

**Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering**

Model-Driven Development of Service Compositions

Masterarbeit

im Studiengang Informatik

von

Leif Singer

**Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Prof. Dr.-Ing. Christian Grimm
Betreuer: Dipl.-Wirt.-Inform. Daniel Lübke**

Hannover, 21. Dezember 2007

Zusammenfassung

Die Business Process Execution Language (BPEL) zielt darauf ab, bei der Orchestrierung von Web Services zu ausführbaren Geschäftsprozessen einen hohen Abstraktionsgrad zu bieten. Obwohl dies teilweise erreicht wurde, sind für die Erstellung von BPEL Prozessen noch einige Aufgaben notwendig, die die mühsame Implementierung von technischen Details erfordern, ohne dabei größeren Nutzen für den eigentlichen Geschäftsprozess zu erschliessen. Auch die Unterstützung durch Werkzeuge kann Benutzer bisher noch nicht befriedigend von diesen Problemen abschirmen. Anstatt lediglich den beabsichtigten Prozess zu modellieren, müssen Entwickler regelmäßig Aufgaben auf niedrigem Abstraktionsniveau bearbeiten, wie bspw. umständliches Kopieren zwischen Variablen oder die manuelle Übersetzung zwischen verschiedenen Datenmodellen. Abgesehen von diesen BPEL-spezifischen Problemen werden Prozessmodellierer durch die sich wiederholende Entwicklung von Standard-Web Services behindert, die heutzutage eigentlich eine Selbstverständlichkeit sein sollten — bspw. die Speicherung und Abfrage von Geschäftsobjekten.

Um diese Probleme zu adressieren, schlägt diese Arbeit eine Notation zur Modellierung von Prozessen, ein Modellierungswerkzeug, sowie einen Generator vor, der schlussendlich eine BPEL-Orchestrierung und Standard-Web Services aus einem im Modellierungswerkzeug erstellten Prozessmodell generiert. Es ist das Ziel, die Orchestrierung von Prozessen stärker auf die Verwendung von domänen-nahen Abstraktionen auszurichten.

Abstract

The Business Process Execution Language (BPEL) aims at enabling the usage of high abstraction levels when orchestrating Web Services to represent business processes. While this has partly been achieved, several tasks required for the creation of a BPEL process demand cumbersome implementation of technical details, adding little value for the actual business process. Real-world tool-support still fails to shield users from these deficiencies. Instead of just modeling the intended process, developers regularly need to perform low-level tasks such as creating and copying variables or translating between different data models. Apart from these BPEL-specific burdens, process modelers get distracted by the repetitive creation of standard Web Services that should be a commodity today — e.g. the persistence and the retrieval of business objects.

To address these problems, this thesis proposes a modeling notation, an accompanying modeling tool and a generator, ultimately producing a BPEL orchestration and Web Services generated from a process model that was created using the modeling tool. It tries to move the act of orchestration more in line with the goal of using high-level abstractions in proximity to the respective domain.

List of Acronyms

| | |
|--------------|--|
| BPEL | Business Process Execution Language |
| BPMN | Business Process Modeling Notation |
| CORBA | Common Object Request Broker Architecture |
| DCOM | Distributed Component Object Model |
| GUI | Graphical User Interface |
| HTTP | Hypertext Transport Protocol |
| IEEE | The Institute of Electrical and Electronics Engineers |
| MDA | Model-Driven Architecture, trademark of the OMG |
| MDSD | Model-Driven Software Development |
| MOF | Meta Object Facility |
| OASIS | Organization for the Advancement of Structured Information Standards |
| OCL | Object Constraint Language |
| OMG | Object Management Group |
| QVT | Query / View / Transformation |
| RMI | Remote Method Invocation (Java API) |
| SOA | Service-oriented Architecture |
| SOAP | Formerly: Simple Object Access Protocol; since 2003: SOAP |
| UML | Unified Modeling Language |
| W3C | World Wide Web Consortium |
| WSDL | Web Services Description Language |
| WSFL | Web Services Flow Language |
| XML | Extensible Markup Language |
| XSD | XML Schema Definition |

Contents

| | |
|--|-----------|
| 1. Introduction | 1 |
| 1.1. Motivation | 1 |
| 1.2. Problem Description | 2 |
| 1.3. Structure of this Thesis | 3 |
| 2. Basic Concepts | 4 |
| 2.1. Service-oriented Architecture | 4 |
| 2.2. Web Services | 5 |
| 2.2.1. Web Services Description Language | 5 |
| 2.2.2. XML Schema | 9 |
| 2.3. Service Compositions | 11 |
| 2.3.1. Business Process Execution Language | 11 |
| 2.4. Model-Driven Development | 16 |
| 2.4.1. Terminology | 16 |
| 2.4.2. UML Profiles | 19 |
| 2.4.3. Consequences | 20 |
| 3. Approach | 22 |
| 3.1. Intended Workflow | 22 |
| 3.2. Scenario | 25 |
| 3.2.1. Process Modeling | 26 |
| 3.2.2. Process Configuration | 27 |
| 3.2.3. Generation and Deployment | 29 |
| 3.3. Graphical User Interface | 29 |
| 3.4. Generator Requirements | 32 |
| 3.5. Commodity Services | 33 |
| 3.6. Summary | 33 |
| 4. The Composr UML Profile | 35 |
| 4.1. Metamodel | 35 |
| 4.1.1. Foundation | 35 |
| 4.1.2. Basic Activities | 38 |
| 4.1.3. Services | 39 |
| 4.1.4. Structured Activities | 40 |
| 4.1.5. Data | 41 |
| 4.2. Notation | 42 |
| 4.2.1. Initial Draft | 42 |
| 4.2.2. Survey | 45 |
| 4.2.3. Results of the Survey | 46 |

Contents

| | |
|---|------------|
| 4.2.4. Final Version | 46 |
| 5. Development | 50 |
| 5.1. Development Process | 50 |
| 5.2. Generator Design | 51 |
| 5.2.1. Technology | 51 |
| 5.2.2. Artifact Dependencies | 53 |
| 5.2.3. Integration with the GUI | 55 |
| 5.2.4. Architecture of the Generator | 56 |
| 5.3. Generator Workflow and Artifacts | 57 |
| 5.3.1. XML Schema | 57 |
| 5.3.2. Web Services Description Language | 59 |
| 5.3.3. Commodity Services | 61 |
| 5.3.4. Business Process Execution Language | 63 |
| 5.4. Graphical User Interface | 64 |
| 5.4.1. Sketches | 65 |
| 5.4.2. Implementation | 67 |
| 6. A real-world Example | 68 |
| 6.1. The Thesis Process | 68 |
| 6.1.1. Student Implementation | 69 |
| 6.1.2. Composr Model | 70 |
| 6.2. Metrics | 72 |
| 6.3. Comparison | 73 |
| 7. Related Work | 77 |
| 7.1. Model-Driven Development of executable Processes | 77 |
| 7.2. Available Products | 79 |
| 8. Conclusions and Outlook | 82 |
| 8.1. Critical Appraisal | 82 |
| 8.2. Outlook | 84 |
| 8.2.1. Graphical Editor and Generator | 84 |
| 8.2.2. Commodity Services | 85 |
| 8.3. Conclusions | 86 |
| Bibliography | 88 |
| A. Survey | 91 |
| B. Final Notation | 96 |
| C. Generated Artifacts | 97 |
| D. Compact Disc | 102 |

1. Introduction

1.1. Motivation

Service-oriented Architecture (SOA) is an architectural style for supporting business processes in computer systems. Instead of using a single, monolithic system or application, business processes are being modelled in an executable language. The resulting executable business processes then use services to carry out the actual tasks. Services are self-contained units of functionality with a defined, platform-neutral interface. The concept of services is not tied to any communications channel, i.e., services can e.g. be provided over the network using various protocols or run directly on the machine executing the process.

