Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diploma Thesis No. 2822

Conceptual Design and Implementation of a BPEL^{light} Workflow Engine With Support for Message Exchange Patterns

Mirko Sonntag

Course of Study:	Software Engineering
Examiner:	Prof. Dr. Frank Leymann
Supervisor:	Dipl.-Inf. Jörg Nitzsche Dipl.-Inf. Tammo van Lessen
Commenced:	April 30th, 2008
Completed:	October 30th, 2008
CR-Classification:	H.4.1, K.1, D.2.12, C.3, D.2.13

Acknowledgements

To my wife and son, Caroline and Luca. Thanks for your support, patience and energy.

To my parents. For making all this possible.

To Markus and Benjamin. For extensive discussions – and relaxing coffee breaks.

To Prof. Dr. F. Leymann. For giving me the opportunity of developing this thesis at the Institute of Architecture of Application Systems.

To Dipl.-Inf. Jörg Nitzsche and Dipl.-Inf. Tammo van Lessen. For your supervision and help in the last months.

Contents

Chapter 1

Introduction

Business process management (BPM) in general and workflow technology in particular gain more and more attention in industry and research. On the one hand workflows contribute to support enterprises making business with each other (business to business, B2B) in an efficient and flexible manner. On the other hand they help solving enterprise application integration (EAI) problems closing gaps in IT landscapes often evolved over years. Anyway, running business processes on IT systems brings significant benefits especially in dynamic environments [WCL+05, LR00].

The Business Process Execution Language (BPEL) [AAA+07] is the common standard for describing executable business processes as a composition of Web services (also referred to as *orchestration*). BPEL's communication is tightly coupled to WSDL[1] 1.1. For example, each BPEL interaction activity corresponds to a WSDL operation, variables are related to WSDL messages, and so on. Under these circumstances a process model designer must be a union of a business economist and a Web services specialist.

Message exchange patterns (MEPs) are abstract definitions of a service's message flow and the relation of these messages. With WSDL 1.1 Web services are emphasized as imperative programming units following the request-response or one-way patterns only. Its successor, WSDL 2.0, makes MEPs an essential part of its interaction model. It is a promising approach that underlines the message-oriented nature of Web services which is the next logical step towards real message orientation in the Web services world.

1.1. Motivation

The situation sketched in the previous section manifests two conceptual issues this work deals with: separating business process logic from Web service technology and incorporating the notion of (arbitrary complex) MEPs into workflow management.

[1] Web Services Description Language

BPEL and WSDL The coupling between BPEL and WSDL 1.1 brings a number of significant drawbacks. First, it prohibits a priori the usage of another interface definition language a user may want to take (e.g. SSDL[2]). Second, modelling a BPEL business process presumes both, knowledge in business and IT level. This hampers designing and reusing process models.

To overcome these deficiencies BPEL^{light} was invented [NLKL07], a BPEL 2.0 extension decoupling process logic from interface definitions. In short, it emulates all BPEL constructs that directly refer to WSDL elements. That way BPEL^{light} facilitates modelling reusable and flexible business processes.

Message Exchange Patterns In WSDL 2.0 a "Template for Message Exchange Pattern" was introduced [BL07] that is able to describe interactions beyond simple one-way or request-response operations as opposed to WSDL 1.1 operations. The template can be compared to an orchestration language describing the message flow within an operation from the service's point of view.

But it lacks the solution of several issues of great importance [NLL08] (e.g. machine readability, communication with multiple instances of a node) that can be overcome by formalizing MEPs with BPEL^{light} [LNL08]. Therefore an abstract process profile was defined to enable the definition of reusable MEPs as abstract BPEL^{light} processes. This brings the advantage of taking an expressive, machine readable language for describing MEPs. In a further step these abstract MEPs can be parameterized towards executable processes being used by business processes realizing certain interaction patterns.

1.2. Task Definition

This diploma thesis consists of two main parts. First, it discusses the notion of MEPs in the context of BPM. In a further step this general approach is incorporated into BPEL^{light}. For this reason BPEL^{light} is extended to support MEPs during development and runtime of process models.

Second, these theoretical considerations are put into praxis. Currently, there is no tool support for BPEL^{light} in the whole lifecycle of business processes. Thus, in the scope of this work a prototypical BPEL^{light} workflow engine is developed bringing the ideas of BPEL^{light} and MEPs together. The open-source project Apache Orchestration Director Engine (ODE) [ODE], a BPEL 2.0 workflow engine, is taken as basis. It is going to be extended to support a part of the lifecycle of BPEL^{light} business processes. This includes the deployment of BPEL^{light} processes, their execution and administration as well as the deployment and execution of parameterized BPEL^{light} MEPs that are used to describe BPEL^{light} interaction activities.

The following tasks are dealt with concerning the development of the prototype. The engine must be able to handle WSDL 2.0 interface descriptions. It is to be extended to understand and process BPEL^{light} workflows. Furthermore the engine is supposed to be capable of using

[2] SOAP Service Description Language

MEPs as description of the behaviour of interaction activities. ODE's legacy functionality should thereby stay untouched.

1.3. Proceeding

The remainder of the document is structured as follows. Chapter 2 reviews technologies and concepts that are essential to understand the topic. The Apache ODE project is presented in detail in chapter 3. The theoretical concepts developed in this work are handled in depth in chapter 4. This is at the same time the specification of the developed workflow engine. Chapter 5 reveals implementation details of the constructed engine. Finally, the thesis is summarized and closes with an outlook in chapter 6.

Chapter 2

Background

This chapter briefly gives insight into the basic technologies and concepts this work depends on. The focus is not on delivering a complete description of the touched topics. It rather helps getting a high level overview for better understanding the subject and problem covered in this thesis, and for having a common comprehension of key terms.

2.1. Service-oriented Architecture

The service-oriented architecture (SOA) is an architectural concept to integrate heterogeneous systems for enabling rich business communication. It is built around pieces of self-contained functionality, the *services*, which provide an abstract view on business functions allowing their users to ignore the underlying component model and implementation (see [WCL+05]).

The key elements of an SOA can be arranged as a temple (the *SOA temple*, see [Mel07]). Its illustration is omitted but the contained components are explained. The fundament consists of following three elements:

- *Simplicity*
  The simplicity does not apply to the technologies as such – they are by far not simple. It rather thinks of their application being eased by means of automation, tool support, and so on.
- *Standards*
  The use of standards supports its adoption in industry and research. Accepted standards increase the likelihood of a technology's establishment.
- *Security*
  A major issue of an SOA is security (i.e. authentication, integrity, confidentiality, and so on). It is crucial for its success in business applications.

Upon the fundament four columns are arranged. Together they carry the service-oriented architecture's roof.

- *Distribution*
  An SOA facilitates the distribution of its services throughout a network. Indeed, an SOA enables services that are running on different platforms, built for different operating systems, and realized by different programming languages to communicate with each other (i.e. *interoperability*).
- *Loose Coupling*
  Reducing the knowledge and the number of assumptions the interacting partners have about each other loosens the coupling between them. On the one hand this fosters interoperability. On the other hand this makes services robust against adaptations if a used partner service changes. A loose coupling makes services reusable.
- *Service Discovery*
  Services are describable pieces of software that can be found and accessed by appropriate discovery mechanisms.
- *Process Orientation*
  Processes permit the orchestration of one or more services to new, higher-level services. This leads to a recursive aggregation model: a service can be composed of other services and can in parallel be a part of a superior service.

The basic principles how an SOA works are reflected by the SOA triangle (see figure 2.1) [WCL+05]. A service is described by an abstract definition. It provides information about its operational functions, its semantics, and how to access the service. A service provider uses this definition to *publish* its service in a discovery facility. This discovery mechanism is a directory or registry collecting service metadata. It is a contact point for service users that are searching for services that meet their requirements. That means a requester *finds* a service by certain criteria with the discovery's help and selects the one he is interested in. After that he is able to *bind* to and use the service due to the fetched metadata (i.e. the service's address and binding details like the transport protocol and message format). This process is referred to as *dynamic binding* if it happens at runtime. The roles service provider and requester cannot be clearly separated. A service requester can be at the same time a service provider; a service provider may also be a service requester in parallel.

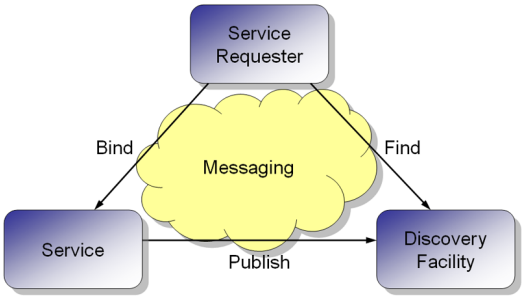

Figure 2.1.: The SOA triangle

2.1.1. Enterprise Service Bus

An Enterprise Service Bus (ESB), or short *service bus*, is a middleware infrastructure that simplifies the tedious dynamic binding process. It fosters a standard-based, loosely-coupled communication between services being connected to the bus. The service bus hooks into the SOA triangle and takes over the part of finding and selecting a service by given criteria,

and binding to it (i.e. *routing*). A service requester only needs to contact the bus by passing the selection data and request message. The bus must also provide data transformation (i.e. *mediation*) capabilities since the services typically do not share a common data format.

That way the ESB realizes a *service virtualization*. That means the services are described in a unified manner. Their implementations are interchangeable without effecting the interface. A service requester does not need to know the concrete partner he is interacting with. The communication is abstracted by the bus.

2.2. Message Exchange Patterns

In an SOA, a message exchange pattern (MEP) is a template defining the message flow between a service and a service consumer and how these messages are related to each other. Thus, it can be seen as a contract between both parties. The abstraction from concrete message types and business contexts allows its reusability in many different situations. MEPs are not only restricted to bilateral interactions but can in fact involve several parties participating in a conversation (e.g. a number of bidders compete in an auction). The complexity of MEPs can vary from a single message sent between two partners up to multiple messages between multiple partners. MEPs emphasize the message-oriented character of Web services in contrast to an imperative programming style. [BDH05] gives a good discussion about MEPs, their categorization and business value. The W3C[1] defines MEPs as follows:

> *A Message Exchange Pattern (MEP) is a template, devoid of application semantics, that describes a generic pattern for the exchange of messages between agents. It describes relationships (e.g., temporal, causal, sequential, etc.) of multiple messages exchanged in conformance with the pattern, as well as the normal and abnormal termination of any message exchange conforming to the pattern.* [BHM+04]

In the course of this work the terms *simple* and *complex* MEP will be used. An MEP is considered simple if it consists of at most two messages exchanged with a single partner. The messages must have opposite directions. A complex MEP can encompass more than two messages with more than one partner. There are no restrictions on the messages' directions.

2.3. Web Services

Often the terms SOA and Web services (WS) are mixed up, but it is important to draw a line between them. While SOA is an abstract architectural style, the Web services technology is the most prominent implementation thereof. Another one is, for example, Jini [Jin] (see

[1] World Wide Web Consortium – a public, international organization that works on the development of Web standards

[BKM07]). However, Web services distinguish themselves in the loose coupling aspect of the architecture and in dynamic components that are adaptable to changes [WCL+05].

The Web services technology is a standard-based, XML[2]-centric approach to offer and request services in a distributed environment. It facilitates the interoperability between applications independent of programming languages, operating systems and platforms. The W3C gives following definition:

> *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP[3] with an XML serialization in conjunction with other Web-related standards.* [BHM+04]

There is not only one way a service requester may discover, choose and use a service. Ideally, this happens by contacting a service directory. Figure 2.2 illustrates this and clarifies at the same time the direct relationship between SOA and Web services. It shows the SOA triangle leveraged by the three most famous WS technologies, SOAP, WSDL[4], and UDDI[5].

The UDDI directory holds information about services published by service providers. These are service descriptions and technical binding information for communicating with it. The service is incorporated into taxonomies to support searching for services by particular criteria (e.g. matter of the service, costs, security). WSDL documents are used to reveal a service's interface. The interface exactly prescribes the way to interact with the service. SOAP is a message format to exchange structured information between peers independent of underlying transport protocols. Although Web services are not limited to use SOAP messages for communication, in fact, SOAP established as the fundamental message format for Web services.

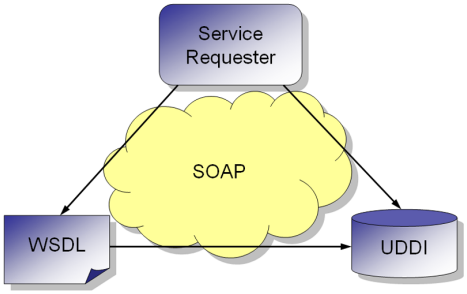

Figure 2.2.: The SOA triangle leveraged by Web service technologies

SOAP and WSDL are presented in more depth in the following. There are a lot more WS specifications and technologies. Since they do not play a role within this work, they are skipped.

[2] eXtensible Markup Language
[3] Hypertext Transfer Protocol
[4] Web Services Description Language
[5] Universal Description, Discovery and Integration

2.3.1. SOAP

Up to version 1.1 [BEK+00] SOAP was the acronym for "Simple Object Access Protocol". From version 1.2 on [GHM+07], SOAP is no longer an acronym since it is not really a protocol for accessing objects. SOAP is an XML-based messaging framework for an information exchange in decentralized, distributed environments. Its design makes it independent of particular programming models. The SOAP binding mechanism allows a diversity of underlying transport protocols like HTTP, or SMTP[6], for example, or even proprietary protocols. SOAP is the most important message format to access Web services.

A SOAP message is shipped from an *initial sender* to an *ultimate receiver*. Between those two the message can pass other nodes, the *intermediaries*. The message's way from the first to the last node is called *message path*. Along the message path the transport protocol can change between each pair of nodes (hop-by-hop processing semantics). Consequently, quality of services (QoS) cannot be guaranteed by the transport protocols. The SOAP message cares about these issues by its extension points where QoS specifications can plug in (WS-Security, WS-Reliable Messaging, etc.) which can assure an end-to-end processing semantics.

SOAP messages consist of an envelope (compare figure 2.3) that contains a body and an optional header element. The header encompasses one or more header blocks, each targeted at any SOAP receiver on the message path. These blocks build the SOAP extension model where other WS specifications or proprietary mechanisms can stick into. The body carries the message payload or business information arranged as one or more child elements. This can be, for instance, a request sent to a service with associated input data. The body is targeted at the ultimate receiver, but intermediaries may change its content on the message path.

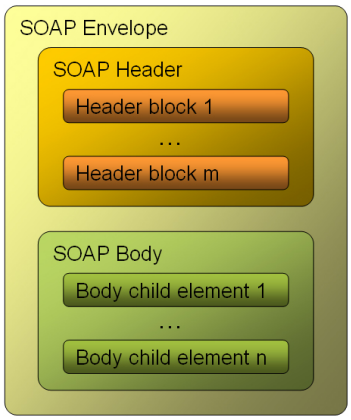

Figure 2.3.: General structure of a SOAP message

SOAP's messaging model provides the transmission of stateless one-way messages. This simple model can be extended to allow more use- and powerful interactions between Web services. Such interactions are referred to as MEPs, explained in detail in 2.2 on page 7. The extension is realized by adequate SOAP headers that might hold a message identifier and a reference to a previous message, for example. WS-Addressing (see 2.3.2 on the next page) provides such a mechanism.

For a description of SOAP's processing model the reader is referred to [GHM+07]. Since it goes far beyond the scope of this work, it is omitted here.

[6] Simple Mail Transfer Protocol

2.3.2. Web Services Addressing

WS-Addressing (WS-A) [BCC+04] provides mechanisms to ensure the correct delivery of messages to their destination. On the one hand Web service endpoints must be identified, which is done by endpoint references. On the other hand these identifiers and additional information must be incorporated into message headers to enable a rich interaction model. Message information headers take over this part. But in terms of this work they play a secondary role and are only touched for the purpose of completeness.

Endpoint References An endpoint reference (EPR) is an XML-based data structure to keep information for accessing a service endpoint. It contains both runtime information and associated metadata. The runtime information is covered by the following three fields:

- *Address* – This mandatory field contains a URI[7] which is the address the service endpoint can be reached at.
- *ReferenceProperties* – These XML elements can be used to identify the entity at the endpoint.
- *ReferenceParameters* – These parameters are XML elements that facilitate the interaction with the endpoint.

Besides these information, the metadata given below finalize the definition of EPRs:

- *PortType* – An optional field containing the QName[8] of the WSDL port type the endpoint is providing.
- *ServiceName* – This optional field embodies the WSDL service's QName and its WSDL port name that form the static representation of the endpoint.
- *Policy* – A set of WS-Policy [BBC+06] elements that need to be met when communicating with the endpoint.

Listing 2.1 shows an example of an EPR. It references the endpoint of the service "aSimpleService" at the URI "http://www.exampleURI.com/endpoint". As parameter the item number "1234" is passed.

Listing 2.1 Example of an endpoint reference

```
<wsa:EndpointReference>
   <wsa:Address>http://www.exampleURI.com/endpoint</wsa:Address>
   <wsa:ReferenceParameters>
      <eg:itemno>1234</eg:itemno>
   </wsa:ReferenceParameters>
   <wsa:ServiceName>tns:aSimpleService</wsa:ServiceName>
</wsa:EndpointReference>
```

[7] Uniform Resource Identifier
[8] Qualified Name – is a triple containing a prefix, namespace, and local part

The final version of WS-A [GHR06] recommends using the address and reference parameters only. Furthermore it introduces a new `MetaData` field that may contain information describing the behaviour, policies and capabilities of the endpoint.

Message Information Headers A bidirectional and asynchronous communication is enabled by message information headers – a set of properties augmenting message headers with additional data. The `<To>` header contains the address of the target. `<Action>` holds the intention of the message as URI (e.g. an update of a record). There are also endpoint information about who is the sender of the message (`<Source>`), where to send a reply (`<ReplyTo>`) or fault (`<FaultTo>`) to, as well as instance information like a message identifier (`<MessageID>`) or the relation to a previous message (`<RelatesTo>`).

WS-A specifies how to insert the information of an EPR into a SOAP message. Only the EPR's runtime information (`Address`, `ReferenceProperty`, and `ReferenceParameter`) are thereby reflected as SOAP headers. The message's intention is taken from the `Action` field of the `input` or `output` of the addressed WSDL operation, if present. Otherwise the WS-A specification provides rules of how to derive an action value on behalf of the port type and message name.

2.3.3. WSDL

The Web Services Description Language (WSDL) is an XML-based notation and by now the de facto standard to convey what can be done with a Web service. Its design allows for high extensibility, for example in using different type systems (like XML Schema, DTD[9], etc.), transport protocols (HTTP, RMI[10], etc.), or message formats (SOAP, plain text, etc.).

The WSDL document is divided into two parts: an abstract and a concrete one. The abstract part reflects the service's operational behaviour by listing its incoming and outgoing messages. The concrete part shows where and how to address the service. WSDL is restricted to represent syntactical information only. Semantical details are committed to other languages, such as RDF[11] or OWL[12]. WSDL supports describing both RPC[13]-style and messaging style Web services.

The WSDL specification comes along in two versions. WSDL 1.1 [CCMW01] is well accepted but poses some problems. Therefore a successor, WSDL 2.0 [BL07], was specified addressing many of these deficiencies. Both are described next with focus on version 2.0 since it underpins (arbitrary) MEPs in its communication model.

[9] Document Type Definition
[10] Remote Method Invocation
[11] Resource Description Framework
[12] Web Ontology Language
[13] Remote Procedure Call

WSDL 1.1 Version 1.1 of WSDL wraps all information with a `<definitions>` element. It contains the data structures (`<types>`), messages (`<message>`) and interfaces (`<portType>`) as abstract part as well as the bindings (`<binding>`) and services (`<service>`) as concrete part. The data types can include both XML and non-XML data. The messages describe pieces of information the Web service is exchanging. Each message consists of one or more parts. A part is a single item to be sent or received referencing a built-in or self-defined data type. Port types are groupings of operations (`<operation>`). Each operation can be used to send (`<output>`) or receive (`<input>`) a message leading to four types of operations (or MEPs):

- *One-way* – The service receives a message.
- *Request-response* – The service first receives a message and produces a message in response.
- *Solicit-response* – The service first sends a message and receives a response.
- *Notification* – The service sends a message.

The binding declares how to translate the abstract messages into a concrete ones (e.g. a SOAP message, or an HTTP GET) by referencing a port type. The different bindings are WSDL 1.1 extensions. That way not only SOAP or HTTP bindings can be defined but arbitrary ones. The `<service>` element defines where the service can be found. It contains one or more `<port>`s, each referencing a binding and giving a location (e.g. a URI) where the port type can be accessed.

WSDL 2.0 A WSDL 2.0 document (see figure 2.4) is structured similar to its predecessor. Some changes are significant, others subtle. The elements `<definitions>`, `<portType>` and `<port>` are renamed to `<description>`, `<interface>` and `<endpoint>`. The concept of `<message>`s is eliminated. Instead, each operation's input, output, infault, or outfault directly references an XML Schema element. This approach centers an exchange of documents. Although still supported RPC-style operations are not focussed anymore.

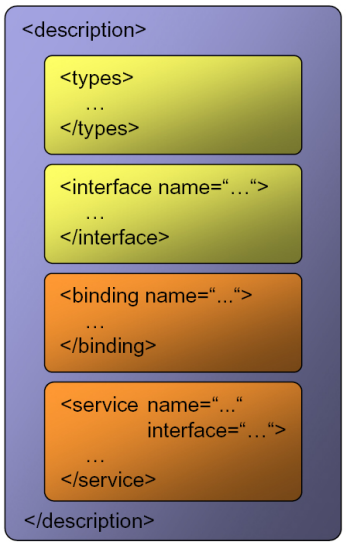

Figure 2.4.: General structure of a WSDL 2.0 document

The declaration of interfaces became more comfortable. An interface can now extend other interfaces as known from object-oriented programming languages. In addition, it is possible to define `<fault>`s on the interface level to be reused in many operations.

Operations are not restricted to one of the four simple types anymore. They can rather consist of an infinite, unordered number of inputs, outputs, in- and outfaults. Each operation refers to an MEP (by the means of an IRI[14] encapsulated in

[14]Internationalized Resource Identifier

the `pattern` attribute) that prescribes the order and cardinality of the messages and faults. WSDL delivers a "Template for Message Exchange Patterns" [CHL+07], a semiformal notation for describing an MEP's course of action.

Up to now, the W3C gives the definition of eight MEPs [CHL+07] [Lew07]:

- *In-only* – The service receives a message.
- *Robust In-only* – The service receives a message. If the received message triggers a fault, fault information are sent in response.
- *In-optional-out* – The service first receives a message and optionally produces a message in response.
- *In-out* – The service first receives a message and produces a message in response.
- *Out-only* – The service sends a message.
- *Robust Out-only* – The service sends a message. If the sent message triggers a fault, fault information are received.
- *Out-optional-in* – The service first sends a message and optionally receives a message in response.
- *Out-in* – The service first sends a message and receives a message in response.

WSDL permits the extension of this list by self-defined MEPs. The W3C recommends applying the WSDL MEP template for the specification of MEPs.

The concept of bindings was redesigned. Typically, a binding refers to an interface. But it is also possible to omit the reference to an interface for defining reusable bindings (i.e. they can be used for more than one interface). The downside of this reusability is that no operation-specific binding details can be given. Therefore only bindings with suitable rules on the interface level are candidates for reusable bindings.

In the end, the relation of services to endpoints has changed. A `<service>` now points to an interface and contains one or more endpoints, all implementing the same interface but possibly with different bindings.

2.4. Business Process Management

Business processes define actions to perform, rules to follow, paths between actions to take, and the data and resources needed to achieve a business goal (e.g. the building or sale of a product). A business process is a model that can be carried out several times. A single execution is called a *process instance*. Business process management (BPM) deals with business processes as enterprise resources [LR00]. It is about process discovery, optimization, analysis and reengineering.

A (part of a) business process running on a computer is referred to as *workflow*. A single execution of a workflow model is accordingly a *workflow instance*. Workflow management

systems (WfMS) are middleware products that support running workflows with all its aspects (user management, staff queries, work items, running different kinds of activities, auditing, and much more).

In this work, the terms *process* and *workflow* are used as synonyms for process and workflow models. The context will reveal the actual meaning. When talking about concrete executions of these models, the terms *process instance*, *workflow instance*, or just *instance* are taken.

Workflows (or processes, respectively) can be seen as three dimensional cubes [LR00]. The first dimension describes *what* actions are performed in which order (process logic). Second, the organization dimension says *who* carries out the behaviour (e.g. a human user, or the WfMS itself). So-called *staff queries* are used to select the performer that, for instance, must meet a specific role, position, and so on. The infrastructure dimension declares *which* resources are utilized to execute a step. This way the business process logic is separated from the concrete implementation of business functions.

2.5. BPEL

The Business Process Execution Language (BPEL) is an XML-based notation to specify machine readable business process behaviour upon Web services (WS) [ACD+03]. It is part of the WS standard stack, enjoys broad attention in industry and research, and became the de facto standard for composing Web services in recent years. BPEL realizes a *recursive aggregation* model [Ley07]. A set of services are linked together to be provided as a new Web service (aggregation). This new service can again be a part of a Web service composition (recursion). This leads to more sophisticated services on a higher level [WCL+05].

The orchestration of Web services can be seen as a kind of programming. Therefore it is also called *programming in the large*. As opposed to *programming in the small*, program code does not need to be written to get a piece of software exposing its functionality. Convenient graphical BPEL editors allow the user sticking Web services together without even getting touched with the BPEL file that is built in the background.

BPEL makes use of several XML specifications like WSDL 1.1, XML Schema[15] 1.0, XPath[16] 1.1, and XSLT[17] 1.0. Especially the coupling to WSDL as its interface definition is very tight as will be recognized in the next section.

The BPEL concepts can be used to define both, abstract and executable processes, described in sections 2.5.2 and 2.5.3.

[15] An XML-based meta language to define structures of XML documents

[16] XML Path Language – a rich expression language to navigate through the trees of XML documents

[17] eXtensible Stylesheet Language Transformation – a programming language to carry over an XML document from one structure into another

2.5.1. BPEL Constructs

In this work, BPEL is used in version 2.0 ([AAA+07]). In the following the most important BPEL 2.0 constructs are presented. Special attention is given to constructs that play an important role in this diploma thesis. Whenever the abbreviation WSDL is mentioned in this section, it refers to version 1.1.

Standard-attributes The standard attributes comprise the `name` and `suppressJoinFailure` attributes of activities. The former one is self-describing. The latter says whether a join failure (i.e. the activity's join condition evaluates to `false`) is to be thrown (`no`) or suppressed (`yes`). If it is suppressed, dead path elimination must be performed when a join failure occurs.

Standard-elements The standard elements encompass the incoming (i.e. target) and outgoing (i.e. source) links of an activity. Links are used to specify synchronization relationships between activities. A join condition can be specified for the set of target links that says whether the activity is to be executed. Each source link can have a transition condition that denotes whether the control flow is passed along the link.

Scope A `scope` is a container for a single activity and several definitions (variables, partner links, correlation sets, message exchanges, as well as event, fault, compensation and termination handlers). The outermost scope is called *process scope*.

Variable Variables are data containers. There are three possible types a variable can be of: (1) WSDL message, (2) XML Schema type (simple or complex), or (3) XML Schema element. Variables can represent message data (received by a partner or to be sent to a partner) or internal data used for calculations, counters, and so on.

Partner Link (Type) Partner link types are declared in a process's WSDL as WSDL extension. They are the glue between two parties defining the port types each service has to provide during the conversation. In case only one partner invokes another partner via request-response operations, the invoking service does not need to provide a port type to receive the answer at as opposed to asynchronous messaging where this is required. Partner links are instances of partner link types used to express peer-to-peer relationships to partners a BPEL process interacts with. They select port types (and indirectly messages) for the communication in both directions.

Receive This `receive` activity catches a single incoming message. It references a partner link, WSDL port type and operation implementing an operation's input message. If the `createInstance` attribute is set to "yes", it instantiates and starts a new process instance.

Pick Pick activities define a set of events waiting for the occurrence of exactly one of it. After selecting an event, all others are disabled. A `pick` can act as a `createInstance` activity. There are two types of events:

- `<onMessage>` waits for the receipt of an incoming message (similar to a `<receive>` activity). Therefore a partner link, port type and operation is specified.
- `<onAlarm>` is a timer-based alarm firing at a specified deadline (`<until>`) or after a given time (`<for>`).

Event Handler The event handler is semantically similar to the `pick` activity but is attached to a `<scope>`. It offers the possibility to react on events that can occur alongside the usual control flow. The events can happen concurrently and recurring as long as the surrounding scope is active. Event handlers must not create new process instances. There are two types of events:

- `<onEvent>` is an inbound message corresponding to a WSDL operation.
- `<onAlarm>` is a time-driven event that can happen after a period of time (`<for>`), at a given deadline (`<until>`) or repeatedly (`<repeatEvery>`).

