bottle_FlowResponse xmlns:tns=\"service.org\" >
38         <tns:result>tns:result</tns:result>
39         </tns:OCN_Bottle_FlowResponse>]]>"
40       responseMessagePart = "response"
41
42       inputParam { name="response" type="{...}..."
43         sourceParamRef="service_out" targetPart="response" }
44   }
45 }
```

The generated language infrastructure also provides different means for custom extensions, which can be implemented in predefined Java and Xtend classes. We implemented the following classes for PubFlow.DSL:

- `PubFlowDSLJavaValidator`: This Java class is used to validate the Ecore model of the PubFlow.DSL whenever a workflow model is changed in the editor. For example, the uniqueness of all `id` attributes is checked. Major parts of the validation are implemented in the supplemental Xtend class `PubFlowDSLJavaValidatorUtil`.
- `PubFlowDSLProposalProvider`: This Java class is used to provide further content assist features for the editor. For example, the directory specified in the attribute `workflowBaseDir` is searched for WSDL files that are proposed when using content assist for editing a `wsdlLocation` attribute. Major parts of the content assist features are implemented in the supplemental Xtend class `PubFlowDSLProposalProviderUtil`.
- `PubFlowDSLGenerator`: This Xtend class is used to transform an EMF model based on the PubFlow.DSL to the target language. In our case, it implements a M2M transformation to BPMN. This class is invoked whenever a PubFlow.DSL model is saved in the workflow editor.

A screenshot of the corresponding editor is shown in Figure 14.6. The generated BPMN models are located in the `src-gen` folder, which can be transformed further to a BPEL process with the BPMN-to-BPEL transformation chain of the MoDFlow framework. Subsequently, the BPEL process can be deployed to an Apache ODE workflow engine.

The PubFlow.DSL editor is used to implement the example data conversion workflow shown in Figure 14.5. Listing 14.6 shows an extract of the data conversion workflow. For example, it contains a sequence (lines 21 to 50) with the three Web service invocations of the data conversion workflow. The full workflow definition is shown in Appendix E.

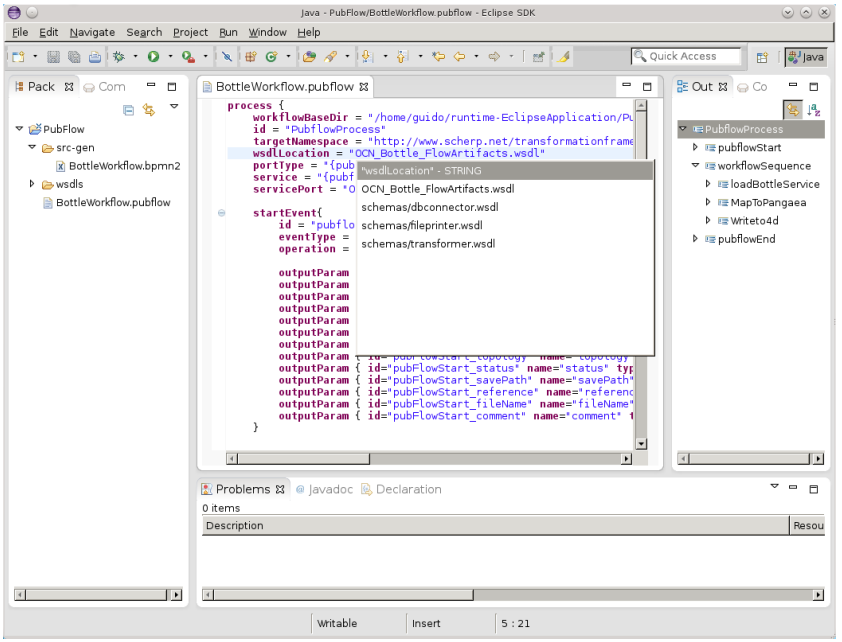

Figure 14.6. PubFlow editor

Listing 14.6. Extract of data conversion workflow definition with PubFlow.DSL

```
process {
  workflowBaseDir = "/home/guido/runtime-EclipseApplication/PubFlow/wsdls"
  id = "PubflowProcess"
  targetNamespace = "http://www.scherp.net/transformationframework/pubflow"
  wsdlLocation = "OCN_Bottle_FlowArtifacts.wsdl"
  portType = "{pubflow.de}OCN_Bottle_Flow"
  service = "{pubflow.de}OCN_Bottle_Flow"
  servicePort = "OCN_Bottle_FlowPort"

  startEvent{
    id = "pubflowStart"
    eventType = "tf.event.message"
    operation = "process"

    outputParam { id="pubFlowStart_input" name="input"
      type="{http://www.w3.org/2001/XMLSchema}int"
      sourcePart="payload" sourceQuery="*[local-name() = 'input']" }
```