Building upon these prerequisites, SOA permits the development of very flexible systems, as service implementations and locations can be modified without affecting the business process. Also, the functionality exposed through services is available to existing and new business processes alike, even allowing for the implementation of cross-organizational processes.

In more concrete terms, the business processes are often being modelled using the Business Process Execution Language (BPEL), while services are predominantly implemented as Web Services. Both approaches are based on standards and have thus been embraced by the industry.

Despite the popularity of BPEL for modelling executable business processes, it has often been criticized for certain short-comings.

Being a *mélange* of two prior languages — WSFL and XLANG, respectively —, it supports different styles of workflow modelling. BPEL inherits its graph-oriented constructs from WSFL, while the block-oriented control-structures stem from XLANG. Even though some may see this combination as a flexibility gain, the consensus is that it adds unnecessary complexity.

BPEL, which is a rather verbose XML language requiring many small steps, is considered cumbersome to write by hand. Thus, many tools exist to support the graphical modelling of BPEL processes. But most of these merely support the graphical arrangement of the exact same constructs found in the plain source, which does not make the modelling process significantly easier. As an accurate knowledge of the available BPEL constructs is therefore still required for both reading and creating such models. Most tools are not adequate for use by novice users or non-technical domain experts, rendering the tools useless for tasks like requirements elicitation.

On the service side, similar improvements in comfort seem desirable. There are sev-

eral standard tasks that appear in almost every SOA project and are basically the same every time – persistence being one example. Nevertheless, developers are forced to manually implement these services again and again, resulting in an avoidable productivity loss.

While tool-support should be able to shield users from many of these problems, appropriate solutions have yet to surface — modeling a BPEL process still requires the modeler to constantly switch between high- and low-level tasks.

1.2. Problem Description

This thesis proposes a model-driven approach to improve the situation in SOA development, addressing the aforementioned problems. The pursued approach will provide a metamodel capable of capturing all aspects of a business process relevant to the generation of executable business processes from model instances. Furthermore, a graphical modelling tool will be provided, allowing the creation of such models. Ultimately, a generator will be developed, taking the created model instance as its input and delivering the artifacts of an executable business process as its output. The following paragraphs add some detail to these requirements.

As a formal basis, a UML profile is developed that will provide a metamodel for the modelling of business processes. The metamodel will define appropriate properties for all elements so a model instance will provide sufficient information for the generation of BPEL processes and standard services from a given model. To permit the development of a graphical modeling tool with this theoretical foundation, a notation for the elements defined in the metamodel must be evaluated and defined.

A graphical modelling tool will be developed, enabling its user to create instances of the aforementioned metamodel. To reduce the complexity found in current tools, the tool will only support the most common tasks directly. To enable the translation of the modeled process into an executable process, the tool will facilitate the generation of all artifacts required to successfully execute the modelled process in a BPEL engine. The user will have access to all generated source code, allowing for subsequent editing in more powerful tools.

To be appropriate for usage in requirements elicitation, the tool requires an unobtrusive user interface that does not hinder the process of business process modelling. Also, the notation used should be intuitively comprehensible by both modellers as well as non-technical domain experts. On the other hand, the tool should be suitable for technical users, wishing to create actual executable business processes, perhaps building upon a model created earlier during the elicitation phase.

The modeling tool will use a generator to create the aforementioned artifacts.

This includes deployment-ready packages for both the BPEL process as well as all generated standard services. Also, the source code for all generated artifacts will be available, supported by custom build scripts. This will allow for further development of

the generated process, should the tool developed in this thesis not suffice at a later stage of process development.

To narrow the scope of this thesis, the generated artifacts may be oriented at specific implementations of a BPEL engine and Web Service engines. Nevertheless, the generator's design should allow for the modular replacement of generation strategies or parts thereof, thus enabling the support of different BPEL engines, different Web Service engines, and even service orchestration languages other than BPEL.

1.3. Structure of this Thesis

The motivation for this thesis and the central problems have now been outlined. Chapter 2 details several basic concepts required to understand the problem domain: Web Services and related technologies, service compositions, and model-driven development.

Chapter 3 gives a high-level view of the approach followed to solve the aforementioned problems and difficulties. In a scenario, the envisioned workflow is outlined.

To formally model business processes, a metamodel for all model instances is required. This is described in chapter 4. Also, a graphical notation for the metamodel is developed and verified in a survey among students.

Chapter 5 explains the details of the development of both the editor and the generator. This includes the requirements for both components as well as their designs. Some user interface concepts are presented and, for the generator, some of the used algorithms are outlined.

To provide a means for comparison of different approaches to process modeling, chapter 6 examines an example process taken from a student project. This process is modelled using the developed tool. The section compares both process models and the tools used to create them. For this, some metrics are defined and their values are discussed.

Related work found relevant during research for this thesis is being presented and put into context in chapter 7. Also, an overview of existing commercial products related to the developed tool is given.

After taking a critical look at the contributions of this thesis, chapter 8 outlines possible future work that could be based on the thesis. The thesis closes with conclusion.

2. Basic Concepts

This chapter introduces several concepts integral to understanding the problem domain of this thesis. The first section is about service-oriented architecture and gives an overview of the approach, while the following section presents Web Services, the most commonly used service implementation in SOAs, presenting core concepts from the Web Services Description Language and XML Schema. Section 2.3 explains the concept of service compositions and introduces BPEL, the Business Process Execution Language. The last section closes this chapter with an overview of model-driven development.

2.1. Service-oriented Architecture

Service-oriented architecture (SOA) is an architectural style for the design of business applications, focussing on the business processes present in an organization. As these processes may cross organizational boundaries, one of the goals of SOA is the use of platform-neutral, self-contained services. Also, SOA takes into consideration that today, much business value is created from business models instead of mere products. Therefore, SOA aims at providing an IT infrastructure that can quickly adapt to changing markets — by recreating the actual business processes in the IT in the form of compositions of services.

At the core of SOA lies the concept of a service. Building on the definition by Lübke [1], this thesis defines a service as follows.

Definition 2.1.1

A service is a loosely-coupled software component that is accessible over a network and provides one or more capabilities to its consumers. It implements a well-defined interface and can be called using standardized protocols without knowledge of the service's actual implementation.

An SOA uses services as the building blocks of executable processes, which are in turn modeled according to existing business processes.

Definition 2.1.2

A business process is a procedure in an organization involving multiple activities, carried out by possibly multiple roles, where each role may be occupied by one or more actors and an actor may occupy one or more roles. An actor may be a person or a computer system. The order of the execution of the activities can be described using workflow patterns as in [2].

2. Basic Concepts

The goal of SOA is to better align business and IT, and to do so in a very flexible manner. On the technical side, this is achieved by implementing the actual business processes as service compositions and implementing the activities as services. Service implementations are easily exchangeable, since their are self-contained and loosely coupled. Compositions of services can be centrally adapted to new business requirements without the need for an organization-wide rollout of new software. Therefore, the implemented business processes can very quickly respond to change in business models.

On the business side, this forces organizations to clearly define their business processes — possibly for the first time. Also, there is less need to worry about whether the IT infrastructure can respond to change as easily as desired, giving business planners more freedom in exploring possible strategies to create business value. The intended effect is an optimization of business processes in terms of time and resources.

2.2. Web Services

Although services can be realized in several ways — RMI, CORBA and DCOM, amongst others —, the currently most promising standard in use is Web Services. HTTP and SOAP, which are the most commonly used transport and communications protocols for Web Services at this time, will be assumed to be known by the reader.