The `<receive>`, `<onMessage>` and `<onEvent>` together build the set of *incoming message activities*.

Reply The `reply` activity is used to send a response to a partner in a synchronous request-response interaction. It references the same partner link, port type and operation as the corresponding incoming message activity implementing the output of a WSDL operation. The `reply` comes in two forms. First, as a normal response – the `faultName` attribute is omitted. Second, with present `faultName` it represents a WSDL operation fault.

Invoke The `invoke` activity is used for calling operations of partner Web services. These operations can be an asynchronous one-way (no `outputVariable` is given) or a synchronous request-response (`outputVariable` is specified). In the asynchronous case the partner's answer can be fetched by a corresponding `<receive>` activity implementing a one-way operation on the process side.

Assign BPEL does not support an explicit data flow. Instead an `assign` activity is introduced with the purpose to copy data from one variable to another. With the help of expressions data can be created (as `<literal>`s) and manipulated (with XPath). It is also possible to copy EPRs from and to partner links. The `assign` comes with an extension mechanism: the `<extensionAssignOperation>` facilitates user-defined operations under its own namespace.

Extension activity The `extensionActivity` is a mechanism to include new activity types into a BPEL process in their own namespace.

Flow The `flow` activity is a container for other activities that might get executed in parallel. It is possible to define `<link>`s that set activities into an execution dependency. This leads to a graph-structured modeling style.

Sequence The `sequence` is a container for activities that are processed in sequential order. This results in a block-structured modeling style.

Besides the `<sequence>` there are even more activities that support a block-structured design (e.g. `<if>`, and `<while>` activities). Both structure styles can be used intermixed in a single process model.

Fault Handler Fault handlers are attached to `<scope>`s. They catch occurring faults, terminate the normal process flow and execute an alternative, custom-defined logic to handle the exception. There are two constructs a fault handler can consist of:

- A `<catch>` fetches a fault by QName and possibly also by variable, message type or element.
- A `<catchAll>` fetches all faults that have not been caught by any `<catch>` of the fault handler.

2.5.2. Executable Processes

Executable processes are fully specified. They contain all details needed to run instances of process models on a workflow engine, that is, variable types, attributes, expressions, and business logic. Often, executable processes reflect the internals of an enterprise and hence are a company secret. Inferred from the rule *processes = products* revealing process models can lead to losing differentiation from competitors.

2.5.3. Abstract Processes

Abstract processes, also referred to as business protocols or message exchange protocols, describe externally visible behaviour only (i.e. the interaction between partners), without exposing the internal logic. Thus an enterprise can present an abstract process of an existing business process to let its partner know how to interact with it. In this way an abstract process works as a projection of an executable one. Another use case for abstract processes is to work as a reusable template for recurring problems. Business process designers that want to make use of the functionality just have to derive an executable process of it.

Abstract processes are explicitly marked abstract. It is permitted to have an abstract process containing complete information as if it were executable. But its abstract status allows any realizations of it to perform additional steps adapting the process.

Common Base The common base defines syntactic rules all abstract processes have to follow. An attribute `abstractProcessProfile` has to exist pointing to a profile that expresses a specific intent for the interpretation of an abstract process. All constructs of executable processes are permitted, but may be omitted with the help of two mechanisms:

- The use of explicit opaque tokens
  Opaque tokens are explicit placeholder for meaningful constructs in an appropriate executable process. There are opaque tokens for activities, expressions, from-specs of `assign` activities, and attributes.
- Omissions
  An omission is a shortcut to opacity being exactly equivalent to replacing the omitted element by an opaque token. Only elements may be omitted whose location can be deterministically detected and which have no default value (activities, expressions, from-specs, and attributes). Omitting an activity is only allowed by replacing it with an opaque token.

Abstract Profiles While the common base provides the syntactic framework for abstract processes, abstract profiles deliver the semantics of an abstract process saying how to interpret it. It may restrict the common base to a subset of allowed omissions/opaque tokens. Moreover, abstract profiles contain a URI the related processes point at via the `abstractProcessProfile` attribute, and a set of executable completions that prescribe the steps to be done to transform the abstract process into an executable one. Two profiles are predefined by the BPEL 2.0 specification ([AAA+07]), one for the observable behaviour, the other for process templates.

2.6. BPEL^{light}

BPEL^{light} [NLKL07] is a BPEL 2.0 extension developed at the Institute of Architecture of Application System (IAAS) at the University of Stuttgart. It is a recent field of research addressing BPEL's most disadvantageous property: its tight coupling of the process logic ('what' dimension) with WSDL 1.1 as interface definition ('which' dimension). This coupling results in three downsides.

First, although BPEL ought to close the Business-IT-Gap [Ley07], it is very IT-weighted and hence difficult to use for business people. A separation of roles during the modelling phase of process models is hampered. A modeller of a BPEL workflow must also understand the concepts of WSDL 1.1.

Second, BPEL's coupling with WSDL 1.1 makes it impossible to use any other IDL[18] (e.g. WSDL 2.0, or SSDL). This influences not only the process model itself (e.g. by activities pointing to WSDL 1.1 port types and operations) but also the discovery and selection of partner services. These must implement a WSDL 1.1 interface that accompanies with the process model's interface concerning the port type and operation names as well as operation types. Partners using another IDL or a WSDL 1.1 with other port type or operation names are rejected a priori. Additionally, the operation types must fit exactly. For example, a workflow implementing a request-response operation is imperatively linked to a partner that uses a request-response operation. Although partners that use and provide a single one-way operation would potentially match they are considered incompatible and have no chance to communicate with the given process.

Third, it inhibits the reusability of (parts of) process models in different contexts. Imagine a workflow model realizing a simple request-response operation – a recurring use case. Unless this model is adapted, it cannot be reused since its receive and reply activities are bound to a specific, case dependent port type, operation and – implicitly – partner link type.

A decoupling of process logic from interface definition addresses these issues. At this point BPEL^{light} plugs in. It makes use of a business centric interaction model that is completely independent of WSDL 1.1 ("WSDL-less BPEL" [NLKL07]) realized by BPEL's extension mechanism. That way, it is especially independent of WS technology and fosters a flexible and reusable modelling and running of business processes. Any IDL can be used to publish BPEL^{light} process models. In particular, even the partner services do not have to be described in terms of interface definitions as opposed to the WSDL 1.1 communication model [NLKL07]. It is sufficient to describe how the process wants to interact with one or more partners in terms of a message exchange. This reduces the assumptions that are made about a partner. The mapping of the process model to an appropriate IDL is done during deployment or even runtime, depending on the supporting middleware.

2.6.1. Interaction Model

BPEL^{light} defines a new interaction model by a single activity (`<bl:interactionActivity>`) that unifies all basic interaction activities of BPEL (`<receive>`, `<reply>`, and `<invoke>`) without referencing any WSDL elements. In fact, BPEL's `<pick>` also gets a WSDL-less counterpart by a BPEL^{light} `<bl:pick>` as well as the event handler's `<onEvent>` by a `<bl:onEvent>`. Additionally, IDL-independent constructs declare which interaction activities belong together as logical unit (`<bl:conversation>`) as well as which partner is addressed by a single message (`<bl:partner>`). The BPEL^{light} constructs are incorporated into the BPEL process under a separate namespace[19]. BPEL's `<extensionActivity>` mechanism is used to introduce the new activity types. Hence, BPEL^{light} is a syntactically correct BPEL 2.0 dialect.

[18] Interface Definition Language

[19] `xmlns:bl=http://iaas.uni-stuttgart.de/bpel-light`

Conversation

The <bl:conversation> element can be seen as a WSDL-less partner link that is not connected to a partner link type. In contrast to partner links, a conversation is not limited in representing a communication between only two parties. Conversations are a powerful frame message exchanges involving one or more partners over one or more messages are running in. Similar to a partnerLink that is declared within a <partnerLinks> element, a conversation gets defined in a <bl:conversations> element in a <process> or <scope> (see listing 2.2 for the latest definition of a conversation as presented in [LLN08]). All activities or elements that step into communication with a partner service

Listing 2.2 BPEL^{light} conversation definition

```
<bl:conversations>
   <bl:conversation name="NCName" />*
</bl:conversations>
```

must reference a conversation (<bl:interactionActivity>, <bl:onMessage> of the pick, and <bl:onEvent> of the event handler; referred to as *messaging elements*). All messages grouped by a conversation can be considered a (complex) message exchange. The usage of conversations instead of partner links changes the expression of requirements a partner service has to meet to communicate with the process: these are no longer expressed by WSDL port types, but instead by the ability of sending and receiving messages as demanded by the conversation.

Partner

BPEL's partnerLink does not only represent an interaction between two parties, but is also a container for an EPR pointing to a partner's location. In BPEL^{light} the abstract partner construct takes over the latter task. Like conversations, partners are defined within a <process> or <scope> (see listing 2.3 for the latest defintion of a partner as presented in [LLN08]). Partners are implicitly connected to conversations by BPEL^{light}'s messaging

Listing 2.3 BPEL^{light} partner definition

```
<bl:partners>
   <bl:partner name="NCName" />*
</bl:partners>
```

elements. Each partner participates in one or more conversations; each conversation involves one or more partners. In [LLN08] the runtime representation of a partner is refined. A partner can be regarded as variable (i.e. container): it encompasses a name, data type and instance value (i.e. a partner instance).

An example for a partner instance is given in listing 2.4 on the facing page. It contains a number of EPRs – all following the same pattern. Note that an EPR is not restricted to the child elements wsa:Address and wsa:ServiceName. Arbitrary EPRs are possible as long as they adhere to the WS-A specification. Basically, partner instances consist of a list of service

Listing 2.4 Example for a partner instance

```
<partnerInstance
     xmlns:sref="http://docs.oasis-open.org/wsbpel/2.0/serviceref"
     xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">
  <current>
     <sref:service-ref id="1">
        <wsa:EndpointReference>
           <wsa:Address>http://www.partnerA.com/endpoint</wsa:Address>
           <wsa:ServiceName>pa:serviceA</wsa:ServiceName>
        </wsa:EndpointReference>
     </sref:service-ref>
  </current>
  <list>
     <sref:service-ref id="1">
        <wsa:EndpointReference>
           <wsa:Address>http://www.partnerA.com/endpoint</wsa:Address>
           <wsa:ServiceName>pa:serviceA</wsa:ServiceName>
        </wsa:EndpointReference>
     </sref:service-ref>
     <sref:service-ref id="2">
        <wsa:EndpointReference>
           <wsa:Address>http://www.partnerB.com/endpoint</wsa:Address>
           <wsa:ServiceName>pb:serviceB</wsa:ServiceName>
        </wsa:EndpointReference>
     </sref:service-ref>
  </list>
</partnerInstance>
```

references (`service-ref`). Each of them contains a single EPR and is unambiguous referable by `id` attribute. One of the service references is assigned to be the current `service-ref`. When invoking a partner during runtime, the EPR of the current `service-ref` is taken to determine the endpoint to send the message to.

A `partner` is instantiated either statically (by a deployment descriptor) or dynamically (during runtime by manual intervention, discovery mechanisms or incoming messages containing an EPR). In both cases the associated partner instance is open to be adapted by appropriate `assign` operations.

Interaction Activity

The interaction activity plays the role of BPEL's `<invoke>`, `<receive>` and `<reply>` activities and furthermore introduces a new kind of behaviour. It is used to exchange messages with partners and as such realizes four use cases:

1. The interaction activity receives a message (like a BPEL `<receive>`).
2. The interaction activity sends a message (like a BPEL `<reply>` or one-way `<invoke>`).
3. The interaction activity first sends and then receives a message (like a BPEL synchronous/two-way `<invoke>`).

4. The interaction activity first receives and then sends a message (no counterpart in BPEL).

In listing 2.5 the interaction activity's syntax is given. The four mentioned behaviour types are syntactically reflected as follows. Interaction activities only receiving a message have to specify an output variable (case 1). Interaction activities only sending a message have to specify an input variable (case 2). For these two cases, the `mode` attribute is not evaluated. In use cases 3 and 4, the interaction activity has to specify both an input and output variable. The `mode` attribute's value then prescribes if it is case 3 (`out-in`) or case 4 (`in-out`). The `createInstance` attribute can be used for interaction activities that only or first receive a message (cases 1 and 4) to indicate that a new process instance must be created (value `yes`). Its default value is `no`. For the other cases, the attribute is ignored. The `partner`

Listing 2.5 BPEL^{light} interaction activity definition

```
<bpel:extensionActivity>
  <bl:interactionActivity inputVariable="NCName"? outputVariable="NCName"?
       mode="in-out|out-in"? conversation="NCName" partner="NCName"
       createInstance="yes|no"? standard-attributes>
     standard-elements
  </bl:interactionActivity>
</bpel:extensionActivity>
```

attribute says with which partner service the communication takes place. The `conversation` can be used to incorporate the interaction activity into a more complex interaction (e.g. consecutively an outgoing and incoming message followed by an acknowledgement).

Pick

BPEL's `pick` activity is required to have at least one `<onMessage>` element that corrupts the idea of BPEL^{light}. Therefore a new `pick` activity is introduced with similar semantics but with WSDL-less `<bl:onMessage>` elements. These elements reference a `conversation` and a `partner` just like the interaction activity.

Event Handler

BPEL's event handler is required to specify at least a WSDL-related `<onEvent>` or an `<onAlarm>` element. Thus extending the handler by an additional BPEL^{light} `<bl:onEvent>` would not allow to model a syntactically correct event handler with a single `<bl:onEvent>` only. Therefore BPEL's event handler is complemented by a BPEL^{light} event handler with WSDL-less `<bl:onEvent>` elements. Each `<bl:onEvent>` – similar to the interaction activity and pick's `onMessage` – points to a `conversation` and a `partner`.

Variable

Variables in BPEL^{light} should not be message-typed since message types are WSDL 1.1 constructs that would corrupt the separation of process models and their interface definition.

Assign

With its `assign` activity BPEL provides a mechanism to copy a partner link's EPR to another partner link. This operation is obsolete in BPEL^{light} as partner links are not needed anymore. Instead EPRs are held by a `partner`. Since these partners are considered variables, the assign activity needs no adaption for an appropriate `copy` operation. Copying EPRs from one partner to another or from a partner's EPR list to the current EPR is simply done by the expression variant of the `<from>` and `<to>` elements (see listing 2.6).

Listing 2.6 Assigning BPEL^{light} partners

```
<bpel:assign>
   <bpel:copy>
      <bpel:from>
         $partnerA/list/service-ref[@id=2]
      </bpel:from>
      <bpel:to>
         $partnerA/current
      </bpel:to>
   </bpel:copy>
</bpel:assign>
```

2.6.2. BPEL^{light} in the WS environment

When using BPEL^{light} in the WS environment it emulates BPEL's communication semantics [NLKL07]. The 'what' and 'which' dimensions remain decoupled but an external mapping between the technology neutral BPEL^{light} interaction model and WSDL is defined. Three possibilities are depicted how this mapping can take place: (i) by an external file (see listing 2.7 on the next page), (ii) by the attachment of WS-Policy information, or (iii) by passing the issue to the underlying middleware (e.g. the ESB).

Bilateral conversations (i.e. conversations being used to communicate with a single partner) take over the part of partner links and are associated with a partner link type. A `myRole` and `partnerRole` can be assigned to each conversation to express if the process is invoked (my role) or if the process invokes a partner service (partner role). Furthermore each BPEL^{light} messaging element must be mapped to a WSDL operation depending on if the message is incoming or outgoing and if the message is a one-way operation or if it belongs to another message with opposite direction (request-response operation).

Listing 2.7 Binding BPEL^{light} to WSDL 1.1

```
<assignmentFor process="QName">
  <conversation name="NCName" partnerLinkType="QName" myRole="NCName" partnerRole="NCName"
      />*
  <activity name="NCName" operation="NCName" />*
</assignmentFor>
```

2.6.3. Formalizing MEPs

In section 2.3.3 on page 11 the generic approach of WSDL 2.0 for describing operation types (i.e. MEPs) by a semiformal notation was presented. In [NLL08, LNL08, NHST08] this notation is discussed. The authors come to the conclusion that it is not powerful enough to describe arbitrary MEPs. Four downsides are stated:

1. The notation is not machine-readable. In the general case, the implementation against operations needs human interpretation and intervention.
2. Its expressivity is limited concerning how to involve several instances of a node.
3. It is not precise enough in expressing optional messages. The user of an operation cannot find out when and if at all an optional message arrives.
4. Only sequential message processing without conditions can be described.

A consequence of BPEL^{light}'s idea to separate process logic and interface definition is that it can not only describe the flow *between* but also *within* Web service operations. Since MEPs are abstract definitions free of concrete data types, BPEL's approach of declaring abstract profiles is used to create an "Abstract Process Profile for Message Exchange Patterns" [LNL08] for BPEL^{light} process models (referred to as *MEP profile*). The profile is associated with an appropriate namespace[20] and the prefix `mep`.

The MEP profile restricts the common base as follows:

- Omission shortcuts are only allowed for timing definitions (i.e. `<for>`, `<until>`, and `<repeatEvery>` in `<onAlarm>` and `<wait>` elements). When omitting a timing definition, a `<mep:timingExpression>` element must be set at its place. The `timingExpression` is defined below.
- Opaque tokens are only allowed for variable references, types and time constraints.
- Faults are explicitly marked by the `faultName` attribute of receiving interaction activities and the `pick` activity's `<bl:onMessage>`.

For a detailed description of the MEP profile the reader is referred to [LNL08]. Abstract MEP processes adhering to this profile address the four disadvantages mentioned above.

[20] `http://iaas.uni-stuttgart.de/bpel-light/abstract/mep/2008/`

Timing Expression The MEP profile's timing expression can replace BPEL timing definitions in `<onAlarm>` and `<wait>` elements to obtain a single point of variability. The `name` uniquely identifies the expression. The values of attributes `type` and `expression` can be abstracted by opaque tokens (similar to the type of variables).

Listing 2.8 Timing expression of the MEP profile

```
<mep:timingExpression name="NCName" expressionLanguage="anyURI"?
    type="for|until|repeatEvery" expression="expr"/>
```

BPEL^{light} MEPs in WSDL 2.0 Abstract BPEL^{light} processes can be used to describe WSDL 2.0 operation types replacing WSDL's template for MEPs. To reference a BPEL^{light} MEP from within a WSDL 2.0 document the WSDL operation's `pattern` attribute points to the abstract process via an IRI that consists of the `targetNamespace` and `name` of the process (e.g. "http://www.example.com/meps#in-out"). A mapping between the operation's messages (`<input>`, `<output>`, `<infault>`, and `<outfault>`) and the MEP's messaging elements have to be defined. A WSDL message belongs to a BPEL^{light} messaging element if the message's `messageLabel` attribute and the activity's `name` attribute contain the same value. There are no means in WSDL to configure the MEP profile's timing expression. Therefore a `<mep:configure>` element is introduced in [LNL08] as extension of the WSDL's `<operation>` element. It allows replacing opaque values of attributes and elements in an MEP process by using XPath for identifying a node.

2.7. Related Work

There are some other approaches of formalizing MEPs or service interactions, respectively. In [DPW06], for example, the π-calculus is used to formalize a subset of service interaction patterns (see [BDH05]). The π-calculus is an algebra for formal description, analysis and verification of interaction models.

The SOAP Service Description Language (SSDL) [PW05] is an XML-based framework for describing Web service interactions on the basis of SOAP messages and WS-A. It focuses on message-oriented contracts between Web services to define arbitrary complex MEPs from a service's point of view. Similar to BPEL^{light} SSDL provides means to replace WSDL MEPs. SSDL is grounded on a composition model around four protocol description frameworks (Message Exchange Pattern, Communication Sequential Processes, Rules and Sequencing Constraints Framework). However, SSDL goes the way of defining a new service description language while BPEL^{light} committed itself to use established standards like WSDL or BPEL. The benefit of the latter approach lies in the sustainment of existing technologies and tools.

Chapter 3

Orchestration Director Engine

The Apache Orchestration Director Engine (ODE) [ODE] is an open-source workflow engine under the patronage of the Apache Software Foundation[1]. ODE runs business processes written in BPEL. It sends messages to and receives messages from Web services, allows complex interactions by correlation mechanisms as well as transactional support by fault handling.

ODE supports both WS-BPEL 2.0 OASIS standard and the legacy BPEL4WS 1.1 vendor specification. There are two communication layers. The first is based on Axis 2, a SOAP implementation. The second works with JBI[2] implementations (e.g. Apache ServiceMix). An API[3] allows to integrate the engine with theoretically any communication layer.

Besides these technical properties ODE provides a management interface for handling processes, instances and messages. Two command line tools are integrated, one for the compilation of a given deployment bundle, another for sending SOAP messages to a destination.

In this work, ODE is the underlying workflow engine that is going to be adapted to understand and execute BPEL^{light} processes supporting MEPs. This chapter gives an insight into ODE's architecture, interfaces and capabilities.

3.1. Architecture

Figure 3.1 on the next page illustrates ODE's high level architecture. Four main components can be outlined by different colors. The compiler (green) translates a deployment unit (DU) into a format especially prepared for the runtime (blue) to process the model. The runtime is ODE's core and thus the major component. It contains the engine that runs process models according to their definition, data access objects as interface to the persistence layer (purple), and an API that allows different communication layers to plug in and that provides an interface for management functions. The persistence layer provides functions to

[1] A community of developers and users of open-source software projects (`http://www.apache.org/`)
[2] Java Business Integration – A standard-based architecture for integration solutions
[3] Application Programming Interface

store information about processes, instances and DUs in the underlying database (DB). The communication layer (orange) exposes ODE's functions to the outside. Service partners can interact with and are invoked by the engine via the communication layer. Furthermore it provides access to the management functions of ODE. Two implementations are delivered, Axis 2 and JBI, but it is possible do add any other messaging framework.

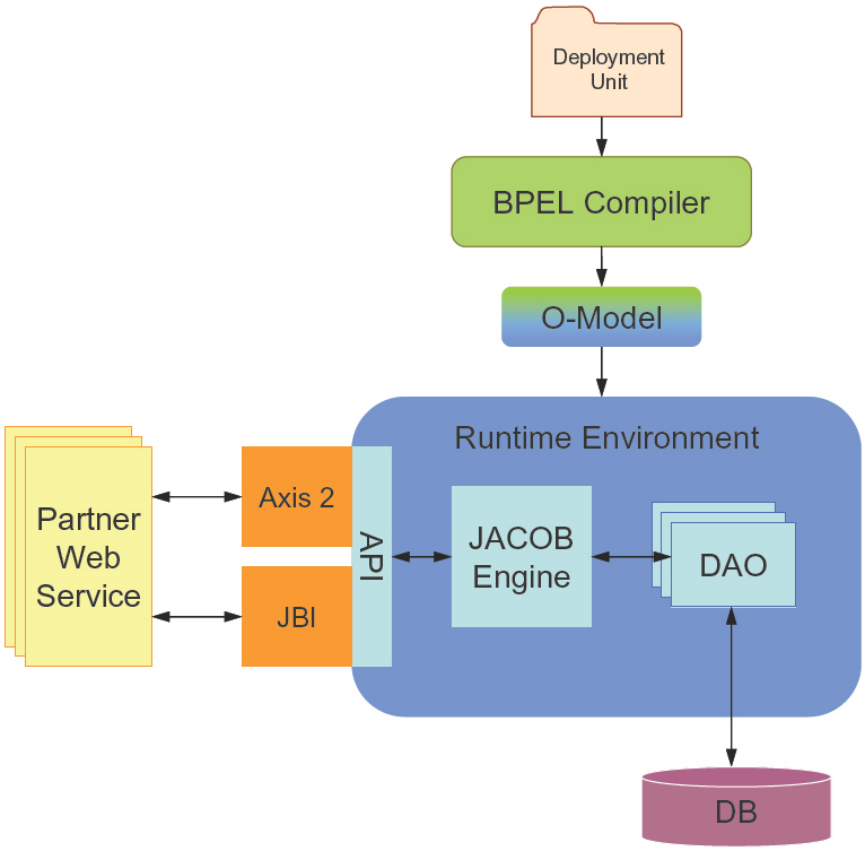

Figure 3.1.: High level architecture of ODE

The architecture orientates itself towards the WfMC[4] Workflow Reference Model (WRM) [Hol95]. In the following sections the single components are presented in detail while a relation to the WRM is established.

3.2. Compiler

The compiler translates a BPEL process into a representation that is understood by the engine and that supports its execution. It can be compared to the WRM's *Interface 1* allowing

[4]Workflow Management Coalition – a nonprofit organization defining standards for WfMSs to achieve interoperability between different WfMS implementations

business modeling tools to provide an input (i.e. a DU) for the WfMS. A modeling tool must provide a DU of the structure given in figure 3.2:

- *BPEL* – One or more BPEL process models in both versions, BPEL 1.1 and 2.0, can be contained. The main process must have the same name as the DU. The other processes can be considered (local) sub- or partner processes.
- *WSDL* – The processes can make use of one or more WSDL 1.1 documents revealing their interfaces.
- *XSD* – One or more XSD files can be referenced by the processes and WSDLs containing the data types for messages and variables.
- *DD* – The deployment descriptor (DD) contains information directed to the compiler saying how to make the process runnable in the engine. This encompasses an activation flag, a mapping of the process's service(s) and partner services on partner links as well as the definition of process interceptors (e.g. for a limitation of the number of process instances for a single process).

The compiled process model is an object representation (thus, the name *O-Model*) of the BPEL file. It is stored to the file system to prevent the process from duplicate compilation after a system reboot. The O-Model is used by the engine during navigation to access static process model constructs like variable names, partner link names, activities and so on. Therefore, in the architecture illustration it is colored both green and blue as it is created during compilation and used during runtime.

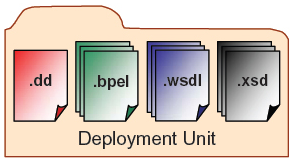

Figure 3.2.: ODE's deployment unit format

After translating a process, it is made persistent in the DB and the publishing of its provided services is delegated to the BPEL server of the runtime component. These three steps, compilation, DB storage and service publishing, are referred to as *deployment*. After the deployment, the process is executable – it can be invoked to carry out its behaviour.

ODE provides a mechanism that is called *hot deployment*. This term denotes that a process can be deployed while the workflow engine is running. A reboot is not necessary. The compiler is addressable via an API that allows to implement different front-ends [Bro07]. Two compilation mechanisms are integrated: one for copying the DU into an appropriate directory (depending on the communication layer), and a command line tool that only compiles a process without deploying it. The latter can be used to just perform a statical analysis of a process and its related documents (i.e. a sanity check).

3.3. Runtime Environment

The runtime environment (or engine, respectively) corresponds to the *WFMS Engine* term of the WRM. It is the core component in a WfMS, so in ODE. The engine drives compiled processes: it creates instances of process models, navigates through them, handles pieces of

data, implements and executes the activities' behaviour (e.g. calling a Web service, managing DB data or the control flow). Moreover it routes incoming messages to existing process instances (so-called *correlation* mechanism) and performs fault handling on behalf of the process.

The runtime component supervises the processes' life cycles. Each process instance goes through different states in its life (e.g. *created* – after it is built; *running* – during its execution; or *finished* – after its completion; and much more). However, in ODE a user can explicitly request a state change with the help of the management interface, for instance, to suspend, resume or terminate a process instance, or to retire or activate a process model. The management functions are comparable to *Interface 2* of the WRM as it makes functions available for users to interact with the WfMS. Besides the state management ODE provides, for instance, user operations for querying processes and process instances as well as getting insight into variable instances and changing variable values manually (e.g. as recovery actions by an administrator).

To publish the management functionality for the user the communication layer implementations must provide an interface for it. The Axis 2 layer does this by exposing the management component as Web service, JBI provides it as JBI component.

The runtime only uses the abstract parts of a WSDL, namely the types, messages, port types and operations. But there is one fact weakening this concept: in the DD the endpoints a process provides (and in case of a static binding also the endpoints the process invokes) must be mapped on adequate partner links. The mapping is used to forward an incoming message to the appropriate partner link implementation which dispatches the message to the correct process instance, or creates a new one if necessary.

ODE implements in parts the WRM's *Interface 5* that defines the audit trail's structure and the events that may occur. An audit trail is a record saving entries about events that take place while running a process instance (e.g. the start of an activity). This can be due to legal requirements or as basis for a process reengineering, for example. However, via the management functions it is possible to query a set of BPEL events happened during process execution (e.g. process instance created, or variable modification event). The granularity of the event generation can be adjusted in the DD to select the events that are to be written into the audit trail. But the audit trail is not recorded into a separate DB – it is strongly connected to the associated process instance.