```
18      ...
19    }
20
21    sequence {
22        id = "workflowSequence"
23
24      serviceTask {
25        id = "loadBottleService"
26        activityType = "tf.activity.webservice"
27        serviceType = "pubflow.services.LoadBottles"
28        wsdlLocation = "workflowbasefiles/schemas/dbconnector.wsdl"
29        portType = "{http://.../}DBConnector"
30        service = "{http://.../}DBConnectorImplService"
31        servicePort = "DBConnectorImplPort"
32        operation = "getData"
33
34        inputParam { name="id" type="{http://www.w3.org/2001/XMLSchema}int"
35          sourceParamRef="pubFlowStart_input" targetPart="id" }
36
37        ...
38      }
39
40      serviceTask {
41        id = "MapToPangaea"
42        ...
43      }
44
45      serviceTask {
46        id = "Writeto4d"
47        ...
48      }
49
50    }
51
52    endEvent {
53      id = "pubflowEnd"
54      ...
55    }
56  }
```

Currently, PubFlow used the standard BPMN-to-BPEL transformation chain. An expansion for a more specific error handling within the invoked Web services is planned for future work, whereby the issue tracking system Jira[11] will be integrated. Therefore, the Xtend class `MODWEExpansions` is to be extended to override the methods `createErrorHandler` and `createDefaultErrorHandler`.

[11] `http://www.atlassian.com/de/software/jira/overview`

As the PubFlow infrastructure is based on Java, the involved developers have learned the Java-like Xtend language very quickly. Besides the transformation implementation, Xtend is also used for other aspects such as model validation and content assist in the editor (see above).

14.6 Discussion and Threats to Validity

In the following, we discuss the results of our evaluation and certain threats to their validity.

The application scenarios I and II have shown that the MoDFlow framework can be used to generate executable scientific workflows for Service Grids. Regarding the generation of executable BPEL code only the *BPMN Process Expansion* step was extended in order to provide an expansion for the GT4 job submission workflow activity and to extend the communication of a client with a workflow instance. All extensions to the standard BPMN-to-BPEL transformation chain are therefore realized with utilization and extension methods provided by the MoDFlow framework. The actual BPEL code generation is completely covered by the standard *BPMN Mapping* step. However, the major efforts to run the application scenarios were caused by the execution infrastructure. We extended the *BPMN Mapping* step to use a BPEL runtime extension of Apache ODE for adding WS-Addressing reference parameters to SOAP headers. We implemented the generation of a BIS-Grid Workflow Engine deployment descriptor for the *Workflow Engine Adaptation* step. We extended the BIS-Grid Workflow Engine to support the invocation of the delegation service of GT4 within a BPEL process. We applied a simple approach to integrate the execution of MATLAB programs in scientific workflows. It should be checked, if this experience can be used to reduce such efforts in further application scenarios.

With the application scenarios I and II, we have further shown that the MoDFlow framework can be used to run parameter sweeps in BPEL. However, the performance capabilities of our setup are limited so that scalability problems have occurred as a consequence. As scalability is not addressed in our approach, these issues provide opportunities for future work.

Application scenario III has shown that the MoDFlow framework can be applied in the PubFlow project. Therefore, the PubFlow.DSL was developed with Xtext that supports the creation of DWMs by developers and its mapping to MoDFlow.BPMN. It was used to realize a sample data conversion workflow needed to examine workflow technologies for publication workflows and to define required workflow activities. The definition of the Xtext grammar for PubFlow.DSL was simple so that a basic language infrastructure could be provided in short time. Thereby, the introduction of Xtext has the positive side effect that the PubFlow developers also learned the Xtend language, e.g., to realize the mapping to MoDFlow.BPMN. This experience is helpful when the standard BPMN-to-BPEL transformation chain has to be customized. However, the capability to use BPMN as graphical notation for designing data publication workflows by scientists has to be shown within the upcoming working packages of PubFlow.

As the PubFlow infrastructure is based on Java, the effort to learn the Java-like syntax of Xtend was quite low. The PubFlow developers are convinced of Xtend so that the language is also used for other purposes, e.g. for validation and content assist in the workflow editor. We therefore conclude that the decision to use Xtend as a transformation language in the MoDFlow framework was good. Other transformation languages such as ATL and QVL Operational Mappings are not completely disregarded as the MoDFlow framework provides the execution of single model transformations with these languages as well. However, the suitability of additional transformation languages compared with Xtend for implementing the MoDFlow.BPMN2BPEL mapping was not examined. We argue that the effort for the introduction of these languages in PubFlow are larger in contrast to Xtend, because the developers would have to learn a complete language.

Chapter 15

Related Work

In this chapter, we discuss related work regarding the utilization of BPEL for Grid and scientific workflows in Section 15.1. The mapping of BPMN to BPEL is considered in Section 15.2 and transformation chaining technologies in Section 15.3.

15.1 Utilization of Business Workflow Technologies for Grid and Scientific Workflows

The suitability of BPEL (WS-BPEL [OASIS 2007] or its predecessor BPEL4WS [BEA et al. 2002, 2003]) for the execution of scientific workflows in service-oriented environments such as Grid infrastructures has been discussed and shown extensively in many publications and projects.

Some approaches defined BPEL extensions, which can be distinguished between *design time only extensions*, *design and runtime extensions*, and *runtime only extensions* [Kopp et al. 2011], or modified the BPEL standard itself. However, the definition of BPEL runtime extensions always imply the modification of an existing BPEL workflow engine. We focus on standard BPEL elements for the technical execution of scientific workflows. Complex BPEL process constructs, which are only required for process execution and not relevant for workflow modeling, are automatically generated within model transformations. All scientific workflow-specific extensions are defined as design time only extensions for BPMN that are mapped to standard BPEL.

We use the Apache ODE workflow engine for BPEL process execution. Regarding the execution of BPEL processes in Grid environments, we additionally use the BIS-Grid Workflow Engine [Gudenkauf et al. 2010a]

as transparent grid proxy that support the required security mechanism of Globus Toolkit 4 and UNICORE 6. To the best of our knowledge, no comparable solution exists to our implementation in the BIS-Grid Workflow Engine to invoke the GT4 delegation service within WS-BPEL.

We currently do not know any SWfMS that utilizes BPMN, e.g., for workflow modeling. However, the graphical representation of BPMN is often used to sketch the process flow of scientific workflows in scientific publications [Yildiz et al. 2009; Sonntag et al. 2010]. The SWfMS Trident[1] from Microsoft is another approach, in which existing business workflow technology is used for scientific workflows in the domain of oceanography. Trident is based on the Windows Workflow Foundation (WWF)[2].

In the following, we discuss existing approaches to use BPEL for Grid and scientific workflow execution in more detail.

Wang et al. [2005] used major concepts of BPEL4WS to create a new workflow language called Grid Process Execution Language (GPEL) that provides specific extensions regarding the execution of Grid workflows. That includes, for example, the support of WSRF as well as the new activity `mInvoke` that represents high computational tasks. GPEL can be executed with the CROWN FlowEngine [Zeng et al. 2007].

Emmerich et al. [2005] successfully conducted experiments in which standard BPEL4WS and the ActiveBPEL workflow engine were used for scientific workflow execution. Grid jobs were submitted via the GridSAM job submission service[3]. Its invocation was encapsulated by an additional BPEL workflow that is invoked by the main scientific workflow. In subsequent publications [Chapman et al. 2006a,b], hierarchical BPEL patterns are motivated and presented in order to encapsulate basic scientific workflow activities such as job execution and data retrieval. Our job submission pattern is inspired by this work. With Sedna [Wassermann et al. 2007], a graphical scientific workflow editor was implemented that provides high-level abstraction of BPEL. Several design-time-only extensions further provide the creation of hierarchically workflows, the inclusion of BPEL code fragments as macros, and the integration of plug-ins that generate BPEL code with a

[1] `http://research.microsoft.com/en-us/projects/trident/`

[2] `http://msdn.microsoft.com/en-us/library/dd489410.aspx`

[3] `http://gridsam.sourceforge.net`

common programming language such as Java. Meanwhile, the implementation has been migrated from BPEL4WS to WS-BPEL. The work was part of the OMII-BPEL project[4] in the context of the OMII-Europe initiative[5].

Leymann [2006] examines the WSRF support by BPEL so that BPEL processes can be used like WSRF services. For example, the support of the implied resource pattern of stateful WSRF services as well as of the specifications for resource lifetime and resource properties is discussed. Solutions are sketched that can be realized with standard BPEL elements. The author encourages the automatic generation of certain complex BPEL constructs, which is also a central aspect in our approach. It is further considered to utilize the BPEL extension mechanism to define standardized Grid/WSRF extensions for BPEL, e.g., to support monitoring.

Ezenwoye et al. [2007a,b] examined the invocation of WSRF services within WS-BPEL. They defined patterns in which standard BPEL elements are used to create and access WSRF service instances. Experiments were conducted with the ActiveBPEL workflow engine and Globus Toolkit 4. We used and extended these patterns for the development of the BIS-Grid Workflow Engine and for our job submission pattern [Gudenkauf et al. 2008, 2009] (see Chapter 14).

Tan et al. [2007] presents a concept called BPEL4Job in which different fault-handling policies can be applied for BPEL-based job executions in distributed computing environments. Initially, a basic BPEL process is defined in which a job execution is represented as a single BPEL invoke activity. This activity is expanded based on a fault-handling policy, e.g., to apply a certain retry strategy with a maximum number of retries, which is defined with WS-Policy [W3C 2007]. The actual job execution is encapsulated by a corresponding job proxy service. Workflow execution is based on the IBM Websphere Process Server[6].

Dörnemann et al. [2007] extended BPEL4WS with the runtime extension elements `gridCreateResourceInvoke`, `gridInvoke`, and `gridDestroyResourceInvoke` that are used for the invocation of WSRF services deployed in Globus Toolkit 4. The authors extended the ActiveBPEL workflow engine to support

[4] `http://www.omii.ac.uk/wiki/BPEL`

[5] `http://omii-europe.omii.ac.uk/`

[6] `http://www-01.ibm.com/software/integration/wps/`

these runtime extensions, which includes the support of the Grid Security Infrastructure (GSI) [Globus Security Team 2005] for the invocation of secured Globus Toolkit 4 services [Dörnemann et al. 2008]. The delegation of proxy certificates is not supported. We adopted the mechanism to invoke secured Globus Toolkit 4 services in a corresponding adapter of the BIS-Grid Workflow Engine [Gudenkauf et al. 2010b].

Görlach et al. [2011] comprehensively discuss the utilization of conventional workflow technology to create a SWfMS for simulation workflows. This includes the utilization of WS-BPEL for workflow execution as well as for workflow modeling. The authors defined main requirements for a SWfMS. As WS-BPEL does not match all of these requirements, the utilization of different design time and/or runtime extensions for WS-BPEL are motivated such as BPEL-D [Khalaf 2008] as the basis for the definition of data dependencies in scientific workflows. The authors also propose the architecture of a corresponding SWfMS that uses the Apache ODE workflow engine. This work is part of the excellence cluster Simulation Technology (SimTech)[7]. Görlach et al. [2011] motivated the utilization of BPEL extensions such as BPEL-D for scientific workflows, which may also be adopted for BPMN. A good overview and classification of existing BPEL extensions is presented by Kopp et al. [2011].

15.2 Mapping of BPMN to BPEL

Stein et al. [2009] distinguish the following approaches to implement a transformation of business process models into executable orchestrations:

- *Control Flow Centred Approaches*: Transformation between control flow-centric workflow languages such as BPMN to BPEL.
- *Approaches Based on Domain-Specific Language Extensions*: Definition of domain-specific languages extensions for process modeling, e.g., for UML activity diagrams that are mapped to BPEL.
- *Framework-Based Approaches*: Environments to implement such transformations.

[7] http://www.simtech.uni-stuttgart.de/

All approaches may be combined, e.g., control flow-centric approaches may provide the foundation to build a framework.

We classify our approach as a *model-driven framework* (MoDFlow) for a BPMN-to-BPEL mapping that defines *custom language extensions for BPMN* (MoDFlow.BPMN) and that *extends an existing control flow-based approach* (MoDFlow.BPMN2BPEL) to map BPMN models to executable BPEL.

In the following, we discuss existing approaches with regard to our work according to the classification by Stein et al. [2009].

Control Flow Centred Approaches:

Mendling et al. [2006, 2008] examined and classified existing control flow-centric strategies to transform graph-based to block-based workflow languages and vice versa. BPEL is used as reference language for block-based workflow languages including the capability of the BPEL `flow` element to define acyclic graph-based control flow with links. Graph-based to block-based (BPEL) transformation strategies are classified as *element-preservation*, *element-minimization*, *structure-identification*, *structure-maximization*, and *event-condition-action-rules* (see Chapter 2). These approaches can be applied to certain workflow languages, e.g., to map BPMN to BPEL.

We applied a structure-identification strategy (see Chapter 13) for implementing a BPMN-to-BPEL mapping. Such a strategy usually consists of an algorithm to decompose the control flow of a graph into components or SESE regions (single entry single exit) and patterns to map the identified components to a target language. Algorithms for identifying components in control flow structures are often generic and independent of a certain workflow language, whereas patterns and mappings are usually defined for specific workflow languages such as BPMN and BPEL. All approaches for mapping BPMN to BPEL support a subset of BPMN only. For example, arbitrary/unstructured loops in graphs with multiple entries and/or multiple exits are often not supported, as there exists no direct mapping to BPEL.

In the following, we discuss existing algorithms and pattern-based approaches with regard to our implementation.

Algorithms to identify components/structures:

Eshuis et al. [2006] presents an algorithm to calculate *structured compositions* for dependency graphs. The algorithm requires that each split node has a corresponding join node of the same type, for example XOR-split and XOR-join. Furthermore, arbitrary loops are explicitly excluded. Due to the high effort to extend this approach for our purpose (see [Kippscholl 2012]), this algorithm was not used in our implementation.

Götz et al. [2008] defined a token analysis algorithm to find components in arbitrary graphs. The idea is to propagate tokens through a graph beginning from the start up to the end node, and to label all arcs with tokens. This mechanism is based on a formalization of the informally defined token propagation in the BPMN 1.0 standard [OMG 2006a]. It requires an initial detection of so-called strong connected components such as loops, e.g., based on the algorithm described in [Tarjan 1972] or [Thomas H Cormen 1994]. The tokens of the labeled arcs are used to decompose the graph into components. The algorithms of Götz et al. [2008] and Tarjan [1972] has been extended and implemented with Xtend within a diploma thesis [Kippscholl 2012] to find and identify components in BPMN workflows. All components are identified with patterns of the BPEL mapping in the BPMN standard.

Vanhatalo et al. [2009] described a so-called *refined process structure tree* that represents a hierarchical decomposition of a biconnected graph into sub-graphs (components). It is a refinement of the concept of *program structure trees* defined in [Johnson et al. 1994]. The algorithm is based on the calculation of so-called *triconnected components* based on the algorithm defined in [Hopcroft and Tarjan 1974]. These triconnected components are used to identify components and to construct a process structure tree. The approach postulates determinism so that two calculations for the same graph always produce the exact same process structure tree. Furthermore, local changes in the graph must only cause local changes in the process structure tree. Due to the high complexity of this approach (see [Kippscholl 2012]), this algorithm was not used in our implementation.

Pattern-based approaches:

Zhao et al. [2006] present an approach to transform so-called unstructured loops (arbitrary loops) in graph-based process models to BPELWS. It is inspired by compiler construction technologies and generally provides a mapping of unstructured loops to structured loops. The algorithm is based on three steps. First, a finite automaton (FA) is derived from an unstructured loop. Second, the FA is mapped to Regular Expression (RE) based on a Regular Expression Language (REL). Third, the REL respectively RE is compiled to BPEL4WS. The approach is based on the emulation of unstructured loops with existing language constructs in BPEL4WS. We currently do not support unstructured loops, because the implementation effort is too high and such loops are usually not needed in scientific workflows.

Ouyang et al. [2006, 2009] provides an event-condition-action-rules approach for transforming all BPMN control-flow structures to BPEL including arbitrary loops. They defined so-called `well-structured components` with a basic mapping to BPEL based on patterns for most common graph-based control flow constructs. These patterns are further extended in [Ouyang et al. 2007]. If a pattern matches in a graph, the corresponding well-structured component is folded to one single task activity. The pattern matching and folding is repeated until no more well-structured components can be found. All task activities (well-structured components) are transformed to BPEL code based on the defined mapping enclosed by BPEL event handler that triggers its execution. A task activity invokes succeeding task activities in the control flow by sending a message to the corresponding BPEL event handlers. The result during execution is a BPEL process instance that sends messages to itself. This mechanism allows the execution of arbitrary loops. However, it makes the debugging of a BPEL process more difficult. We applied the folding strategy to create a structure tree for BPMN workflows (see [Kippscholl 2012]). Arbitrary loops are currently not supported (see above) and thus identified as unknown components.

OMG [2011a] and all preceding BPMN standards [OMG 2006a, 2008, 2009] define patterns for the mapping of certain BPMN elements to BPEL, whereby the created BPEL code is not executable. A BPMN process must be *sound*, which means it must not contain any *deadlock* or *lack of synchronization*. We elaborated an extension of this BPEL mapping for MoDFlow.BPMN2BPEL (see Chapter 9), which has been implemented as trans-

formation chain in the MoDFlow framework (see Chapter 13).

Approaches Based on Domain-Specific Language Extensions:

We defined custom BPMN metamodel extensions for scientific workflows in MoDFlow.BPMN (see Chapter 8), e.g., to define workflow activities and parameter sweeps. These extensions can be regarded as design time only as they are mapped to standard BPEL elements. We currently do not know a comparable solution that defines BPMN metamodel extensions in the scientific workflow domain. Further metamodel extensions may be based on existing BPEL extensions [Kopp et al. 2011].

Framework-Based Approaches:

Zdun and Dustdar [2007] present a model-driven approach for the development of so-called process-driven SOAs. They defined a common meta-meta-model as basis for the definition of DSLs focusing on different aspects of process-driven SOAs such as message flow models, business process models, and architecture models. BPEL is conceptually considered as target language for the generation of executable business process models. The adoption of this approach is too complex and impracticable for our purposes and the effort for an implementation is too high.

[Roser et al. 2007] describe a model and code generation framework in order to transform *domain-specific (workflow) models* to BPEL code. A domain-specific model can be represented by different workflow languages. The authors defined domain-specific language extensions for UML 2 activity diagrams. The transformation to BPEL is executed within three steps. First, the domain-specific model is loaded via a corresponding adapter and transformed to an own common process modeling format (M2M transformation). Second, the common process model is structured via a structure-identification algorithm (M2M transformation). Third, the structured common process model is transformed to BPEL based on a visitor pattern transformation approach and code generation templates (M2T transformation). A prototype was implemented within the AgilPro

project[8], in which the code generation is based on Java Emitter Templates (JET)[9] combined with pure Java. The introduction of our intermediate workflow model (IWM) was inspired by this approach. However, we are using a BPMN subset with custom extensions defined in MoDFlow.BPMN instead of a complete new common process modeling format. Our BPMN-to-BPEL transformation chain is further realized with M2M transformations implemented with Xtend.

15.3 Transformation Chaining

Transformation chaining (or external composition) is often used to both divide a complex transformation into several steps and to integrate different transformation technologies. Existing approaches in the scientific literature focus on the foundations of transformation chaining, tool interoperability, transformation reuse, and model-driven software development processes.

The purpose of the transformation framework is to execute single model transformations and sequential transformation chains on EMF models based on different transformation technologies. None of the existing implementations, which are described below, could be used out of the box for our purpose. The efforts to adapt an implementation or to implement one of these approaches are too high. Thus, we realized an own implementation based on MWE2 (see Chapter 13), which is inspired by the ideas of the existing approaches. Most approaches provide an own metamodel to define single model transformations, transformation chains, and input and output models while abstracting from certain transformation technologies. We reused the capabilities of MWE2, whose language for defining so-called modules is already defined by a corresponding metamodel. A modification of this metamodel was not necessary to use MWE2 modules for defining single model transformations and sequential transformation chains represented by *Transformation Executor* and *Transformation Chain Executor*.

In the following, we briefly present the relevant existing approaches.

Marvie [2004] proposes a *transformation composition framework* that is