This section will examine the core technologies behind Web Services, namely the Web Services Description Language (WSDL) and XML Schema. The former is used as a platform-neutral way to define the interfaces of Web Services, while the latter is the predominant language used to describe the data types required for communication with a Web Service.

2.2.1. Web Services Description Language

WSDL, the Web Services Description Language, is an XML language used for the definition of the interface a Web Service exposes — i.e., its operations, their arguments and return values, as well as the protocols and endpoints required to access the service. As of this writing, the current version of WSDL is 2.0, superseding WSDL 1.1.

The approach presented in this thesis includes generating both WSDL and BPEL artifacts. As the current version of the Business Process Execution Language, WS-BPEL 2.0, does not support WSDL 2.0, this thesis will focus on WSDL 1.1 entirely.

Inside its `<definitions>` root element, a WSDL description contains the following subelements.

- One `<types>` element, defining the data types used in the service interface. Most of the time, this is done using XML Schema, although other languages are possible.

2. Basic Concepts

- Zero or more `<message>` elements, defining the input and output signatures of the exposed interface.
- Zero or more `<portType>` elements, defining the actual operations using the defined messages and bundling them into so-called ports.
- Zero or more `<binding>` elements, associating concrete protocols with the port types. Common protocols include SOAP for message encoding and HTTP for transport.
- Zero or more `<service>` elements, publishing the bindings at concrete endpoint URLs.

Figure 2.1 illustrates this structure by dividing it into an abstract section and a concrete section. The abstract service interface consists of the operations, which use messages for their input and output. Each message references one or more types defined in the types element. The concrete service interface is a collection of endpoints, each referencing a binding that specifies a concrete protocol. The bindings employ the abstract service interface by referencing the port types that consist of the operations.

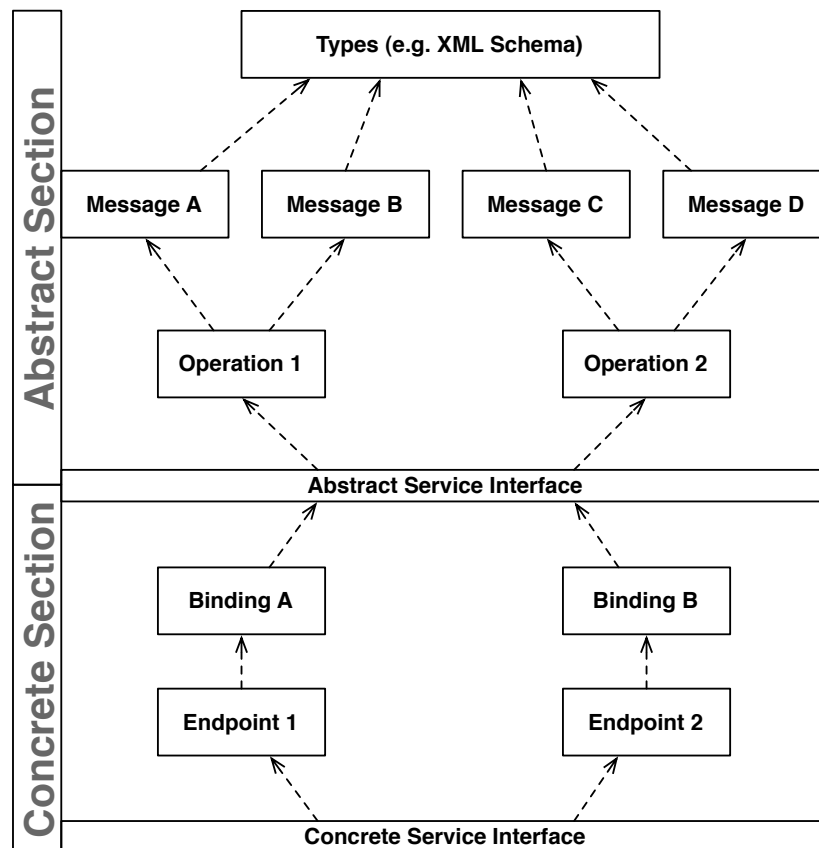


Figure 2.1.: The structure of a WSDL description.

Since WSDL is an extensible language, elements from other namespaces can also be

2. Basic Concepts

added. E.g., BPEL adds the `<partnerLinkType>` element, amongst others. This will be discussed further in the section about the Business Process Execution Language (2.3.1).

Listing 2.1 shows a simplified example for a WSDL description.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <definitions>
3   <types>
4     <xs:schema>
5       <xs:import schemaLocation="Person.xsd"/>
6       <xs:import schemaLocation="Messages.xsd"/>
7     </xs:schema>
8   </types>
9   <message name="saveResponse">
10     <part name="parameters" element="tns:saveResponse"/>
11   </message>
12   <message name="saveRequest">
13     <part name="parameters" element="save"/>
14   </message>
15   <portType name="PPSPortType">
16     <operation name="save">
17       <input message="saveRequest"/>
18       <output message="saveResponse"/>
19     </operation>
20   </portType>
21   <binding name="PPSBinding" type="PPSPortType">
22     <soap:binding style="document"/>
23     <operation name="save">
24       <soap:operation soapAction="urn:save" style="document"/>
25       <input>
26         <soap:body use="literal"/>
27       </input>
28       <output>
29         <soap:body use="literal"/>
30       </output>
31     </operation>
32   </binding>
33   <service name="PersonPersistenceService">
34     <port name="PPSPort" binding="PPSBinding">
35       <soap:address location="http://example.org/PersonPersistenceService"/>
36     </port>
37   </service>
38 </definitions>
```

Listing 2.1: A simplified WSDL description of the interface of a Web Service offering to persist person objects

The example in listing 2.1 defines a service called *PersonPersistenceService* at the endpoint URL `http://example.org/PersonPersistenceService` (lines 33-37). For this, it uses a SOAP binding (lines 21-32) employing HTTP as transport protocol and using the document/literal style. Its operations are defined in the referenced port type (lines 15-20) which specifies a single operation called *save*. The *save* operation takes a *saveRequest* message (lines 12-14) as its argument and returns a *saveResponse* message (lines 9-11). The definitions of the elements used by the messages

2. Basic Concepts

are being imported in the *types* element. The referenced XML Schema definitions will be explained in the next section.

The WSDL supports different styles for defining the structure of the service interface. An outdated one is the RPC/encoded-style, which could not be validated against a schema. By now, the most frequently used style is the document/literal-style, in which the types of the messages have been defined in a schema and can thus be validated. Listing 2.2 shows an example for a SOAP message sent using this style. It invokes an operation called `anOperation` with two integer parameters, called `aParameter` and `anotherParameter`, respectively.

```
1 <soap:envelope>
2   <soap:body>
3     <aParameter>12</aParameter>
4     <anotherParameter>144</anotherParameter>
5   </soap:body>
6 </soap:envelope>
```

Listing 2.2: A SOAP message with the document/literal-style defined in the WSDL.

Unfortunately, this would violate a recommendation from the WS-I Basic Profile [3], as the `body` element of a SOAP message might contain multiple elements when used in this form. Therefore, it is common practice to wrap a message's parts in a wrapper element, named after the operation the message belongs to. For responses, "Response" is appended to the name of the wrapper element. Listing 2.3 shows an example for a SOAP message sent using this style variation. It invokes the same operation as in the previous example.