The WfMC suggests an *Interface 4* in its WRM for handling sub-processes between different *Workflow Enactment Services* (i.e. WfMS implementations). The interface should define functions, for instance, to start a sub-process or to query its state. However, ODE does not explicitly provide this kind of interface. Instead such an interface might be implicitly defined by a sophisticated process model exposed as Web service that acts as a wrapper for sub-processes controlled by another WfMS implementation.

Data Access Objects To reliably execute process instances storing instance data persistently is indispensable. ODE's data access objects (DAOs) build an interface to the persistence layer, typically a transactional relational DB. Two different persistence layers are provided by ODE

but the design allows integrating custom DAO implementations. The information persistently saved include process instance data, variable values, EPRs in partner links, message routing information and process execution states.

However, besides these two persistent DAO implementations ODE also provides a non-persistent approach, namely in-memory processes. A process must be explicitly signaled in the DD to be executed in-memory. In this case running a process is realized by in-memory objects holding the "stored" information in the runtime environment. On the one hand this speeds up process execution due to the lack of expensive DB accesses. On the other hand the pieces of data are lost in case of a system crash. There are some more limitations on in-memory processes, for example, their instances cannot be queried by the management functions and they are only allowed to include exactly one `<receive>` activity, namely the one creating the instance (i.e. only short running, synchronous processes are allowed, so-called *microflows*).

JACOB The engine's BPEL implementation relies on the JACOB (JAva Concurrent OBjects) framework [ODE], a message-passing virtual machine. It addresses two major issues, making the execution state persistent and allowing a parallel execution of BPEL constructs. By leaving these tasks to JACOB the implementation of BPEL elements can concentrate on BPEL logic and does not have to deal with the surrounding infrastructure. JACOB achieves concurrency in a single thread by using communication channels for realizing the control flow (i.e. the communication between activities). The *TerminationChannel*, for instance, is used to pass the control back to an activity's parent when the activity has completed. The parent then decides how to proceed, for example starting the next child activity of a `<sequence>`.

3.4. Integration Layer

The integration layer (IL) qualifies ODE to interact with the outside world. On its own ODE is not runnable. It needs to be incorporated into an execution environment to be configured and started. This is satisfied by an IL. It is primarily an abstract term embracing which interfaces must be implemented and what tasks must be met by a concrete IL instance. Amongst others these are sending and receiving of messages, controlling the runtime's life cycle, initiating the deployment of process models, or publishing the management functions. As opposed to the runtime, the IL uses the WSDL's concrete part, that is, binding and service information, to make services available and to establish communication channels for interacting with partners. It fits to *Interface 3* of the WRM by invoking activity implementations (i.e. Web services).

ODE comes with two IL implementations, Axis 2 and JBI, described in the following.

Axis 2 Apache Axis 2 is a SOAP implementation supporting SOAP versions 1.1 and 1.2 as well as REST[5]-style Web service invocations. It comes with a lot of features that are in parts

[5] REpresentational State Transfer – the architecture behind the World Wide Web

interesting for this work, for instance, a support for WS-A, asynchronous Web service calls and WSDL 2.0. The Axis 2 IL makes it possible to embed ODE as web archive (WAR) in a servlet container, for instance, in Apache Tomcat, an open-source implementation of the Java servlet and Java Server Pages (JSP) technologies.

JBI Java Business Integration (JBI) is a standard-based architecture for EAI and B2B integration [THW05]. Adopting the concepts of SOA it provides an infrastructure for plugging in decoupled components (i.e. services) allowing them to interoperate in a reliable fashion. The component interaction relies on a mediated message exchange model that is based on WSDL 1.1 and 2.0. The JBI IL incorporates ODE as JBI component into a JBI container like Apache ServiceMix, an open-source ESB with integrated JBI components for HTTP, JMS[6], SOAP, etc.

3.5. Persistence Layer

The persistence layer's task is primarily to back up volatile (i.e. in-memory) process and process instance data to prevent a loss of information in situations that require a system reboot. Typically this done by a database management system (DBMS) that also provides transactional support and concurrency control but can be done by any other approach, for instance, simply storing the information as files to the file system.

In ODE the DAOs work as interface to the persistence layer. This design abstracts from concrete persistence layer implementations and allows plugging in custom persistence mechanisms. ODE delivers two DAO implementations. The first uses Hibernate, an open-source object-relational persistence framework [Hib]. It allows the development of object-oriented classes that encapsulate the DB operations. The second is based on Apache OpenJPA, a JEE[7] persistence project implementing the Java Persistence API (JPA) to persistently store Java objects [Ope]. Similar to Hibernate these objects can be used like normal objects while the DB accesses are running in the background. Both implementations make use of a Derby DB.

3.6. BPEL Extensions

ODE does not only implement the BPEL specification but also extends it in certain issues. A unique session identifier is introduced that is associated with each partner link instance. When sending a message over a partner link, its identifier is sent along with the message to be used by the recipient in follow-on messages to find the corresponding process instance (*implicit correlation*).

[6] Java Message Service
[7] Java Platform, Enterprise Edition – The industry standard for implementing SOA and web applications

ODE separates BPEL faults from system or communication channel failures. In case of a failure (e.g. an incorrect DNS resolution) a process instance does not terminate. Instead it joins a failure state and waits for administrator interventions. An administrator may fix the problem and tell the engine to retry the waiting activity. In contrast to that, BPEL faults cause a process instance to terminate.

There is support for external variables. These are, for instance, records in a DB, or REST resources. External variables can be used in expressions and assignments and can ease data sharing between a process and external systems.

Besides the message payload ODE also stores message headers. This allows a process model to access headers, assign them to other messages, or even execute header-dependent business logic.

ODE adds support for XPath 2.0 [BBC+07] as expression and query language and offers a few utility functions for use in `assign` activities. These extension functions are defined under a specific namespace[8] and associated with the prefix *ode*. There are functions to delete, insert, or rename nodes in XML documents.

[8] `http://www.apache.org/ode/type/extension`

Chapter 4

Concept and Specification

This chapter first introduces the concept of MEPs in a general BPM environment. After that, the BPEL^{light} specification is extended to match the outlined MEP concept. Furthermore, the workflow engine requirements are given. The chapter can be seen as the specification of the implemented workflow engine.

4.1. Using Message Exchange Patterns

As already mentioned MEPs are reusable templates simply prescribing which messages are exchanged with which partners in which order. In the context of BPM this means that an MEP is a process model without concrete message types and interaction partners (i.e. an abstract process; compare section 2.6.3 on page 24). When integrating an MEP in a concrete context (e.g. in a loan approval scenario), these abstract information must be made concrete. This procedure is called *parameterization*. The parameterization can take place at design time, deployment, or even runtime.

There are three scenarios how MEPs can be incorporated into a process model. First, the pattern is a part of the process. This is the obvious use case. Each activity is inherently related to a message sent or received by the engine. Consequently this activity has to be reflected in the interface description of the process. For example, when using WSDL 2.0, each activity has an incoming or outgoing WSDL message (or fault, respectively) as counterpart and hence belongs to an MEP. Second, the pattern is not part of the process (called *main process* in the following). It is rather outsourced into another process, an *MEP process*, and referenced by a special activity (an *MEP activity*). The MEP process can be seen as the description of this activity's behaviour. Third, the two scenarios can occur intermixed. An MEP consists of message sending/receiving activities and of MEP activities. However, interacting partners are not influenced by the choice of composing a process model of inline, external or intermixed MEPs. The process model's interface looks the same and the process is accessed the same way independent of the used MEP types.

The goal of using MEPs in the BPM context touches three aspects. First, modelling business processes can be simplified. Instead of choosing the different types of activities, placing them within the control flow and configuring them to meet the required needs to solve a particular

problem, a matching predefined pattern needs to be found, incorporated into the process and concretized (i.e. the usage of inline MEPs). Second, using external MEPs makes huge processes easier to read as chunks of the processes' logic are outsourced. Third, external MEPs enable the reuse of process model parts at runtime.

In the following, a closer look is taken on the three approaches. As far as possible the discussion is held apart from technical details or concrete languages. It focusses on forming a basic understanding of terms used in the course of this work.

4.1.1. Inline MEPs

These MEPs are inline from the process model's point of view. That means that a number of interaction activities (activities that send or receive a message) belongs together to be arranged as an MEP. These MEPs can be simple or complex. The activities' control flow dependency determines the message order. A single incoming and two outgoing activities can result, for instance, into an in-out-out, an out-in-out, or an out-out-in MEP depending on their arrangement.

Each interaction activity of the process belongs to exactly one MEP even if it is as simple as an out-only. Figure 4.1 shows a process model with three interaction activities arranged in a sequence, one of them incoming, and two outgoing. Activities A and C build an in-out MEP whereas activity B represents an out-only. That is, this process consists of two inline MEPs.

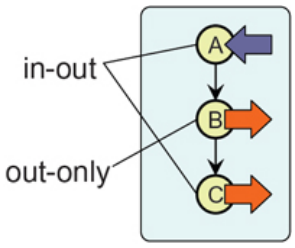

Figure 4.1.: A process model with inline MEPs only

The workflow's description language has to provide a mechanism to indicate which activities belong together. In BPEL this is done by `partnerLink` and `operation` for simple MEPs. In BPEL^{light} a `conversation` plays this role allowing arbitrary complex MEPs. In our example, activities A and C would belong to another `conversation` than activity B when using BPEL^{light} .

Inline MEPs can be reused at design time: an MEP definition can be integrated as often as needed while modelling a process. But there is no reusability at runtime. Imagine a process model containing two out-in MEPs somewhere in the control flow. These are then realized by two interaction activity pairs, that is, four interaction activities. No reuse takes place.

Inline MEPs must be parameterized at design time as they are incorporated into a concrete business process with a certain context. A parameterization during deployment (with the help of an eligible DD) would also work but makes little sense as the MEP statically belongs to the particular process. The benefit of less work during design is decreased by preparing the more extensive DD. Furthermore the deployment procedure will be more complex and time-consuming. Parameterizing inline MEPs at runtime is not possible since abstract process models cannot be deployed on a workflow engine.

4.1.2. External MEPs

This type of MEPs is external from the process model's point of view. Process models are thereby not self-contained. There is rather a single *main process* and one or more outsourced *MEP processes*. While the main process contains the business logic of the workflow (i.e. business conditions, staff queries, etc.), the external MEP processes should be just to transfer messages. Another difference between main and MEP processes is their degree of abstraction. While a main process contains concrete message types, expressions, and so on, MEP processes are abstract definitions that are enriched with concrete information in the course of their parameterization.

In contrast to the main process each MEP process implements exactly one MEP on its own. They are referenced by an MEP activity in the main process. At design time this activity prescribes the data types needed to parameterize the MEP process. At runtime the parameterized MEP process describes the MEP activity's behaviour.

In figure 4.2 a simple example for an external MEP can be found. The main process (A) consists of two MEPs. The inline in-out is realized by the interaction activities A and C. The external out-in is implemented by process B with its activities D and E. The MEP process B is referenced by activity B. In this way a reuse of MEPs during runtime is possible. Consider again the previous example of a process model containing two out-in MEPs. Realized as external MEP, the main process just needs two activities pointing to the out-in MEP process that carries out the actual behaviour of sending a message to and receiving a message from a communication partner. But there is one constraining requirement for this reusability. In both MEP process calls the same message types have to be passed from the main to the MEP process. Otherwise two different parameterized MEP processes would be addressed by the main process and hence no reuse would take place. Although the abstract MEP processes might be identical, parameterizing them with different message types leads to different concrete MEP processes.

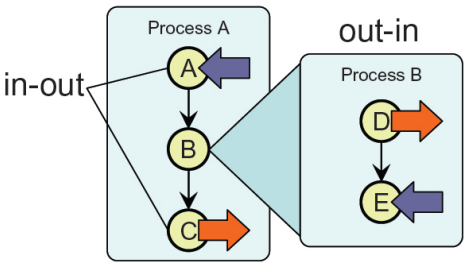

Figure 4.2.: A process model incorporating an external MEP

External MEPs can be parameterized at any time with different benefits and handicaps. As the main process exactly prescribes which message types have to be used to concretize the abstract MEP process, the parameterization is eligible for an automated operation. However, parameterization at design time, be it automated or by hand, brings the advantage of a faster execution at deployment or runtime since the work does not have to be done anymore. The downsides are more effort at design, a bigger DU that can contain a lot of concrete MEP processes, and especially a more complicated procedure when changing a message type (main process *and* MEP process would have to be adapted). Parameterizing the MEP processes at deployment requires a sophisticated deployment algorithm. It liberates the process designer of extra work. The DD needs no or little additive information since the necessary information can be found in the main process. The approach of parameterization at runtime is also imaginable but burdens the engine with a lot of time-consuming operations.

4.1.3. Intermixed MEPs

Intermixed MEPs are a special case bringing the two other approaches together. There is also a single main process and a number of MEP processes as in the external MEPs scenario. From the process model's point of view an intermixed MEP consists of interaction activities sending and/or receiving single messages as well as of MEP activities pointing to external MEPs. Thus, on the one hand, the behaviour of such an MEP activity is described by an external MEP process, but on the other hand, it is also a part of an MEP on a higher level. That means the process's interaction activities extend the external MEP by additional messages. Again a mechanism is needed to mark an arbitrary number of activities that belong together. BPEL^{light}'s `conversation` meets this requirement. In BPEL a `partnerLink` is used to group messaging activities – but restricted to only two parties.

This model allows for a recursive MEP composition: an MEP gets described by an external MEP process. Another process may extend it and can again expose its functionality as an MEP process. A hierarchy of related processes evolves comparable to inheritance structures in object-oriented programming languages: the inheriting object (here: the higher level process with an MEP activity) knows the extended object (here: the MEP process), but not vice versa. The extension stops at the level of the main process. It cannot be referenced by an MEP activity of another process due to a couple of reasons. First, it can implement more than one MEP. Second, it is a concrete process definition instead of an abstract one.

Figure 4.3 illustrates this issue. There is an MEP process (C) implementing an in-out with activities E and F. Process B, another MEP process, extends the in-out towards an out-in-out by prepending an outgoing message (activity C). MEP activity D, that references process C, must be explicitly marked to express its relationship to activity C. The main process A incorporates process B by MEP activity B and prepends activity A resulting in an in-out-in-out MEP. Besides this complex intermixed MEP, process A also implements the simple inline out-only MEP. It is obvious at first glance that process A cannot be an MEP process as those are not allowed to consist of more than one MEP. At this point, the inheritance structure finds an end. It is not possible to extend process A by another process, say D, without changing A.

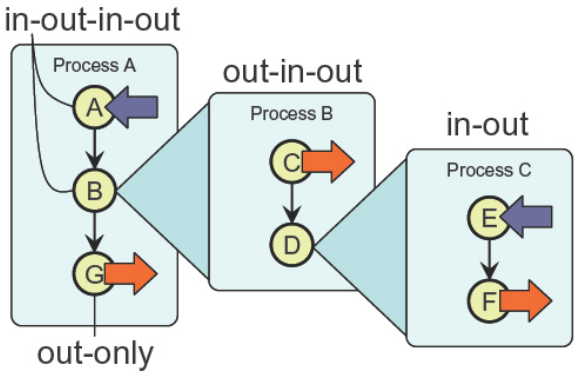

Figure 4.3.: A process model with intermixed MEPs

The usage of intermixed MEPs during design time can lead to a high degree of reusability of process models due to the recursive composition concept. Imagine there is already an MEP definition realizing a pattern sending notifications to a number of partners. Extending this pattern towards an *Atomic Multicast Notification* [BDH05] may be wished (a pattern that sends a number of notifications and waits for a number of responses meeting a certain criteria, for instance, at least five responses out of ten contacted partners). With the concept

of intermixed MEPs this Atomic Multicast Notification has not to be defined new. The MEP for sending notifications can be rather extended by creating a new MEP process with an MEP activity referencing the notifications pattern, and appending interaction activities and control flow constructs realizing the receiving of answers and a counter triggering the exit.

At runtime intermixed MEPs do not bring any significant advantages. Quite the contrary, it means much more effort to execute processes that are arranged in a deep hierarchy of intermixed MEPs. Carrying out the behaviour of an MEP activity means more than just executing an MEP process. Considerations about correlation, error handling, and data passing have to be made. This complexity increases with each level of the hierarchy. For more information about intermixed MEPs, especially the recursive composition model, the reader is referred to [Höh08].

4.1.4. MEP Processes vs. Sub-processes and Fragments

External MEP processes seem to be related to sub-processes and fragments. This section discusses similarities and differences to these terms.

Fragments

There is no common definition for process fragments but rather a general understanding. Fragments are connected process model elements that can be reused when designing new process models. That means MEP processes can be seen as process fragments. The difference is situated in the notion "process model elements". MEP processes concentrate on the use of messaging elements whereas fragments can encompass arbitrary elements.

Sub-processes

BPEL sub-processes [KKL+05] are fragments that can be reused within a single process or by multiple processes. As such they seem to be very similar to MEP processes. But there are a lot of differences. Like fragments sub-processes do not have to contain messaging elements, as opposed to MEP processes. MEPs do not describe business logic. They are focussed on the exchange of messages. Sub-processes can be arbitrary complex whereas MEP processes should be smaller pieces of reusable process elements. MEPs are abstract definitions. Sub-processes are executable workflows.

Besides these statically differences there are also runtime disparities. The lifecycle of a sub-process is tightly connected to its parent process. If a parent process terminates its sub-processes are cancelled, too. Furthermore a parent process can delegate compensation to its sub-processes. In contrast to that, MEP processes describe the behaviour of MEP activities. This does not necessarily result in the execution of the (parameterized) MEP process. There are other eligible approaches like a state machine that supervises the MEP execution, or a number of hard coded MEP processes a workflow engine chooses of at runtime depending on the referenced MEP. However, if an MEP process is executed like a sub-process, the life-cycle

dependency is softened: it is not defined whether an MEP process is influenced by main process failures or not.

4.2. Extending BPEL^{light}

BPEL^{light} is a WSDL-less BPEL dialect to express the interaction between two or more partners. Primarily, it was developed to decouple BPEL processes' 'what' and 'which' dimensions facilitating role assignment during business process development and loosening BPEL's dependency on WSDL 1.1 as its interface definition. As mentioned, BPEL^{light} can also be used as a powerful mechanism to express MEPs since it can describe the flow *between* and *within* operations [LNL08].

However, the language as summarized in section 2.6 on page 18 lacks a number of important aspects concerning the support of MEPs just described. For that purpose BPEL^{light} is extended to achieve the goal of full MEP support, be it inline, external or intermixed. Furthermore some BPEL constructs are restricted when used in BPEL^{light} processes for pushing forward the separation of BPEL and WSDL. BPEL allows the definition of both, synchronous and asynchronous interactions, but it is restricted to the synchronous sending (by a `reply` activity) and receiving (by a two-way `invoke`) of fault messages. BPEL^{light} relies on an asynchronous communication model that can emulate synchronous interactions if necessary. Up to now BPEL^{light} is only able to handle incoming fault messages. Its fault handling mechanism is pursued by expanding it on all messaging elements to facilitate an asynchronous sending and receiving of fault messages.

To match the presented MEP concepts with BPEL^{light} some general considerations have to be done before dipping into the syntax descriptions. A main process corresponds to an executable BPEL^{light} process model. MEP processes map to abstract BPEL^{light} process models that follow the MEP profile described in section 2.6.3 on page 24. When used in abstract processes BPEL^{light}'s `conversation` fits the term MEP, that is, a `conversation` comprises one or more messages interchanged with one ore more partners. The enhanced `<bl:interactionActivity>` is, among other things, the realization of the MEP activity that references MEP processes. Parameterizing an MEP process means providing it with information needed to make it executable (i.e. variable names and types, substituting MEP timing expression placeholders by appropriate BPEL constructs, etc.). All these issues will become clear in the next sections.

4.2.1. Boolean Expression

Up to now the MEP profile allows to omit timing definitions and to use opaque tokens for variable types, variable references and timing definitions (i.e. duration and deadline expressions). However, BPEL's boolean expressions are also considered eligible candidates for opacity. Boolean expressions are used for decisions in control flow activities (i.e. `if` and `while`) and for conditions (`joinCondition` in activities, `transitionCondition` in links). An MEP designer may want to use expressions to define alternative routes or recurring events.

As an MEP process should not contain concrete values, there should be the possibility to define such expressions as opaque tokens. Therefore the MEP profile's restriction on the usage of opaque tokens concerning expressions is relaxed. BPEL's concept of opaque expressions is not sufficient for our use case. Although it enables omitting concrete expressions, the particular expressions are not referenceable what is crucial for the concept of parameterizing MEPs. Therefore BPEL's opaque expressions are set aside and a proprietary BPEL^{light} expression element is defined on a process or scope level similar to partners, variables and conversations (see listing 4.1). The definition of an expression comprises three attributes.

Listing 4.1 Syntax of BPEL^{light}'s boolean expression declaration

```
<bl:expressions>?
   <bl:expression expressionLabel="NCName" expression="bool-expr"
       expressionLanguage="anyURI"? />*
</bl:expressions>
```

The `expressionLabel` contains a name that must be unique concerning expressions *and variables*. This is due to the application of expressions as described in the next paragraph. The label makes expressions referenceable. The `expression` contains a boolean term or can be set to `##opaque` to hide a concrete expression. The optional `expressionLanguage` can refer to a language that is used to evaluate the expression. In case of an omission the behaviour is implementation-dependent. For example, the standard language can be used, or the language defined in the parent element the expression is applied in.

The use of BPEL^{light} expressions in boolean expressions is done – similar to referencing variables with XPath – by a "$" followed by the expression label. Due to this fact expressions and variables with same names should be avoided. Listing 4.2 on the following page clarifies this by an example. There is a BPEL^{light} MEP process that contains an abstract expression definition in the process scope. The process's `empty` activity is the source of two links, each declares a transition condition. With the help of the BPEL^{light} expression these conditions are undetermined specifications. Both reference the expression "XorExpr" – link "Path2" in combination with the XPath function `fn:not()` that returns the inverse of a boolean value. That way the process's control flow follows an XOR semantics at this point.

This concept enables an MEP designer to use structured activities for arranging messaging elements in a sophisticated control flow (e.g. to form alternative routes) and at the same time to leave concrete details open. These can later be added when parameterizing the MEP process by referencing the expression per label.

Additionally, the use of BPEL^{light} expressions is beneficial in executable BPEL^{light} process models to reuse boolean expressions since BPEL lacks in providing a similar concept.

4.2.2. Timing Expression

The MEP profile's timing expression is left as is with a single exception. Its `name` attribute is renamed to `timingExpressionLabel` to accord with BPEL^{light} boolean expressions and messages. The adapted syntax is presented in listing 4.3 on the next page.

Listing 4.2 Boolean expression example

```
<process xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
    xmlns:mep="http://iaas.uni-stuttgart.de/bpel-light/abstract/mep/2008/"
    abstractProcessProfile="http://iaas.uni-stuttgart.de/bpel-light/abstract/mep/2008/"
    ...>
  <bl:expressions>
    <bl:expression expressionLabel="XorExpr" expression="##opaque" />
  </bl:expressions>
  ...
  <empty>
    <sources>
      <source linkName="Path1">
        <transitionCondition>
            $XorExpr
        </transitionCondition>
      </source>
      <source linkName="Path2">
        <transitionCondition>
          fn:not($XorExpr)
        </transitionCondition>
      </source>
    </sources>
  </empty>
  ...
</process>
```

Listing 4.3 Adapted timing expression of the MEP profile

```
<mep:timingExpression timingExpressionLabel="NCName" expressionLanguage="anyURI"?
    type="for|until|repeatEvery" expression="expr"/>
```

4.2.3. Conversation

The `conversation` is extended by an additional optional `mep` attribute. It can be used to specify the MEP that is implemented by the messages of a `conversation`. Its value is an IRI, similar to the interaction activity's `mep` attribute as will become clear in the next section. Listing 4.4 shows the `conversation`'s syntax with the new attribute.

Listing 4.4 Extended BPEL^{light} conversation definition

```
<bl:conversations>?
  <bl:conversation name="NCName" mep="anyURI"? />*
</bl:conversations>
```

4.2.4. Interaction Activity

The `interactionActivity` as introduced in the first version of BPEL^{light} satisfies two requirements: On the one hand it implements one of four MEPs, namely *in-only*, *out-only*, *in-out*, or *out-in*. On the other hand it can be part of a more complex MEP when using a `conversation` to group several interaction activities.

Concerning the first point, the syntax of a single interaction activity is extended from implementing one of four predefined patterns towards arbitrary ones. This is done by realizing the concept of external MEPs that can be referenced by interaction activities (i.e. it is a realization of the MEP activity). The second point remains as is: conversations can be used to aggregate interaction activities that build a logical unit. The new syntax of the interaction activity can be found in listing 4.5.

Listing 4.5 Extended BPEL^{light} interaction activity definition

```
<bpel:extensionActivity>
  <bl:interactionActivity conversation="NCName" mep="IRI"? createInstance="yes|no"?
      standard-attributes>
    standard-elements
    <bl:input messageLabel="NCName"? variable="NCName" messageRef="NCame"? />*
    <bl:output messageLabel="NCName"? variable="NCName" messageRef="NCame"? />*
    <bl:infault faultName="QName" messageLabel="NCName"? variable="NCName"
        messageRef="NCame"? />*
    <bl:outfault faultName="QName"? messageLabel="NCName"? variable="NCName"
        messageRef="NCame"? />*
    <bl:timingExpression expression="expr" type="for|until|repeatEvery"
        expressionLanguage="anyURI"? timingExpressionLabel="NCName"?
        timingExpressionRef="NCName" />*
    <bl:expression expressionLabel="NCName" expressionRef="NCName"/>*
    <bl:partner name="NCName" partnerRef="NCame"? />*
    <bl:variable name="NCName" variableRef="NCame" />*
    <bpel:correlations>?
      <bpel:correlation set="NCName" initiate="yes|no|join" />+
    </bpel:correlations>
  </bl:interactionActivity>
</bpel:extensionActivity>
```

At first glance, it seems to be massive with a lot of optional attributes. But on the other hand it is an expressive, multi-purpose syntax that combines two usage scenarios: it allows to define single-message interaction activities (i.e. receiving or sending) that can be composed to more complex message exchanges, and it facilitates to create multi-message multi-partner interaction activities the behaviour of which is described by external MEPs. These two scenarios are considered separately in the following for giving the reader the chance to keep the overview.

Simple Interaction Activity

In this scenario (see listing 4.6 on the next page) the interaction activity realizes a single message, namely an input or an output. That is, it implements one of the basic MEPs in-only or out-only. In this context the interaction activity is called *Simple Interaction Activity*. It can have either a `<bl:input>` or `<bl:output>` node as child element (`infault` and `outfault` are special cases discussed below). That way it is able to emulate BPEL's `receive`, `reply`, and `invoke` activities (a two-way `<invoke>` is implemented by two interaction activities – a sending and a receiving). The `conversation` attribute references a conversation in the process. Conversations relate interaction activities or other messaging contructs (the

messages in a <bl:pick>, or the events in an event handler) by grouping them to a message exchange. The createInstance attribute is reserved for receiving interaction activities that are used to create an instance of the process model, that is, activities receiving the first message of the overall interaction of the process. The correlation mechanism is the same as in BPEL's messaging elements since there is no need to change it.

Listing 4.6 BPEL^{light} simple interaction activity scenario

```
<bpel:extensionActivity>
   <bl:interactionActivity conversation="NCName" createInstance="yes|no"?
       standard-attributes>
      standard-elements
      (
      <bl:input messageLabel="NCName" variable="NCName" />
      |
      <bl:output messageLabel="NCName" variable="NCName" />
      |
      <bl:infault faultName="QName" messageLabel="NCName" variable="NCName" />
      |
      <bl:outfault faultName="QName"? messageLabel="NCName" variable="NCName" />
      )
      <bl:partner name="NCName" />
      <bpel:correlations>?
         <bpel:correlation set="NCName" initiate="yes|no|join" />+
      </bpel:correlations>
   </bl:interactionActivity>
</bpel:extensionActivity>
```

The <bl:partner> child element simply refers to a partner definition in the process. A partner at runtime (i.e. a *partner instance*) holds an EPR a message can be sent to or received from, respectively. Finally, a look inside the input and output messages is taken. The variable attribute references a process or scope variable the content of which is sent to the partner (<bl:input> element), or the payload of an incoming message is copied to (<bl:ouput> element). Thus, the variable's data type determines the type of the message. It may be confusing that an incoming message is reflected by the <bl:output> element and vice versa, but this conforms to BPEL's invoke activity syntax having an input variable for the outgoing message and an output variable for the incoming one. From the process model's point of view the activity gets an *input* to invoke a partner, and provides the received answer as *output* to the process. The messageLabel makes the message referenceable. It must be unique within its conversation.