[8] http://sourceforge.net/projects/agilpro/

[9] http://www.eclipse.org/modeling/m2t/?project=jet

used to build development tools for the support of model-driven software development processes. Therefore, the author created a *primitive transformation metamodel* to define single model transformations and a *composite transformation metamodel* to define transformation chains. A transformation chain can consist of single transformations as well as of composite transformations. The approach was applied for the creation of a filtering system based on an abstract model that is transformed to Java within four transformation steps.

Blanc et al. [2005] focuses on the interoperability of modeling tools by applying a *Model Bus* to integrate and connect different *modeling services*. A modeling service can execute certain operations on models such as editing, transformation, and code generation. The signature of a modeling service, e.g., to define the consumed and produced model types, can be described based on a *Functional Description* metamodel. Such signatures are also used for compatibility checks between modeling services. Furthermore, based on a modeling service signature a Java class is generated, which is used as *EntryPoint* for the invocation of a modeling service. A proof-of concept was implemented as so-called Model Bus Integrated Environment (MBIE) based on Eclipse.

Oldevik [2005] presents a *transformation composition modeling framework* to apply model-driven software development processes. The author proposes a metamodel for a high-level definition of transformation types. A general transformation is represented by the abstract class `GenericTransformation` that defines input models, output models, and a transformation artifact. This class is extended by `ManualTransformation`, `ModelTransformation` (single model transformation), and `ComplexTransformation` (transformation chain). A `ComplexTransformation` is further separated into `SequentialTransformation` and `ParallelTransformation` for a set of `GenericTransformation` elements. All classes that extend `GenericTransformation` can be linked together based on their input and output models.

Kleppe [2006] describes a model transformation environment called MDA Control Center (MCC). The author distinguishes between *executable units* and *non-executable units*. Executable units are *Creators* (load model), *Transformers* (single M2M transformation), and *Finishers* (save model). Non-executable units are *ModelTypes* (metamodel). Executable units can be

combined as *Sequence*, *Parallel*, or *Choice*. MCC has been implemented as an Eclipse plugin.

Vanhooff et al. [2007] describes the Unified Transformation Infrastructure (UniTI) that provides a common metamodel to define model transformations separated by specification (based on a metamodel), implementation (transformation language), and execution (transformation instance). The foundations for the metamodel are presented in [Vanhooff et al. 2006]. A `TFSpecification` is used to specify a transformation that can be either an `AtomicTFSpecification` for a model transformation based on a specific technology or a `CompositeTFSpecification` to define a transformation chain. UniTI is based on Eclipse and supports EMF.

Part IV

Conclusion and Future Work

Chapter 16

Summary and Conclusion

The thesis is motivated by recent efforts in adopting standardized and well-accepted business workflow technologies for SWfMSs. We utilize the business workflow language BPEL for the execution of scientific workflows in Service Grids [Scherp et al. 2010; Gudenkauf et al. 2010a]. Our general objective is to further foster the adoption of business workflow technologies in the scientific workflow domain, which have been implemented with the help of the BPMN standard and technologies from model-driven software development (MDSD) [Scherp and Hasselbring 2010a,b].

We have addressed the problem that BPEL is well suited for the technical scientific workflow execution while it is not easily applicable by scientists for the purpose of domain-specific scientific workflow modeling, since it was originally designed for IT experts. Thus, an abstraction for BPEL is required that provides domain-specific modeling of executable scientific workflows by scientists without having to deal with technical details regarding its execution.

Our general approach introduces the intermediate layer, which serves as common exchange layer between the domain-specific layer (workflow modeling) and technical layer (workflow execution). The intermediate layer allows for the combination of different technologies for workflow modeling, for example, from the scientific workflow domain, with different technologies for workflow execution from the business workflow domain. The mapping of scientific workflows models between these layers are based on model transformations. We thus distinguish between a domain-specific workflow model (DWM), an intermediate workflow model (IWM), and an executable workflow model (EWM) as well as between a DWM2IWM mapping and an IWM2EWM mapping. Our work focused on the definition

and technical realization of an IWM based on the BPMN metamodel and corresponding model transformations for a BPMN-to-BPEL mapping to an EWM based on BPEL. Thus, the intermediate layer provides a central interface to utilize business workflow technologies in the scientific workflow domain.

This is implemented with MoDFlow, a conceptual approach for ***Mo**del-**D**riven **S**cientific **W**orkflow **E**ngineering*. It consists of MoDFlow.BPMN and MoDFlow.BPMN2BPEL as well as several utilization and extension methods. MoDFlow.BPMN defines a BPMN metamodel subset with custom extensions for the representation of IWMs. It encapsulates common aspects of scientific workflows, e.g., the definition of workflow activities and data dependencies between them, as well as high-level technical aspects for execution that are regarded as hidden from scientists, e.g. the configuration of a Web service invocation for a workflow activity. MoDFlow.BPMN2BPEL defines a BPMN-to-BPEL mapping (IWM2EWM mapping) that maps an IWM based on MoDFlow.BPMN to an EWM based BPEL. It is separated into three single model transformations that are aggregated into one model transformation chain. MoDFlow further describes different ways to utilize MoDFlow.BPMN and MoDFlow.BPMN2BPEL to implement the representation of a DWM for workflow modeling and a corresponding DWM2IWM mapping. One possibility is the creation of domain-specific languages (DSLs), which are particularly important means to apply MDSD. Thereby, different extension mechanisms can be exploited that are provided by MoDFlow.BPMN and MoDFlow.BPMN2BPEL.

The MoDFlow framework is an implementation of MoDFlow approach that is based on the Eclipse Modeling Framework (EMF) and published at `http://sourceforge.net/projects/bpmn2bpel/`. It includes a transformation framework for the execution of single model transformations and model transformation chains on EMF models, which has been used for the MoDFlow.BPMN2BPEL mapping. All model transformations are implemented in Xtend.

We have evaluated the MoDFlow framework with three application scenarios, in which different utilization and extension mechanisms of MoDFlow were applied. The first two application scenarios make use of scientific workflows with parameter sweeps that are executed in a Grid infrastructure.

They have proven the technical feasibility of the implementation of the MoDFlow framework and its application. The third application scenario has proven the practicability of MoDFlow and was conducted in collaboration with the project PubFlow, which aims at creating an infrastructure for data publication workflows. In the first project phase, PubFlow intends to examine the potential of workflow technologies for realizing data publication processes. Based on the Xtext framework, we have created a textual DSL called PubFlow.DSL with a corresponding language infrastructure that supports developers to create data publication workflows, which can be regarded as DWMs. The workflow editor includes a mapping from the DSL to MoDFlow.BPMN (DWM2IWM mapping) so that the standard BPMN-to-BPEL transformation chain can been used for BPEL code generation and workflow execution with the Apache ODE workflow engine. We have implemented and tested one sample data conversation workflow, which is also used to define domain-specific workflow activities. In the next steps, the further integration of the MoDFlow framework in the PubFlow infrastructure is fostered. To this end, several extension methods will be applied, for example, for custom error handling on Web service invocation errors. PubFlow also plans to provide a graphical DSL based on the BPMN notation and a corresponding workflow editor so that scientists can design and execute data publication workflows.

We can conclude that BPMN has a great potential since version 2.0 for its utilization in the scientific workflow domain. With MoDFlow.BPMN, we have achieved a central step and have shown that the BPMN metamodel is capable of representing common aspects of scientific workflows, for which we have defined several custom metamodel extensions. Due to our focus on BPEL and the strong relation between BPMN and BPEL, we provide model transformations for a BPMN-to-BPEL mapping with MoDFlow.BPMN2BPEL in order to generate executable BPEL code for scientific workflows, whereby only standard BPEL elements are used. Thus, it appears reasonable for us to further examine the utilization of the graphical notation and execution semantics of BPMN for scientific workflows, so that BPMN represents the domain-specific, intermediate, and technical layer. This progress may also foster a standardization process in the scientific workflow domain, in which standardized domain-specific extensions are

defined for existing business workflow standards and technologies such as BPMN and BPEL.

The implementation of the MoDFlow framework has especially benefited from the application of MDSD technologies based on EMF. The domain knowledge for the complex MoDFlow.BPMN2BPEL mapping could be encapsulated and separated in reusable and individually extendable model transformations, which can be flexibly combined and executed as transformation chain using the transformation framework. This approach facilitated the realization of the application scenarios, whereby certain steps of the BPMN-to-BPEL transformation chain have been specifically extended or replaced.

One important decision was to use the Xtend programming language for the implementation of model transformations instead of classical transformation languages such as ATL and QVT. Xtend is easy to learn by Java developers and provides many additional features that are not available in Java. Features such as extension methods, multiple dispatch and template expressions provide powerful means for the implementation of model transformations that are comparable to ATL and QVT, while existing knowledge from Java programming can be applied. This significantly fostered the adoption of Xtend in the Java-centric project PubFlow. Meanwhile, the developers in PubFlow use Xtend beyond the implementation of model transformations. This significantly contributed to the successful evaluation in PubFlow and the MoDFlow framework is now a central component of the PubFlow infrastructure. We believe that ATL and QVT Operational Mappings would have hampered the acceptance in this Java-dominated environment.

The development of PubFlow.DSL has further shown the benefits of the MDSD approach in MoDFlow. Xtext is very suited for the quick creation of a textual DSL and to generate a basic language infrastructure. PubFlow.DSL has supported the developers in PubFlow to examine the capabilities of the MoDFlow framework and the utilization of BPEL/Apache ODE for the execution of data publication workflows at an early stage of the project. This has significantly fostered the integration of the MoDFlow framework in the PubFlow infrastructure. The applied DSL concept will be extended in PubFlow with the examination of the BPMN notation to create a graphical

DSL for data publication workflows.

Finally, the MoDFlow approach and the MoDFlow framework has contributed to our central objective to foster the integration of business workflow technologies in the scientific workflow domain with the help of BPMN and MDSD technologies. The MoDFlow framework has been successfully evaluated and is further used and enhanced in the PubFlow project.

Chapter 17

Future Work

The discussion of future work is separated by the MoDFlow approach, the MoDFlow framework, and the utilization of the MoDFlow framework.

MoDFlow approach:

- *Further adoption of the capabilities of BPMN for scientific workflows*: This includes the utilization of the BPMN notation for workflow modeling, which is also part of the agenda of the PubFlow project, and the BPMN execution semantics for workflow execution. Technically, the adoption of the BPMN notation requires the utilization of the BPMN metamodel for diagram interchange (BPMN DI). With custom metamodel extensions for BPMN DI, a graphical representation for MoDFlow.BPMN could be defined, such that the separation between the domain-specific and intermediate layer is dropped. To utilize the BPMN execution semantics at the technical layer, the BPMN-to-BPEL mapping defined in MoDFlow.BPMN2BEL must be replaced by a BPMN-to-BPMN mapping, whereby the first step *BPMN Process Expansion* of MoDFlow.BPMN2BPEL can be reused. As the BPMN execution semantics provides the invocation of Web services, a major requirement is fulfilled in order to replace BPMN with BPEL. However, based on the experiences with the application scenarios, major efforts may be needed for the technical interoperability of BPMN process engines with execution infrastructures for scientific workflows, e.g., to support the security features of the Globus Toolkit 4 middleware.
- *Utilization of semantic technologies and support of data references*: The MoD-

Flow approach currently uses basic XML means to define data types for input and output parameters of workflow activities. Values for input parameters are interpreted by the workflow activities itself. Data dependencies are represented by referencing output parameters and sweep parameters by input parameters. The definition of data types can be extended by utilizing semantic technologies so that specified metadata about consumed and produced data of workflow activities can better be automatically interpreted. This also helps to create valid scientific workflow models. Furthermore, a concept to define data references for external data sources can be added so that required data transfers between data sources and processing locations can be automatically supported, e.g., by adding special data transfer workflow activities to the workflow model.

- ▻ *Utilization of Clouds as execution infrastructures for scientific workflows*: A Cloud is an infrastructure that provides services for an on-demand access to different types of resources, whereby such Cloud services are often distinguished between *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS), and *Software as a Service* (SaaS) [Mell and Grace 09]. The utilization of computing resources in a Cloud is often referred to as Cloud computing. The Cloud computing paradigm is increasingly adopted in the scientific workflow community [Berriman et al. 2013], in which certain workflow activities are executed by corresponding Cloud services. On reason is that Cloud computing scales very well, especially for applications with massive data parallelism. Since many Cloud services offer Web service interfaces for their utilization, the technical foundation for its integration in MoDFlow is given. The MoDFlow approach and framework provide means to define appropriate workflow activities for Cloud services and a mapping to the required Web service calls.

MoDFlow framework:

- ▻ *Support for process fragments within M2M transformations*: Especially in the model transformations for the *BPMN Process Expansion* step, complex

process structures are created from one source element. The required lines of manual code required for such mapping are high. One possibility to facilitate the creation of complex mappings is the definition of BPMN process fragments with an appropriate editor, which are then translated to corresponding transformation code via a model transformation. A developer uses the generated code to complete the mapping implementation. A concept for process fragments that is applied to BPEL is given by Schumm et al. [2011].

- *Support for data provenance*: Provenance for data contain information about the origin of processed data and the applied processing steps. Thus, provenance is an important research topic for scientific workflows [Gil et al. 2007]. An approach based on the monitoring framework Kieker [van Hoorn et al. 2012] to collect provenance data within a workflow execution is described by Brauer and Hasselbring [2012]. It is implemented and evaluated within the PubFlow project [Brauer and Hasselbring 2013] and can also be used in the MoDFlow framework, An existing data provenance model, for example, is the Open Provenance Model (OPM)[1].

- *Creation of a model transformation DSL for Xtend*: One advantage of transformation languages such as ATL and QVT Operational Mappings in contrast to Xtend is that their language syntax is designed for model transformations. As Xtend is a general-purpose (high-level) programming language, we have applied special class structures and code guidelines for implementing model transformations. These code guidelines mainly concerns the structure of the methods of an Xtend class and not its implementation code, because the features of the expression language Xbase used in Xtend are sufficient to implement a transformation method. Thus, a model transformation DSL for Xtend could provide special syntactic elements to define the structure of a model transformation, whereby Xbase is further used to implement transformation methods. As the complete language infrastructure of Xtend can be reused, we believe that the effort to create such a DSL is manageable.

- *Extensions for defining transformation chains*: We currently use existing

[1] `http://openprovenance.org/`

features of MWE2 to define transformation chains, which includes the MWE2 editor. This can be extended for a better support of the creation and validation of transformation chains. Therefore, the MWE2 editor can be extended by corresponding features, for which the metamodel of the MWE2 language to define MWE2 modules may also be modified. Furthermore, the definition input and output models for a model transformation step can be additionally supported.

Utilization of the MoDFlow framework:

- *Adoption of an existing scientific workflow language*. It would be a valuable contribution for the MoDFlow approach and framework, if applied with an existing scientific workflow language. Therefore, the language concepts that can be mapped to MoDFlow.BPMN have to be examined, whereby an extension of MoDFlow.BPMN or MoDFlow.BPMN2BPEL may be required. As most scientific workflow languages are data flow-centric, a mapping of data flow constructs to the corresponding control flow constructs in MoDFlow.BPMN must be provided.
- *Adoption of MoDFlow in the business workflow domain*. MoDFlow.BPMN is designed for its utilization in the scientific workflow domain. But major parts of MoDFlow.BPMN and MoDFlow.BPMN2BPEL have to deal with technical aspects of Web service orchestration. Thus, it would also be a valuable contribution to examine the applicability of the MoDFlow approach and framework in the business workflow domain. This includes an examination of comparable approaches in the business workflow domain, e.g. for BPMN-to-BPEL mappings, which may be supported in existing commercial software products.

Part V

Appendix

Appendix A

WSDL Definition for Scientific Workflows

Listing A.1. WSDL Definition for Scientific Workflows

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <wsdl:definitions name="workflow" targetNamespace="http://scherp.net/tf/workflow/wsdl"
       xmlns:tns="http://scherp.net/tf/workflow/wsdl"
       xmlns:tnstypes="http://scherp.net/tf/workflow/wsdl/types"
       xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
       xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
3
4    <wsdl:types>
5      <xsd:schema
6        targetNamespace="http://scherp.net/tf/workflow/wsdl/types"
7        xmlns:tns="http://scherp.net/tf/workflow/wsdl/types"
8        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
9
10       <xsd:element name="startWorkflowRequest" type="tns:startWorkflowRequestType"/>
11       <xsd:complexType name="startWorkflowRequestType">
12         <xsd:sequence>
13           <xsd:element name="workflowid" type="xsd:string"/>
14         </xsd:sequence>
15       </xsd:complexType>
16
17       <xsd:element name="startWorkflowResponse" type="tns:startWorkflowResponseType"/>
18       <xsd:complexType name="startWorkflowResponseType">
19         <xsd:sequence>
20           <xsd:element name="workflowid" type="xsd:string"/>
21           <xsd:element name="message" type="xsd:string"/>
22         </xsd:sequence>
23       </xsd:complexType>
24
25
26       <xsd:element name="fetchWorkflowStateRequest" type="tns:fetchWorkflowStateRequestType"/>
27       <xsd:complexType name="fetchWorkflowStateRequestType">
28         <xsd:sequence>
29           <xsd:element name="workflowid" type="xsd:string"/>
```

```
30          </xsd:sequence>
31        </xsd:complexType>
32
33        <xsd:element name="fetchWorkflowStateResponse" type="tns:fetchWorkflowStateResponseType"/>
34        <xsd:complexType name="fetchWorkflowStateResponseType">
35          <xsd:sequence>
36            <xsd:element name="workflowid" type="xsd:string"/>
37            <xsd:element name="state" type="xsd:string"/>
38          </xsd:sequence>
39        </xsd:complexType>
40
41
42        <xsd:element name="endWorkflowRequest" type="tns:endWorkflowRequestType"/>
43        <xsd:complexType name="endWorkflowRequestType">
44          <xsd:sequence>
45            <xsd:element name="workflowid" type="xsd:string"/>
46          </xsd:sequence>
47        </xsd:complexType>
48
49        <xsd:element name="endWorkflowResponse" type="tns:endWorkflowResponseType"/>
50        <xsd:complexType name="endWorkflowResponseType">
51          <xsd:sequence>
52            <xsd:element name="workflowid" type="xsd:string"/>
53            <xsd:element name="message" type="xsd:string"/>
54          </xsd:sequence>
55        </xsd:complexType>
56      </xsd:schema>
57    </wsdl:types>
58
59
60    <wsdl:message name="startWorkflowRequestMessage">
61      <wsdl:part name="startWorkflowRequestPart" type="tnstypes:startWorkflowRequestType"/>
62    </wsdl:message>
63    <wsdl:message name="startWorkflowResponseMessage">
64      <wsdl:part name="startWorkflowResponsePart" type="tnstypes:startWorkflowResponseType"/>
65    </wsdl:message>
66
67    <wsdl:message name="fetchWorkflowStateRequestMessage">
68      <wsdl:part name="fetchWorkflowStateRequestPart"
              type="tnstypes:fetchWorkflowStateRequestType"/>
69    </wsdl:message>
70    <wsdl:message name="fetchWorkflowStateResponseMessage">
71      <wsdl:part name="fetchWorkflowStateResponsePart"
              type="tnstypes:fetchWorkflowStateResponseType"/>
72    </wsdl:message>
73
74    <wsdl:message name="endWorkflowRequestMessage">
75      <wsdl:part name="endWorkflowRequestPart" type="tnstypes:endWorkflowRequestType"/>
76    </wsdl:message>
77    <wsdl:message name="endWorkflowResponseMessage">
78      <wsdl:part name="endWorkflowResponsePart" type="tnstypes:endWorkflowResponseType"/>
79    </wsdl:message>
```

```
80
81
82    <wsdl:portType name="workflowPort">
83      <wsdl:operation name="startWorkflow">
84        <wsdl:input message="tns:startWorkflowRequestMessage"/>
85        <wsdl:output message="tns:startWorkflowResponseMessage"/>
86      </wsdl:operation>
87      <wsdl:operation name="fetchWorkflowState">
88        <wsdl:input message="tns:fetchWorkflowStateRequestMessage"/>
89        <wsdl:output message="tns:fetchWorkflowStateResponseMessage"/>
90      </wsdl:operation>
91      <wsdl:operation name="endWorkflow">
92        <wsdl:input message="tns:endWorkflowRequestMessage"/>
93        <wsdl:output message="tns:endWorkflowResponseMessage"/>
94      </wsdl:operation>
95    </wsdl:portType>
96
97  </wsdl:definitions>
```