```
1 <soap:envelope>
2   <soap:body>
3     <AnOperation>
4       <aParameter>12</aParameter>
5       <anotherParameter>144</anotherParameter>
6     </AnOperation>
7   </soap:body>
8 </soap:envelope>
```

Listing 2.3: A SOAP message with the document/literal-style defined in the WSDL using a wrapper element.

Because of the absence of such a wrapper element, the first variation shown is referred to as "nonwrapped" document/literal, whereas the second variation is called "wrapped" document/literal [4].

Several Web Services engines support the automatic "unwrapping" of these wrapped messages, resulting in source code stubs that omit the wrapper element. E.g. in Java, the signature of the above operation would be *public void anOperation(AnOperation parameters)* when not using the unwrapping option. The actual parameters would be inside the parameters object. The unwrapped signature would be *public void anOperation(int aParameter, int anotherParameter)*, which is easier to read and use.

The "unwrapped" term must be clearly distinguished from the "nonwrapped" term. It is

impossible to unwrap a nonwrapped SOAP message, consequently unwrapping can only be done when using the “wrapped” variation. It will be used throughout this thesis.

2.2.2. XML Schema

As has been mentioned before, WSDL commonly uses XML Schema to define the data types used in the provided service interface. XML Schema is a W3C recommendation [5] used to define the types, elements and attributes permissible in an XML document, thus formally defining an XML language.

The root of an XML Schema definition is a `<schema>` element. Its most important sub-elements are

- the `<import>` element, which can be used to import existing type definitions;
- the `<complexType>` element for the definition of new types;
- the `<element>` element, which defines new elements and has a type attribute referencing a valid XML Schema type.

Inside the type and element definitions, additionally to all imported types and elements, many types built into XML Schema ¹ can be used. Employing a few more constructs from the XML Schema recommendation, users of XML Schema can model the business objects required in their Web Services.

To resume the WSDL example from listing 2.1, the two referenced XML Schema files will now be presented and explained. Listing 2.4 defines the *save* and *saveResponse* elements.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <schema xmlns="http://www.w3.org/2001/XMLSchema">
3   <xs:element name="save">
4     <xs:complexType>
5       <xs:sequence>
6         <xs:element name="person" type="Person"/>
7       </xs:sequence>
8     </xs:complexType>
9   </xs:element>
10  <xs:element name="saveResponse">
11    <xs:complexType>
12      <xs:sequence>
13        <xs:element name="person" type="Person"/>
14      </xs:sequence>
15    </xs:complexType>
16  </xs:element>
17 </schema>
```

Listing 2.4: The definition of the *save* and *saveResponse* elements in XML Schema

¹<http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#built-in-datatypes>

2. Basic Concepts

Each element is set to contain a sequence of just one element called *person* which is of the type *Person*.

Listing 2.5 shows the simple definition of a *Person* data type. As should be rather obvious, it defines a *Person* to have the sub-elements *id*, *adult*, *age*, *birthday*, and *name*. For the elements' types, it uses some of the simple data types already built into XML Schema.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <schema xmlns="http://www.w3.org/2001/XMLSchema">
3   <complexType name="Person">
4     <sequence>
5       <element name="id" type="long"/>
6       <element name="adult" type="boolean"/>
7       <element name="age" type="int"/>
8       <element name="birthday" type="dateTime"/>
9       <element name="name" type="string"/>
10    </sequence>
11  </complexType>
12 </schema>
```

Listing 2.5: The definition of a *Person* data type in XML Schema

XML Schema provides many more possibilities for more exact definitions of data types — value range restrictions, substitution groups and type derivation, amongst others. Nevertheless, the language constructs presented in this section already represent the basis of what is needed for the definition of types used in Web Services.

To achieve compatibility with different Web Service engines and the languages Web Services are written in, developers must often restrict themselves to only using a subset of what is expressible with XML Schema. Practical interoperability between Web Services would not be achievable otherwise.

Although maximizing interoperability and platform neutrality was the reasoning behind choosing XML Schema as the default type system for WSDL [6] (*Section 2.2: Types*), development reality has relativized the extent to which this can be achieved using XML Schema. Following are some examples for what is possible with XML Schema, but not so in the Java programming language:

- restricting the `int` type to values between two integer numbers;
- default values for primitive types;
- optional attributes;
- unsigned numbers.

Other restrictions may apply to other languages — Java is used as a mere example.

On the other hand, the `char` primitive type found in Java cannot be mapped to XML Schema. Disparities like these must be taken into account when creating XML Schema definitions to be used with Web Services.

2.3. Service Compositions

Service composition is an important idea in service-oriented architecture. It combines invocations of existing services into a new, usually more abstract service. To regulate the transition between the invocations, some kind of flow control is required. Depending on the language used to assemble a composition, different elements are available — an extensive description of possible workflow patterns and their support in several languages can be found in [2].

There are two distinct possible topologies when creating a service composition — orchestrations / choreographies and point-to-point compositions. Orchestrations and choreographies realize a *hub and spoke* or *star* topology on the service layer, providing central flow control. They must be distinguished from point-to-point topologies, which linearly connect services that exert local flow control. The latter also creates a service composition, but is not relevant to the subject of this thesis and will thus not be discussed further. The difference between orchestrations and choreographies will be explained in section 2.3.1.

Since a service composition can itself provide yet another service interface and thus look like a regular service to any service consumer, cascades of compositions become possible. These allow for the creation of sophisticated abstraction hierarchies, thereby fostering the services' decoupling and domain-orientation and thus critically supporting central aims of service-oriented architecture.

2.3.1. Business Process Execution Language

BPEL, the Business Process Execution Language, is an XML language defined for the creation of service orchestrations. A process defined in BPEL can be executed in a service engine. The core capabilities of a BPEL process include receiving messages from clients, invoking Web Services, sending reply messages back to clients, and structuring the flow of all these activities. A BPEL process is a Web Service itself and can, in turn, be invoked by other BPEL processes.

The first BPEL specification, called BPEL4WS [7], was based on the workflow languages WSFL and XLANG. As each of these supported a different approach to the structuring of workflows, BPEL now does so, as well, combining both approaches into one language. The current version of BPEL, WS-BPEL 2.0, was specified by OASIS [8]. The name change was given to reflect the considerable amount of changes and to align it better with existing Web Services standards. This thesis will be restricted to WS-BPEL 2.0.

There are two concepts of composition support in BPEL: orchestration and choreography. Orchestration has been explained in the preceding section. Choreographies are abstract process contracts that cannot be executed, but can be used as an inter-organizational process interface, defining the interactions between each organization's processes. The following analogy makes the difference very accessible: in orchestrations, there is a central conductor overseeing the execution, as in an orchestra — or in

2. Basic Concepts

the case of BPEL, a central BPEL engine executing the process. In a choreography, every participant knows the contract by which to behave — as in choreographies of dancers or, in the case of BPEL, the contract defined by the abstract process. Choreographies are out of the scope of this thesis and will not be pursued further.

BPEL processes use invocations of Web Services and several flow control constructs to model business processes in an executable manner. The following paragraphs will take a look at the most important elements present in BPEL, omitting those not necessary to understand the basic concepts of BPEL.

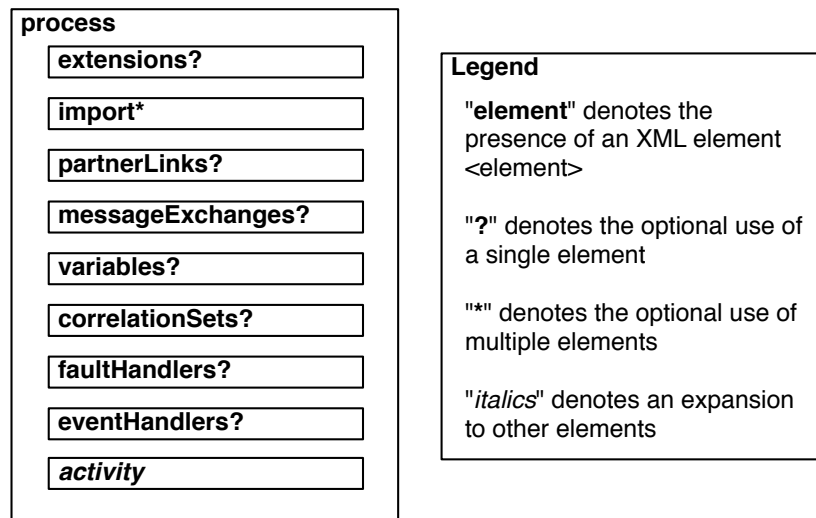


Figure 2.2.: The structure of a BPEL process.

To give a first overview, figure 2.2 presents the structure of a BPEL process. The following list gives a short explanation for each of the elements shown in the figure.

- The **extensions** element serves as a place to declare possible extensions to WS-BPEL 2.0, e.g. vendor-specific features not found in the standard, but desired by the process and supported by the concrete process engine.
- Using **import** elements, the process may reference existing WSDL and XML Schema definitions, e.g. for the declaration of Web Services invoked in the process or messages accepted by the process.
- Using **partnerLinks**, a process defines the roles of itself and other Web Services when communicating with each other. In simple terms, this answers the questions about who calls someone and who gets called by someone. BPEL adds the <partnerLinkType> extension to WSDL to support these definitions.
- If the process definition contains a situation in which there are two possible reply paths to a single request from a client, the **messageExchanges** element must be used to disambiguate communication.

2. Basic Concepts

- In the **variables** element, the process may declare multiple variables for the storage of received messages, the results of service invocations and intermediary results.
- Since of a given process definitions multiple instances are likely to run in a process engine, **correlationSets** must be used to create a mapping of client requests to running process instances. In its most simple form, a correlation set may consist of a single integer used as a primary key to clearly identify process instances.
- **faultHandlers** are a means of catching faults occurred during the execution of a process, e.g. reporting the fault to the process administrator or terminating the process gracefully. They specify a fault they want to catch and an activity to execute when that fault is being thrown. Optionally, there can also be a general fault handler catching all faults.
- To receive messages and react to them parallel to the execution of the process, **eventHandlers** may be used. Upon receipt of a specified message, they execute the activity contained within them, without being subject to the regular control flow.

The elements explained so far have not yet touched the main control flow of the process — defining this, the place of the *activity* must be taken by exactly one activity as defined in the specification for WS-BPEL 2.0. Table 2.1 presents the most important basic activities available in BPEL and provides a description for each.

Table 2.2 shows the most important structured activities found in BPEL. Structured activities are containers for other activities and required to model the control flow of a BPEL process.

Listing 2.6 shows the XML for a simplified BPEL process. This process cannot be executed by a process engine, because much detail has been omitted and some of the expressions have been replaced with pseudo-code for improved readability. The example is shown here to further illustrate the structure and the semantics of a BPEL process and its elements.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <process name="OrderProcess">
3   <import location="OrderProcess.wsdl"/>
4   <import location="WarehouseService.wsdl"/>
5   <import location="InvoiceService.wsdl"/>
6   <import location="ShippingService.wsdl"/>
7   <partnerLinks>...</partnerLinks>
```