Interaction activities that specify a <bl:infault> or <bl:outfault> element implement a fault message to send or receive, respectively. These two cases are not considered an MEP since fault messages define exceptional instead of regular behaviour. Hence, such interaction activities only make sense in a larger context, that is, in an MEP that contains at least one regular message. The faultName should denote a business fault (such as ex:notExistingUser) that is meaningful in terms of the message exchange, as opposed to a BPEL failure a partner may not understand. Business faults should be reflected in the process's interface definition, if the chosen language allows for it. If the fault name of an outfault is omitted, the interaction activity is able to receive a fault message independent of the fault's name. The variable attribute points to a variable that contains additional fault

information to send, or that is the container fault information are copied to when receiving a fault. Again, the variable's data type specifies the message type. When a fault message is received the simple interaction activity behaves like a `throw` activity (i.e. it acts like an implicit `throw`). That means the fault is internally thrown to be catched by a fault handler.

Simple interaction activities are the implementation of inline MEPs. One or more of them can be comprised with the help of a `conversation` to realize an MEP. Thus, a single simple interaction activity in a conversation is one of the patterns *in-only* or *out-only*, depending on the message direction. A process model can contain several conversations and can therefore realize several inline MEPs.

BPEL^{light} processes do not contain references to any interface description. However, to make such a BPEL^{light} process executable on a workflow engine, it has to be mapped via DD to an interface. Simple interaction activities reflect messages sent or received by the process, that is, they constitute the communication to the outside. Consequently they must have a related counterpart in the process's interface description. Figure 4.4 shows a BPEL^{light} process model fragment (the box on the left) with two simple interaction activities, one of them outgoing, the other incoming, both belonging to the single conversation "out-in-conv". The `sequence` activity prescribes their order resulting in the implementation of the *out-in* pattern. The example illustrates a mapping of the conversation to a WSDL 2.0 operation (the box on the right) by an appropriate DD. The input's and output's message labels of the interaction activities must match those of the WSDL messages. This uniquely relates WSDL messages to BPEL^{light} activities. When using another interface description language, this mapping has to be adapted to its needs. Section 4.5.2 on page 65 provides more information about mapping BPEL^{light} processes to interfaces.

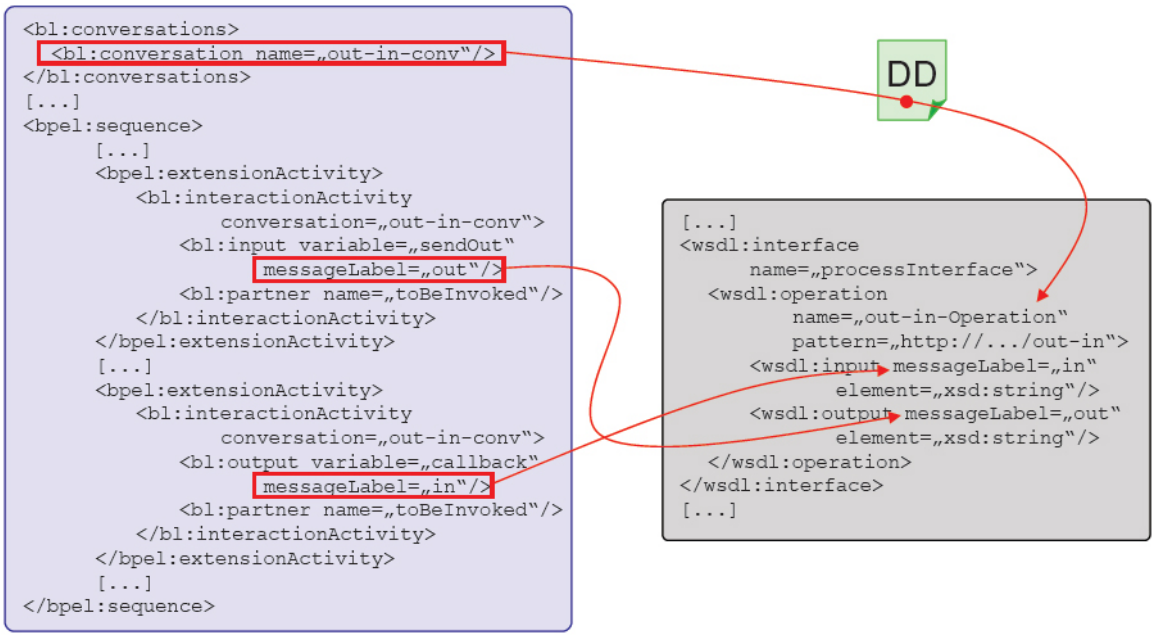

Figure 4.4.: Mapping simple interaction activities on a WSDL 2.0 operation

Simple Interaction Activities in MEP processes When using a simple interaction activity in an abstract MEP process, the syntax and semantics are as before. But the MEP profile allows to hide concrete variable names in `<bl:input>`, `<bl:output>`, `<bl:infault>`, and `<bl:outfault>` elements. Instead of a variable name an opaque token can be used as value for the `variable` attribute. Furthermore the `faultName` of fault messages can be abstracted by using an opaque token. Consider the code snippet in listing 4.7. It shows a simple

Listing 4.7 Example of a simple interaction activity in an MEP process

```
<bpel:process ...>
   ...
   <bpel:extensionActivity>
      <bl:interactionActivity name="in" conversation="in-out-conv">
         <bl:output messageLabel="request" variable="##opaque" />
         <bl:partner name="aPartner" />
      </bl:interactionActivity>
   </bpel:extensionActivity>
   ...
</bpel:process>
```

interaction activity that is part of the conversation "in-out-conv". The `<bl:output>` child element indicates that it is a receiving activity. The message is referenceable via the message label "request" which must be unique within the MEP process (i.e. a message label is in parallel unique within a `conversation`). The `output` denotes an abstract message, signaled by an opaque token instead of a concrete variable name in the `variable` attribute. That means no message type is prescribed. The `<bl:partner>` element points to the partner "aPartner" the message is received from when executing the process.

Complex Interaction Activity

In this scenario the interaction activity's behaviour is described by an externally defined abstract BPEL^{light} MEP process. In this context the interaction activity is called *Complex Interaction Activity*. It consists of all information needed to parameterize the referenced abstract MEP process, that is primarily to substitute abstract data types by concrete ones. Listing 4.8 on the next page illustrates the complex interaction activity's syntax for use in an executable process. When applied in an abstract MEP, the syntax is enriched by additional attributes and is therefore described below. However, the differences are only located at the child nodes, the attributes of the `<bl:interactionActivity>` element are the same in both cases and are explained next.

The MEP process is referred to by an IRI contained in the `mep` attribute. As proposed in [LNL08] its value is the MEP process's target namespace followed by a "#" as delimiter plus the process name (e.g. "http://iaas.uni-stuttgart.de/mep-in-bpel#in-out"). This way the called MEP process can be uniquely identified. The `conversation` attribute points to a `<bl:conversation>` of the process model. It can be used to relate the complex interaction activity to other messaging elements. This concept allows both the realization of external and intermixed MEPs. If the complex interaction activity is the only activity in its conversation, this conversation stands for an external MEP. If there are other messaging elements, it is

an intermixed MEP. The `createInstance` attribute can be set to `yes` if the referenced MEP starts with an incoming message. MEPs that start with an outgoing message cannot create a new process instance.

Listing 4.8 BPEL^{light} complex interaction activity scenario

```
<bpel:extensionActivity>
  <bl:interactionActivity conversation="NCName" mep="IRI" createInstance="true|false"?
      standard-attributes>
    standard-elements
    <bl:input variable="NCName" messageRef="NCame" />*
    <bl:output variable="NCName" messageRef="NCame" />*
    <bl:infault faultName="QName" variable="NCName" messageRef="NCame" />*
    <bl:outfault faultName="QName"? variable="NCName" messageRef="NCame" />*
    <bl:variable name="NCName" variableRef="NCame" />*
    <bl:partner name="NCName" partnerRef="NCame" />*
    <bl:timingExpression expression="expr" type="for|until|repeatEvery"
        expressionLanguage="anyURI"? timingExpressionRef="NCName" />*
    <bl:expression expressionLabel="NCName" expressionRef="NCName" />*
  </bl:interactionActivity>
</bpel:extensionActivity>
```

The child nodes of the `<bl:interactionActivity>` element provide the MEP process with initial values. The `<bl:input>` and `<bl:output>` elements are very similar. They only differ in the message direction: inputs represent outgoing whereas outputs stand for incoming messages. Both are referring to an MEP process message: therefore the `messageRef` attribute must contain the same NCName as the message's `messageLabel` it points at. The message label is unique within an MEP process as it consists of a single `conversation` only. Furthermore a variable of the main process must be specified per `variable` attribute. It denotes the message payload to send *during* the execution of the MEP process (in case of a `<bl:input>`), or the container to copy received data *after* the MEP process execution finishes (in terms of a `<bl:output>`). The elements `<bl:infault>` and `<bl:outfault>` are similar to `input` and `output` but refer to messages that are to send or receive faults. Their `faultName` attribute is used to provide the handled fault with a name. It can be omitted in case of an `outfault` to make the appropriate incoming messaging element able to receive faults independent of their name. The `<bl:variable>` element is used to initialize variables of the MEP process (referenced via the `variableRef` attribute) that are not related to messages (e.g. counter variables). The `<bl:partner>` child copies the EPRs of the given main process partner (`name` attribute) to the MEP process partner (`partnerRef` attribute). The `<bl:timingExpression>` element provides values to substitute a `<mep:timingExpression>` with appropriate BPEL constructs when parameterizing an MEP process. With the help of the `<bl:expression>` element an MEP process's abstract boolean expression (denoted by the `expressionRef` attribute) is parameterized. The `expressionLabel` attribute points to a boolean expression defined in the complex interaction activity's process. Its expression value is moved to the MEP process.

Essentially, all messages, variables, partners, timing and boolean expressions of the MEP process are expected to be initialized by the main process. After MEP execution the content of main process variables related to incoming MEP messages (i.e. output and outfault)

are refreshed with the particular message content the corresponding messaging element received.

In figure 4.5 two business processes are given as an example to illustrate the interplay of a main with an MEP process via a complex interaction activity. The MEP process realizes the *out-in* pattern sending a message ("out") to a partner ("aPartner") and expecting a message back ("in") from the same partner. This interaction takes course within a single conversation

Figure 4.5.: Using a BPEL^{light} MEP process to describe the behaviour of an interaction activity

("in-out-conv"). If the main process wants to use this MEP, a complex interaction activity is required with an `mep` attribute pointing to the target namespace and name of the MEP process and a number of parameterizing elements. There has to be an `input` element for the outgoing message. It says that the content of variable "sendOut" is passed via the message "out" to the partner. The `output` element makes sure that the answer of the partner – indicated by the message "in" – is copied into the variable "callback". The `partner` element passes the EPRs of the main process partner "toBeInvoked" to the MEP process partner "aPartner".

Complex Interaction Activities in MEP processes A complex interaction activity in an MEP process must provide partly different information than its counterpart in executable processes. Listing 4.9 on the next page shows its syntax with additional attributes only. All other attributes are omitted to provide a clear view and to avoid redundancy. Due to the MEP profile

<bl:input>, <bl:output>, <bl:infault>, and <bl:outfault> elements can – similar to simple interaction activities in MEPs – specify an opaque token as variable instead of a concrete variable name. This again abstracts the messages from concrete types. Furthermore these elements get the additional messageLabel attribute. Each message label must be unique within a conversation and hence within an MEP process. Via the label a message can be referenced by a complex interaction activity on a higher level. In <bl:infault> and <bl:outfault> elements the faultName can be left abstract by an opaque token. The <bl:timingExpression> does not need to contain concrete values. Therefore the attributes expression and type can contain opaque tokens. Like messages, timing expressions also need to be labelled (by a timingExpressionLabel attribute) to be referable by a complex interaction activity. The elements <bl:variable>, <bl:partner>, and <bl:expression> are the same in executable and abstract processes.

Listing 4.9 BPEL^{light} complex interaction activity in abstract MEP processes

```
<bpel:extensionActivity>
   <bl:interactionActivity ...>
      standard-elements
      <bl:input messageLabel="NCName" .../>*
      <bl:output messageLabel="NCName" .../>*
      <bl:infault messageLabel="NCName" .../>*
      <bl:outfault messageLabel="NCName" .../>*
      <bl:variable .../>*
      <bl:partner .../>*
      <bl:timingExpression timingExpressionLabel="NCName" .../>*
      <bl:expression .../>*
   </bl:interactionActivity>
</bpel:extensionActivity>
```

4.2.5. Pick

Up to now there are considerations and examples about BPEL^{light}'s pick activity in [NLKL07, LNL08]. This section provides a syntax for the pick (see listing 4.10) and some slight adaptations compared to the former approach. Similar to the interactionActivity the pick is embedded in BPEL's extension activity. It defines a set of events: onMessages for receiving incoming messages, onFaultMessages for receiving fault messages, and onAlarms that signal the end of a countdown or the achievement of a deadline. Only one of all defined events can fire. The contained activity is executed then.

A pick can create new process instances only if it exclusively consists of onMessage events. A <bl:onMessage> element waits for a message of the given partner and is part of a conversation. The renamed variable attribute (i.e. the outputVariable in the former BPEL^{light} pick) points to a variable the message payload is copied to. The former name attribute is substituted by a messageLabel. Like in interaction activities, it uniquely names the message within its conversation. This is important to make the message unambiguously identifiable. Besides regular messages a BPEL^{light} pick can also receive fault messages. In this case a <bl:onFaultMessage> element must be specified. Its faultName attribute can

Listing 4.10 BPEL^{light} `pick` activity definition

```
<bpel:extensionActivity>
  <bl:pick createInstance="yes|no" standard-attributes>
    standard-elements
    <bl:onMessage messageLabel="NCName" conversation="NCName" partner="NCName"
        variable="NCName" />*
      <bpel:correlations>?
        <bpel:correlation set="NCName" initiate="yes|no|join" />+
      </bpel:correlations>
      activity
    </bl:onMessage>
    <bl:onFaultMessage faultName="QName"? messageLabel="NCName" conversation="NCName"
        partner="NCName" variable="NCName" />*
      <bpel:correlations>?
        <bpel:correlation set="NCName" initiate="yes|no|join" />+
      </bpel:correlations>
      activity
    </bl:onFaultMessage>
    <bpel:onAlarm/>*
      (
      <bpel:for expressionLanguage="anyURI"?>duration-expr</bpel:for>
      |
      <bpel:until expressionLanguage="anyURI"?>deadline-expr</bpel:until>
      )
      activity
    </bpel:onAlarm>
  </bl:pick>
</bpel:extensionActivity>
```

contain a QName referencing a fault that should be reflected in the process model's interface description. If the fault name is omitted, a fault message can be received independent of the fault's name. When receiving a fault the pick executes the contained activity (as opposed to the interaction activity which acts like an implicit `throw`). That means it acts like a fault handler for external faults as long as it is active. BPEL's correlation tokens are adopted. Their semantics can be found in [AAA+07].The definition of `fromParts` like in BPEL's `pick` is not allowed. Parts imply the usage of WSDL 1.1 messages which should not be used in BPEL^{light} process models. As BPEL's `onAlarms` are WSDL-independent, there is no need to adapt them. Therefore they stay unchanged.

Picks in MEP processes If a `pick` activity is used in an MEP process, the MEP profile allows to hide certain details. The `onMessage`'s and `onFaultMessage`'s `variable` attribute can contain an opaque token instead of a concrete value to represent an abstract message. The same holds for the `onFaultMessage`'s `faultName`. In an `onAlarm` the concrete duration or deadline expression can be substituted by a `<mep:timingExpression>` element (see 2.6.3 on page 24) to achieve both a single point of variability and opacity.

Listing 4.11 on the next page presents an example of a `pick` activity in an MEP process. The conversation's name "robust-out-only" indicates that the `<bl:pick>` is a part of a greater MEP. The *robust out-only* pattern encompasses an outgoing message and optionally an incoming

Listing 4.11 Example of a `pick` activity in an MEP process

```
<bpel:process xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
    xmlns:mep="http://iaas.uni-stuttgart.de/bpel-light/abstract/mep/2008/"
    abstractProcessProfile="http://iaas.uni-stuttgart.de/bpel-light/abstract/mep/2008/"
    ...>
  ...
  <bpel:extensionActivity>
    <bl:pick name="faultCatcher">
      <bl:onFaultMessage messageLabel="inFault" conversation="robust-out-only" partner=
          "aPartner" faultName="##opaque" variable="##opaque">
        <bpel:empty />
      </bl:onFaultMessage>
      <bpel:onAlarm>
        <mep:timingExpression name="timeout" type="##opaque" expression="##opaque" />
        <bpel:empty />
      </bpel:onAlarm>
    </bl:pick>
  </bpel:extensionActivity>
  ...
</bpel:process>
```

fault message as response. The `pick`'s `onFaultMessage` realizes this waiting for the fault message. The message type and fault name are abstract – signaled by opaque tokens as attribute values. Additionally, an `onAlarm` is specified that acts as timeout mechanism. If no fault message arrives after a certain time or until a point in time, it executes an `empty` activity and hence finishes the `pick`. Instead of a concrete `<for>` or `<until>` an abstract `<mep:timingExpression>` is defined for the `onAlarm`. This allows deciding on the timing mechanism when parameterizing the MEP.

4.2.6. Assign

[NLKL07] proposes to extend the `assign` activity's to-spec by an additional `partner` attribute that points to a partner of the process. This proposal was not followed anymore since the concept of regarding partners as variables solved the problem of copying EPRs by using the `assign`'s expression variant (see 2.6.1 on page 23). However, the `<assign>` extension is considered worth to be pursued. Therefore a `bl:partner` attribute is introduced in the empty from-spec (see listing 4.12). It is able to select a single partner of the process – similar to the partner link from-spec – as source for the copy operation. Similarly the empty to-spec

Listing 4.12 BPEL^{light} `assign` activity extension: from-spec

```
<from bl:partner="NCName" />
```

is extended as shown in listing 4.13 on the following page. Like the appropriate from-spec it selects a single partner, but in this case as the destination of the copy operation. Using this alternative for copying partners is referred to as the *partner variant* in the following. Thus, there are two possibilities to handle partner EPRs. Both the partner and expression variant can be used to assign complete partner instances. If a single EPR or only a part out of

Listing 4.13 BPEL^{light} `assign` activity extension: to-spec

```
<to bl:partner="NCName" />
```

a partner instance needs to be copied, the expression variant must be chosen. The variants can be used mixed up, for instance, the literal from-spec can be taken to initialize a partner instance. But in general, an assign operation is only valid if the data items selected by the from- and to-spec are compatible as required by the BPEL 2.0 specification [AAA+07].

Note that BPEL^{light} variables should not be WSDL messages. Therefore the `part` attribute should not be used in the variable variant (compare [AAA+07]).

4.2.7. Event Handling

This section provides the syntax for the BPEL^{light} event handler (see listing 4.14). It is geared to the BPEL event handler but provides WSDL-less `<bl:onEvent>` elements. Each `onEvent` can be compared to a receiving interaction activity. If a matching message arrives (in terms of `messageLabel`, `conversation`, and `partner`), the associated `scope` is executed. The

Listing 4.14 BPEL^{light} event handler definition

```
<bl:eventHandlers>
  <bl:onEvent messageLabel="NCName" conversation="NCName" partner="NCName"
      variable="NCName" element="QName" />*
    <bpel:correlations>?
      <bpel:correlation set="NCName" initiate="yes|no|join" />+
    </bpel:correlations>
    <bpel:scope ...>...</bpel:scope>
  </bl:onEvent>
  <bl:onFaultEvent faultName="NCName"? messageLabel="NCName" conversation"NCName" partner=
      "NCName" variable="NCName" element="QName" />*
    <bpel:correlations>?
      <bpel:correlation set="NCName" initiate="yes|no|join" />+
    </bpel:correlations>
    <bpel:scope ...>...</bpel:scope>
  </bl:onFaultEvent>
  <bpel:onAlarm>*
    (
    <bpel:for expressionLanguage="anyURI"?>duration-expr</bpel:for>
    |
    <bpel:until expressionLanguage="anyURI"?>deadline-expr</bpel:until>
    )?
    <bpel:repeatEvery expressionLanguage="anyURI"?>duration-expr</bpel:repeatEvery>?
    <bpel:scope ...>...</bpel:scope>
  </bpel::onAlarm>
</bl:eventHandlers>
```

combination of the attributes `variable` and `element` is an implicit declaration of a local variable for the event handler, that is, it is comparable to a variable explicitly defined in the associated `scope`. As opposed to BPEL `onEvents` the declaration of a message-typed variable is not allowed due to the separation from WSDL 1.1. For this reason the `messageType`

attribute does not appear in the <bl:onEvent> element. Furthermore the specification of fromParts is interdicted since only WSDL 1.1 messages can contain parts which should not be used in BPEL^{light} process models. The correlation mechanism is adopted from the BPEL 2.0 specification [AAA+07].

The <bl:onFaultEvent> is similar to the <bl:onEvent> but can be used to receive a fault message. A faultName can be specified that denotes a business fault that should be reflected in the process's interface definition. When a fault message is received the event handler executes the appropriate scope (similar to the pick activity). It behaves like a fault handler for external faults as long as its surrounding scope is active.

The <bpel:onAlarm> element stays unchanged and is still in the BPEL namespace. An onAlarm event occurs after a countdown ran out (for and repeatEvery), or when a deadline is reached (until). Unlike the pick activity not only a single event can be triggered. Instead multiple events can be handled concurrently and recurring as long as the enclosing scope is active.

Event handling in MEP processes In an event handler of an MEP process the MEP profile allows to abstract from certain details. The onEvent's and onFaultEvent's element attribute can contain an opaque token to represent an abstract incoming message as well as the onFaultEvent's fault name. In an onAlarm concrete duration or deadline expressions can be hidden by declaring a <mep:timingExpression> element instead.

4.2.8. Fault Handling

BPEL's fault handler aims at catching and handling internal faults. It is not considered normal processing behaviour but rather reverse work for the failing scope. For application in BPEL^{light} process models BPEL's fault handler can be used with a single note: declaring message-typed variables should be avoided in BPEL^{light} processes. Therefore the faultMessageType attribute in the <catch> element should not be used as shown in listing 4.15.

Listing 4.15 Adapted BPEL fault handler

```
<bpel:faultHandlers>
   <bpel:catch faultName="QName"? faultVariable="NCName"? faultElement="QName"?>*
      activity
   </bpel:catch>
   <bpel:catchAll>?
      activity
   </bpel:catchAll>
</bpel:faultHandlers>
```

4.2.9. Correlation

BPEL's concept of correlation, that allows stateful interactions independent of implementation-specific tokens or the underlying transport infrastructure, stays unchanged. It is built upon message properties – certain fields in well-defined message structures that can be identified by a query [AAA+07].

BPEL provides the definition of such properties by the `<property>` element. `<propertyAlias>` elements are responsible for mapping properties on fields of messages and variables. Since users of a process model should be aware of the demanded correlation mechanism, these elements are declared within a WSDL 1.1 interface description under their own namespace[1].

In BPEL^{light} two issues concerning BPEL's correlation concept are addressed. First, as BPEL^{light} process models come without a coupling to an IDL, `<property>` and `<propertyAlias>` elements should be incorporated into the BPEL^{light} process as children of the `<process>` element. An example is given in listing 4.16. The property "CustomerID" of type `xsd:string` is mapped via property alias on the "ex:identifier" field of XSD element "ex:Order". The appropriate XSD file is referenced by an `import`. Second, property aliases are restricted to

Listing 4.16 Example of BPEL^{light} correlation tokens

```
<process xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
     xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
     xmlns:ex="http://example.com/dt" ... >
   ...
   <import namespace="http://example.com/dt" location="dt.xsd"
       importType="http://www.w3.org/2001/XMLSchema" />
   <vprop:property name="CustomerID" element="xsd:string">
   <vprop:propertyAlias propertyName="tns:CustomerID" element="ex:Order">
      <vprop:query>ex:identifier</vprop:query>
   </vprop:propertyAlias>
   ...
</process>
```

map properties on XSD elements and types only. A mapping on message types and parts would be WSDL 1.1 dependent and should therefore not be used. The adapted syntax can be regarded in listing 4.17 on the facing page. It is the same as before but now declared within a process and without the attributes `messageType` and `part`. A downside of this approach is that the correlation mechanism is opaque to process partners as they do not have access to the BPEL file. Therefore, when a BPEL^{light} process is bound against an IDL, the correlation elements (`property` and `propertyAlias`) should be copied to it. This is at least possible for WSDL 1.1 and 2.0 bindings. This copying may require adjustments in the chosen interface (e.g. import of an XSD file) or in the copied elements (e.g. adaptation of a namespace prefix). If the IDL does not provide means to incorporate those BPEL extensions, for instance, when it is a non-XML language, an inbuilt or extension mechanism of the chosen IDL must be used instead – as far as supported – to express the correlation capacity. It is the task of an appropriate interface mapping to relate the modelled correlation elements to the IDL's

[1] `xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop"`

correlation mechanism. When using WSDL 1.1 or 2.0 such a relation is not needed since the correlation elements are simply copied. However, integrating correlation information into the IDL reveals them for communication partners. It is important that the correlation elements are left in the BPEL^{light} process model. This way correlation sets (that reference properties) do not have to be adapted.

Listing 4.17 Adapted BPEL property alias

```
<process xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop" ...>
   ...
   <vprop:propertyAlias propertyName="QName" type="QName"? element="QName"?>
      <vprop:query queryLanguage="anyURI"?>?
         queryContent
      </vprop:query>
   </vprop:propertyAlias>
   ...
</process>
```

Declaring correlation sets in `<process>` or `<scope>` elements stays unchanged. A correlation set is a named collection of properties. This is reasonable in cases where more than one token is needed for correlation (e.g. name and birthday). Using these correlation sets in messaging elements does also follow BPEL's specification. For a more detailed explanation of correlation sets the reader is referred to [AAA+07].

4.2.10. Import

BPEL uses an import mechanism to define dependencies of process models on XML Schema or WSDL 1.1 files. In an `<import>` element a `namespace` and `location` for the imported document can be specified. The `importType` attribute indicates the document type by an absolute URI. At this point the import declaration is extended for making it possible to import WSDL 2.0 definitions. In this case a particular URI[2] must be given as import type.

There is a need for importing WSDL 2.0 documents since a process model may need to reference XML Schema elements or types declared within it. Of course, this is only possible if a BPEL^{light} process model is bound to a WSDL 2.0 interface at design time. Listing 4.18 exemplarily illustrates such an import. There is a link to the document "interface.wsdl" with target namespace "http://example.com/wsdl". The import type signals that a WSDL 2.0 document is referenced.

Listing 4.18 Extended BPEL import

```
<process xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
     xmlns:ex="http://example.com/wsdl" ...>
   ...
   <import namespace="http://example.com/wsdl" location="interface.wsdl"
      importType="http://www.w3.org/ns/wsdl" />
   ...
</process>
```

[2] `http://www.w3.org/ns/wsdl`

4.3. Summary of the Concept

This section summarizes major statements of the previously introduced concepts and provides comprehensive examples to illustrate their relation.

4.3.1. Statements

BPEL^{light} can be used to define executable processes. A process consists of one or more internal MEPs – each represented by a single `conversation`. These MEPs' messages are signaled by appropriate messaging elements. The message order depends on the process's control flow structure. A process can also integrate external MEPs with the help of complex interaction activities. These external MEPs are defined using abstract BPEL^{light}. The behaviour of a complex interaction activity is thereby described by such an abstract MEP process that leaves open concrete details (e.g. data types, or fault names). The concrete realization of the complex interaction activity is implementation-dependent. MEP processes contain only a single `conversation` since they implement exactly one MEP. The application of intermixed MEPs is done by complex interaction activities that are integrated into a `conversation` that encompasses also additional messages. The concept of intermixed MEPs makes it possible to extend existing MEPs. This can lead to a dependency structure similar to inheritance in object-oriented programming languages.

4.3.2. Examples

The following two examples illustrate the incorporation of external MEPs into a main process.

Listing 4.19 on the facing page illustrates parts of a main process that is used to log in at a server. The external MEP "Robust-out-only" is thereby taken to realize the authentication, incorporated by the complex interaction activity "logInMep". The MEP process can be found in appendix A on page 95. In short, the MEP sends a request and may receive a fault message in response by the same partner. The complex interaction activity parameterizes the MEP with an `ex:AuthToken` element for the outgoing message and an `ex:FaultContent` element for the optional incoming fault message. As communication opponent the partner "server" is chosen. The MEP provides a timeout to abort waiting for the fault message. It is initialized with a duration expression for five minutes ("PT5M", the expression language is XPath 2.0).