Appendix B

BPMN Workflow for Application Szenario I

Listing B.1. BPMN Workflow for Application Szenario I

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <bpmn2:definitions id="workflowBPMNId" name="workflowBPMN"
       targetNamespace="http://www.scherp.net/transformationframework"
       xmlns:tf-ext="http://scherp.net/tf/extensions"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:bpmn2="http://www.omg.org/spec/BPMN/20100524/MODEL" >
3
4  <bpmn2:process id="workflowProcess" name="workflowProcess">
5    <bpmn2:extensionElements>
6      <tf-ext:tfExtensions>
7        <tf-ext:processConfig>
8          <tf-ext:wsdlLocation>scientificworkflow.wsdl</tf-ext:wsdlLocation>
9          <tf-ext:portType>workflowPort</tf-ext:portType>
10       </tf-ext:processConfig>
11     </tf-ext:tfExtensions>
12   </bpmn2:extensionElements>
13
14 <bpmn2:startEvent id="startWorkflow" name="startWorkflow">
15   <bpmn2:extensionElements>
16     <tf-ext:tfExtensions>
17       <tf-ext:eventConfig>
18         <tf-ext:eventType>tf.event.message</tf-ext:eventType>
19       </tf-ext:eventConfig>
20       <tf-ext:messageStartEventConfig>
21         <tf-ext:operation>startWorkflow</tf-ext:operation>
22       </tf-ext:messageStartEventConfig>
23     </tf-ext:tfExtensions>
24   </bpmn2:extensionElements>
25 <bpmn2:outgoing>startWorkflowToImageService</bpmn2:outgoing>
26 </bpmn2:startEvent>
27
28 <bpmn2:sequenceFlow id="startWorkflowToImageService" name="startWorkflowToImageService"
       sourceRef="startWorkflow" targetRef="imageService"/>
29
```

```
<bpmn2:serviceTask id="imageService" name="imageService">
  <bpmn2:extensionElements>
    <tf-ext:tfExtensions>
      <tf-ext:activityConfig>
        <tf-ext:activityType>tf.activity.webservice</tf-ext:activityType>
        <tf-ext:outputParam id="imageService_leftImage" name="leftImage"
              type="{http://www.w3.org/2001/XMLSchema}string"
            part="fetch3DImagesResponse" query="//*[local-name()='leftImageGridFTPURL']" />
        <tf-ext:outputParam id="imageService_rightImage" name="rightImage"
              type="{http://www.w3.org/2001/XMLSchema}string"
            part="fetch3DImagesResponse" query="//*[local-name()='rightImageGridFTPURL']" />
        <tf-ext:outputParam id="imageService_date" name="date"
              type="{http://www.w3.org/2001/XMLSchema}string"
            part="fetch3DImagesResponse"
                query="replace(//*[local-name()='rightImageGridFTPURL'],'_r.jpg','')" />
      </tf-ext:activityConfig>
      <tf-ext:serviceTaskConfig>
        <tf-ext:serviceType>tf.services.3DImageService</tf-ext:serviceType>
        <tf-ext:wsdlLocation>schemas/tf/Image3D.wsdl</tf-ext:wsdlLocation>
        <tf-ext:portType>3DServicePort</tf-ext:portType>
        <tf-ext:operation>getImageURLs</tf-ext:operation>
        <tf-ext:requestMessageContent><![CDATA[
            <dim:fetch3DImagesRequest xmlns:dim="http://scherp/3DImageSerivce" />
        ]]></tf-ext:requestMessageContent>
        <tf-ext:requestMessagePart>fetch3DImagesRequest</tf-ext:requestMessagePart>
      </tf-ext:serviceTaskConfig>
    </tf-ext:tfExtensions>
  </bpmn2:extensionElements>
  <bpmn2:incoming>startWorkflowToImageService</bpmn2:incoming>
  <bpmn2:outgoing>imageServiceTocalc3DImage</bpmn2:outgoing>
</bpmn2:serviceTask>

<bpmn2:sequenceFlow id="imageServiceTocalc3DImage" name="imageServiceTocalc3DImage"
      sourceRef="imageService" targetRef="calc3DImage"/>

<bpmn2:serviceTask id="calc3DImage" name="calc3DImage">
  <bpmn2:extensionElements>
    <tf-ext:tfExtensions>
      <tf-ext:activityConfig>
        <tf-ext:activityType>tf.globus.jobsubmission</tf-ext:activityType>
        <tf-ext:inputParam name="leftImage" type="{http://www.w3.org/2001/XMLSchema}string"
              sourceParamRef="imageService_leftImage" targetPart="parameters"
              targetQuery="//*[local-name()='argument'][1]" />
        <tf-ext:inputParam name="rightImage" type="{http://www.w3.org/2001/XMLSchema}string"
              sourceParamRef="imageService_rightImage" targetPart="parameters"
              targetQuery="//*[local-name()='argument'][2]" />
        <tf-ext:inputParam name="date" type="{http://www.w3.org/2001/XMLSchema}string"
              sourceParamRef="imageService_date" targetPart="parameters"
              targetQuery="//*[local-name()='argument'][3]" />
        <tf-ext:inputParam name="AT" type="{http://www.w3.org/2001/XMLSchema}int"
              sourceParamRef="sweep_AT" targetPart="parameters"
              targetQuery="//*[local-name()='argument'][4]" />
```

```
69          <tf-ext:inputParam name="BN" type="{http://www.w3.org/2001/XMLSchema}int"
                sourceParamRef="sweep_BN" targetPart="parameters"
                targetQuery="//*[local-name()='argument'][5]"  />
70          <tf-ext:inputParam name="DN" type="{http://www.w3.org/2001/XMLSchema}int"
                sourceParamRef="sweep_DN" targetPart="parameters"
                targetQuery="//*[local-name()='argument'][6]"  />
71          <tf-ext:inputParam name="sourceimage" type="{http://www.w3.org/2001/XMLSchema}string"
72  sourceExpression="concat('gsiftp://srvgrid01.offis.uni-oldenburg.de/home/d-grid-users/ \\
73  dgbi0005/3dimagessweep/', $imageService_date,'_AT=',string($sweep_AT),'_BT=', \\
74  string($sweep_DT),'_DN=',string($sweep_DN),'__.jpg')"
75              targetPart="parameters" targetQuery="//*[local-name()='sourceUrl']"  />
76          <tf-ext:inputParam name="targetimage" type="{http://www.w3.org/2001/XMLSchema}string"
77  sourceExpression="concat('gsiftp://scherp.net/tmp/',$imageService_date,'_AT=', \\
78  string($sweep_AT),'_BT=',string($sweep_DT),'_DN=',string($sweep_DN),'__.jpg')"
79              targetPart="parameters" targetQuery="//*[local-name()='destinationUrl']"  />
80          <tf-ext:invidualConfigParam
                name="server">srvgrid01.offis.uni-oldenburg.de</tf-ext:invidualConfigParam>
81          <tf-ext:invidualConfigParam name="credentialDelegation">true</tf-ext:invidualConfigParam>
82          <tf-ext:invidualConfigParam name="jobTemplate"><![CDATA[
83  <des:job xmlns:job="http://www.globus.org/namespaces/2004/10/gram/job"
84      xmlns:wsn="http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-01.xsd"
85      xmlns:add="http://schemas.xmlsoap.org/ws/2004/03/addressing"
86      xmlns:des="http://www.globus.org/namespaces/2004/10/gram/job/description"
87      xmlns:typ="http://www.globus.org/namespaces/2004/10/gram/job/types"
88      xmlns:rft="http://www.globus.org/namespaces/2004/10/rft">
89
90    <des:factoryEndpoint>
91      <add:Address>http://srvgrid01.offis.uni-oldenburg.de:8443/wsrf/services/ \\
92          ManagedExecutableJobService</add:Address>
93      <add:ReferenceProperties>
94        <job:ResourceID>PBS</job:ResourceID>
95      </add:ReferenceProperties>
96    </des:factoryEndpoint>
97
98    <des:stagingCredentialEndpoint />
99    <des:executable>/home/d-grid-users/dgbi0005/generate3dImageExtended.sh</des:executable>
100   <des:argument></des:argument>
101   <des:argument></des:argument>
102   <des:argument></des:argument>
103   <des:argument></des:argument>
104   <des:argument></des:argument>
105   <des:argument></des:argument>
106   <des:queue>test</des:queue>
107
108   <des:fileStageIn>
109     <rft:transferCredentialEndpoint />
110   </des:fileStageIn>
111
112   <des:fileStageOut>
113     <rft:transferCredentialEndpoint />
114     <rft:transfer>
115       <rft:sourceUrl />
```