2. Basic Concepts

| Basic Activities in WS-BPEL 2.0 | |
|---------------------------------|---|
| BPEL elements | Description |
| <receive> | Receive a SOAP message from a remote client as defined in the process's WSDL and may instantiate the process — consequently, the first activity in a BPEL process must be either a <receive> or a <pick> (see table 2.2) activity. Both activities block any subsequent activities, they complete as soon as the message has been received. |
| <reply> | Sends a SOAP message back to a remote client as defined in the process's WSDL. |
| <invoke> | Invokes a Web Service whose WSDL must have been imported into the process. It uses previously defined variables for input and output data. |
| <assign> | Copies a source value to a target variable. The source value may be extracted from an existing variable using a specified query language or may be a literal value. The exact place where to put the value in the target variable can also be specified using the query language. The most commonly used query language is XPath, but a different query language is also allowed. To execute the process in a process engine, the engine must of course support the query language. |

Table 2.1.: Basic Activities in WS-BPEL 2.0

| Structured Activities in WS-BPEL 2.0 | |
|---|--|
| BPEL elements | Description |
| <pick> | The <pick> activity allows the reception of multiple different message types, whereas the <receive> (see 2.1) activity allows for only one message type to be received. |
| <sequence> | A container for other activities which will be executed in sequential order. |
| <flow> | A container for other activities which will be executed in the order given by the graph implied by <link> elements attached to the contained activities, marking them as sources or targets of a link. This activity enables parallel execution of activities. |
| <if> | Provides for conditional branching between alternative activities. |
| <while>, <repeatUntil>, <forEach> | Used for repeating the contained activities, based on a termination condition. <forEach> permits the parallel execution of the repetition and requires the usage of a counter for termination. |

Table 2.2.: Structured Activities in WS-BPEL 2.0

```

8      <variables>...</variables>
9      <sequence>
10         <receive portType="OrderProcess" createInstance="yes" operation="
              orderBooks" variable="order"/>
11         <foreach counterName="bookCounter" parallel="yes">
12            <startCounterValue>1</startCounterValue>

```

2. Basic Concepts

```
13      <finalCounterValue>sum(/orderBooks/books)</finalCounterValue>
14      <scope>
15          <sequence>
16              <assign><!-- copy the current book to the temporary variable
17                  book -->
18                  ...
19              </assign>
20              <invoke portType="WarehouseService" operation="allocateBook"
21                  inputVariable="book"/>
22          </sequence>
23      </scope>
24  </foreach>
25  <if>
26      <condition>sum(/orderBooks/books/book/@price) > 30</condition>
27      <assign><!-- add 5 EUR to shipping & handling -->
28          ...
29      </assign>
30  </if>
31  <flow>
32      <links>
33          <link name="l1"/>
34      </links>
35      <invoke portType="InvoiceService" operation="sendInvoiceForOrder"
36          inputVariable="orderBooks"/>
37      <invoke portType="ShippingService" operation="shipOrder"
38          inputVariable="orderBooks"/>
39  </flow>
40  <assign><!-- add a success message to variable orderBooksResponse -->
41      ...
42  </assign>
43  <reply operation="orderBooks" variable="orderBooksResponse"/>
44  </sequence>
45  </process>
```

Listing 2.6: A simplified BPEL process.

Before declaring any activities, the process shown in listing 2.6 first imports its own WSDL — so its own message types can be used — and those of the Web Services it uses. In addition it specifies some partnerLinks to define its relationship to said services and declares some variables used to store data in. The following paragraph explains the activities contained in the process.

A new instance of the process will be created as soon as a message of the type `orderBooks` is received. For each book contained in that message, the service called `WarehouseService` will be invoked in parallel, sending it a message of the type `allocateBook` with the respective book copied into the message sent. If the total price of all the books exceeds 30 EUR, there will be no shipment costs — otherwise, 5 EUR for shipping and handling will be added to the final cost of the order. Now, the services `InvoiceService` and `ShippingService` will be invoked in parallel. The `InvoiceService` is sent a message of the type `sendInvoiceForOrder`, while the `ShippingService` is sent a message of the type `shipOrder`. Finally, the process replies to the initially received request using a message of the type `orderBooksResponse`.

There are many more elements and aspects to BPEL processes. Nevertheless, this

thesis will concentrate on the already presented core concepts, as the aim of the thesis is not the creation of a complete tool, but the development, exploration and evaluation of a model-driven approach to service composition, involving the generation of BPEL processes from a given model.

2.4. Model-Driven Development

In conventional software development, models — Class Diagrams, Entity Relationship Models, Activity Diagrams, etc. — are part of the documentation. They are created during the requirements and design phases and serve as structured views upon the product to be developed. This is what is commonly referred to as *model-based* development.