Listing 4.20 on page 58 illustrates parts of a main process that is used to react to an offer provided by a customer. The appropriate message exchange is defined by the external MEP "In-xor-out", integrated by interaction activity "offerReaction". The MEP process can be found in appendix A on page 95. It consists of an incoming and two outgoing messages, all exchanged with the same partner. Only one of these two outgoing messages is sent depending on a boolean expression. The complex interaction activity parameterizes the MEP with an `ex:offer` element for the incoming message, an `ex:agree` element for the first

Listing 4.19 Example: integration of a robust-out-only MEP into a main process

```
<bpel:process name="authenticate"
     targetNamespace="http://.../bpel-light-process"
     xmlns:bpel="http://.../wsbpel/2.0/process/executable"
     xmlns:bl="http://iaas.uni-stuttgart.de/bpel-light"
     xmlns:ex="http://example.com/dt">
   ...
   <bl:conversations>
      <bl:conversation name="extMepConv" />
   </bl:conversations>
   <bl:partners>
      <bl:partner name="server" />
   </bl:partners>
   <bpel:variables>
      <bpel:variable name="logIn" element="ex:AuthToken" />
      <bpel:variable name="faultMessage" element="ex:FaultContent" />
   </bpel:variables>
   ...
   <bpel:extensionActivity>
      <bl:interactionActivity name="logInMep" conversation="extMepConv"
          mep="http://iaas.uni-stuttgart.de/mep-in-bpel#Robust-out-only">
        <bl:input variable="logIn" messageRef="out" />
        <bl:outfault variable="faultMessage" faultName="ex:UnknownUser"
            messageRef="inFault" />
        <bl:partner name="server" partnerRef="aPartner" />
        <bl:timingExpression expression="PT5M" type="for"
            expressionLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0"
            timingExpressionRef="timeout" />
      </bl:interactionActivity>
   </bpel:extensionActivity>
   ...
</bpel:process>
```

outgoing message, and an `ex:decline` for the second outgoing message. All these messages are exchanged with partner "customer". The MEP's boolean expression gets the value of the "lowerLimit" expression of the main process that uses XPath 2.0 as expressionLanguage. The parameterization leads to following semantics: a customer sends an offer to the process. Depending on how much money the customer is willing to pay an agreement ($>=$ 20000) or decline ($<$ 20000) is responded.

4.3.3. Best Practices

This section gives pieces of advice about the definition of MEPs with BPEL^{light}. An MEP declaration should not grow too large since this would hamper its reusability. On the other hand an MEP that incorporates a single message only is hardly worth to be defined because this behaviour can also be defined by single activities. A good middle way should be found for creating MEPs. The usage of intermixed MEPs at design time can lead to an added value. It can save time and resources to extend existing MEPs instead of defining a complete new one. Additional information can be added to an MEP (e.g. control flow dependencies, or

Listing 4.20 Example: integration of an in-xor-out MEP into a main process

```
<bpel:process name="sales"
     targetNamespace="http://.../bpel-light-process"
     xmlns:bpel="http://.../wsbpel/2.0/process/executable"
     xmlns:bl="http://iaas.uni-stuttgart.de/bpel-light"
     xmlns:ex="http://example.com/dt">
   ...
   <bl:conversations>
      <bl:conversation name="extMepConv" />
   </bl:conversations>
   <bl:partners>
      <bl:partner name="customer" />
   </bl:partners>
   <bl:expressions>
      <bl:expression expressionLabel="lowerLimit" expression="$offer/ex:amount >= 20000"
          expressionLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0" />
   </bl:expressions>
   <bpel:variables>
      <bpel:variable name="offer" element="ex:offer" />
      <bpel:variable name="agree" element="ex:agree" />
      <bpel:variable name="decline" element="ex:decline" />
   </bpel:variables>
   ...
   <bpel:extensionActivity>
      <bl:interactionActivity name="offerReaction" conversation="extMepConv"
          mep="http://iaas.uni-stuttgart.de/mep-in-bpel#In-xor-out">
         <bl:output variable="offer" messageRef="in" />
         <bl:input variable="agree" messageRef="out-option1" />
         <bl:input variable="decline" messageRef="out-option2" />
         <bl:partner name="customer" partnerRef="aPartner" />
         <bl:expression expressionLabel="lowerLimit" expressionRef="expr" />
      </bl:interactionActivity>
   </bpel:extensionActivity>
   ...
</bpel:process>
```

conditions) by using it inline. That way it is open for proprietary adaptations. Note that the actual semantics should thereby not be destroyed.

4.4. Application of the Concept

The presented concept of BPEL^{light} is held apart from details concerning its implementation by a workflow engine. This section explains how it is applied to specify the prototype's functionality. The conceptual generality is thereby restricted at certain points. Appropriate discussions outline why the choice fell on a presented solution.

4.4.1. Execution of MEPs

The behaviour of an interaction activity can be described by an MEP process (i.e. the complex interaction activity scenario). There is a number of approaches how such an interaction activity can be implemented. Three of them are discussed in the following.

- *Hard coded*
  A BPEL^{light} engine can implement the behaviour of established MEPs. Depending on the `mep` attribute's value a certain hard-coded MEP is executed. On the one hand this eases the deployment of processes and may lead to a faster execution of the interaction activity. On the other hand an interaction activity is restricted to the implemented MEPs. New MEPs cannot easily be integrated.

- *State machine*
  A BPEL^{light} engine can make use of a state machine that supervises the correct execution of the referenced MEP. The advantage is the flexibility to insert new MEPs. The downsides are manifold: the BPEL^{light} MEP processes must be parameterized, transformed into a state machine representation and deployed afterwards. The execution may be slower compared to the hard-coded case since the communication between engine and state machine must be managed. Furthermore the execution infrastructure gets an additional component.

- *BPEL^{light} engine*
  MEP processes can be executed by the BPEL^{light} engine itself. No additional execution component is needed. The incorporation of new MEPs can easily be realized. There is neither a need for a transformation of MEP processes into another language nor for managing the interaction with an external component. Beside these benefits there are also handicaps. The MEP processes must be parameterized and deployed on the engine. Invoking an MEP process means spending time and resources.

The latter is obviously the best choice for this diploma thesis as hard-coded MEPs are a bad design and the use of a state machine unnecessarily complicates the situation.

That means in the complex scenario an interaction activity synchronously invokes a parameterized MEP process and passes data denoted by its elements (inputs, partners, etc.). The MEP process carries out its behaviour with the delivered data and replies with the content of all messages that were received during its execution. Meanwhile, the interaction activity waits for the MEP process to finish. If an answer is transmitted it copies the results into the appropriate variables and completes.

To realize an asynchronous invocation of MEP processes another type of activity would be needed that receives and unpacks the MEP process's response. This approach does not promise significant advantages. There are other possibilities to let a process do work in parallel while an MEP is executed. For instance, this can be achieved by a sophisticated control flow structure in the main process, or by using the MEP within the main process as inline MEP and inserting the parallel running activities among the MEP activities.

MEP processes are abstract definitions and therefore by implication not executable. They must be made executable by a sophisticated parameterization algorithm. That means more

than concretizing abstract information like message types. As MEP process models have no explicit start activity [LNL08], two activities must be wrapped around the MEP logic: one for receiving the call, creating a process instance and unpacking the passed data, the other for packing the response and sending it back to the main process. The parameterization can happen at design time, deployment, or runtime. In this work parameterization at deployment is focussed as it seems to be the optimal solution as the discussion at the end of section 4.1.2 on page 37 reveals.

The communication between main and MEP process entails a number of requirements. The invocation of an MEP process creates a new MEP process instance that carries out the MEP's functionality. When finished, it must send its response to the correct main process instance. That means these two instances must be correlated. The concrete realization of this correlation is up to the engine's implementation.

4.4.2. Use of Intermixed MEPs

With the help of intermixed MEPs process models can be designed that build a deep hierarchy of outsourced processes. This dependency graph – although beneficial at design time concerning reusability – can cause an inefficient execution. Especially a structure where MEP process models again reference MEP processes on their own brings more disadvantages. When calling an outsourced process, a new process instance must be created, data must be passed between the processes, and the answer of the outsourced process must be correlated to the correct instance of the calling process. Furthermore fault and compensation handling may be considered. This effort is eligible for the main process that may profit by reuse and that may get clearly arranged by outsourcing complicated or duplicated code. For MEP processes it is unnecessary. Therefore the hierarchy should be flattened. This work concentrates on process model dependencies with at most two levels. This is considered a precondition for deployment, that is, the DU is expected to be structured as described in the following.

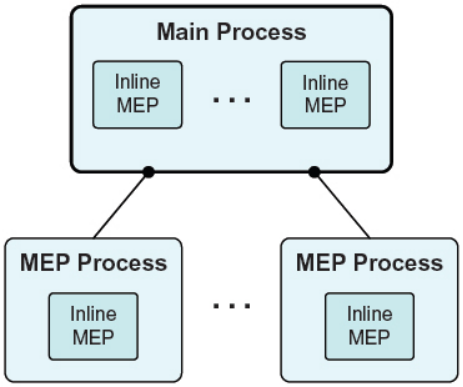

Figure 4.6.: Hierarchy of process models in the deployment unit

The main process must consist of inline and external MEPs only. The prototype ensures this by not allowing complex interaction activities to be part of a conversation (i.e. the `conversation` attribute in complex interaction activities is omitted). MEP processes are restricted to realize a single inline MEP. This leads to a structure as illustrated in figure 4.6: a two-level tree with the main process as root node and MEP processes as leaves. That way no intermixed (i.e. recursive) MEPs are used at runtime.

An algorithm to level the hierarchy can be regarded in [Höh08] as it is beyond the scope of this work. Figure 4.7 on the facing page exemplarily shows how to flatten the hierarchy of MEP process models. There are three processes: process A is a main process with an inline

and an external MEP, process B an MEP process with an intermixed MEP, and process C an MEP process with an inline MEP. Process B violates the structural requirement in referencing an MEP process (C) on its own. To solve this problem its complex interaction activity (D) is substituted by a `scope` that contains all constructs of process C. After that process C becomes obsolete and can be removed from the DU. The result is the unchanged main process A that references MEP process B which consists now of an inline MEP.

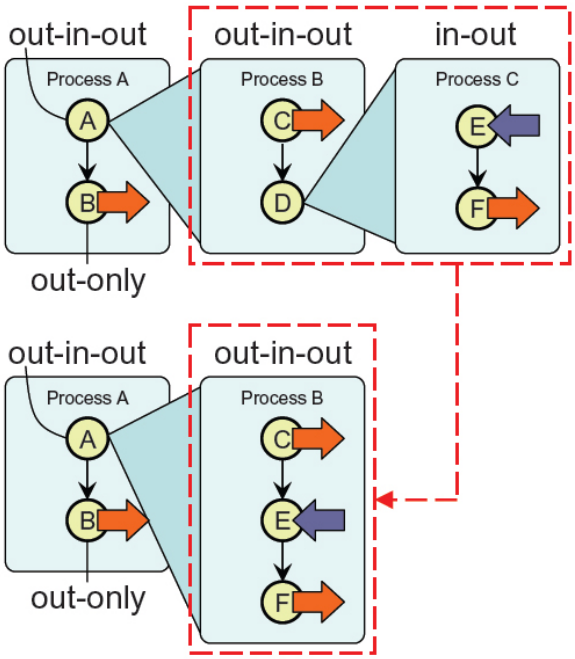

Figure 4.7.: Flattening the dependency hierarchy of process models

This flattening can happen either between design time and deployment, or during deployment. Since such an algorithm is time consuming, the former approach is focussed in this work to keep the deployment mechanism simpler and faster. It provides advantages especially when redeploying process models, for instance, to clean the DB or when preparing test cases.

4.5. Deployment

This section specifies the deployment of BPEL^{light} processes on a workflow engine. It relies on the application of the concept as presented in the previous section.

The deployment of a process is an important step during its lifecycle. After finishing the process modelling, it can be put "into production" [LR00] to be executable on a workflow engine. Instances of the process can then be created to carry out the modelled behaviour.

Deploying a process model requires – besides the BPEL file(s) – a whole bundle of files. Figure 4.8 shows a general view on the DU. There are four types of files described next. Dotted lines mark optional files, solid lines stand for mandatory ones.

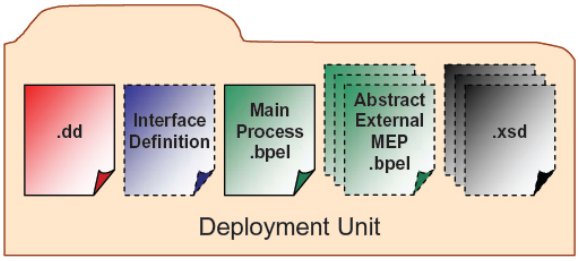

Figure 4.8.: Components of the deployment unit

1. BPEL^{light}
   There is at least one process model in the bundle, namely the main process. Optionally, one or more abstract MEP definitions can be given the main process points at by complex interaction activities. An alternative could be a repository MEP processes are stored in and fetched during deployment. This work focusses on the former approach.
2. XSD
   BPEL^{light} process models can import XSD files used data types are defined in. They are marked optional since data types may also be defined in an interface definition, if present.
3. Deployment Descriptor
   The DD contains information prescribing how to make the DU executable on a specific workflow engine. There are two conceptual requirements on a DD. First, it must incorporate an interface mapping that relates BPEL^{light} messaging elements to IDL constructs. Second, it should be possible to declare partner instances for an early (i.e. static) binding (compare [LLN08]). The setup of a partner instance can be looked up in section 2.6.1 on page 20.
4. Interface Definition
   An interface definition may be part of a DU if a BPEL^{light} process model is bound at design time. If no interface is given, it must be generated during deployment with the help of the DD's interface mapping.

If a main process makes use of external MEPs, the DU is augmented by additional files generated during deployment. Each complex interaction activity parameterizes an external MEP process that is stored as separate file. If two or more complex interaction activities reference the same MEP process and all parameterize it with same message types, variables, expressions, and timing expressions, it should be concretized only once. For the communication between main and MEP processes a well-defined data type is needed that is stored in an XSD file created during deployment.

Two use cases for putting a process model into production can be pointed out. First, an interface definition is given a priori. This requires fewer steps during deployment but means

more effort at design time. Second, there is no interface definition. It must be generated during deployment depending on the interface mapping information of the DD. Note that in either case the main process and all parameterized external MEP processes share the same interface.

4.5.1. Parameterizing MEP Processes

When using external MEPs a lot of steps have to be performed during deployment. Each complex interaction activity of the main process references an external MEP being available as abstract process definition. This abstract process has to be transformed into an executable one for being runnable on a workflow engine. Particularly the following tasks have to be performed:

1. The MEPs abstract[3] target namespace is substituted by the executable[4].
2. The `abstractProcessProfile` attribute of the `<process>`-tag is removed.
3. All opaque tokens (variable references, variable types, expressions, and timing expressions) are substituted by the information of the appropriate complex interaction activity element (input, output, infault, outfault, variable, expression or timing expression).
4. An MEP process as such is designed to carry out the behaviour of the realized MEP only. The communication with a potential main process is ignored. Therefore an MEP process must be enriched with additional activities and elements to facilitate the interaction with an appropriate main process. Figure 4.9 on the next page shows how to wrap the MEP's root activity. Newly created activities and elements are marked by dotted lines. In particular a number of steps are required:
   - The root activity is pushed into a new `sequence` activity. This step allows to pre- and append additional activities. Alternatively, a `flow` activity with appropriate links can be used but means more effort due to the larger number of XML nodes.
   - Since an MEP process model has no explicit starting activity, a simple interaction activity is prepended to the (former) root activity initializing the process and creating new process instances.
   - The instance creating simple interaction activity gets a package of data from the main process. This data contains partner instances and contents of variables used to execute the MEP process's behaviour. The single pieces of information must be extracted and copied to their destination (partners and variables). This is done by an appropriate `assign` activity located behind the receiving interaction activity.
   - An activity is needed that marks the MEP process's ending and that sends data back to the main process.
   - This response contains the content of all incoming messages carried out during MEP execution. It is packaged by another `assign` activity.

[3] `http://docs.oasis-open.org/wsbpel/2.0/process/abstract`
[4] `http://docs.oasis-open.org/wsbpel/2.0/process/executable`

- The two new interaction activities must be incorporated into a `conversation`. As they are strictly separated from the actual message flow in the MEP process, a new `conversation` is defined to encompass the communication activities with the main process.
- Furthermore a new `partner` is created that stands for the main process and that is associated with the new interaction activities.
- Both interaction activities need references to a variable. Therefore two `variables` are defined – one as container for the incoming message of the main process, the other as response package.

5. The new interaction activities' variables must be of a well-defined type which is added to the DU as XML schema definition.

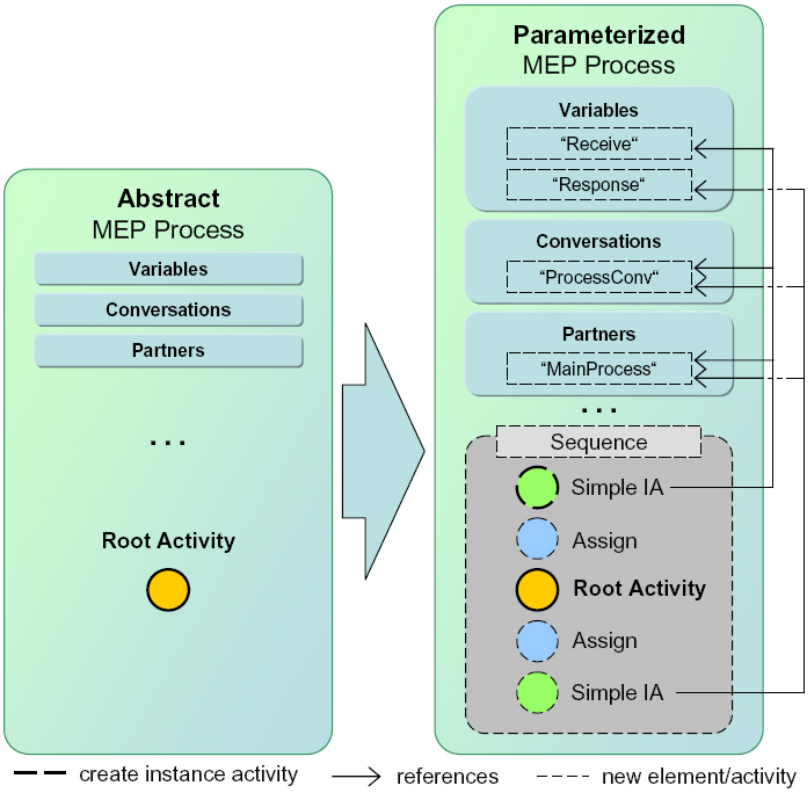

Figure 4.9.: Inserting activities and elements into an MEP process during parameterization

Two or more complex interaction activities referencing the *same* external MEP process can lead to several distinct parameterized MEP processes depending on message, variable and fault types as well as expressions and timing expressions. If all of them are equal, a duplicated parameterization would be redundant and should be avoided. Anyhow, a redundant parameterization works as well but violates the concept of reusing external MEP processes during runtime.

4.5.2. Interface Mapping

Due to BPEL^{light}'s idea of separating process models from their interface definition, BPEL^{light} processes are self-describing: they do not associate process model elements with interface elements. At design time this eases modelling processes. But by runtime a process model must be bound to an interface definition for being runnable. An interface mapping in the DD connects process models and interface definitions.

Potentially every interface can be mapped on a process. Each of these mappings has to be defined individually as each IDL consists of particular constructs with their own syntax and semantics. Up to now, there is an interface mapping specification for WSDL 2.0 and consideration about a mapping on WSDL 1.1. A main process and its MEP processes must be bound to the same type of IDL (e.g. all are mapped to WSDL 2.0, or all are mapped to WSDL 1.1, but not mixed up). This is due to the fact that main and MEP processes are exposed by a single interface. Thus, from a partner's point of view the provided services are realized by a single process.

An interface mapping must make it possible to map process model on interface elements. This is sufficient if an interface is given a priori in the DU. Additionally, an interface mapping may be powerful enough to support the generation of an interface during deployment if none is delivered with the DU.

There are general considerations each interface mapping should follow:

- Simple interaction activities must be mapped on interface messages.
- Conversations must be mapped on interface operations/functions (i.e. on a concept that groups messages).
- Correlation elements must be mapped on adequate interface concepts. This is redundant if the interface's correlation mechanism equals BPEL's (i.e. it uses properties and property aliases to determine message fields as correlation tokens).
- If no interface definition is provided in the DD, additional information concerning the interface generation must be provided (bindings, endpoints, etc.). If this point is omitted, the mapping does not support interface generation during deployment.

There are two major approaches about mapping a main process and its MEP processes on a single interface. First, an interface mapping can be separately defined for the main and each MEP process. Second, an interface mapping is declared for the main process only. Such a mapping must be able to relate a referenced MEP process (i.e. indirectly its `conversation`) to an IDL operation. In this work the latter solution is chosen. On the one hand it ensures that main and MEP processes share the same IDL type. On the other hand it entails the requirement that complex interaction activities must be uniquely referenceable throughout the main process. This is best done by a unique name. Therefore the `name` is considered a mandatory attribute for complex interaction activities. Referencing complex interaction activities by `mep` attribute is not possible since one and the same MEP can be used by several interaction activities of a process.

WSDL 2.0

Mapping a BPEL^{light} process model to a WSDL 2.0 description is the obvious solution: each `<bl:conversation>` is nicely reflected by a WSDL 2.0 operation with multiple incoming and outgoing messages or faults, respectively.

The structure of the mapping is shown in listing 4.21 on the facing page. The outer tag, `<interfaceMapping>`, is the general container for concrete mappings. Its `fileName` attribute, if present, points to the interface file the mapping is defined for. The `mustGenerate` attribute signals whether an interface is to be generated. In this case the created interface is stored under the given filename, if present, or under a generated one if the filename is omitted.

The WSDL 2.0 mapping is geared to the WSDL 2.0 specification (compare [BL07]). There are two child elements, `<interface>` and `<service>`. At least one `interface` must be given. It realizes the mapping of conversations on WSDL 2.0 operations. If the WSDL is to be generated, the `<interface>`s are used to create its abstract part. An `interface` contains one or more `operations`, each referring by `name` attribute to a WSDL 2.0 operation. The `<conversation>` element points to the process's `conversation` that is associated with the operation. Duplicate mappings for conversations or operations should be avoided. There is no mapping of process model messaging elements on WSDL 2.0 messages. This is due to the implicit mapping that is defined per message label: a WSDL 2.0 message must have the same value in its `messageLabel` attribute like the corresponding BPEL^{light} messaging element. The WSDL message's XSD element is fetched from the BPEL^{light} message's variable type.

Several `services` can be declared. A `service` provides information about which interface is implemented, where (endpoint) and how (binding) to access it. The `<service>` elements can be omitted in two cases. First, if a given WSDL 2.0 interface already contains service information. Second, if the supporting middleware does not need service information, independent of the presence of a WSDL 2.0 file in the DU, for instance, when it creates these information on its own. On the other hand `<service>` elements must be given if the middleware needs service information and these are contained in the delivered WSDL 2.0 interface (for mapping reasons) or the interface has to be generated. However, the unique service name is held in the `name` attribute. The implemented interface is referenced by `interface` attribute. One and the same interface may be exposed by different services. Each `service` can be assigned to one or more `endpoints` being the location where the service can be reached. The endpoint's URI is stored in the `<address>` element. The `binding` provides information about how the endpoint is bound to a specific message format and underlying transport protocol. Currently, there are two binding types, SOAP and HTTP. The SOAP binding comes with two properties: the SOAP message version (either 1.1 or 1.2, where 1.2 is the default value), and the transport protocol type. The HTTP binding is used to send and receive messages natively without encoding them in SOAP. It consists of a lot of optional settings. The `<methodDefault>` element signals which HTTP operation to use for all WSDL 2.0 operations. If an operation needs special handling, the default method can be overridden by specifying a method within the `<operation>` element. The `name` attribute must thereby refer to an operation defined in the operation-conversation map of the `<interface>` element. Per `location` attribute a pattern can be specified for serializing input message data into the request URI. Only a very small subset of the whole SOAP and HTTP binding extensions of

Listing 4.21 Interface mapping for WSDL 2.0

```
<interfaceMapping fileName="xsd:string"? mustGenerate="xsd:boolean"? >
   <wsdl20>
      <interface name="xsd:NCName">+
         <operation name="xsd:NCName">+
            (
            <conversation name="xsd:NCName" />
            |
            <complexIA name="xsd:NCName" />
            )
         </operation>
      </interface>
      <service name="xsd:NCName" interface="xsd:NCName">*
         <endpoint name="xsd:NCName">+
            <address>xsd:anyURI</address>
            <binding name="xsd:NCName"?>
               (
               <soap>
                  <version>1.1|1.2</version>?
                  <protocol>HTTP|JMS|...</protocol>
               </soap>
               |
               <http>
                  <methodDefault>GET|PUT|POST|DELETE|...</methodDefault>?
                  <operation name="xsd:NCName" location="xsd:anyURI"?
                      method="GET|PUT|POST|DELETE|..."? inputSerialization="xs:string"?
                      outputSerialization="xs:string"? faultSerialization="xs:string"?
                      contentEncodingDefault="xs:string"? />*
               </http>
               )
            </binding>
         </endpoint>
      </service>
   </wsdl20>
</interfaceMapping>
```

WSDL 2.0 is reflected here constraining the number of usage scenarios to the conventional ones. For the entire picture or for deeper explanations of the binding properties the reader is referred to [CHL+07].

The interface mapping does not offer a correlation map. The correlation tokens are simply copied into the WSDL 2.0 file. Property aliases may be adapted concerning namespace prefixes if the BPEL^{light} process and the WSDL file use different prefixes for the same `import`. Thus, a correlation mapping is not needed.

WSDL 1.1

The focus of this work lay on binding BPEL^{light} process models against WSDL 2.0 interface definitions. Nevertheless some considerations were made about mapping BPEL^{light} processes on WSDL 1.1 interfaces.

The authors of [NLKL07] propose to map conversations on roles of partner link types. Since the concept of conversations and partners is renewed [LLN08], this will not work anymore because conversations and partners behave orthogonally. That means a `conversation` can communicate with several partners; a `partner` has exactly one opponent. Therefore a `partner` is an eligible candidate to be mapped on a partner link type. Listing 4.22 illustrates a proposal for a WSDL 1.1 interface mapping that may need further refinements especially concerning interface generation.

Partners are related to partner link types. The `myRole` and `partnerRole` can be specified depending on the partner link type's role definition. The `myRole` denotes that the operation is implemented by the process itself whereas the `partnerRole` signals an operation on the partner's side. Since a `conversation` can be composed of several messages in both directions it cannot be simply mapped on a single WSDL 1.1 operation. Therefore the single messages of a `conversation` are related by message label to an operation. Note that two messages of opposite directions can be related to one and the same operation if both address the same partner. An NCName is sufficient to specify an operation since the partner link type and hence the operation's port type can be found over the message's partner.

Similar to the WSDL 2.0 mapping a correlation map is not needed since the correlation tokens can be simply copied from the BPEL^{light} process model.

Listing 4.22 Interface mapping for WSDL 1.1

```
<interfaceMapping fileName="xsd:string"? mustGenerate="xsd:boolean"? >
  <wsdl11>
    <partner name="NCName" partnerLinkType="QName" myRole="NCName"? partnerRole="NCName"?
        />*
    <conversation name="NCName">
      <message label="NCName" operation="NCName" />
    </conversation>*
  </wsdl11>
</interfaceMapping>
```

4.5.3. Scenarios

There are four scenarios about how BPEL^{light} process models of a single DU can be put into production. The existence of MEP processes thereby plays a secondary role. In the following the term "process" is therefore used in its singular form although it denotes a main process and possibly related MEP processes.

First, a DU can contain a BPEL^{light} process model and an a priori given interface description with abstract and concrete parts. The DD's interface mapping relates the process's conversations and messages to suitable abstract interface elements. The interface's concrete part is relevant for publishing the process model on a workflow engine.

The second case is very similar to the first one. It differs only in the interface description's concrete part which is absent in this scenario. A workflow engine publishing such a DU must rely on the chosen server to create an appropriate endpoint the process can be accessed at.

Third, a DU can be given with a BPEL^{light} process model but without an interface description. The DD's interface mapping is used to generate an interface with abstract and concrete parts during deployment of the process.