```
116         <rft:destinationUrl />
117       </rft:transfer>
118     </des:fileStageOut>
119
120     <des:fileCleanUp>
121       <rft:transferCredentialEndpoint />
122     </des:fileCleanUp>
123   </des:job>
124           ]]></tf-ext:invidualConfigParam>
125           <tf-ext:invidualConfigParam name="resourceID">PBS</tf-ext:invidualConfigParam>
126         </tf-ext:activityConfig>
127       </tf-ext:tfExtensions>
128     </bpmn2:extensionElements>
129     <bpmn2:incoming>imageServiceTocalc3DImage</bpmn2:incoming>
130     <bpmn2:outgoing>calc3DImageToEndWorkflow</bpmn2:outgoing>
131     <bpmn2:multiInstanceLoopCharacteristics >
132       <bpmn2:extensionElements>
133         <tf-ext:tfExtensions>
134           <tf-ext:multiInstanceLoopCharacteristicsConfig>
135             <tf-ext:sweepParam id="sweep_AT" name="AT" type="{http://www.w3.org/2001/XMLSchema}int"
                    startValue="4" endValue="5" incrementValue="1" />
136             <tf-ext:sweepParam id="sweep_BN" name="BN"
                    type="{http://www.w3.org/2001/XMLSchema}string" values ="0.8;1.0;1.2"
                    valuesSeparator=";" />
137             <tf-ext:sweepParam id="sweep_DN" name="DN"
                    type="{http://www.w3.org/2001/XMLSchema}string" values ="0.8;1.0;1.2"
                    valuesSeparator=";" />
138           </tf-ext:multiInstanceLoopCharacteristicsConfig>
139         </tf-ext:tfExtensions>
140       </bpmn2:extensionElements>
141       <bpmn2:loopCardinality><![CDATA[2]]></bpmn2:loopCardinality>
142     </bpmn2:multiInstanceLoopCharacteristics>
143   </bpmn2:serviceTask>
144
145   <bpmn2:sequenceFlow id="calc3DImageToEndWorkflow" name="calc3DImageToEndWorkflow"
        sourceRef="calc3DImage" targetRef="endWorkflow"/>
146
147   <bpmn2:endEvent id="endWorkflow" name="endWorkflow">
148     <bpmn2:extensionElements>
149       <tf-ext:tfExtensions>
150         <tf-ext:eventConfig>
151           <tf-ext:eventType>tf.event.message</tf-ext:eventType>
152         </tf-ext:eventConfig>
153         <tf-ext:messageEndEventConfig>
154           <tf-ext:operation>startWorkflow</tf-ext:operation>
155           <tf-ext:reponseMessageContent><![CDATA[
156               <typ:startWorkflowResponse xmlns:typ="http://scherp.net/tf/workflow/wsdl/types">
157                   <message>Workflow finished</message>
158               </typ:startWorkflowResponse>
159           ]]></tf-ext:reponseMessageContent>
160           <tf-ext:reponseMessagePart>startWorkflowResponsePart</tf-ext:reponseMessagePart>
161         </tf-ext:messageEndEventConfig>
```

```
162         </tf-ext:tfExtensions>
163       </bpmn2:extensionElements>
164       <bpmn2:incoming>calc3DImageToEndWorkflow</bpmn2:incoming>
165     </bpmn2:endEvent>
166
167     </bpmn2:process>
168     </bpmn2:definitions>
```

Appendix C

BPMN Workflow for Application Szenario II

Listing C.1. BPMN Workflow for Application Szenario II

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <bpmn2:definitions id="workflowBPMNId" name="workflowBPMN"
       targetNamespace="http://www.scherp.net/transformationframework"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:bpmn2="http://www.omg.org/spec/BPMN/20100524/MODEL"
       xmlns:tf-ext="http://scherp.net/tf/extensions" >
3
4  <bpmn2:process id="workflowProcess" name="workflowProcess">
5    <bpmn2:extensionElements>
6      <tf-ext:tfExtensions>
7        <tf-ext:processConfig>
8          <tf-ext:wsdlLocation>workflow.wsdl</tf-ext:wsdlLocation>
9          <tf-ext:portType>workflowPort</tf-ext:portType>
10       </tf-ext:processConfig>
11     </tf-ext:tfExtensions>
12   </bpmn2:extensionElements>
13
14 <bpmn2:startEvent id="startWorkflow" name="startWorkflow">
15   <bpmn2:extensionElements>
16     <tf-ext:tfExtensions>
17       <tf-ext:eventConfig>
18         <tf-ext:type>tf.event.message</tf-ext:type>
19       </tf-ext:eventConfig>
20       <tf-ext:messageStartEventConfig>
21         <tf-ext:operation>startWorkflow</tf-ext:operation>
22       </tf-ext:messageStartEventConfig>
23     </tf-ext:tfExtensions>
24   </bpmn2:extensionElements>
25   <bpmn2:outgoing>startWorkflowTorunMATLAB</bpmn2:outgoing>
26 </bpmn2:startEvent>
27
28 <bpmn2:sequenceFlow id="startWorkflowTorunMATLAB" name="startWorkflowTorunMATLAB"
       sourceRef="startWorkflow" targetRef="runMATLAB"/>
29
```

```
<bpmn2:serviceTask id="runMATLAB" name="runMATLAB">
  <bpmn2:extensionElements>
    <tf-ext:tfExtensions>
      <tf-ext:activityConfig>
        <tf-ext:type>tf.globus.jobsubmission</tf-ext:type>
        <tf-ext:inputParam name="randomBRicker" type="{http://www.w3.org/2001/XMLSchema}string"
              part="parameters" query="//*[local-name()='argument'][1]" >1</tf-ext:inputParam>
        <tf-ext:inputParam name="randomW" type="{http://www.w3.org/2001/XMLSchema}string"
              part="parameters" query="//*[local-name()='argument'][2]" >1</tf-ext:inputParam>
        <tf-ext:inputParam name="outputdir" type="{http://www.w3.org/2001/XMLSchema}string"
              part="parameters" query="//*[local-name()='argument'][3]"
              >/home/d-grid-users/dgbi0005/sensitivityBalticCod</tf-ext:inputParam>
        <tf-ext:inputParam name="invocationCount" type="{http://www.w3.org/2001/XMLSchema}int"
              part="parameters" query="//*[local-name()='argument'][4]"
              source="sweep_invocationCount" />
        <tf-ext:invidualConfigParam
              name="server">srvgrid01.offis.uni-oldenburg.de</tf-ext:invidualConfigParam>
        <tf-ext:invidualConfigParam name="jobTemplate"><![CDATA[
<des:job xmlns:job="http://www.globus.org/namespaces/2004/10/gram/job"
    xmlns:wsn="http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-01.xsd"
    xmlns:add="http://schemas.xmlsoap.org/ws/2004/03/addressing"
    xmlns:des="http://www.globus.org/namespaces/2004/10/gram/job/description"
    xmlns:typ="http://www.globus.org/namespaces/2004/10/gram/job/types"
    xmlns:rft="http://www.globus.org/namespaces/2004/10/rft">

  <des:factoryEndpoint>
    <add:Address>http://srvgrid01.offis.uni-oldenburg.de:8443/wsrf/services/ \\
        ManagedExecutableJobService</add:Address>
    <add:ReferenceProperties>
      <job:ResourceID>PBS</job:ResourceID>
    </add:ReferenceProperties>
  </des:factoryEndpoint>

  <des:executable>/home/d-grid-users/dgbi0005/sensitivityBalticCod.sh</des:executable>
  <des:argument></des:argument>
  <des:argument></des:argument>
  <des:argument></des:argument>
  <des:argument></des:argument>
  <des:queue>test</des:queue>
</des:job>
        ]]></tf-ext:invidualConfigParam>
        <tf-ext:invidualConfigParam name="resourceID">PBS</tf-ext:invidualConfigParam>
      </tf-ext:activityConfig>
    </tf-ext:tfExtensions>
  </bpmn2:extensionElements>
  <bpmn2:incoming>startWorkflowTorunMATLAB</bpmn2:incoming>
  <bpmn2:outgoing>runMATLAToendWorkflow</bpmn2:outgoing>
  <bpmn2:multiInstanceLoopCharacteristics >
    <bpmn2:extensionElements>
      <tf-ext:tfExtensions>
        <tf-ext:multiInstanceLoopCharacteristicsConfig>
```

```
74            <tf-ext:sweepParam id="sweep_invocationCount" name="invocationCount"
                    type="{http://www.w3.org/2001/XMLSchema}int" startValue="1" endValue="10000"
                    incrementValue="1" />
75          </tf-ext:multiInstanceLoopCharacteristicsConfig>
76        </tf-ext:tfExtensions>
77      </bpmn2:extensionElements>
78      <bpmn2:loopCardinality><![CDATA[5]]></bpmn2:loopCardinality>
79    </bpmn2:multiInstanceLoopCharacteristics>
80  </bpmn2:serviceTask>
81
82  <bpmn2:sequenceFlow id="matlabServiceToEndWorkflowEvent" name="runMATLAToendWorkflow"
        sourceRef="runMATLAB" targetRef="endWorkflow"/>
83
84  <bpmn2:endEvent id="endWorkflow" name="endWorkflow">
85    <bpmn2:extensionElements>
86      <tf-ext:tfExtensions>
87        <tf-ext:eventConfig>
88          <tf-ext:type>tf.event.message</tf-ext:type>
89        </tf-ext:eventConfig>
90        <tf-ext:messageEndEventConfig>
91          <tf-ext:operation>startWorkflow</tf-ext:operation>
92          <tf-ext:reponseMessageContent><![CDATA[
93              <typ:startWorkflowResponse xmlns:typ="http://scherp.net/tf/workflow/wsdl/types">
94                <message>Workflow finished</message>
95              </typ:startWorkflowResponse>
96          ]]></tf-ext:reponseMessageContent>
97        </tf-ext:messageEndEventConfig>
98      </tf-ext:tfExtensions>
99    </bpmn2:extensionElements>
100   <bpmn2:incoming>runMATLAToendWorkflow</bpmn2:incoming>
101 </bpmn2:endEvent>
102
103 </bpmn2:process>
104 </bpmn2:definitions>
```