Model-driven development, however, assigns a new role to models. They are not only a documentation artifact, but rather constitute parts of the implementation — or sometimes even all of it. Using suitable transformations, all kinds of artifacts, among them program source code, are being generated from the models. This section explains the terminology and the theoretical basics of model-driven development and provides an overview of the consequences of this methodology.

2.4.1. Terminology

The basis of model-driven development is the definition of a domain-specific language (DSL), matching transformations and a supporting platform, enabling the conversion of the model of an application that is to be developed into an actual application. This approach is depicted in figure 2.3.

Model-driven development starts with a reference implementation — an already functional application that has the characteristics desired in the final, generated application. This not only includes the use of certain selected technologies, but also requires the architecture and other quality aspects to already be determined and implemented. Since model-driven development is primarily targeted at the generation of several applications sharing some fundamental characteristics, this is no hindrance, as the initial effort will pay off later, when application variants can be generated based on changes in the model. The set of these application variants is referred to as an application family.

The reference implementation, once available, needs to be analyzed to determine which parts of it fall into which of the following categories.

- Schematic, repetitive code that can later be generated from model properties.
- Generic code that will be the same in every application variant based on the reference implementation.

2. Basic Concepts

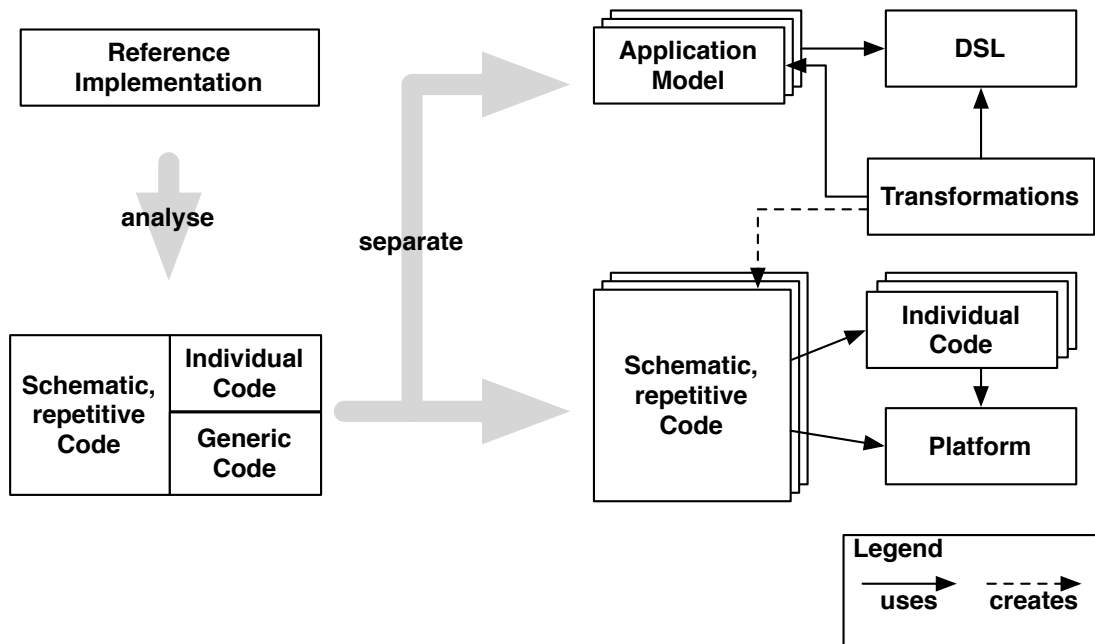


Figure 2.3.: The basic idea of model-driven development, according to [9].

- Individual code that will need to be programmed manually for each of the application variants.

A DSL needs to be chosen, if not newly created, to permit the modeling of the reference application in high-level terms. Transformations that understand the semantics of the DSL can then be created to process the application model, eventually converting it into the parts of the reference implementation that were found to be schematic and / or repetitive.

The final application will consist of this generated code, some parts that were initially found to require individual programming, and the platform. The platform consists of those parts found to be generic across all possible applications that are to be generated and thus must only be written once. An example for a platform might be an enterprise framework like Java EE [10] or a web application framework like Ruby on Rails². In some cases it might be necessary to mix existing frameworks with new source code to assemble the platform or even to exclusively rely on self-made code.

The transformation from the initial, high-level model to the final code artifacts will usually involve a set of cascaded transformations, increasing specificity with every step. As illustrated in figure 2.4, the initial platform independent model (PIM) is being transformed into more and more low-level platform-specific models (PSM), eventually becoming source code — which is actually just another PSM, but is often noted separately for clarity. According to its name, every PSM relies on a platform, of which there must be one for each abstraction level.

²<http://rubyonrails.org/>

2. Basic Concepts

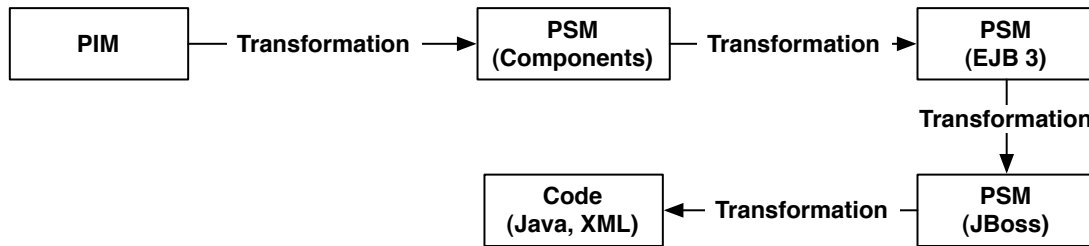


Figure 2.4.: An example for cascaded transformations. According to [9].

The cascaded transformations shown in figure 2.4 use several different DSLs — e.g., EJB 3 [12] and JBoss³, which contains a concrete implementation of the EJB 3 specification. This leads to the question about the origins of the specification of a DSL. To explain this, the concept of metamodels and meta-metamodels must be introduced.

An application model on its own has no semantic value. To be interpreted, it must be seen in the context of its DSL, which specifies the elements available to model an application along with their meaning. To formally create a DSL, another model is being created: a model describing the DSL. As this is a “model about a model”, it is referred to as the metamodel of the DSL.

But this only shifts the definition of the model semantics up to a more abstract level. Again, it is reasonable to question the semantics of the elements used in the meta-model. To be able to define the elements permitted to be used in the description of a metamodel, another shift up the abstraction hierarchy must be made: those elements are defined in a meta-metamodel.

To avoid handing off the responsibility for the semantics to more and more abstract meta-levels indefinitely, the meta-metamodel usually uses its own elements to describe itself. These relationships lead to the hierarchy shown in figure 2.5.

The following example will clarify these relationships: A class diagram could define a part of the *model* (level M1) for an application. In it, there might be a class called `Circle`. A runtime system executing the application described in the model would then create *instances* of the class `Circle` (level M0). But when `Circle` was defined to be a class in M1, the semantics of “something being a class” had to be made clear — so a *metamodel* (level M2) was referenced, which again is a model, only this time declaring the `Class` element. To define what a `Class` is, the metamodel references the *meta-metamodel* (level M3) — which is actually the metamodel’s meta-model, since the “meta” relationship is always a relative one and the metamodel is a model in itself. So the meta-metamodel provides the definition of a `Classifier`. As it would not make any sense to abstract any further, the meta-metamodel uses itself to define its own elements.