The fourth scenario resembles the third. By means of an interface mapping an interface is created by the deployment mechanism – but only the abstract part. The concrete part is added on behalf of the server the process model is published on.

Chapter 5

Design and Implementation

This chapter depicts how the presented concepts are put into practice. It works as design for the realized workflow engine and describes the implementation. The Apache ODE workflow engine is taken as basis to be extended in particular points. The focus is thereby not on implementing the complete bunch of considerations but the most important parts to prove the concepts. Thus, the developed software can be regarded as prototype.

ODE's current capabilities were introduced in chapter 3 on page 27. An architecture was given and the components were described on a high level. Concerning this chapter it is considered previous knowledge for a full understanding of the handled issues. The developed workflow engine is referred to as *BPEL^{light} engine* or simply *prototype*.

Although the workflow engine was highly adapted in the course of this work the former functions were preserved. For instance, a BPEL 2.0 process with WSDL 1.1 can be deployed and executed as before. In the particular sections the prototype's capabilities and courses of action are described. ODE's former functionality is thereby not considered.

5.1. Development Environment

ODE was used in revision 658269 of the development trunk. The project advancement was conducted in Eclipse 3.3.2 with Java 5.0. ODE provides two ILs, Axis 2 and JBI. In this work the decision fell on using the approach of Axis 2 for it is powerful enough to meet the requested requirements. ODE came with Axis 2 version 1.3, but the realized prototype relies on version 1.4. This decision influenced the selection of an appropriate runtime environment. With Axis 2 as IL ODE needs an application server to run on. The choice fell on Apache Tomcat in version 6.0.16.

5.2. Deployment

In ODE the deployment of a process means copying a DU into the `processes` directory of the engine (when using the Axis 2 IL). The deployment mechanism underlies significant

changes. The DU stringently contained at least one WSDL 1.1 file as IDL for the process. This requirement is relaxed as a BPEL^{light} DU can come without any interface definition. In this case the DD's interface mapping must be used to generate one. Furthermore, if the main process model references MEP processes, these must be parameterized – a complex operation.

The prototype is able to process BPEL^{light} workflows that are bound against WSDL 2.0 interfaces – are they given a priori or generated. But extension points for WSDL 1.1 interfaces were implemented in the deployment descriptor, parser and Axis 2 IL. Furthermore the design supports to easily insert proprietary program code that handles different IDLs.

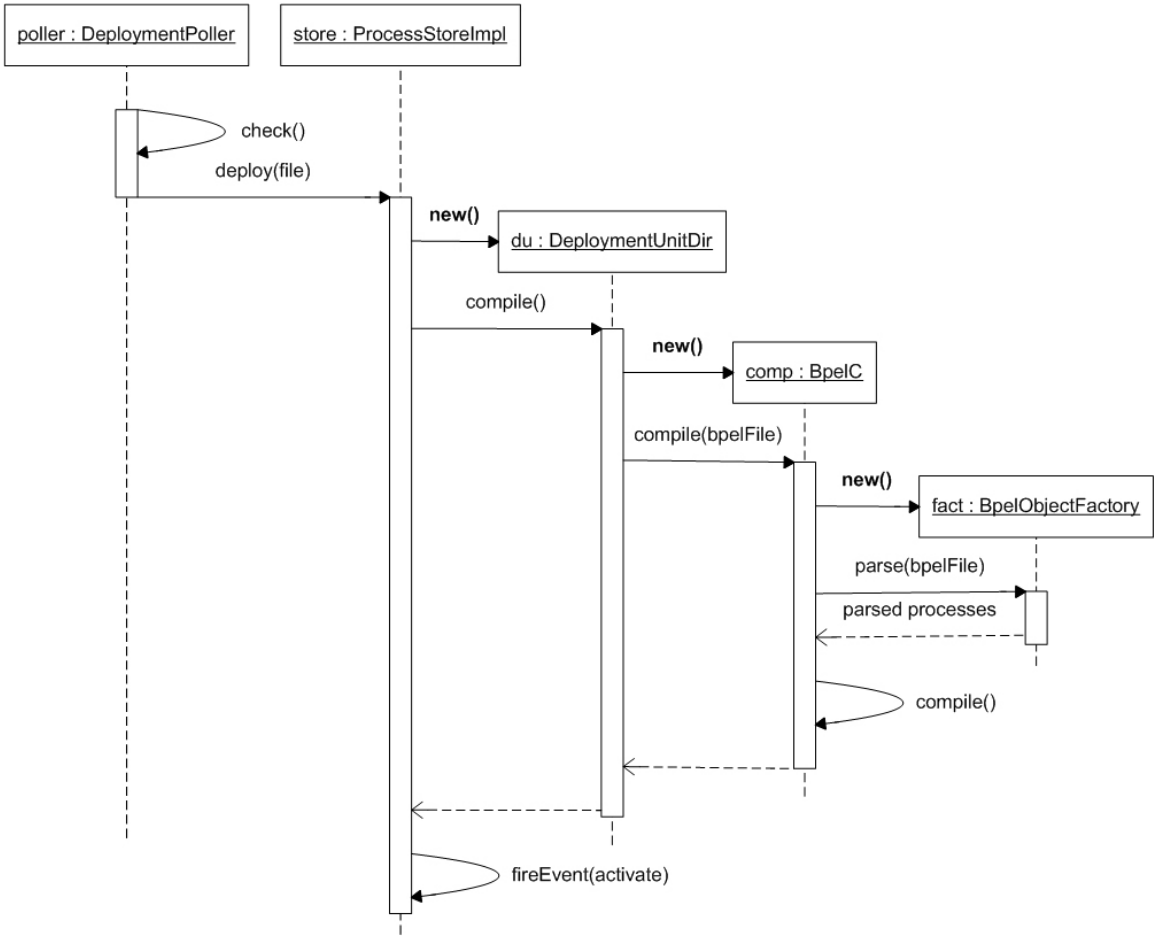

Figure 5.1.: Sequence diagram of the deployment mechanism with Axis 2 IL

Figure 5.1 shows a sequence diagram to illustrate the (adapted) deployment process. The `DeploymentPoller` is part of the Axis 2 IL. It periodically checks the `processes` directory for a changed setting (i.e. new or deleted DUs). The sketched situation happens when a new DU is detected. An object of the class `ProcessStore`, which manages the deployment, is called to deploy the DU (i.e. an object of `DeploymentUnitDir`). The deployment consists of two major steps. First, the contained files are compiled, that is, they are translated into a manageable object representation. Second, the compiled processes are activated in the IL. Both steps

are started by the process store. The entry point for the compilation process is built by the class `BpelC`, a wrapper for actual BPEL compilers. It uses the `BpelObjectFactory` to parse all BPEL processes into a BOM (BPEL Object Model), an object representation of BPEL files based on DOM[1]. After parsing, `BpelC` checks the BPEL files' versions (either 1.1 or 2.0) and delegates the actual compilation to an appropriate compiler. Note that BPEL^{light} process models get compiled by a BPEL 2.0 compiler as BPEL^{light} is a BPEL dialect, not a new version. During compilation the BPEL files are read out with the help of the BOM to create an object representation (the *O-Model*) that is used during runtime to execute the process model. An O-Model integrates a BPEL with all referenced WSDL and XSD files (e.g. message, variable and partner link types are resolved). After compilation the installation of the particular process models in the IL is initiated by an activation event.

During deployment each single BPEL file of the DU is considered. The `DeploymentUnitDir` must reject MEP processes as these are abstract and deployable only after parameterization in the course of the main process's deployment. This is done with the help of the deployment descriptor. Note that MEP processes do not need to be mentioned in the DD since all deployment information (e.g. interface mapping, and partner instances) are assigned to their main process(es). Therefore, if there is no entry in the DD for the regarded process, it is an abstract MEP process. Otherwise it is a main process. The class `BpelObjectFactory` parses BPEL files and returns their BOM representation. It is also the starting point for parameterizing MEP processes if contained in the DU as well as for generating the IDL according to the DD, if necessary. The parsing does not only return the considered BPEL file's BOM but also those of all related parameterized MEP processes. Thus, the class `BpelC` gets a list of BOM processes. These are then passed to the BPEL compiler that creates the appropriate O-Models.

The parameterization and interface generation are complex functions and therefore outsourced to keep a structured design. Figure 5.2 on the next page shows a diagram with all involved classes. For clarity reasons only the most important methods, attributes and return types are given. Parameters are left open since they would hamper the diagram's readability. Essentially they are irrelevant for understanding the coherences. The `BpelObjectFactory` is extended to support MEP processes and the creation of an interface definition. The class `MepParameterizer` contains the functionality needed to convert an abstract MEP process into a concrete one with the help of the main process's complex interaction activity that refers to the MEP process. There are two class attributes. The first denotes the name of the virtual operation that is used to invoke an MEP process. The second contains the name of the generated MEP process conversation that stands for the interaction with its main process. The `InterfaceGenerator` is a wrapper for all operations on interface definitions. Depending on a process's interface mapping it delegates requests to the responsible generator/handler. All handlers must implement the interface `Generator` that is used to decouple IDL specific and compiler logic. There are IDL handlers for WSDL 1.1 and 2.0. The former is only a class with method stubs that acts as a starting point to implement the WSDL 1.1 support for BPEL^{light}. The latter fully implements the binding of BPEL^{light} process models against WSDL 2.0 interface descriptions. The `FileOperations` is a helper class that provides a couple of

[1] Document Object Model – An API to access XML documents

methods to cope with XML documents in general, especially with BPEL^{light} documents and the DD.

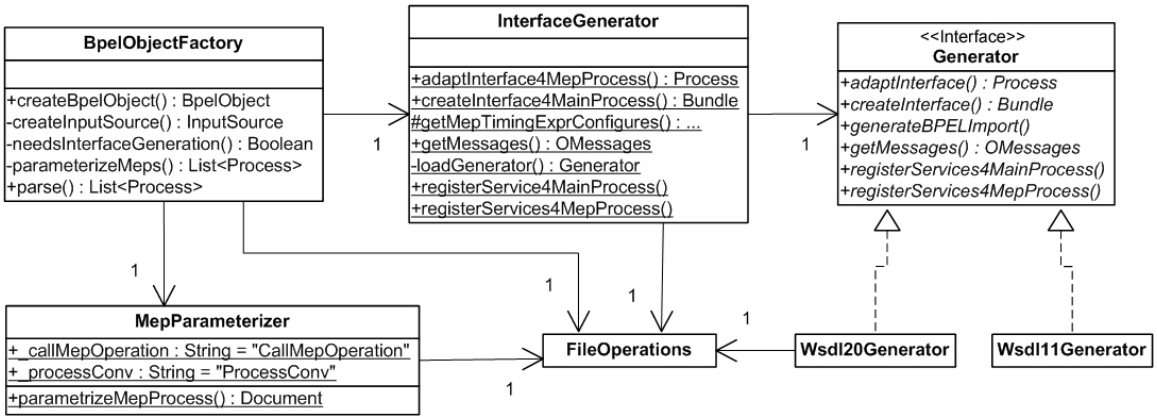

Figure 5.2.: Class diagram of the IDL generation mechanism

5.2.1. Supported Deployment Scenarios

In section 4.5.3 on page 68 four deployment scenarios are outlined depending on two circumstances. First, whether an interface definition is given at all. Second, whether the DD's interface mapping encompasses an interface's abstract *and* concrete parts, or only its abstract part.

In ODE it is not possible to publish a DU as service with an interface including an abstract part only. The service's concrete endpoint must be present before the service can be published: the `ProcessConfImpl` class uses the endpoints to define a mapping on process models. This mapping is later used to activate the endpoints in the appropriate IL. Due to this fact, scenarios two and four are discarded a priori. Only settings that include concrete endpoint information are therefore permitted.

In contrast to that, scenarios one and three are supported with WSDL 2.0 as IDL. That means a DU for a BPEL^{light} process either contains a WSDL 2.0 with an abstract *and* concrete part, or a WSDL 2.0 (again with an abstract *and* concrete part) is generated during deployment with the help of the DD's WSDL 2.0 interface mapping.

5.2.2. Deployment Descriptor

Each DU must contain an XML DD with name `deploy.xml` which is sketched in listing 5.1. It is used to specify how to make a process runnable. The `<deploy>` element is the root node. It contains a declaration for each BPEL process model in the DU, that is, there must be at least one `<process>` element. An exception are BPEL^{light} MEP processes: these are not mentioned in the DD as they inherit all needed data from their main process. The `name` attribute is a mandatory QName value that incorporates the process's name and target

namespace. If the BPEL process is of version 1.1, the `fileName` must be given. A lot of

Listing 5.1 Deployment descriptor of the BPEL^{light} engine

```
<deploy xmlns="http://www.apache.org/ode/schemas/dd/2007/03" ...>
   <process name="xsd:QName" fileName="xsd:string"? ...>
      <active>true|false</active>?
      <mep ciaName="NCName"? mainProcess="QName"? />?
      <in-memory>true|false</in-memory>?
      <interfaceMapping fileName="xsd:string"? mustGenerate="xsd:boolean"? >
         ...
      </interfaceMapping>?
      <partnerInstances>
         <partnerInstance ...>...</partnerInstance>+
      </partnerInstances>?
      ...
   </process>+
</deploy>
```

adjustments can be taken for a process. The `<active>` child node allows deactivating a process after deployment (`false`). The default value is `true`. The `<mep>` element is only relevant for parameterized BPEL^{light} MEP processes: during parameterizing an abstract MEP process an entry is made in the DD that contains only an `<mep>` element referencing the main process (`mainProcess` attribute) and the appropriate complex interaction activity (`ciaName` attribute). This reference is important during activating an MEP process to find the interface mapping of the main process. The `<in-memory>` node declares a process to be executed persistently (`false`) or volatile (`true`). The default value is `false`.

Binding a BPEL^{light} process model against an IDL is realized by a mapping that is declared in the `<interfaceMapping>` element. Its content is omitted here as it adheres strictly to the interface mapping specification in section 4.5.2 on page 65. That is, the prototype comes with two mapping definitions, one for WSDL 1.1, one for WSDL 2.0, whereat the focus lies on the latter. There are two important but optional attributes concerning interface generation, `mustGenerate` and `fileName`. The former says whether to generate an interface definition with the help of the mapping (`true`), or to take a given one (`false`). Its default value is `false`. The latter gives the interface definition's filename. Depending on the `mustGenerate` attribute this can have two meanings. First, if an interface is given, it contains its filename. In this case the attribute is mandatory. Second, if an interface is required to be created, it proposes a filename that is used for the interface. In this case the attribute is optional. If it is omitted, an adequate name is generated. If the interface mapping is omitted, the process model is assumed to be non-BPEL^{light}.

The WSDL 2.0 interface mapping as introduced in section 4.5.2 on page 66 is extended by two additional optional attributes in the `<conversation>` and `<complexIA>` elements: the `service` and `endpoint` are used to reference an endpoint of a predefined WSDL 2.0. An endpoint is related to a conversation during activation to be able to delegate incoming messages to the correct conversation.

The DD permits a static binding of partners against EPRs. The `<partnerInstances>` element works as container for such partner instances. The concrete structure of a `partnerInstance` is not mentioned here as it is described in detail in section 2.6.1 on page 19.

Much more settings can be taken like the specification of process interceptors or properties. These are omitted here as they do not play an important role for deploying BPEL^{light} process models.

5.2.3. Parameterization

If a DU contains a main process that references one or more abstract MEP processes, these must be parameterized during deployment. In short, this means substituting all abstract information by concrete ones. The concrete information are fetched from the main process's complex interaction activity that points to the MEP process. Indeed some more steps have to be taken to make an MEP process executable as shown in section 4.5.1 on page 63. This section dips deeper into the concrete implementation of these steps by the class `MepParameterizer`.

During parameterizing an MEP process the prototype does not consider whether the same MEP process with the same concrete information was parameterized before. This results in a parameterized MEP process per complex interaction activity. At deployment this may lead to a lower throughput. At runtime no reuse of MEP processes takes place leading to an increased usage of resources (e.g. DB space).

MEP Call Bundle

A well-defined XSD element is needed for enabling a main process to communicate with its MEP processes at runtime. On the one hand partner and variable instances as well as the content of outgoing messages have to be passed to an MEP process to be able to carry out its behaviour. On the other hand an MEP process must respond with the value of all incoming messages. Listing 5.2 on the next page presents an XSD file containing an `<MepCallBundle>` element that meets these requirements.

It consists of four child elements which are all of the same type: a `content` element works as container for the passed instance value. The attribute `source` denotes the partner or variable the value is taken from. The attribute `target` points to the partner, variable, or outgoing message the value is to be copied to.

Each of the four elements copies a certain value type from a main to an MEP process or vice versa:

- `<input>` – is used to copy a main process variable instance to an outgoing MEP process message.
- `<output>` – is used to copy an incoming MEP process message value to a main process variable.
- `<variable>` – is used to copy a main process variable instance to an MEP process variable.
- `<partner>` – is used to copy a main process partner instance to an MEP process partner.

Listing 5.2 XML schema of the MEP call bundle data type

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://iaas.uni-stuttgart.de/mep-dt"
           xmlns:tns="http://iaas.uni-stuttgart.de/mep-dt"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="MepCallBundle">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="input"   minOccurs="0" maxOccurs="unbounded"
                type="tns:tContent"/>
            <xs:element name="output"  minOccurs="0" maxOccurs="unbounded"
                type="tns:tContent"/>
            <xs:element name="partner" minOccurs="0" maxOccurs="unbounded"
                type="tns:tContent"/>
            <xs:element name="variable" minOccurs="0" maxOccurs="unbounded"
                type="tns:tContent"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:complexType name="tContent">
      <xs:all>
         <xs:element name="content" minOccurs="1" maxOccurs="1">
            <xs:complexType>
               <xs:all>
                  <xs:any namespace="##other" processContents="lax"/>
               </xs:all>
            </xs:complexType>
         </xs:element>
      </xs:all>
      <xs:attribute name="source" type="xs:string" use="required"/>
      <xs:attribute name="target" type="xs:string" use="required"/>
   </xs:complexType>
</xs:schema>
```

Concretion

One of the most important steps in parameterizing an MEP process is the concretion of its abstract process elements. The MEP process is fetched from the DU by its IRI as given in the referencing complex interaction activity in the main process. The complex interaction activity (directly or indirectly) delivers all information needed to make the MEP process concrete.

First, the MEP process is stored under its own name which is the old one plus "4" plus the complex interaction activity's name (i.e. *oldMepName4complexIAName*). The target namespace stays as is. The complex interaction activity's `mep` attribute is therefore updated to point to the concrete MEP process now. Second, the complex interaction activity's child elements are parsed. Each delivers a concrete piece of information for the MEP process except for the `<bl:partner>` element that has only a meaning at runtime. Following actions are done depending on the kind of child element:

- `<bl:input>` – The input denotes an outgoing message of the MEP process (i.e. a sending simple interaction activity). A new variable is created and added to the MEP

process. Its type is taken from the main process variable that is referenced by the input's `variable` attribute. The `variable` attribute of the MEP process's simple interaction activity is assumed to be set to `##opaque`. It gets the new variable's name.

- `<bl:output>` – The output stands for an incoming message in the MEP process (i.e. an incoming simple interaction activity, a `<bl:onMessage>` of a `pick`, or a `<bl:onEvent>` of an event handler). Similar to the `<bl:input>` a new variable is created the incoming message's opaque `variable` attribute is made pointing at.
- `<bl:infault>` – The infault entails the same behaviour like an input but additionally a `faultName` attribute is set in the outgoing messaging element (i.e. a `<bl:infault>` in a simple interaction activity).
- `<bl:outfault>` – The outfault references an incoming fault message (i.e. a `<bl:outfault>` in a simple interaction activity, a `<bl:onFaultMessage>` in a `pick`, or a `<bl:onFaultEvent>` in an event handler). Again a new variable is created which is then referred to by the incoming fault message element. Furthermore the `faultName` is specified, if present.
- `<bl:variable>` – The variable points to an abstract variable of the MEP process (i.e. a variable with an opaque type). Its type is made concrete according to the main process's variable type.
- `<bl:timingExpression>` – A timing expression concretizes its abstract counterpart, `<mep:timingExpression>`, by simply substituting it by an adequate BPEL construct depending on the expression's type (i.e. a `<for>`, `<until>`, or `<repeatEvery>`). The expression and expression language are set according to the given values.
- `<bl:expression>` – Paremeterizing boolean expressions is not implemented in the prototype.

All cases that create a new process variable or that concretize a variable's data type (i.e. input, output, infault, outfault, and variable) may result in an additional BPEL XSD import that is generated and inserted.

Main Process Interaction

An MEP process exactly describes the behaviour of an MEP as such (i.e. the messages that belong to the MEP). But since the presented concept envisages the invocation of MEP processes by complex interaction activities from within main processes, this interaction with a main process must also be taken into account. As described in section 4.5.1 on page 63 a BPEL^{light} `conversation` and `partner` are therefore created. An incoming, instance creating simple interaction activity, that is placed prior to the regular control flow, receives a bundle of instance data. An outgoing simple interaction activity appended to the regular control flow is used to send a response back to the main process. Two XSD element variables of the type `MepCallBundle` are inserted – one for the incoming, the other for the responding simple interaction activity.

The most complex task in the course of making the MEP process ready to communicate with its main process is the generation of two assign activities that unpack or pack the data bundle, respectively. The unpacking assign activity gets a <copy> element for each partner, local variable and outgoing message of the MEP. Each <copy> assigns an instance value to the specified destination. Furthermore it initializes all variables that belong to incoming messages of the MEP. Figure 5.3 shows an example of an unpacking `assign`. The main process's complex interaction activity has three child elements: an input, output and partner. The MEP call bundle obtained by the MEP process is stored in the variable "receive". The first `copy` initializes the incoming variable of the message "in". The second `copy` gets the content of the input with target "out" and assigns it to the generated variable "outVar". The third `copy` fetches the partner instance with target "mepPartner" and copies it to this partner.

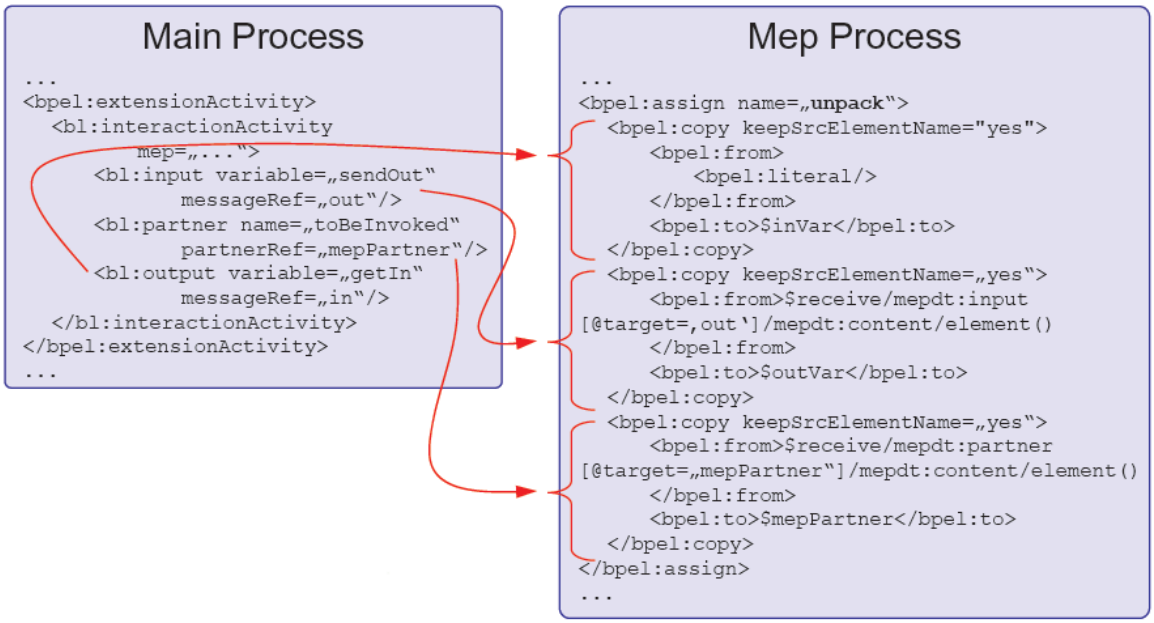

Figure 5.3.: Generating the unpacking `assign` activity in an MEP process

The `assign` packing the response for the main process consists of three parts. First, it initializes the response variable with an <mepdt:MepCallBundle> element and a number of <mepdt:output> children stubs. This number depends on the quantity of incoming messages of the MEP process: The variable instance of each incoming message is sent back to the main process. Second, the `target` attribute of each <mepdt:output> element is set with the help of <mepdt:output> stubs of the incoming MEP call bundle (see 5.7 on page 85). Third, the variable instances are copied into the bundle. Figure 5.4 on the next page continues the just considered example by an appropriate packing `assign`. It initializes the "response" variable by a single `output` the source of which is set to the name of the related incoming message. The content is filled with a placeholder element that is substituted in part three of the assignment. After that the `source` attribute's value of the `output` stub of the incoming MEP call bundle is fetched. It denotes the main process's target variable the result is to be copied

to (i.e. the variable "getIn"). Finally, the "inVar" variable's instance data is inserted into the `output`'s `<mepdt:content>` element of the bundle. The placeholder thereby disappears.

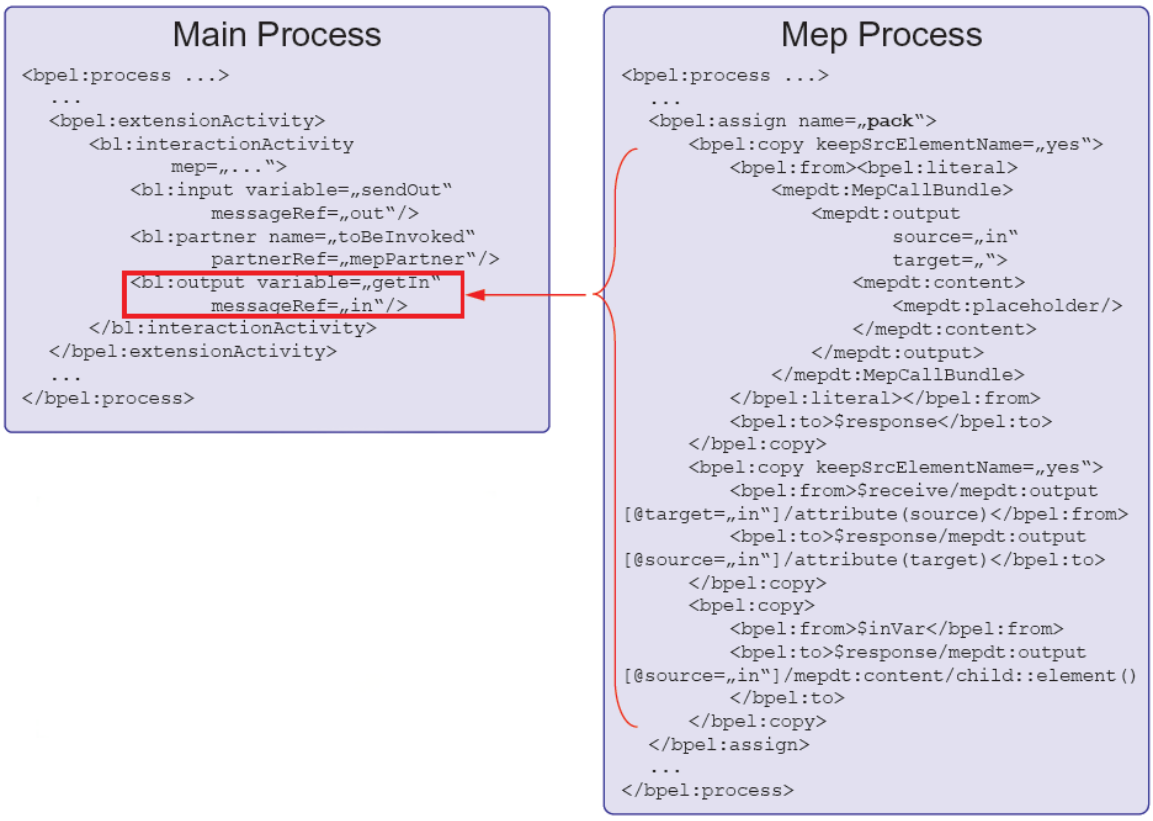

Figure 5.4.: Generating the packing `assign` activity in an MEP process

Deployment Descriptor Adaptation

After an MEP process is parameterized an entry must be made in the DD. This is important to be able to activate the concrete MEP process after deployment. The entry is restricted to a number of important information. These are the concrete MEP process's name, the referencing main process's QName as well as the main process's complex interaction activity that points to the MEP process.

The executable MEP process does not need static partner EPRs as it gets partner instance information at runtime by the main process. An interface mapping is also redundant as the MEP process inherits the interface mapping from its main process. That means the reference to its main process and the corresponding complex interaction activity is all information needed to publish an MEP process. Listing 5.3 on the facing page shows an example of a DD entry generated for a parameterized MEP process. Its generated name contains the abstract MEP process's name ("Out-optional-in") and the referencing complex interaction activity's name ("complexIA"). The `<mep>` element signals that the entry is for a

parameterized MEP process instead of a main process. It contains a reference to its main process ("pns:LoanApproval") and its complex interaction activity. The latter is used to find the main process's interface mapping operation that makes the MEP accessible.

Listing 5.3 Example of an entry in the deployment descriptor for a parameterized MEP process