Appendix D

Xtext grammar PubFlow.DSL for Application Szenario III

Listing D.1. Xtext grammar PubFlow.DSL for Application Szenario III

```
1  grammar net.scherp.tf.pubflow.dsl.PubFlowDSL with org.eclipse.xtext.common.Terminals
2
3  generate pubFlowDSL "http://www.scherp.net/tf/pubflow/dsl/PubFlowDSL"
4
5  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
6
7  Process :
8      'process'
9      '{'
10         ('workflowBaseDir' '=' workflowBaseDir = STRING)?
11         'id' '=' id = STRING
12         'targetNamespace' '=' targetNamespace = STRING
13         'wsdlLocation' '=' wsdlLocation = STRING
14         ('portType' '=' portType = STRING)?
15         ('service' '=' service = STRING)?
16         ('servicePort' '=' servicePort = STRING)?
17         startEvent = StartEvent
18         processElementRoot = ProcessElementRoot
19         endEvent = EndEvent
20     '}'
21 ;
22
23 ProcessElementRoot:
24     Sequence | Flow
25 ;
26
27 ProcessElement:
28     ServiceTask | Task | SubProcess | Sequence | Flow | Switch
29 ;
30
31 Sequence : {Sequence}
32     'sequence'
33     '{'
```

```
34          'id' '=' id = STRING
35          processElements += ProcessElement+
36      '}'
37  ;
38
39  Flow : {Flow}
40      'flow'
41      '{'
42          'id' '=' id = STRING
43          processElements += ProcessElement+
44      '}'
45  ;
46
47  SubProcess :
48      'subProcess'
49      '{'
50          'id' '=' id = STRING
51          'activityType' '=' activityType = STRING
52          processElementRoot = ProcessElementRoot
53      '}'
54  ;
55
56  Switch :
57      'switch'
58      '{'
59          'id' '=' id = STRING
60          case += Case+
61          default = Default?
62      '}'
63  ;
64
65  Case :
66      'case' '(' condition = STRING ')'
67      '{'
68          processElement = ProcessElement
69      '}'
70  ;
71
72  Default : {Default}
73      'default'
74      '{'
75          processElement = ProcessElement
76      '}'
77  ;
78
79  StartEvent :
80      'startEvent'
81      '{'
82          'id' '=' id = STRING
83          'eventType' '=' eventType = STRING
84          'operation' '=' operation = STRING
85          outputParam += OutputParameter*
```

```
86      '}'
87  ;
88
89  EndEvent :
90      'endEvent'
91      '{'
92          'id' '=' id = STRING
93          'eventType' '=' eventType = STRING
94          'operation' '=' operation = STRING
95          ('responseMessageContent' '=' responseMessageContent = STRING)?
96          ('responseMessagePart' '=' responseMessagePart = STRING)?
97          ('responseMessageKeepSrcElementName' '=' responseMessageKeepSrcElementName = BOOLEAN)?
98          inputParam += InputParameter*
99      '}'
100 ;
101
102 ServiceTask :
103     'serviceTask'
104     '{'
105         'id' '=' id = STRING
106         'activityType' '=' activityType = STRING
107         'serviceType' '=' serviceType = STRING
108         'wsdlLocation' '=' wsdlLocation = STRING
109         'portType' '=' portType = STRING
110         'service' '=' service = STRING
111         'servicePort' '=' servicePort = STRING
112         'operation' '=' operation = STRING
113         ('requestMessageContent' '=' requestMessageContent = STRING)?
114         ('requestMessagePart' '=' requestMessagePart = STRING)?
115         ('requestMessageKeepSrcElementName' '=' requestMessageKeepSrcElementName = BOOLEAN)?
116         inputParam += InputParameter*
117         outputParam += OutputParameter*
118     '}'
119 ;
120
121 Task :
122     'task'
123     '{'
124         'id' '=' id = STRING
125         'activityType' '=' activityType = STRING
126         inputParam += InputParameter*
127         outputParam += OutputParameter*
128     '}'
129 ;
130
131 InputParameter :
132     'inputParam'
133     '{'
134         'name' '=' name = STRING
135         'type' '=' type = STRING
136         ('collection' '=' collection = BOOLEAN)?
137         ('sourceParamRef' '=' sourceParamRef = STRING)?
```

```
138         ('sourceParamQuery' '=' sourceParamQuery = STRING)?
139         ('sourceExpression' '=' sourceExpression = STRING)?
140         ('sourceValue' '=' sourceValue = STRING)?
141         ('targetPart' '=' targetPart = STRING)?
142         ('targetQuery' '=' targetQuery = STRING)?
143         ('targetExpression' '=' targetExpression = STRING)?
144         ('targetKeepSrcElementName' '=' targetKeepSrcElementName = BOOLEAN)?
145     '}'
146 ;
147
148 OutputParameter :
149     'outputParam'
150     '{'
151         'id' '=' id = STRING
152         'name' '=' name = STRING
153         'type' '=' type = STRING
154         ('collection' '=' collection = BOOLEAN)?
155         ('sourcePart' '=' sourcePart = STRING)?
156         ('sourceQuery' '=' sourceQuery = STRING)?
157         ('sourceExpression' '=' sourceExpression = STRING)?
158     '}'
159 ;
160
161 terminal BOOLEAN returns ecore::EBoolean:
162     'true' | 'false' | 'yes' | 'no'
163 ;
```

Appendix E

PubFlow.DSL Workflow for Application Szenario III

Listing E.1. PubFlow.DSL Workflow for Application Szenario III

```
1  process {
2    workflowBaseDir = "/home/guido/runtime-EclipseApplication/PubFlow/wsdls"
3    id = "PubflowProcess"
4    targetNamespace = "http://www.scherp.net/transformationframework/pubflow"
5    wsdlLocation = "OCN_Bottle_FlowArtifacts.wsdl"
6    portType = "{pubflow.de}OCN_Bottle_Flow"
7    service = "{pubflow.de}OCN_Bottle_Flow"
8    servicePort = "OCN_Bottle_FlowPort"
9
10     startEvent{
11       id = "pubflowStart"
12       eventType = "tf.event.message"
13       operation = "process"
14
15       outputParam { id="pubFlowStart_input" name="input"
               type="{http://www.w3.org/2001/XMLSchema}int" sourcePart="payload"
               sourceQuery="*[local-name() = 'input']" }
16       outputParam { id="pubFlowStart_pid" name="pid"
               type="{http://www.w3.org/2001/XMLSchema}string" sourcePart="payload"
               sourceQuery="*[local-name() = 'pid']" }
17       outputParam { id="pubFlowStart_login" name="login"
               type="{http://www.w3.org/2001/XMLSchema}string" sourcePart="payload"
               sourceQuery="*[local-name() = 'login']" }
18       outputParam { id="pubFlowStart_source" name="source"
               type="{http://www.w3.org/2001/XMLSchema}string" sourcePart="payload"
               sourceQuery="*[local-name() = 'source']" }
19       outputParam { id="pubFlowStart_author" name="author"
               type="{http://www.w3.org/2001/XMLSchema}string" sourcePart="payload"
               sourceQuery="*[local-name() = 'author']" }
20       outputParam { id="pubFlowStart_type" name="type"
               type="{http://www.w3.org/2001/XMLSchema}string" sourcePart="payload"
               sourceQuery="*[local-name() = 'type']" }
```

```
21        outputParam { id="pubFlowStart_project" name="project"
                type="{http://www.w3.org/2001/XMLSchema}string" sourcePart="payload"
                sourceQuery="*[local-name() = 'project']" }
22        outputParam { id="pubFlowStart_topology" name="topology"
                type="{http://www.w3.org/2001/XMLSchema}string" sourcePart="payload"
                sourceQuery="*[local-name() = 'topology']" }
23        outputParam { id="pubFlowStart_status" name="status"
                type="{http://www.w3.org/2001/XMLSchema}string" sourcePart="payload"
                sourceQuery="*[local-name() = 'status']" }
24        outputParam { id="pubFlowStart_savePath" name="savePath"
                type="{http://www.w3.org/2001/XMLSchema}string" sourcePart="payload"
                sourceQuery="*[local-name() = 'savePath']" }
25        outputParam { id="pubFlowStart_reference" name="reference"
                type="{http://www.w3.org/2001/XMLSchema}string" sourcePart="payload"
                sourceQuery="*[local-name() = 'reference']" }
26        outputParam { id="pubFlowStart_fileName" name="fileName"
                type="{http://www.w3.org/2001/XMLSchema}string" sourcePart="payload"
                sourceQuery="*[local-name() = 'fileName']" }
27        outputParam { id="pubFlowStart_comment" name="comment"
                type="{http://www.w3.org/2001/XMLSchema}string" sourcePart="payload"
                sourceQuery="*[local-name() = 'comment']" }
28      }
29
30      sequence {
31        id = "workflowSequence"
32
33      serviceTask {
34        id = "loadBottleService"
35        activityType = "tf.activity.webservice"
36        serviceType = "pubflow.services.LoadBottles"
37        wsdlLocation = "workflowbasefiles/schemas/dbconnector.wsdl"
38        portType = "{http://dbconnector.service.ocn.sample.pubflow/}DBConnector"
39        service = "{http://dbconnector.service.ocn.sample.pubflow/}DBConnectorImplService"
40        servicePort = "DBConnectorImplPort"
41        operation = "getData"
42
43        inputParam { name="id" type="{http://www.w3.org/2001/XMLSchema}int"
                sourceParamRef="pubFlowStart_input" targetPart="id" }
44
45        outputParam { id="loadBottleService_return" name="return"
                type="{http://www.w3.org/2001/XMLSchema}anyType" sourcePart="return" }
46      }
47
48      serviceTask {
49        id = "MapToPangaea"
50        activityType = "tf.activity.webservice"
51        serviceType = "pubflow.services.MapToPangaea"
52        wsdlLocation = "workflowbasefiles/schemas/transformer.wsdl"
53        portType = "{http://transformer.service.ocn.sample.pubflow/}Transformer"
54        service = "{http://transformer.service.ocn.sample.pubflow/}TransformerImplService"
55        servicePort = "TransformerImplPort"
56        operation = "replaceArtefacts"
```

```
57
58        inputParam { name="id" type="{http://www.w3.org/2001/XMLSchema}int"
                sourceParamRef="loadBottleService_return" targetPart="id" }
59
60        outputParam { id="MapToPangaea_return" name="return"
                type="{http://www.w3.org/2001/XMLSchema}anyType" sourcePart="return" }
61      }
62
63      serviceTask {
64        id = "Writeto4d"
65        activityType = "tf.activity.webservice"
66        serviceType = "pubflow.services.Write_to_4d"
67        wsdlLocation = "workflowbasefiles/schemas/fileprinter.wsdl"
68        portType = "{http://fileprinter.service.ocn.sample.pubflow/}FilePrinter"
69        service = "{http://fileprinter.service.ocn.sample.pubflow/}FilePrinterImplService"
70        servicePort = "FilePrinterImplPort"
71        operation = "toCSV"
72
73        inputParam { name="input" type="{http://www.w3.org/2001/XMLSchema}anyType"
                sourceParamRef="MapToPangaea_return" targetPart="input" }
74        inputParam { name="pid" type="{http://www.w3.org/2001/XMLSchema}anyType"
                sourceParamRef="pubFlowStart_pid" targetPart="pid" }
75        inputParam { name="login" type="{http://www.w3.org/2001/XMLSchema}anyType"
                sourceParamRef="pubFlowStart_login" targetPart="login" }
76        inputParam { name="source" type="{http://www.w3.org/2001/XMLSchema}anyType"
                sourceParamRef="pubFlowStart_source" targetPart="source" }
77        inputParam { name="author" type="{http://www.w3.org/2001/XMLSchema}anyType"
                sourceParamRef="pubFlowStart_author" targetPart="author" }
78        inputParam { name="project" type="{http://www.w3.org/2001/XMLSchema}anyType"
                sourceParamRef="pubFlowStart_project" targetPart="project" }
79        inputParam { name="topology" type="{http://www.w3.org/2001/XMLSchema}anyType"
                sourceParamRef="pubFlowStart_topology" targetPart="topology" }
80        inputParam { name="status" type="{http://www.w3.org/2001/XMLSchema}anyType"
                sourceParamRef="pubFlowStart_status" targetPart="status" }
81        inputParam { name="savePath" type="{http://www.w3.org/2001/XMLSchema}anyType"
                sourceParamRef="pubFlowStart_savePath" targetPart="savePath" }
82        inputParam { name="reference" type="{http://www.w3.org/2001/XMLSchema}anyType"
                sourceParamRef="pubFlowStart_reference" targetPart="reference" }
83        inputParam { name="fileName" type="{http://www.w3.org/2001/XMLSchema}anyType"
                sourceParamRef="pubFlowStart_fileName" targetPart="fileName" }
84        inputParam { name="comment" type="{http://www.w3.org/2001/XMLSchema}anyType"
                sourceParamRef="pubFlowStart_comment" targetPart="comment" }
85
86        outputParam { id="Writeto4d_return" name="return"
                type="{http://www.w3.org/2001/XMLSchema}anyType" sourcePart="return" }
87      }
88
89    }
90
91    endEvent {
92      id = "pubflowEnd"
93      eventType = "tf.event.message"
```

```
94      operation = "process"
95      responseMessageContent = "<![CDATA[<tns:OCN_Bottle_FlowResponse xmlns:tns=\"pubflow.de\" >
96          <tns:result>tns:result</tns:result>
97          </tns:OCN_Bottle_FlowResponse>]]>"
98      responseMessagePart = "payload"
99
100     inputParam { name="output" type="{http://www.w3.org/2001/XMLSchema}anyType"
            sourceParamRef="Writeto4d_return" sourceParamQuery="*[local-name()='result']"
            targetPart="payload" }
101   }
102 }
```

Bibliography

[Barga and Gannon 2007] R. Barga and D. Gannon. Scientific versus Business Workflows. In *Workflows for e-Science*, pages 9–16. Springer London, 2007. doi: 10.1007/978-1-84628-757-2. ISBN 978-1-84628-519-6 (Print) 978-1-84628-757-2 (Online). (cited on page 39)

[Bärisch 2010] S. Bärisch. *Domain-Specific Model-Driven Testing*. Software Engineering Research. Vieweg+Teubner Verlag, 2010. ISBN 978-3-8348-0931-5. URL `http://www.viewegteubner.de/Buch/978-3-8348-0931-5/Domain-Specific-Model-Driven-Testing.html`. (cited on page 207)

[BEA et al. 2002] BEA, IBM, and Microsoft. Business Process Execution Language for Web Services Version 1.0, July 2002. (cited on pages 26 and 235)

[BEA et al. 2003] BEA, IBM, Microsoft, SAP, and S. Systems. Business Process Execution Language for Web Services Version 1.1, May 2003. (cited on pages 26 and 235)