Summing this up, one might say: “This circle with a radius of 5 cm is an instance of the class `Circle`. A `Circle` is the instance of the classifier `Class`. A `Classifier` is an instance of the classifier `Classifier`. ”

³JBoss Application Server, <http://jboss.org>

2. Basic Concepts

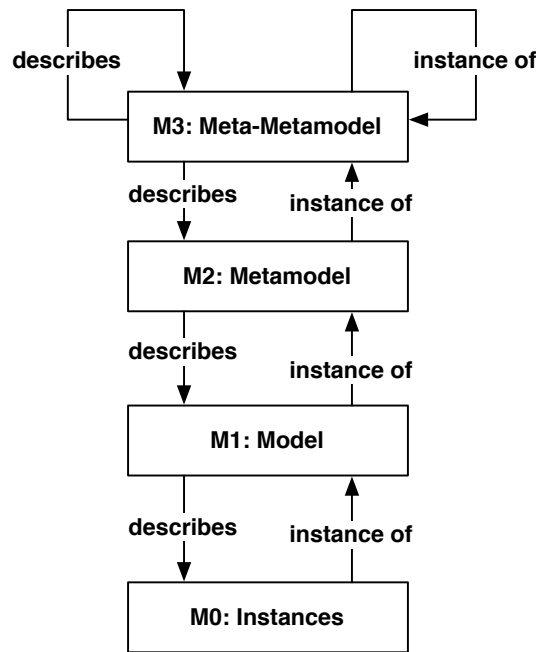


Figure 2.5.: The relationships between the different meta layers, according to [9].

Since the general concept of model-driven development is completely independent of specific standards and technologies, an overview of the standards used in the OMG's MDA approach [13] will now be given to put them in context, as some of the standards will be referenced later in this thesis.

MDA uses the Unified Modeling Language (UML) [14] for modeling. MOF, the Meta Object Facility [15], acts as the UML's metamodel and can thus be used as the meta-metamodel for all UML models. To annotate models with constraints, the Object Constraint Language (OCL) [16] is used. The "Query / Views / Transformations" (QVT) [17] specification defines mechanisms to create transformations between models — also called model-to-model transformations. Model-to-text transformations, i.e., transformations that convert a given model to actual source code or other text-based artifacts, are out of the scope of QVT and are not yet addressed by any specification. XML Metadata Interchange (XMI) [18] is used to serialize models to XML to enable transmission and exchange.

2.4.2. UML Profiles

A UML profile is an extension of the existing UML or a part thereof and is a common method to create a domain-specific language. By introducing new stereotypes and tags into the UML, models can use terms that domain-experts can understand — e.g., an EJB expert could use an `EJB Entity Bean` element in their models, while an insurance expert might use an `Insurance Policy` element when modeling an application. Employing a DSL already has the advantage of enabling domain experts to

use a terminology they already know. The use of UML profiles as a means of creating the DSL adds to that all the benefits of using standards — improving both comprehensibility for human readers and processibility for machine readers.

In the context of model-driven development, the creation of UML profiles is a suitable strategy to define metamodels for application families. On the syntactic side, the UML is a widely accepted standard understood by most developers and brings its own standardized notation. On the semantic side, the UML already provides its own meta-model, the Meta Object Facility (MOF), which has been mentioned before in the MDA context. The MOF supplies a sufficiently rich and proven set of constructs for the semantic decoration of custom metamodels created as UML profiles.

2.4.3. Consequences

Supported by a suitable development process, model-driven development can reach several desirable improvements compared to conventional software development. While the automatic generation of artifacts from models is a very obvious and tangible advancement, other effects are a little less apparent. Some of them will be outlined in the following paragraphs to motivate the adoption of model-driven development.

In model-based development, models created in the requirements and design phases merely serve as documentation. If something specified in these models changes during the implementation phase, it is easy to just ignore the models and to introduce the change in source code only. As models are part of the implementation in model-driven development, this mismatch cannot occur. Changing the implementation results in a semantically equivalent change in the model — and vice versa.

The creation of domain-specific languages improves communication between domain experts and developers. As both parties use the same language to express the application's functionality, misunderstandings and therefore defects in the application are far less likely than in traditional development. In some cases, it might for the same reasons even be feasible to let the domain experts create the models — or at least parts of them. Thus, overall application quality is improved as far as functional correctness is concerned.

The separation of generated code from platform code is not merely a technical one, but can also be organizational. While functional requirements are implemented in the models — and, therefore, in the generated code —, technical requirements are being solved by the platform and the generators. This permits the establishment of two separate development branches and decouples functional from technical issues. Thereby, greater independence between these branches is achieved, as changes in either branch do not require the other one to replicate them. E.g., a technical change like replacing a certain technology only needs to be reflected in the platform and the transformations, while functional development can proceed undisturbed.

Once a DSL along with its transformations and a platform are available, functional development does not need to restrict itself to a single application. By adding a model for an application, functionally different software can be produced with greatly reduced

2. Basic Concepts

effort. Improvements such as better performance or corrected defects in parts shared by multiple applications — i.e., those found in the platform or the transformations — are removed in all of them without requiring additional care. This results in a consistent level of quality for a whole application family.

3. Approach

This chapter describes the approach taken by this thesis to solve the problems stated in section 1.2. Based on the description of the workflow of a process modeling tool, the requirements for the graphical user interface of the tool will be presented. To solve the problem of repetitive service implementations, the concept of Commodity Services will be introduced. This chapter closes with the requirements for the generator to be developed, while the next chapter will focus on the metamodel of the processes and its graphical notation.

3.1. Intended Workflow

This section describes a workflow that aims at simplifying problems identified in section 1.2. For this workflow to be realized, software implementing it needs to be written, from now on referred to as *Composr*. After this section and the following one introduce the workflow, the sections following them will provide details about the supporting software that the workflow will require.

To permit the modeling of processes, the software will provide the user with a palette of elements representing activities possible in the model, as well as a canvas on which the elements can be arranged. To connect the activities, a transition element will be provided. Below is a list of the elements and their meaning in the context of BPEL.

- `receive` element
Equivalent to BPEL's `<receive>` element, signifying the blocking wait for a message to be received by the process.
- `reply` element
Equivalent to BPEL's `<reply>` element, to realize the returning of messages back to client.
- `invoke` element
Equivalent to BPEL's `<invoke>` element. Used to invoke Web Services with a message possibly containing a payload, and to acquire the resulting message from the Web Service.
- `flow` element
Equivalent to BPEL's `<flow>` element, to permit the parallel execution of activities.

3. Approach

- **pick element**
Equivalent to BPEL's `<pick>` element. Used to wait for incoming messages of multiple possible types or the occurrence of a timer event.
- **if element**
Equivalent to BPEL's `<if>` element, providing branching into different activities based on the evaluation of an expression that has a logical value.
- **loop element**
Equivalent to BPEL's `<while>`, `<repeatUntil>`, and `<forEach>` elements. Permits the optionally parallel repetition of an activity based on a counter or a logical value.

The process itself will be represented implicitly by the canvas containing all these elements.

Employing these elements, the Composr software will support a workflow consisting of three phases for the user and one phase executed by the software. Each phase corresponds to one role held by the user of the software. While each role may be assigned to a different person, this is not a requirement; it is perfectly reasonable for a single person to hold each of the participating roles or to have multiple persons fulfil the tasks of a single role. Figure 3.1 shows a diagram illustrating the workflow.