```
<deploy xmlns="http://www.apache.org/ode/schemas/dd/2007/03" ...>
   ...
   <process name="pns2:Out-optional-in4complexIA">
      <mep ciaName="complexIA" mainProcess="pns:LoanApproval" />
   </process>
</deploy>
```

5.2.4. Interface Generation

If a DU comes with BPEL^{light} process models but without an interface definition, it is generated with the help of the DD during deployment. The prototype allows binding BPEL^{light} processes against WSDL 2.0 interfaces. Its design provides extension points for different IDLs. In principle, each IDL can be taken to publish BPEL^{light} process models if an adequate interface mapping is defined. But up to now only XML-based IDLs are supported due to their import as DOM documents. Another requirement is that the used IL must be able to handle the IDL. The BPEL^{light} engine makes use of the Axis 2 IL. Since Axis 2 supports WSDL 2.0 and 1.1 only, different IDLs cannot simply be integrated although suitable extension points are given. An appropriate IL must also be provided on top of ODE that understands the selected IDL.

The interface mapping of the DD is the grounding for creating an interface for a main process. MEP processes do not declare an interface mapping on their own. They rather inherit the mapping from the specifying main process.

In the following the interface generation is described with focus on WSDL 2.0 as this is the implemented scenario. When using another IDL all WSDL 2.0 specific tasks must be adapted accordingly. The creation of the interface is carried out in several steps, independent of the selected IDL. First, when the BPEL^{light} main process is considered, its interface mapping is taken to create a new interface file. Second, the file is adapted for each related MEP process.

The WSDL 2.0 interface generator (i.e. the class `Wsdl20Generator`) places the WSDL 2.0 file in the DU's directory. Its name is specified per `filename` attribute if present, or, if absent, generated with the main process's name as basis. Its target namespace is set to the main process's target namespace with "`/wsdl`" as appendix. The interface is filled with all abstract and concrete information as given by the DD or prescribed by the main process itself.

First, the main process's XSD imports are candidates for being incorporated in the WSDL file. They cannot be simply copied as the WSDL imports slightly differ from BPEL imports. They are therefore transformed and added to the definition. Second, the interfaces and operations are generated according to the interface mapping's abstract part. An operation

is created for each conversation. The conversation's `mep` attribute is thereby taken as value for the operation's `pattern` attribute. Incoming and outgoing regular and fault messages are not defined in the interface mapping. They are rather fetched out of the main process's BPEL file by analyzing all messaging elements. Only operations that are directly implemented by the main process are considered (i.e. operations that are described by inline MEPs). Third, for each specified endpoint a binding is created. There are two binding types: SOAP and HTTP. In case of a SOAP binding the version and transport protocol are read out of the interface mapping. Furthermore an entry for each operation is added with the `soap:action` attribute set to "urn:" plus operation name (e.g. `urn:abortOrder`). When using an HTTP binding the global property `methodDefault` has to be set in the binding. Additionally, operation-specific characteristics (e.g. location, method and different serializations) are evaluated and inserted as separate `<operation>` elements. Fourth, the WSDL's services and endpoints are generated. In ODE it is not possible to publish one and the same interface under two different services or endpoints.

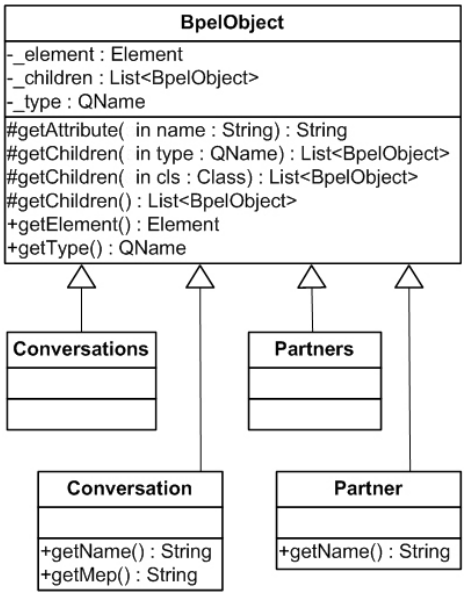

Figure 5.5.: BPEL^{light} extension of ODE's BPEL Object Model

For each MEP process the WSDL 2.0 is enriched with additional information for its abstract part. The MEP process's XSD imports are incorporated into the WSDL. A single operation is added to the specified interface. This operation encompasses all messages (including fault messages) the MEP process sends and receives. Additionally, for each parameterized timing expression two `<mep:configure>` elements are inserted into the operation. One of them assigns the timing expressions type, the other its concrete expression. The injection of a `<mep:configure>` element for boolean expressions is not implemented yet.

To insert a proprietary IDL generator an appropriate class must be added to the BPEL^{light} compiler package[2]. There are two requirements such a class must meet. First, it needs to implement the `Generator` interface. Second, its name is assumed to be composed of the local name of the DD's interface mapping element and the appendix "Generator" (e.g. "SsdlGenerator").

5.2.5. BPEL Object Model

ODE's BOM is used by the compiler to parse BPEL processes. Each BPEL element has therefore a counterpart in the BOM. As BPEL^{light} introduces additional elements that must be parsed during the compilation process appropriate classes must be injected into the BOM. This encompasses BPEL^{light} elements that are not wrapped by a BPEL extension mechanism, that is, `<bl:conversations>`, `<bl:conversation>`, `<bl:partners>` and `<bl:partner>`. The

[2] `org.apache.ode.bpel.compiler.bpellight`

`interactionActivity`, for example, does not need a matching BOM class as its enclosing `extensionActivity` is used to parse it.

The BOM's basis is built by the class `BpelObject`. It helps accessing a single BPEL element, its attributes and children. All BOM classes are specializations of `BpelObject`. Figure 5.5 on the preceding page shows a class diagram with the BPEL^{light} extension of the BOM: a class is created for each mentioned BPEL^{light} element, all inheriting from `BpelObject`. The class `conversations` and `partners` have the only function to deliver their child elements, that is the nested `conversations` and `partners`, respectively. The `partner` and `conversation` classes facilitate accessing the particular attributes. These new BOM classes are registered in the element-object mapping of the `BpelObjectFactory`.

The prototype does not yet implement BPEL^{light}'s `<bl:expressions>` and `<bl:expression>` elements (compare section 4.2.1 on page 40). If they may be added in future, the appropriate classes should inherit from `BpelObject` and should be registered in the `BpelObjectFactory`.

5.3. BPEL^{light} Activities

As presented in the previous section BPEL^{light} constructs that are not supported by a BPEL extension mechanism have to be integrated in ODE's BOM. This is the most reasonable approach to access them. In contrast to that, the BPEL^{light} activities (i.e. `<bl:interactionActivity` and `<bl:pick>`) are regular BPEL extensions, that is, they are nested in a BPEL `extensionActivity`. ODE provides a generic interface for the integration of extension bundles that must be registered in the `ode-axis2.properties` file of the `conf` directory. A bundle defines a mapping of an element's name on the realizing class which contains the element's validation and runtime logic. That way extension activities and operations can be added to ODE without touching the engine's code.

This mechanism is therefore the eligible candidate for enriching ODE with BPEL^{light} activities. The BPEL^{light} bundle is built in its own Eclipse project with name "bpel-light". Figure 5.6 on the next page illustrates its class structure. Methods, attributes and parameters of secondary importance are omitted to ease the readability. All displayed classes are contained in the package `org.apache.ode.extension.bpellight` except for `AbstractAsyncExtensionOperation` and `AbstractExtensionBundle` which are part of ODE's extension point. The bundle is registered to ODE in the Axis 2 properties file with following entry:

ode-axis2.extension.bundles=org.apache.ode.extension.bpellight.BpelLightBundle

The `BpelLightBundle` class maps the BPEL^{light} elements `<bl:interactionActivity>` and `<bl:pick>` on the classes `IAOperation` and `PickOperation`, respectively. Both extend an abstract class that is the plug point for asynchronous extension operations. Two functions are realized: validating the extension element during compilation and executing it at runtime. In both cases these functions are delegated to helper classes to keep the design sound.

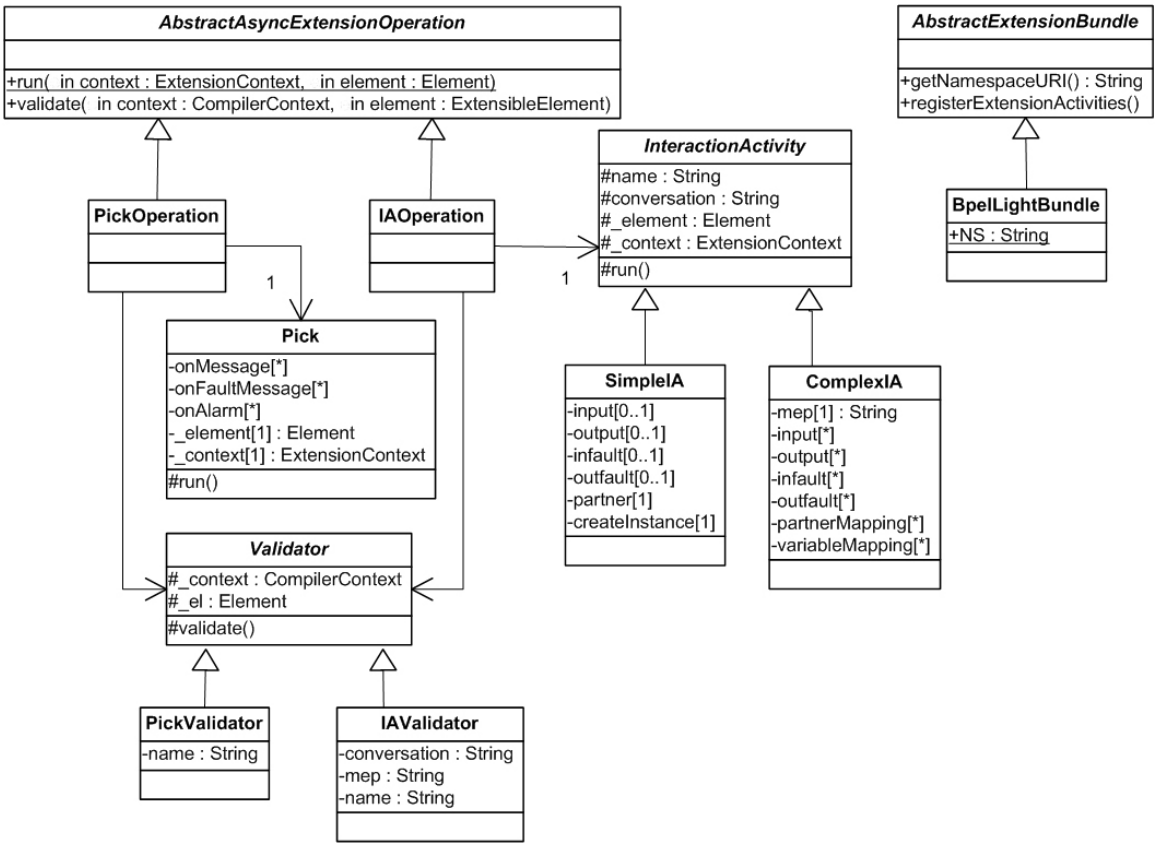

Figure 5.6.: Class diagram of the BPEL^{light} extension bundle

The different validators are generalized by an abstract `Validator` class that holds common attributes. The abstract class `InteractionActivity` encapsulates the two usage scenarios of interaction activities, simple and complex. Depending on the case the interaction activity can contain one (simple interaction activity) or more (complex interaction activity) messages. The BPEL^{light} pick activity's runtime behaviour is implemented by the class `pick`.

The following steps should be considered to extend the bundle by additional activities:

- Implement an abstract extension operation (synchronous or asynchronous) to have a starting point for the validation and runtime behaviour.
- Extend the element-class mapping of the `BpelLightBundle`.
- Implement a validator class that realizes a static analysis of the activity's XML element.
- Implement a runtime class that forms the activity's execution logic.
- Plug these two classes into the operation class of the activity.

5.3.1. Interaction Activity

The abstract interaction activity implementation holds common attributes for the two interaction activity scenarios. It contains a factory that instantiates and returns a simple or complex interaction activity depending on the `mep` attribute of the `<bl:interactionActivity>` element. Both implement the `run` method that is the interaction activity's starting point.

Simple Interaction Activity

If the interaction activity element does not contain the `mep` attribute, it is a simple interaction activity scenario. The prototype's simple interaction activity implementation focusses on sending and receiving non-fault messages. The behaviour is divided into the incoming and outgoing case.

The receiving interaction activity either creates a new process instance or receives a message destined for a running process instance. The former does not need correlation handling. The arrived message is fetched and its value is stored to the `<bl:output>` element's variable. If the partner provides a callback EPR within the message, it is set as *current* EPR to the appropriate partner instance. Furthermore, if the message contains a session identifier, the partner is also stateful. The identifier (ID) is assigned to the partner instance to be inserted into a follow-up message that may be sent to this partner later on in the process. If the interaction activity receives a message in a running process instance, the correlation mechanism plays an important role. The prototype is able to handle ODE's *implicit correlation* that is based on session IDs exchanged between stateful partners. The BPEL^{light} correlation is not implemented yet. If the engine was able to correlate the message to the simple interaction activity instance, processing it is similar to the create instance case. The process's control flow may activate an incoming interaction activity before the expected message arrives. In this case the interaction activity waits for it.

MepCallBundle

```
<mepdt:MepCallBundle>
    <mepdt:input source=„sendOut"
            target=„out">
        <mepdt:content>
            ...
        </mepdt:content>
    </mepdt:input>
    <mepdt:partner source=„toBeInvoked"
            target=„mepPartner">
        <mepdt:content>
            <partnerInstance ...>
                ...
            </partnerInstance>
        /mepdt:content>
    </mepdt:partner>
    <mepdt:output source=„getIn"
            target=„in"/>
</mepdt:MepCallBundle>
```

Figure 5.7.: Example of an MEP process call

The sending interaction activity transmits the content of the `<bl:input>` element's variable to the specified partner. If a session ID is assigned to the partner instance, it is injected into the message. Again the prototype does not provide BPEL^{light} correlation.

Complex Interaction Activity

If the interaction activity element contains the `mep` attribute, it is a complex interaction activity scenario. The BPEL^{light} engine prohibits the specification of a `conversation` for a

complex interaction activity. This ensures that a complex interaction activity is not part of a `conversation` that contains additional activities (i.e. the use of intermixed MEPs is avoided). As a `conversation` is essential for the communication mechanism in ODE the interaction activity validator generates a `conversation` for each complex interaction activity during compilation and adds it to the process scope.

At runtime a complex interaction activity invokes a parameterized MEP process denoted by the IRI of the `mep` attribute. It creates an outgoing message bundle (i.e. an *MEP call bundle*) depending on its inputs, outputs, variables and partners. Each results in an entry in the bundle. The prototype disregards `infaults` and `outfaults`. Figure 5.7 is the continuation of the example introduced in figure 5.3 on page 79. It shows an MEP message bundle that is built by an interaction activity with an `input`, a `partner` and an `output`. The source variable/partner is specified as well as the target variable/partner/message in the appropriate attribute. The particular instance content is copied into the bundle. The `output` is only reflected by a stub in the bundle without instance data. This stub is used by the MEP process to get the `output`'s variable name for its response. After creating the message the MEP process is invoked synchronously. As it is an internal invocation (i.e. excluding the detour of an IL) the correlation between main and MEP process is simply done by process instance ID. If the MEP process's answer arrives, it is parsed for all `output` elements. Each denotes a message the MEP process received during its execution. The particular instance data is copied into the variable identified by the value of the `target` attribute. Figure 5.8 (the continuation of the sample in illustration 5.4 on page 80) exemplarily shows an MEP process response that contains a single `output`. Its content is assigned to a variable with name "getIn".

MepCallBundle

```
<mepdt:MepCallBundle>
   <mepdt:output source=„in"
         target=„getIn">
      <mepdt:content>
         ...
      </mepdt:content>
   </mepdt:output>
</mepdt:MepCallBundle>
```

Figure 5.8.: Example of an MEP process response

5.3.2. Pick

The prototype's BPEL^{light} `pick` activity focusses on receiving regular messages. Fault messages are not considered. The `pick` acts like several incoming simple interaction activities. That means it waits for a message that matches one of the defined `<bl:onMessage>` elements. If a matching message arrives a new process instance may be created (`createInstance` attribute set to `yes`). In this case no correlation is done. The message content is stored to the appropriate variable and the contained activity is invoked. Possibly delivered callback EPR and session ID are assigned to the related partner instance. If the message is destined for a running process instance, implicit correlation is done. BPEL's correlation mechanism is not implemented by the prototype. If the engine was able to correlate the message, the behaviour is the same as in the create instance case.

If the `pick` declares one or more `<bpel:onAlarm>` elements, waiting for an incoming message can be aborted due to a timeout. In this case the appropriate `onAlarm`'s activity is executed.

However, only a single child of a `pick` is allowed to be executed. All remaining children are handled by a dead path elimination (see [AAA+07]).

5.4. Runtime Model

ODE's engine underlies significant changes due to BPEL^{light}'s interaction model that is based on arbitrary complex `conversations` that are spanned over multiple partners and messages. Figure 5.9 on the following page shows the class diagram of the BPEL^{light} engine's runtime model. It contains both legacy and new classes. The legacy core classes (i.e. `BpelServer` and its implementation, `BpelProcess`, `BpelRuntimeContext` and its implementation) are extended to support BPEL^{light}. The former functionality is thereby preserved, that is, BPEL 1.1 and 2.0 process models are managed as before. The diagram only shows the most important classes, attributes and methods. All method parameters, return types as well as legacy attributes and methods are omitted completely to increase the diagram's readability. The class `PartnerLinkInstance` is just mentioned for integrity reasons.

The BPEL server is the starting point for accessing a BPEL process. It routes incoming messages to the correct process that creates an incoming message holder. This helper class's function is – as its name says – to hold the incoming message and additional context information. It triggers the message correlation and invocation of the intended process.

The `BpelProcess` class is the center of the runtime module. For each deployed process model there is a `BpelProcess` object that delegates the tasks of all appropriate process instances. The particular process instances are managed by the BPEL runtime context. It uses the JACOB engine to process BPEL workflows. Methods are provided to access instance data, select incoming messages, invoke a partner or MEP process, and more.

5.4.1. Conversations and Partners

In ODE the communication model is based on message exchange classes in different varieties. A message exchange stands for a combination of a partner link and operation. That means it is a container for at most two messages (a request and an optional response). Depending on the direction of the first message (i.e. the request) ODE distinguishes partner role (outgoing request) and my role message exchanges (incoming request). The destination of outgoing messages is stored in EPRs that are assigned to partner links.

This paradigm is renewed by BPEL^{light}. A conversation now stands for an operation (when using WSDL 2.0 as interface mapping). It may cover more than two messages, that is, arbitrary complex message exchanges. There is no distinction of partner and my role message exchanges anymore. As a conversation can include more than one partner, it must be able to hold multiple EPRs. That means the BPEL^{light} engine's runtime model is conversation based. Therefore a new message exchange is introduced (the `ConversationMessageExchange`) that contains interaction related data. The `ConversationImpl` implements a conversation's runtime behaviour (i.e. creating a new process instance, correlation of incoming messages, invoking of external partners). The implementation of a partner (i.e. the class `PartnerImpl`)

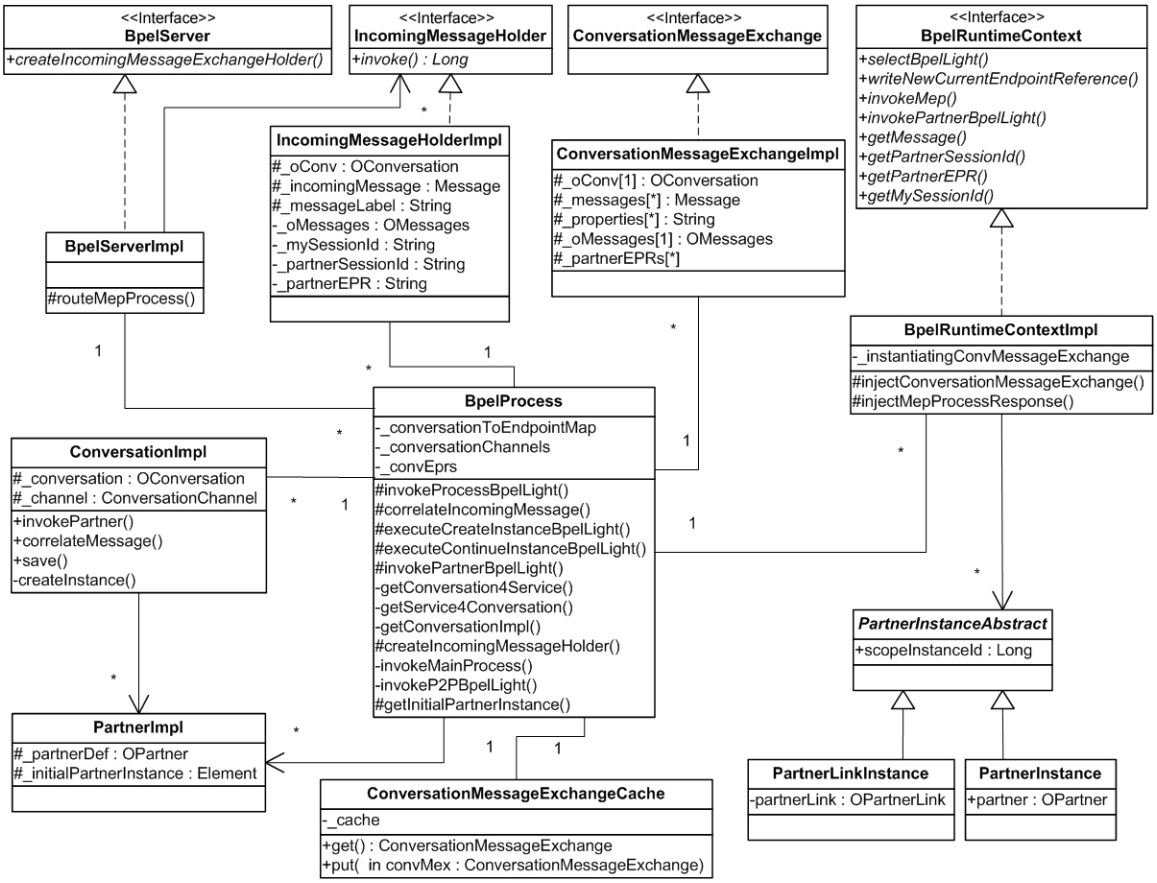

Figure 5.9.: BPEL^{light} extension of ODE's runtime environment

holds the compiled partner definition and its static EPR if one was assigned with the help of the DD. A `PartnerInstance` does not directly contain partner instance information (i.e. the current EPR and the EPR list). A BPEL^{light} partner is rather considered an XSD-element variable with a fixed data type as introduced in section 2.6.1 on page 20. A `PartnerInstance` object can be used to find its scope instance (by scope instance ID) that has access to the partner variable instance.

The prototype is restricted to support ODE's *implicit correlation* based on session IDs but not yet BPEL^{light}'s correlation mechanism.

5.4.2. Assign Activity

As BPEL^{light} partners are handled like BPEL variables the `assign` activity is inherently able to cope with them by using the expression variant. Nevertheless the `assign`'s runtime implementation probably possesses a bug that hampers copying partner instances (e.g. with the from-spec `<from>$partnerA</from>` and the to-spec `<to>$partnerB</to>`). In this case the parent of the target partner's `<partnerInstance>` element delivers `null` instead of the

document root leading to a `NullPointerException`. A workaround that takes the target's document root (not the parent) and appends the source element solves the problem.

The BPEL^{light} extension of the `assign` (i.e. the *partner variant*) is not yet implemented by the prototype. This can be done by adapting the class `ASSIGN` of the runtime component.

5.4.3. O-Model

ODE's O-Model is the compiled representation of BPEL processes. O-Model constructs are used at runtime to access a process's data items (e.g. a variable type, or a partner link's role). BPEL^{light} elements that are not covered by BPEL's extension mechanism (i.e. extension activity and operation) must be reflected by the O-Model to be accessible at runtime. This extension encompasses conversations (i.e. the class `OConversation`) and partners (i.e. the class `OPartner`) as can be seen in the class diagram of figure 5.10. The diagram only shows new attributes of the legacy classes `OBase`, `OScope` and `OProcess`. Methods and attribute types are omitted for increasing the readability. The dotted generalization arrow of `OScope` is not UML[3] conform. It denotes an indirect inheritance over classes that are unimportant in the context of this diagram. Partners and conversations can be declared on process and scope level. The `OMessages` class is an IDL-independent container for regular and fault messages that are covered by a `conversation`.

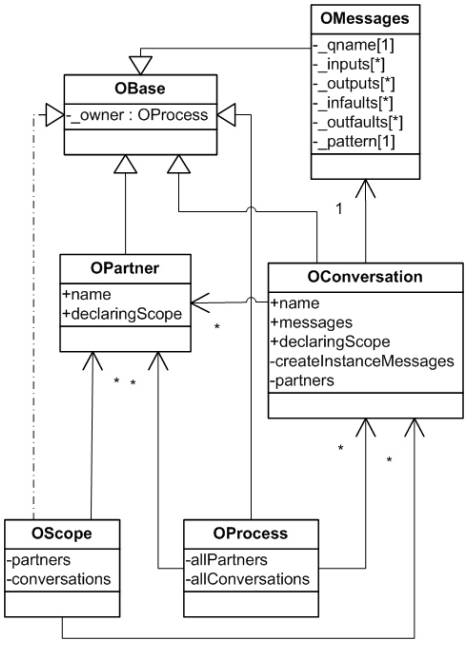

Figure 5.10.: Class diagram of BPEL^{light}'s O-Model extension

The BPEL^{light} engine does not yet implement BPEL^{light}'s `<bl:expression>` element (compare section 4.2.1 on page 40). If it will be added in future, an adequate O-Model construct should be implemented that is associated with `OProcess` and `OScope` – similar to conversations and partners.

5.5. Integration Layer

ODE supports the use of two ILs, Axis 2 and JBI. In the course of this work the Axis 2 IL is extended to support BPEL^{light}'s communication model. Therefore the Axis 2 framework version 1.4 is adapted in a diploma thesis running coexistent (see [Tos08]). In short, Axis 2 is enriched with ESB functionality. On that account it is referred to as the *service bus* in the following.

[3]Unified Modeling Language

The class diagram in figure 5.11 on the facing page illustrates the extension of ODE's Axis 2 IL. Only the most important classes, data types, and methods are shown. Parameters and return types are omitted. In taking a closer look at the diagram two use cases are observable. First, binding a BPEL^{light} process model against a WSDL 2.0 interface description. Second, selecting a WSDL 1.1 interface definition for publishing BPEL^{light} process models. However, the focus lies on the WSDL 2.0 use case. The WSDL 1.1 solution is not implemented by the prototype; the appropriate classes and methods are empty stubs for the most parts, specified to provide an extension point for a follow-on development. The remainder of this section concentrates on the WSDL 2.0 approach.

The legacy Axis 2 IL strictly separated the handling of operations that first receive a message (request-response or one-way from the *process's* point of view) from those that first send a message (request-response or one-way from the *partner's* point of view). This approach was acceptable since an operation contained at most two messages that were considered synchronous and therefore transmitted over the same channel. Complex operations with more than two messages over more than one channel were not supported.

With WSDL 2.0 such complex operations come into play the handling of which is not distinct by the direction of their initial message anymore. That means the IL implements an asynchronous interaction model as intended by BPEL^{light}. It is up to the service bus how to lay synchronous interactions over pairs of messages, and if at all, depending on the operations a partner supports. This makes BPEL^{light} process models more flexible in the choice of matching partners. For instance, a BPEL^{light} process that implements a request-response operation is not restricted to communicate with partners that implement a solicit-response operation. Partners that realize a notification and one-way operation can also be considered.

The service bus controls the message flow within an operation by an operation context. All incoming and outgoing messages of an operation instance must be associated to the same operation context object. This context is indirectly accessed via an operation client from the IL's perspective. The `ODEServer` is extended to hold references to operation client objects that can be accessed when a message is received and when a message is sent. This cache is important to work on the correct operation context object. The two cases are examined in the following.

5.5.1. Incoming Messages

The `ODELocalPartner` is a receiver of all incoming message. It is the implementation of a local partner interface provided by the service bus. A local partner object is hooked into all Axis operations of an Axis service. That means there is a single local partner object that handles all incoming messages for a service. The messages are delegated by the `BpelServer` to the process that implements the service and then correlated to the correct process instance (compare section 5.4 on page 87).

If the incoming message is the first of this operation instance, a new operation client object is created and stored to the `ODEServer`. The delivered operation context is attached to this client. This operation client object must be used for outgoing follow-on messages of this

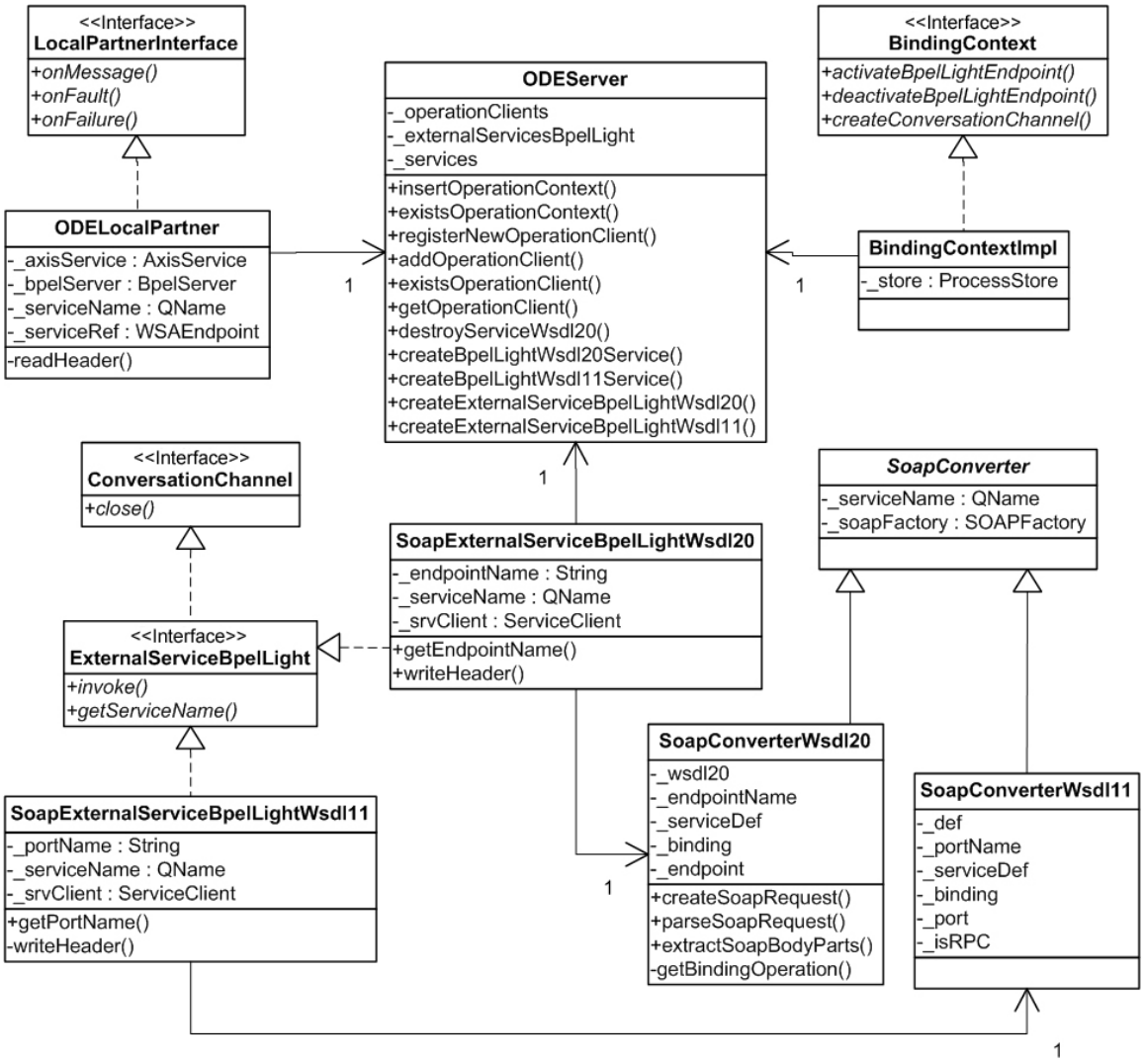

Figure 5.11.: Class diagram of the Axis 2 IL with BPEL^{light} extension

operation instance. If the incoming message is not the first of its operation, an appropriate operation client object already exists.

5.5.2. Outgoing Messages

Outgoing messages are sent over a WSDL 2.0 implementation of the interface `ExternalServiceBpelLight`. Sending a message is done with an Axis operation client object. If the message is the first of its operation, a new operation client object is created, stored to the `ODEServer`, and used to deliver the message. If the message's operation instance is already running, the appropriate operation client is fetched from the `ODEServer` and used to send the message.

5.5.3. Binding

Since SOAP is a widespread message format the prototype supports a SOAP binding only. Although an HTTP binding is technically possible, it is not pursued. On the one hand it would bypass the service bus. On the other hand it would have an impact on the complexity of MEPs.

5.6. Persistence Layer

ODE comes with two built-in persistence layers, OpenJPA and Hibernate, as well as with an in-memory implementation of the persistence interface. The prototype follows the approach of OpenJPA. The execution of in-memory BPEL^{light} process models is considered secondary as it lacks asynchronous interactions and is therefore not implemented. However, the Hibernate and in-memory modules are provided with stubs to easily realize them later on.

The BPEL^{light} engine's conversation message exchange and partner instances are candidates for being stored in a DB during execution. As partners are handled like variables no adaptation of the persistence layer is needed. In contrast to partners, conversation message exchanges are not reflected by the legacy DAOs. Therefore a new DAO interface for the conversation message exchanges is created as well as an appropriate OpenJPA implementation class. This `ConversationMessageExchangeDAO` persistently stores a complete conversation message exchange, that is, its messages, process instance ID, EPRs, channels, correlation keys, properties, and so on. This new DAO is weaved into the persistence layer structure. That means it is associated with message, property and correlator DAOs as well as the connection DAO which is used to create a new or fetch an existing conversation message exchange DAO.

Chapter 6

Summary and Outlook

BPEL^{light} is a WSDL-less BPEL dialect that facilitates modelling executable, IDL-independent process models resulting in a set of advantages. The development of business processes is eased due to a separation of roles – a process designer defines a process; an IT specialist makes it runnable on a workflow engine. Each role can thereby concentrate on its core competencies. Additionally, the reusability of (parts of) process models is increased since they are independent of concrete port types and operations. A process model can be bound against any IDL. BPEL^{light} leverages an asynchronous interaction model (that can simulate synchronous interactions when needed) being flexible to communicate with a broader spectrum of services. The decision on a blocking or non-blocking communication is thereby first made at deployment or runtime.

MEPs prescribe the interaction between partners in an abstract manner. WSDL 2.0 introduces arbitrary complex interface operations describable by a "Template for Message Exchange Patterns". As discussed this template is limited in its expressiveness. These limitations can be overcome by using BPEL^{light} as MEP definition since it can describe the message flow between and within Web service operations.

This thesis discussed the application of MEPs in a general business process context. It extended the first version of BPEL^{light} to achieve two objectives. First, the separation of BPEL and WSDL is pushed. Previously unregarded concepts like event handling, fault handling, and correlation were considered. The new `<bl:expression>` construct is introduced that allows the definition of referenceable and therefore reusable expressions. The interaction model was adapted to an asynchronous sending and receiving of fault messages. Second, the concept of defining MEPs with BPEL^{light} is integrated into BPEL^{light} process models as description of the interaction activity's behaviour. On the one hand this enables an interaction activity to express an arbitrary complex message exchange. On the other hand it facilitates a recursive definition of MEPs.

Furthermore the presented concepts were proven by the specification and prototypical implementation of a BPEL^{light} workflow engine on basis of the Apache ODE project. It allows binding BPEL^{light} process models against WSDL 2.0 interfaces at deployment. The integration with an Axis 2 service bus [Tos08] makes it possible to publish process models and to communicate with eligible partners. BPEL^{light} MEP processes, which describe the behaviour of interaction activities, are also used to carry out the specified behaviour at

runtime. They are therefore parameterized during deployment and synchronously invoked when the appropriate interaction activity is executed. A sound and extensible design was emphasized in the development of the prototype. There are extension points for WSDL 1.1 as well as for additional BPEL^{light} activities and elements.

Since the BPEL^{light} workflow engine is a prototypical implementation a number of issues can be addressed in a follow-on development. BPEL^{light}'s event handling is not realized so far. The `assign` activity's empty from- and to-spec may be extended to provide a partner variant. BPEL^{light}'s expression concept may be reflected in a further version. It will result in a higher flexibility and expressiveness in defining MEP processes. The proposed BPEL^{light} correlation mechanism is not yet implemented. In general, considerations about outsourcing correlation to the service bus are worthwhile. This would avoid redundancy since the ESB also uses correlation to find correct operation instances. The prototype's interaction model concentrates on regular messages. Sending and receiving fault messages as well as handling fault messages in parameterized MEP processes should be considered in future. Finally, the reuse of parameterized MEP processes at runtime is still to be realized. The deployment mechanism should be extended to check whether an MEP process was already parameterized before concretizing it.

Appendix A

Message Exchange Patterns

This appendix applies abstract BPEL^{light} to define a set of MEPs. The focus is thereby on non-trivial MEPs to prove the concept.

- *Robust-out-only*
  A request is sent to a partner. If the partner fails to process the message, a fault message may be sent in response. A timeout aborts waiting for the optional fault message depending on a timing expression.
- *In-optional-out*
  A message is received from a partner. Depending on a boolean condition a message is sent in response to the same partner or nothing is done.
- *Robust-in-only*
  A message is received from a partner. Depending on a boolean condition a fault message is sent in response to the same partner or nothing is done.
- *Out-optional-in*
  A request is sent to a partner. A message may be sent in response. A timeout aborts waiting for the optional message depending on a timing expression.
- *In-xor-out*
  A message is received from a partner. Depending on a boolean condition one of two (different) messages is sent in response to the same partner.

Listing A.1 "Robust-out-only" MEP in abstract BPEL^{light}.