[Berriman et al. 2013] G. B. Berriman, G. Juve, J.-S. Vöckler, E. Deelman, and M. Rynge. The Application of Cloud Computing to Scientific Workflows: A Study of Cost and Performance. *Proceedings of the Royal Society A*, 371 (1983), January 2013. (cited on page 256)

[Biehl 2010] M. Biehl. Literature Study on Model Transformations. Technical Report ISRN/KTH/MMK/R-10/07-SE, Royal Institute of Technology, July 2010. (cited on pages 46, 47, and 150)

[Blanc et al. 2005] X. Blanc, M.-P. Gervais, and P. Sriplakich. Model Bus: Towards the Interoperability of Modelling Tools. In U. Aßmann, M. Aksit, and A. Rensink, editors, *Model Driven Architecture*, volume 3599 of *Lecture Notes in Computer Science*, pages 17–32. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-28240-2. doi: 10.1007/11538097_2. (cited on page 244)

[Brauer and Hasselbring 2012] P. C. Brauer and W. Hasselbring. Capturing provenance information with a workflow monitoring extension for the Kieker framework. In *Proceedings of the 3rd International Workshop on Semantic Web in Provenance Management*, volume 856 of *CEUR Workshop Proceedings*. CEUR-WS, Mai 2012. (cited on page 257)

[Brauer and Hasselbring 2013] P. C. Brauer and W. Hasselbring. PubFlow: provenance-aware workflows for research data publication. In *5th USENIX Workshop on the Theory and Practice of Provenance (TaPP '13)*, April 2013. (cited on page 257)

[Bézivin and Kurtev 2005] J. Bézivin and I. Kurtev. Model-based Technology Integration with the Technical Space Concept. In *Proceedings of the Metainformatics Symposium, Springer-Verlag*. Springer-Verlag, 2005. (cited on page 152)

[Chapman et al. 2006a] C. Chapman, A. Walker, M. Calleja, R. Bruin, M. Dove, and W. Emmerich. Managing Scientific Processes on the eMinerals Mini-Grid using BPEL. In *Proceedings of the UK e-Science All Hands Meeting*, 2006a. (cited on page 236)

[Chapman et al. 2006b] C. Chapman, A. Walker, M. Calleja, R. Bruin, M. Dove, and W. Emmerich. Simple Grid Access using the Business Process Execution Language. In *Proceedings of the UK e-Science All Hands Meeting*, 2006b. (cited on page 236)

[Czarnecki and Helsen 2006] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45:621–645, July 2006. ISSN 0018-8670. doi: 10.1147/sj.453.0621. (cited on pages 46, 47, and 150)

[Dörnemann et al. 2007] T. Dörnemann, T. Friese, S. Herdt, E. Juhnke, and B. Freisleben. Grid Workflow Modelling Using Grid-Specific BPEL Extensions. In *Proceedings of German e-Science Conference 2007*, pages 1–9, 2007. (cited on pages 4 and 237)

[Dörnemann et al. 2008] T. Dörnemann, M. Smith, and B. Freisleben. Composition and Execution of Secure Workflows in WSRF-Grids. *Cluster*

Computing and the Grid, IEEE International Symposium on, 0:122–129, 2008. doi: 10.1109/CCGRID.2008.74. (cited on page 238)

[Efftinge et al. 2012] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. Massow, W. Hasselbring, and M. Hanus. Xbase: implementing domain-specific languages for Java. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering - GPCE '12*, pages 112–121, New York, NY, USA, 2012. ACM. (cited on page 49)

[Emmerich et al. 2005] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. Price. Grid Service Orchestration Using the Business Process Execution Language (BPEL). *Journal of Grid Computing*, 3(3-4):283–304, September 2005. ISSN 1570-7873. doi: 10.1007/s10723-005-9015-3. (cited on pages 4 and 236)

[Eshuis et al. 2006] R. Eshuis, P. Grefen, and S. Till. Structured service composition. In *Proc. of the 4th International Conference on Business Process Management (BPM 2006), volume 4102 of Lecture Notes in Computer Science*, pages 97–112. Springer, 2006. (cited on page 239)

[Eusgeld et al. 2008] I. Eusgeld, F. C. Freiling, and R. Reussner, editors. *Dependability Metrics: Advanced Lectures [result from a Dagstuhl seminar, October 30 - November 1, 2005]*, volume 4909 of *Lecture Notes in Computer Science*, 2008. Springer. ISBN 978-3-540-68946-1. (cited on page 207)

[Ezenwoye et al. 2007a] O. Ezenwoye, S. M. Sadjadi, A. Cary, and M. Robinson. Orchestrating WSRF-based Grid Services. Technical report, School of Computing and Information Sciences, Florida International University, April 2007a. (cited on pages 4 and 237)

[Ezenwoye et al. 2007b] O. Ezenwoye, S. M. Sadjadi, A. Cary, and M. Robinson. Grid Service Composition in BPEL for Scientific Applications. In R. Meersman and Z. Tari, editors, *OTM Conferences (2)*, volume 4804 of *Lecture Notes in Computer Science*, pages 1304–1312. Springer, 2007b. ISBN 978-3-540-76835-7. (cited on page 237)

[Favre and Nguyen 2005] J.-M. Favre and T. Nguyen. Towards a Megamodel to Model Software Evolution Through Transformations. *Electr. Notes Theor. Comput. Sci.*, 127(3):59–74, 2005. (cited on page 44)

[Foster 2002] I. Foster. What is the Grid? - a three point checklist. *GRIDtoday*, 1(6), July 2002. URL http://www.gridtoday.com/02/0722/100136.html. (cited on pages 1 and 58)

[Foster et al. 2001] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, August 2001. ISSN 1094-3420. doi: 10.1177/109434200101500302. (cited on pages 57 and 58)

[Foster et al. 2002] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, 2002. (cited on pages 4 and 58)

[Foster et al. 2006] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. V. Reich. The Open Grid Services Architecture, Version 1.5, Open Grid Forum Final Document GFD.80. Technical report, OGF, September 2006. URL http://www.ogf.org/documents/GFD.80.pdf. (cited on page 58)

[Fowler 2010] M. Fowler. *Domain-Specific Languages*. The Addison-Wesley Signature Series. Addison-Wesley, 2010. ISBN 9780321712943. (cited on pages 7, 45, and 48)

[Gil et al. 2007] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. Examining the Challenges of Scientific Workflows. *Computer*, 40(12):24–32, December 2007. ISSN 0018-9162. doi: 10.1109/MC.2007.421. (cited on pages 82 and 257)

[Globus Security Team 2005] Globus Security Team. Globus Toolkit Version 4 Grid Security Infrastructure: A Standards Perspective, 2005. URL http://www.globus.org/toolkit/docs/4.0/security/GT4-GSI-Overview.pdf. (cited on pages 64, 65, and 238)

[Goble and Roure 2009] C. A. Goble and D. D. Roure. The impact of workflow tools on data-centric research. In Hey et al. [2009], pages 137–145. URL http://research.microsoft.com/en-us/collaboration/fourthparadigm/. (cited on pages 2 and 37)

[Görlach et al. 2011] K. Görlach, M. Sonntag, D. Karastoyanova, F. Leymann, and M. Reiter. *Conventional Workflow Technology for Scientific Simulation*, pages 1–31. Guide to e-Science. Springer-Verlag, März 2011. ISBN 978-0-85729-438-8. (cited on pages 4, 5, 39, 40, 41, 81, 82, 83, 85, 147, and 238)

[Götz et al. 2008] M. Götz, S. Roser, F. Lautenbacher, and B. Bauer. Using Token Analysis to Transform Graph-Oriented Process Models to BPEL. Technical Report 2008-08, Informatik, 2008. (cited on pages 188 and 240)

[Gray 2007] J. Gray. eScience – A Transformed Scientific Method, January 2007. URL http://research.microsoft.com/en-us/um/people/gray/talks/NRC-CSTB_eScience.ppt. Talk to National Research Council - Computer Science and Telecommunications Board (NRC-CSTB). (cited on page 1)

[Gudenkauf et al. 2008] S. Gudenkauf, A. Höing, and G. Scherp. BIS-Grid Deliverable 2.1: Catalogue of WS-BPEL Design Patterns. Technical report, BIS-Grid, August 2008. (cited on pages 214 and 237)

[Gudenkauf et al. 2009] S. Gudenkauf, W. Hasselbring, A. Höing, O. Kao, G. Scherp, H. Nitsche, H. Karl, and A. Brinkmann. Employing WS-BPEL Design Patterns for Grid Service Orchestration using a Standard WS-BPEL Engine and a Grid Middleware. In *The 8th Cracow Grid Workshop*, pages 103 – 110, Cracow, Poland, March 2009. Academic Computer Center CYFRONET AGH. (cited on pages 214 and 237)

[Gudenkauf et al. 2010a] S. Gudenkauf, A. Höing, D. Meister, H. Nitsche, and G. Scherp. BIS-Grid Deliverable 3.4: Final Version of the WS-BPEL Engine. Technical report, BIS-Grid, April 2010a. (cited on pages 66, 73, 235, and 249)

[Gudenkauf et al. 2010b] S. Gudenkauf, A. Höing, and G. Scherp. BIS-Grid Deliverable 3.5: GT4 Interoperability. Technical report, BIS-Grid, April 2010b. (cited on pages 68 and 238)

[Hasselbring 2010] W. Hasselbring, editor. *Betriebliche Informationssysteme: Grid-basierte Integration und Orchestrierung*. GITO mbH Verlag, 2010. ISBN 9783942183208. (cited on page 66)

[Hey et al. 2009] T. Hey, S. Tansley, and K. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, Redmond, Washington, 2009. URL http://research.microsoft.com/en-us/collaboration/fourthparadigm/. (cited on pages 1, 37, and 289)

[Hey et al. 2012] T. Hey, D. Gannon, and J. Pinkelman. The Future of Data-Intensive Science. *Computer*, 45(5):81 –82, may 2012. ISSN 0018-9162. doi: 10.1109/MC.2012.181. (cited on page 1)

[Hille-Doering 2010] R. Hille-Doering. Making of the BPMN 2.0 Meta Model for Eclipse: Merge and Conquer, October 2010. (cited on pages 77, 149, and 159)

[Höing et al. 2009] A. Höing, G. Scherp, and S. Gudenkauf. The BIS-Grid Engine: an Orchestration as a Service Infrastructure. *International Journal of Computing*, 8(3):96–104, 2009. (cited on page 66)

[Hollingsworth 1995] D. Hollingsworth. Workflow Management Coalition - The Workflow Reference Model. Technical report, Workflow Management Coalition, January 1995. (cited on pages 15, 18, and 38)

[Hopcroft and Tarjan 1974] J. E. Hopcroft and R. E. Tarjan. Dividing a Graph into Triconnected Components. *SIAM J. Comput.*, 2(3):135–158, 1974. (cited on page 240)

[Huber 2008] P. Huber. The Model Transformation Language Jungle - An Evaluation and Extension of Existing Approaches. Master's thesis, Universität Wien Business Informatics Group Institut für Softwaretechnik und Interaktive Systeme, May 2008. (cited on pages 47, 150, 155, and 174)

[Höing 2010] A. Höing. *Orchestrating secure workflows for cloud and grid services.* PhD thesis, Berlin Institute of Technology, 2010. URL http://d-nb.info/1010030981. (cited on page 66)

[Johnson et al. 1994] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 171–185, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. doi: 10.1145/178243.178258. (cited on page 240)

[Kesselman and Foster 1998] C. Kesselman and I. Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998. ISBN 1558604758. (cited on pages 1 and 57)

[Khalaf 2008] R. Khalaf. *Supporting business process fragmentation while maintaining operational semantics : a BPEL perspective*. PhD thesis, Universität Stuttgart, Holzgartenstr. 16, 70174 Stuttgart, 2008. (cited on page 238)

[Kippscholl 2012] D. Kippscholl. Structure Identification in BPMN Workflows. Master's thesis, University of Kiel - Department of Computer Science, 2012. (cited on pages 130, 188, 190, 191, 199, 240, and 241)

[Kleppe 2006] A. Kleppe. MCC: A Model Transformation Environment. In A. Rensink and J. Warmer, editors, *Model Driven Architecture – Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 173–187. Springer Berlin / Heidelberg, 2006. URL 10.1007/11787044_14. 10.1007/11787044_14. (cited on page 244)

[Kopp et al. 2011] O. Kopp, K. Görlach, D. Karastoyanova, F. Leymann, M. Reiter, D. Schumm, M. Sonntag, S. Strauch, T. Unger, M. Wieland, and R. Khalaf. A Classification of BPEL Extensions. *Journal of Systems Integration*, 4(2):3–28, 2011. ISSN 1804-2724. (cited on pages 29, 73, 109, 235, 238, and 242)

[Koziolek 2008] H. Koziolek. *Parameter Dependencies for Reusable Performance Specifications of Software Components*. PhD thesis, University of Oldenburg, Germany, March 2008. URL http://oops.uni-oldenburg.de/742/. (cited on page 207)

[Leymann 2006] F. Leymann. Choreography for the Grid: towards fitting BPEL to the resource framework: Research Articles. *Concurr. Comput. :*

Pract. Exper., 18(10):1201–1217, 2006. ISSN 1532-0626. doi: 10.1002/cpe.v18:10. (cited on pages 4 and 237)

[Leymann and Roller 1999] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, September 1999. ISBN 0130217530. (cited on pages 15 and 40)

[Lin et al. 2009] C. Lin, S. Lu, X. Fei, A. Chebotko, D. Pai, Z. Lai, F. Fotouhi, and J. Hua. A Reference Architecture for Scientific Workflow Management Systems and the VIEW SOA Solution. *IEEE Transactions on Services Computing*, 2(1):79–92, 2009. ISSN 1939-1374. doi: 10.1109/TSC.2009.4. (cited on pages 3, 37, 38, and 39)

[Ludäscher et al. 2009] B. Ludäscher, M. Weske, T. McPhillips, and S. Bowers. Scientific Workflows: Business as Usual? In U. Dayal, J. Eder, J. Koehler, and H. Reijers, editors, *7th Intl. Conf. on Business Process Management (BPM)*, LNCS 5701, Ulm, Germany, 2009. (cited on pages 2, 37, 39, and 40)

[Marvie 2004] R. Marvie. A transformation composition framework for model driven engineering. Technical report, University of Lille - Computer Science Research Lab, 2004. (cited on page 243)

[Mell and Grace 09] P. Mell and T. Grace. The NIST Definition of Cloud Computing, July 09. URL `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`. (cited on page 256)

[Mendling et al. 2006] J. Mendling, K. B. Lassen, and U. Zdun. Transformation Strategies between Block-Oriented and Graph-Oriented Process Modelling Languages. In *Multikonferenz Wirtschaftsinformatik 2006. Band 2*, pages 297–312. GITO-Verlag, 2006. (cited on page 239)

[Mendling et al. 2008] J. Mendling, K. B. Lassen, and U. Zdun. On the Transformation of Control Flow between Block-Oriented and Graph-Oriented Process Modeling Languages. *IJBPIM*, 2008. (cited on pages 29, 31, 32, 33, 34, 35, and 239)

[Mens and Gorp 2006] T. Mens and P. V. Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, March 2006. doi: 10.1016/j.entcs.2005.10.021. (cited on pages 46, 47, 150, and 151)

[OASIS 2006] OASIS. Web Services Resource Framework (WSRF), Version 1.2, April 2006. URL http://www.oasis-open.org/committees/wsrf/. (cited on pages 4 and 58)

[OASIS 2007] OASIS. Web Services Business Process Execution Language Version 2.0, April 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. (cited on pages 4, 19, 26, and 235)

[Oldevik 2005] J. Oldevik. Transformation Composition Modelling Framework. In L. Kutvonen and N. Alonistioti, editors, *Distributed Applications and Interoperable Systems*, volume 3543 of *Lecture Notes in Computer Science*, pages 1135–1136. Springer Berlin / Heidelberg, 2005. doi: 10.1007/11498094_10. (cited on page 244)

[OMG 2006a] OMG. Business Process Modeling Notation, V1.0, February 2006a. URL http://www.omg.org/bpmn/Documents/OMG_Final_Adopted_BPMN_1-0_Spec_06-02-01.pdf. (cited on pages 20, 240, and 241)

[OMG 2006b] OMG. Meta Object Facility (MOF), February 2006b. URL http://www.omg.org/spec/MOF/. (cited on pages 20 and 45)

[OMG 2006c] OMG. Object Contraint Language (OCL), February 2006c. URL http://www.omg.org/spec/OCL/. (cited on page 44)

[OMG 2008] OMG. Business Process Modeling Notation, V1.1, February 2008. URL http://www.omg.org/bpmn/Documents/BPMN_1-1_Specification.pdf. (cited on pages 20 and 241)

[OMG 2009] OMG. Business Process Model and Notation, V1.2, January 2009. URL http://www.omg.org/spec/BPMN/1.2/. (cited on pages 20 and 241)

[OMG 2011a] OMG. Business Process Model and Notation (BPMN) Version 2.0, January 2011a. URL http://www.omg.org/spec/BPMN/2.0/. (cited on pages 6, 19, 21, 25, 131, 132, and 241)

[OMG 2011b] OMG. Query/View/Transformation (QVT), V1.1, January 2011b. URL http://www.omg.org/spec/QVT/1.1/. (cited on page 47)

[OMG 2011c] OMG. XML Metadata Interchange (XMI), Version 2.4.1, August 2011c. URL http://www.omg.org/spec/XMI/2.4.1/. (cited on page 45)

[Ouyang et al. 2006] C. Ouyang, W. van der Aalst, M. Dumas, and A. Hofstede. Translating BPMN to BPEL. BPM Center Report BPM-06-02, BPMcenter.org, 2006. (cited on pages 32, 33, and 241)

[Ouyang et al. 2007] C. Ouyang, M. Dumas, A. H. ter Hofstede, and W. M. van der Aalst. Pattern-based translation of bpmn process models to bpel web services. *International Journal of Web Services Research (JWSR)*, 2007. (cited on page 241)

[Ouyang et al. 2009] C. Ouyang, M. Dumas, W. M. P. V. D. Aalst, A. H. M. T. Hofstede, and J. Mendling. From business process models to process-oriented software systems. *ACM Trans. Softw. Eng. Methodol.*, 19:2:1–2:37, August 2009. ISSN 1049-331X. doi: 10.1145/1555392.1555395. (cited on pages 32, 33, and 241)

[Reussner and Hasselbring 2008] R. Reussner and W. Hasselbring, editors. *Handbuch der Software-Architektur*. dpunkt, Heidelberg, 2. edition, 2008. ISBN 978-3-89864-559-1. (cited on pages 7, 43, 44, 45, 46, 47, and 150)

[Roser et al. 2007] S. Roser, F. Lautenbacher, and B. Bauer. Generation of Workflow Code from DSMs. In *In Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*, 2007. (cited on page 242)

[Scherp and Hasselbring 2010a] G. Scherp and W. Hasselbring. Towards a model-driven transformation framework for scientific workflows. *Procedia Computer Science*, 1(1):1513 – 1520, 2010a. ISSN 1877-0509. doi: DOI: 10.1016/j.procs.2010.04.169. ICCS 2010. (cited on pages 74 and 249)

[Scherp and Hasselbring 2010b] G. Scherp and W. Hasselbring. Ein modellgetriebener Ansatz zur Nutzung von WS-BPEL für scientific workflows. In G. Engels, M. Luckey, A. Pretschner, and R. H. Reussner, editors, *Software Engineering 2010 – Workshopband*, volume 160 of *Lecture Notes in*

Informatics, pages 201–208. Gesellschaft für Informatik e.V., 2010b. (cited on pages 74 and 249)

[Scherp and Hasselbring 2011] G. Scherp and W. Hasselbring. Interoperability of the BIS-Grid Workflow Engine with Globus Toolkit 4. In K.-D. Warzecha and L. Packschies, editors, *Proceedings of the Grid Workflow Workshop 2011*, volume 826. CEUR Workshop Proceedings, 2011. URL http://ceur-ws.org/Vol-826/paper06.pdf. (cited on page 212)

[Scherp et al. 2010] G. Scherp, A. Höing, S. Gudenkauf, W. Hasselbring, and O. Kao. Using UNICORE and WS-BPEL for Scientific Workflow Execution in Grid Environments. In *Euro-Par 2009 Workshops - Parallel Processing*, volume Lecture Notes in Computer Science, 2010. (cited on pages 4, 39, 73, and 249)

[Schumm et al. 2011] D. Schumm, D. Karastoyanova, O. Kopp, F. Leymann, M. Sonntag, and S. Strauch. Process Fragment Libraries for Easier and Faster Development of Process-based Applications. *Journal of Systems Integration*, 2(1):39–55, 2011. ISSN 1804-2724. (cited on page 257)

[Sonntag et al. 2010] M. Sonntag, D. Karastoyanova, and E. Deelman. Bridging the Gap between Business and Scientific Workflows: Humans in the Loop of Scientific Workflows. In *e-Science (e-Science), 2010 IEEE Sixth International Conference on*, pages 206 –213, dec. 2010. doi: 10.1109/eScience.2010.12. (cited on page 236)

[Stein et al. 2009] S. Stein, S. Kühne, and K. Ivanov. Business to IT Transformations Revisited. In D. Ardagna, M. Mecella, and J. Yang, editors, *Business Process Management Workshops : BPM 2008 Internation Workshops, Milano, Italy, September 2008, Revised Papers*, number 17 in LNBIB, pages 176–187, Berlin, 2009. Springer. ISBN 978-3-642-00327. (cited on pages 238 and 239)

[Tan et al. 2007] W. Tan, L. Fong, and N. Bobroff. BPEL4Job: A Fault-Handling Design for Job Flow Management. In *ICSOC '07: Proceedings of the 5th international conference on Service-Oriented Computing*, pages 27–42, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-74973-8. doi: 10.1007/978-3-540-74974-5_3. (cited on pages 4 and 237)

[Tarjan 1972] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. (cited on page 240)

[Taylor et al. 2006] I. J. Taylor, E. Deelman, and D. B. Gannon. *Workflows for e-Science: Scientific Workflows for Grids*. Springer, December 2006. ISBN 1846285194. (cited on pages 2 and 37)

[Thomas H Cormen 1994] R. L. R. Thomas H Cormen, Charles E Leiserson. *Introduction to algorithms*. The MIT electrical engineering and comp. sci. series, MIT Press, Cambridge, 1994. (cited on page 240)

[Treadwell 2007] J. Treadwell. Open Grid Services Architecture - Glossary of Terms Version 1.6. Technical report, OGF, December 2007. URL `http://www.ogf.org/documents/GFD.120.pdf`. (cited on page 60)

[van Hoorn et al. 2012] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM, April 2012. (cited on page 257)

[Vanhatalo et al. 2009] J. Vanhatalo, H. Völzer, and J. Koehler. The refined process structure tree. *Data Knowl. Eng.*, 68(9):793–818, 2009. ISSN 0169-023X. doi: 10.1016/j.datak.2009.02.015. (cited on page 240)

[Vanhooff et al. 2006] B. Vanhooff, S. Van Baelen, A. Hovsepyan, W. Joosen, and Y. Berbers. Towards a Transformation Chain Modeling Language. In S. Vassiliadis, S. Wong, and T. Hämäläinen, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 4017 of *Lecture Notes in Computer Science*, pages 39–48. Springer Berlin / Heidelberg, 2006. doi: 10.1007/11796435_6. 10.1007/11796435_6. (cited on page 245)

[Vanhooff et al. 2007] B. Vanhooff, D. Ayed, S. V. Baelen, W. Joosen, and Y. Berbers. UniTI: A Unified Transformation Infrastructure. In *MoDELS*, pages 31–45, 2007. (cited on page 245)

[W3C 1999] W3C. XSL Transformations (XSLT) Version 1.0, November 1999. URL `http://www.w3.org/TR/xslt`. (cited on page 47)

[W3C 2006] W3C. Web Services Addressing 1.0, May 2006. URL http://www.w3.org/TR/ws-addr-core/. (cited on pages 28 and 59)

[W3C 2007] W3C. Web Services Policy 1.5 - Framework, April 2007. URL http://www.w3.org/TR/ws-policy/. (cited on page 237)

[Wang et al. 2005] Y. Wang, C. Hu, and J. Huai. A New Grid Workflow Description Language. In *SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing*, pages 257–260, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2408-7-02. doi: 10.1109/SCC.2005.14. (cited on pages 4 and 236)

[Wassermann et al. 2007] B. Wassermann, W. Emmerich, B. Butchart, N. Cameron, L. Chen, and J. Patel. Sedna: A BPEL-Based Environment for Visual Scientific Workflow Modeling. In I. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 428–449. Springer London, 2007. ISBN 978-1-84628-519-6. doi: 10.1007/978-1-84628-757-2_26. (cited on pages 4, 39, and 236)

[WfMC 1999] WfMC. *Workflow Management Coalition Terminology & Glossary (Document No. WFMC-TC-1011).* Workflow Management Coalition Specification, 1999. (cited on pages 15, 16, 17, 18, 19, and 37)

[Yildiz et al. 2009] U. Yildiz, A. Guabtni, and A. H. H. Ngu. Towards scientific workflow patterns. In *WORKS '09: Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, pages 1–10, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-717-2. doi: 10.1145/1645164.1645177. (cited on page 236)

[Zdun and Dustdar 2007] U. Zdun and S. Dustdar. Model-Driven and Pattern-Based Integration of Process-Driven SOA Models. *International Journal of Business Process Integration and Management (IJBPIM)*, 2(2):109–119, 2007. (cited on page 242)

[Zeng et al. 2007] J. Zeng, Z. Du, C. Hu, and J. Huai. CROWN FlowEngine: A GPEL-Based Grid Workflow Engine. In *HPCC*, pages 249–259, 2007. (cited on page 236)

[Zhao et al. 2006] W. Zhao, R. Hauser, K. Bhattacharya, B. R. Bryant, and F. Cao. Compiling business processes: untangling unstructured loops in irreducible flow graphs. *Int. J. Web Grid Serv.*, 2:68–91, February 2006. ISSN 1741-1106. doi: 10.1504/IJWGS.2006.008880. (cited on pages 32 and 240)