```
<bpel:process name="Robust-out-only"
     abstractProcessProfile="http://iaas.uni-stuttgart.de/bpel-light/abstract/mep/2008/"
     targetNamespace="http://iaas.uni-stuttgart.de/mep-in-bpel"
     xmlns:bpel="http://.../wsbpel/2.0/process/abstract"
     xmlns:bl="http://iaas.uni-stuttgart.de/bpel-light"
     xmlns:mep="http://iaas.uni-stuttgart.de/bpel-light/abstract/mep/2008/">
   <bl:partners>
      <bl:partner name="aPartner"/>
   </bl:partners>
   <bl:conversations>
      <bl:conversation name="robust-out-only"/>
   </bl:conversations>
   <bpel:sequence>
      <bpel:extensionActivity>
         <bl:interactionActivity name="sendRequest" conversation="robust-out-only">
            <bl:input messageLabel="out" variable="##opaque"/>
            <bl:partner name="aPartner"/>
         </bl:interactionActivity>
      </bpel:extensionActivity>
      <bpel:extensionActivity>
         <bl:pick name="receiveFault">
            <bl:onFaultMessage messageLabel="inFault" faultName="##opaque"
                conversation="robust-out-only" partner="aPartner" variable="##opaque">
               <bpel:empty/>
            </bl:onFaultMessage>
            <bpel:onAlarm>
               <mep:timingExpression timingExpressionLabel="timeout"
                   expressionLanguage="##opaque" type="##opaque" expression="##opaque" />
               <bpel:empty/>
            </bpel:onAlarm>
         </bl:pick>
      </bpel:extensionActivity>
   </bpel:sequence>
</bpel:process>
```

Listing A.2 "In-optional-out" MEP in abstract BPEL^{light}.

```
<bpel:process name="In-optional-out"
     abstractProcessProfile="http://iaas.uni-stuttgart.de/bpel-light/abstract/mep/2008/"
     targetNamespace="http://iaas.uni-stuttgart.de/mep-in-bpel"
     xmlns:bpel="http://.../wsbpel/2.0/process/abstract"
     xmlns:bl="http://iaas.uni-stuttgart.de/bpel-light"
     xmlns:mep="http://iaas.uni-stuttgart.de/bpel-light/abstract/mep/2008/"
     expressionLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0"
     suppressJoinFailure="yes">
   <bl:partners>
      <bl:partner name="aPartner"/>
   </bl:partners>
   <bl:conversations>
      <bl:conversation name="in-optional-out"/>
   </bl:conversations>
   <bl:expressions>
      <bl:expression expressionLabel="expr" expression="##opaque"/>
   </bl:expressions>
   <bpel:flow>
      <bpel:links>
         <bpel:link name="option1"/>
         <bpel:link name="option2"/>
      </bpel:links>
      <bpel:extensionActivity>
         <bl:interactionActivity name="receiveMessage" conversation="in-optional-out">
            <bpel:sources>
               <bpel:source linkName="option1">
                  <bpel:transitionCondition>$expr</bpel:transitionCondition>
               </bpel:source>
               <bpel:source linkName="option2">
                  <bpel:transitionCondition>fn:not($expr)</bpel:transitionCondition>
               </bpel:source>
            </bpel:sources>
            <bl:output messageLabel="in" variable="##opaque"/>
            <bl:partner name="aPartner"/>
         </bl:interactionActivity>
      </bpel:extensionActivity>
      <bpel:extensionActivity>
         <bl:interactionActivity name="sendResponse" conversation="in-optional-out">
            <bpel:targets>
               <bpel:target linkName="option1"/>
            </bpel:targets>
            <bl:input messageLabel="out" variable="##opaque"/>
            <bl:partner name="aPartner"/>
         </bl:interactionActivity>
      </bpel:extensionActivity>
      <bpel:empty name="doNothing">
         <bpel:targets>
            <bpel:target linkName="option2"/>
         </bpel:targets>
      </bpel:empty>
   </bpel:flow>
</bpel:process>
```

Listing A.3 "Robust-in-only" MEP in abstract BPEL^{light}.

```
<bpel:process name="Robust-in-only"
     abstractProcessProfile="http://iaas.uni-stuttgart.de/bpel-light/abstract/mep/2008/"
     targetNamespace="http://iaas.uni-stuttgart.de/mep-in-bpel"
     xmlns:bpel="http://.../wsbpel/2.0/process/abstract"
     xmlns:bl="http://iaas.uni-stuttgart.de/bpel-light"
     xmlns:mep="http://iaas.uni-stuttgart.de/bpel-light/abstract/mep/2008/"
     expressionLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0"
     suppressJoinFailure="yes">
   <bl:partners>
      <bl:partner name="aPartner"/>
   </bl:partners>
   <bl:conversations>
      <bl:conversation name="robust-in-only"/>
   </bl:conversations>
   <bl:expressions>
      <bl:expression expressionLabel="expr" expression="##opaque"/>
   </bl:expressions>
   <bpel:flow>
      <bpel:links>
         <bpel:link name="option1"/>
         <bpel:link name="option2"/>
      </bpel:links>
      <bpel:extensionActivity>
         <bl:interactionActivity name="receiveMessage" conversation="robust-in-only">
            <bpel:sources>
               <bpel:source linkName="option1">
                  <bpel:transitionCondition>$expr</bpel:transitionCondition>
               </bpel:source>
               <bpel:source linkName="option2">
                  <bpel:transitionCondition>fn:not($expr)</bpel:transitionCondition>
               </bpel:source>
            </bpel:sources>
            <bl:output messageLabel="in" variable="##opaque"/>
            <bl:partner name="aPartner"/>
         </bl:interactionActivity>
      </bpel:extensionActivity>
      <bpel:extensionActivity>
         <bl:interactionActivity name="sendResponse" conversation="robust-in-only">
            <bpel:targets>
               <bpel:target linkName="option1"/>
            </bpel:targets>
            <bl:infault messageLabel="out" faultName="##opaque" variable="##opaque"/>
            <bl:partner name="aPartner"/>
         </bl:interactionActivity>
      </bpel:extensionActivity>
      <bpel:empty name="doNothing">
         <bpel:targets>
            <bpel:target linkName="option2"/>
         </bpel:targets>
      </bpel:empty>
   </bpel:flow>
</bpel:process>
```

Listing A.4 "Out-optional-in" MEP in abstract BPEL^{light}.

```
<bpel:process name="out-optional-in"
     abstractProcessProfile="http://iaas.uni-stuttgart.de/bpel-light/abstract/mep/2008/"
     targetNamespace="http://iaas.uni-stuttgart.de/mep-in-bpel"
     xmlns:bpel="http://.../wsbpel/2.0/process/abstract"
     xmlns:bl="http://iaas.uni-stuttgart.de/bpel-light"
     xmlns:mep="http://iaas.uni-stuttgart.de/bpel-light/abstract/mep/2008/">
   <bl:partners>
      <bl:partner name="aPartner"/>
   </bl:partners>
   <bl:conversations>
      <bl:conversation name="out-optional-in"/>
   </bl:conversations>
   <bpel:sequence>
      <bpel:extensionActivity>
         <bl:interactionActivity name="sendRequest" conversation="out-optional-in">
            <bl:input messageLabel="out" variable="##opaque"/>
            <bl:partner name="aPartner"/>
         </bl:interactionActivity>
      </bpel:extensionActivity>
      <bpel:extensionActivity>
         <bl:pick name="receiveBid">
            <bl:onMessage messageLabel="in" conversation="out-optional-in" partner=
                "aPartner" variable="##opaque">
               <bpel:empty/>
            </bl:onMessage>
            <bpel:onAlarm>
               <mep:timingExpression timingExpressionLabel="timeout" type="##opaque"
                   expression="##opaque" />
               <bpel:empty/>
            </bpel:onAlarm>
         </bl:pick>
      </bpel:extensionActivity>
   </bpel:sequence>
</bpel:process>
```

Listing A.5 "In-xor-out" MEP in abstract BPEL^{light}.

```
<bpel:process name="In-xor-out"
     abstractProcessProfile="http://iaas.uni-stuttgart.de/bpel-light/abstract/mep/2008/"
     targetNamespace="http://iaas.uni-stuttgart.de/mep-in-bpel"
     xmlns:bpel="http://.../wsbpel/2.0/process/abstract"
     xmlns:bl="http://iaas.uni-stuttgart.de/bpel-light"
     xmlns:mep="http://iaas.uni-stuttgart.de/bpel-light/abstract/mep/2008/"
     expressionLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0"
     suppressJoinFailure="yes">
  <bl:partners>
     <bl:partner name="aPartner"/>
  </bl:partners>
  <bl:conversations>
     <bl:conversation name="in-xor-out"/>
  </bl:conversations>
  <bl:expressions>
     <bl:expression expressionLabel="expr" expression="##opaque"/>
  </bl:expressions>
  <bpel:flow>
     <bpel:links>
        <bpel:link name="option1"/>
        <bpel:link name="option2"/>
     </bpel:links>
     <bpel:extensionActivity>
        <bl:interactionActivity name="receiveMessage" conversation="in-xor-out">
           <bpel:sources>
              <bpel:source linkName="option1">
                 <bpel:transitionCondition>$expr</bpel:transitionCondition>
              </bpel:source>
              <bpel:source linkName="option2">
                 <bpel:transitionCondition>fn:not($expr)</bpel:transitionCondition>
              </bpel:source>
           </bpel:sources>
           <bl:output messageLabel="in" variable="##opaque"/>
           <bl:partner name="aPartner"/>
        </bl:interactionActivity>
     </bpel:extensionActivity>
     <bpel:extensionActivity>
        <bl:interactionActivity name="sendResponseOpt1" conversation="in-xor-out">
           <bpel:targets>
              <bpel:target linkName="option1"/>
           </bpel:targets>
           <bl:input messageLabel="out-option1" variable="##opaque"/>
           <bl:partner name="aPartner"/>
        </bl:interactionActivity>
     </bpel:extensionActivity>
     <bpel:extensionActivity>
        <bl:interactionActivity name="sendResponseOpt2" conversation="in-xor-out">
           <bpel:targets>
              <bpel:target linkName="option2"/>
           </bpel:targets>
           <bl:input messageLabel="out-option2" variable="##opaque"/>
           <bl:partner name="aPartner"/>
        </bl:interactionActivity>
     </bpel:extensionActivity>
  </bpel:flow>
</bpel:process>
```

List of Acronyms

API	Application Programming Interface
B2B	Business to Business
BOM	BPEL Object Model
BPEL	Business Process Execution Language
BPM	Business Process Management
DAO	Data Access Object
DB	Database
DD	Deployment Descriptor
DOM	Document Object Model
DU	Deployment Unit
EAI	Enterprise Application Integration
EPR	Endpoint Reference
ESB	Enterprise Service Bus
HTTP	Hypertext Tranfer Protocol
ID	Identifier
IDL	Interface Definition Language
IL	Integration Layer
IRI	Internationalized Resource Identifier
JACOB	JAva Concurrent OBjects
JBI	Java Business Integration
JMS	Java Message Service
JPA	Java Persistence API
MEP	Message Exchange Pattern
NCName	Non-colonized Name
QName	Qualified Name
QoS	Quality of Service
REST	REpresentational State Transfer
SSDL	SOAP Service Description Language
UDDI	Universal Description, Discovery and Integration

URI ..... Uniform Resource Identifier
W3C ..... World Wide Web Consortium
WfMC ..... Workflow Management Coalition
WfMS ..... Workflow Management System
WRM ..... Workflow Reference Model
WS ..... Web Service
WS-A ..... Web Services Addressing
WSDL ..... Web Services Description Language
XML ..... eXtensible Markup Language
XPath ..... XML Path Language
XSD ..... XML Schema Definition

List of Figures

List of Listings

Bibliography

[AAA+07] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, A. Yiu. *Web Services Business Process Execution Language Version 2.0*. 2007.

[ACD+03] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, S. Weerawarana. *Business Process Execution Language for Web Services Version 1.1*. May, 2003.

[BBC+06] S. Bajaj, D. Box, D. Chappell, F. Curbera, G. Daniels, P. Hallam-Baker, M. Hondo, C. Kaler, D. Langworthy, A. Nadalin, N. Nagaratnam, H. Prafullchandra, C. von Riegen, D. Roth, J. Schlimmer, C. Sharp, J. Shewchuk, A. Vedamuthu, Ümit Yalçinalp, D. Orchard. Web Services Policy 1.2 - Framework (WS-Policy). April, 2006. URL `http://www.w3.org/Submission/2006/SUBM-WS-Policy-20060425/`. W3C Member Submission.

[BBC+07] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, J. Siméon. XML Path Language (XPath) 2.0. Jan. 2007. URL `http://www.w3.org/TR/xpath20`.

[BCC+04] D. Box, E. Christensen, F. Curbera, D. Ferguson, J. Frey, M. Hadley, C. Kaler, D. Langworthy, F. Leymann, B. Lovering, S. Lucco, S. Millet, N. Mukhi, M. Nottingham, D. Orchard, J. Shewchuk, E. Sindambiwe, T. Storey, S. Weerawarana, S. Winkler. Web Services Addressing (WS-Addressing). August, 2004. URL `http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/`. W3C Member Submission.

[BDH05] A. Barros, M. Dumas, A. H. ter Hofstede. *Service Interaction Patterns: Towards a Reference Framework for Service-Based Business Process Interconnection*. April, 2005.

[BEK+00] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, D. Winer. Simple Object Access Protocol (SOAP) 1.1. May, 2000. URL `http://www.w3.org/TR/2000/NOTE-SOAP-20000508`. W3C Note.

[BHM+04] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, D. Orchard. Web Services Architecture. February, 2004. URL http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/. W3C Working Group Note.

[BKM07] P. Bianco, R. Kotermanski, P. Merson. Evaluating a Service-Oriented Architecture. September, 2007.

[BL07] D. Booth, C. K. Liu. Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. June, 2007. URL http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626.

[Bro07] P. Brown. An Introduction to Apache ODE. 2007. URL http://www.infoq.com/articles/paul-brown-ode.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. Web Services Description Language (WSDL) 1.1. March, 2001. URL http://www.w3.org/TR/2001/NOTE-wsdl-20010315. W3C Note.

[CHL+07] R. Chinnici, H. Haas, A. A. Lewis, J.-J. Moreau, D. Orchard, S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts. June, 2007. URL http://www.w3.org/TR/2007/REC-wsdl20-adjuncts-20070626/. W3C Recommendation.

[DPW06] G. Decker, F. Puhlmann, M. Weske. Formalizing Service Interactions. In *4th International Conference on Business Process Management (BPM)*. September, 2006.

[GHM+07] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielson, A. Karmarkar, Y. Lafon. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). April, 2007. URL http://www.w3.org/TR/2007/REC-soap12-part1-20070427/. W3C Recommendation.

[GHR06] M. Gudgin, M. Hadley, T. Rogers. Web Services Addressing 1.0 - Core. May, 2006. URL http://www.w3.org/TR/2006/REC-ws-addr-core-20060509. W3C Recommendation.

[Höh08] B. Höhensteiger. Conceptual Design and Implementation of a BPEL^{light} Editor with Support for Message Exchange Patterns. October, 2008.

[Hib] Hibernate - *An Object-relational Persistence And Query Service.* URL http://www.hibernate.org/.

[Hol95] D. Hollingsworth. Workflow Management Coalition - The Workflow Reference Model. January, 1995. URL http://www.wfmc.org/standards/docs/tc003v11.pdf.

[Jin] Jini. URL http://www.jini.org.

[KKL+05] M. Kloppmann, D. König, F. Leymann, G. Pfau, A. Rickayzen, C. von Riegen, P. Schmidt, I. Trickovic. *WS-BPEL Extension for Sub-processes - BPEL-SPE*. September, 2005.

[Lew07] A. A. Lewis. Web Services Description Language (WSDL) Version 2.0: Additional MEPs. June, 2007. URL `http://www.w3.org/TR/2007/NOTE-wsdl20-additional-meps-20070626/`. W3C Working Group Note.

[Ley07] F. Leymann. The Business Value of WS-BPEL for Business Analysts and Managers - Part 1. *OASIS BPEL Webinar*, March 12th, 2007.

[LLN08] T. von Lessen, F. Leymann, J. Nitzsche. Extending BPEL^{light} for Expressing Multi-Partner Message Exchange Patterns. In *12th IEEE International EDOC Conference (EDOC 2008)*. September, 2008. To appear.

[LNL08] T. van Lessen, J. Nitzsche, F. Leymann. Formalising Message Exchange Patterns using BPEL^{light}. In *5th International Conference on Services Computing (SCC), To appear*. July 2008. Honululu, Hawaii, USA.

[LR00] F. Leymann, D. Roller. *Production Workflow*. Prentice Hall, 2000.

[Mel07] I. Melzer. *Service-orientierte Architekturen mit Web Services. Konzepte - Standards - Praxis*. 2007. 2nd Edition.

[NHST08] J. Nitzsche, B. Höhensteiger, M. Sonntag, M. Tost. Defining the Behaviour of BPEL^{light} Interaction Activities Using Message Exchange Patterns. August, 2008.

[NLKL07] J. Nitzsche, T. van Lessen, D. Karastoyanova, F. Leymann. BPEL^{light}. In *5th International Conference on Business Process Management (BPM)*. Sept. 2007. Brisbane, Australia.

[NLL08] J. Nitzsche, T. van Lessen, F. Leymann. WSDL 2.0 Message Exchange Patterns: Limitations and Opportunities. In *3rd International Conference on Internet and Web Applications and Services (ICIW)*. June 2008. Athens, Greece.

[ODE] Apache Orchestration Director Engine (ODE) - *A Workflow Engine Implementing BPEL*. URL `http://ode.apache.org`.

[Ope] Apache OpenJPA - *Implementing The Java Persistence API*. URL `http://openjpa.apache.org/`.

[PW05] S. Parastatidis, J. Webber. The SOAP Service Description Language. April, 2005.

[THW05] R. Ten-Hove, P. Walker. Java Business Integration (JBI) 1.0. August, 2005.

[Tos08] M. Tost. Conceptual Design and Implementation of an Enterprise Service Bus with Native Support for Complex Message Exchange Patterns. October, 2008.

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005.

All links were last followed on October 28th, 2008.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author.

(Mirko Sonntag)