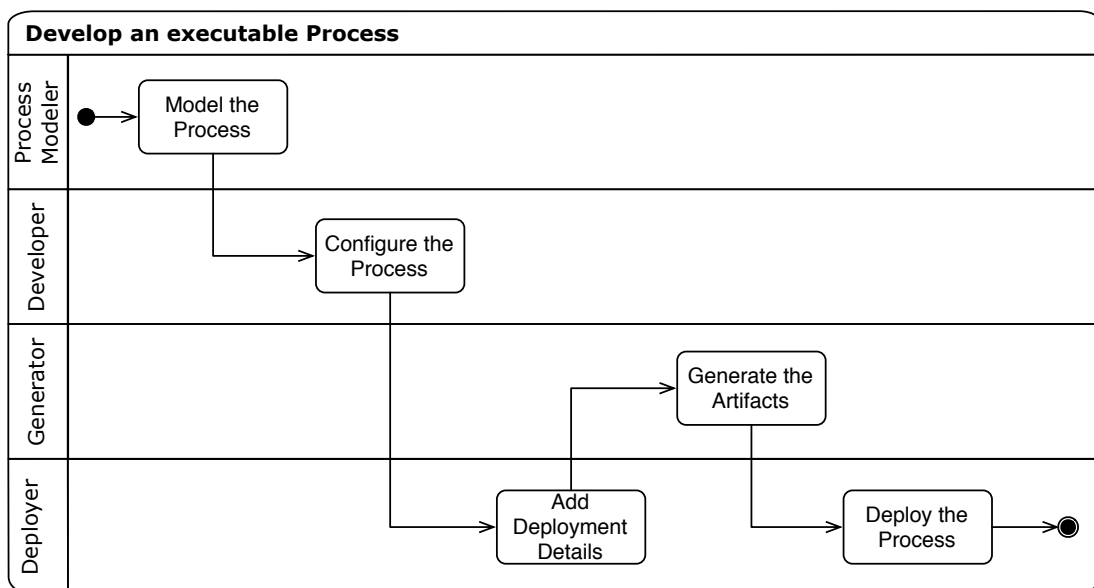


Figure 3.1.: The workflow of the Composr application. The notation is UML Activity Diagrams [19].

In the first phase, the user holds the role of the *Process Modeler*, who will model a business process using a graphical notation provided by the tool without entering any element-specific configuration data. This will result in a very compact diagram of the process which might not be easy to understand by persons not familiar with the

3. Approach

modeled process, but will provide the involved roles with a general overview of the executable process they are creating.

The role of the *Developer* will take over in the second phase, in which the modeled process is being configured. The Developer role is being introduced as this phase requires some technical knowledge of a lower abstraction level than before. Below is a list of configuration options for each element.

- `process` element
 - a name
- `receive` element
 - an operation name
 - a list of parameters, i.e., (name, data type) tuples
- `reply` element
 - a reference to a receive element that is able to reach this element
 - a list of named return types, i.e., (name, data type) tuples
 - for each data type, a value to be returned
- `invoke` element
 - a service to invoke
 - an operation from the service's interface
 - a value for each parameter of the chosen operation
- `pick` element
 - for each outgoing transition, an operation name to support, and additionally, if the `pick` element is not the first receiving activity in the process, an alarm, i.e., a time span or a point in time; with at least one operation in this list
 - for each operation, a list of parameters, i.e., (name, data type) tuples
- `if` element
 - for each outgoing transition, an XPath expression with boolean value, with all these expressions being mutually exclusive
- `loop` element
 - whether to use a boolean expression or a counter for looping
 - if the boolean option was chosen: an XPath expression with boolean value to decide whether to continue looping after each iteration
 - if the boolean option was chosen: whether to execute the contained activity at least once

3. Approach

- if the counter option was chosen: an XPath expression with unsigned integer value, defining the final value of the counter
- if the counter option was chosen: whether to execute the repetitions in parallel

In the third and final phase, the role of the *Deployer* will create deployable archives from the modeled process. The tool will ask the deployer for several additional details — shown below — that can vary from deployment to deployment.

- a base endpoint at which the generated services will be deployed, e.g., `http://localhost:8080/axis`, if the service with the name “AService” would be deployed at `http://localhost:8080/axis/AService`
- a namespace to use for the process, which will also be the base namespace for the generated services
- a directory to save the generated artifacts to

Once these details have been entered and confirmed by the Deployer, the tool will start the *Generator*, which will create deployable archives for the process and for each Commodity Service. The Deployer will be informed once generation is done and can start deploying the generated archives.

3.2. Scenario

For a better illustration of the vision directing the development of the workflow and its associated software, this section describes the intended workflow depicted in 3.1 in a story-like manner. For better illustration of the process, at some points in the scenario fragments of the current model will be presented in a generic notation.

The workflow starts with a user *U* having a goal. *U* is a developer at a mid-sized company that wants to start exploring SOA, especially to integrate better with the IT processes of two large suppliers — namely, a supplier for cables and a supplier for cable connectors.

U has already read some literature about SOA, Web Services, and BPEL, and also developed some simple BPEL processes and Web Services. But *U* quickly recognized that the tool he used to create the BPEL didn't really let him concentrate on what he was trying to achieve. Instead, he was trying to overcome namespace problems he didn't really understand and was busy creating `assign` activities when all he wanted to do was invoking a Web Service or returning a value to the client. Also, *U* quickly got weary of developing yet another Web Service when all he just needed was to persist another class of data objects. When *U* found out about Composr, he was interested and tried it out.

3. Approach

Composr presents to U a canvas on which to draw his process and a palette of the elements he can use. As he already has some knowledge about BPEL and Web Services, he quickly identifies the meaning of the notation elements. The first thing he notices is a symbol for transitions from one activity to another. Then, he finds the equivalents to BPEL's `receive`, `reply`, and `invoke` activities as well as those for the `flow`, `pick`, and `if` activities. He also finds an element for repeating activities, but instead of the three different ones BPEL provides, he can see only a single one, called `loop`. Additionally, he finds elements marking the `start` and the `end` of the process — he does not know these from BPEL, but since the palette provides tooltips, he soon understands what they mean. He does not find an element for the `assign` activity, which he notices with relief.

3.2.1. Process Modeling

U starts modeling a process that he wants to use to place orders with the suppliers. In his prior exploration of BPEL and Web Services, he has written local mocks for their services already. One supplier published his `CableService`, providing the operations `searchCables` and `orderCables`. The other supplier published the `ConnectorService`, providing the `searchConnectors` and `orderConnectors` operations.

In the first stage, he does not care about details like data types, condition expressions or service endpoints. To gain an overview of his process model, he just wants to create all the model elements he needs. He knows, he can always configure them later.

First, he places a `start` and end `end` element on the canvas, so he does not forget them. Then, he adds a `receive` element, as he wants to receive keywords for a search. He means to use the keywords in a product search, using the searching operations of the suppliers' services. So he adds an `invoke` element for each service, enclosing them in a `flow` for parallel execution.

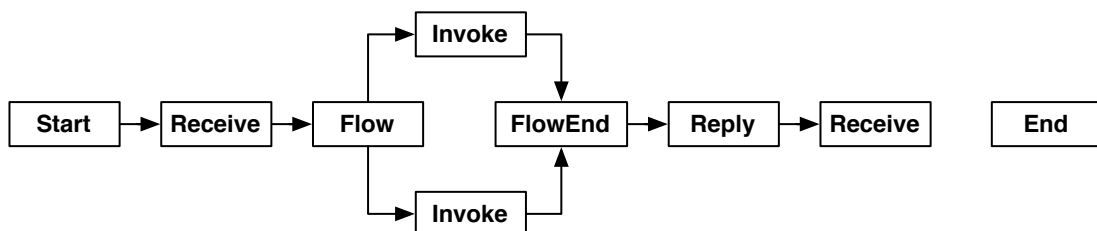


Figure 3.2.: An excerpt from U's current process model.

After the `flow` element, U adds a `reply` so he can return the search results to the client application, followed by another `receive`. He intends this to accept the actual order from the client (figure 3.2).

U appends another `flow`, this time to invoke the ordering operations of the services. After adding the `invoke` elements, he attaches a `pick` event to each and closes the

3. Approach

`flow`. He uses the `pick` elements since the order services are asynchronous services — he wants the process to wait for a reply from each service (figure 3.3).

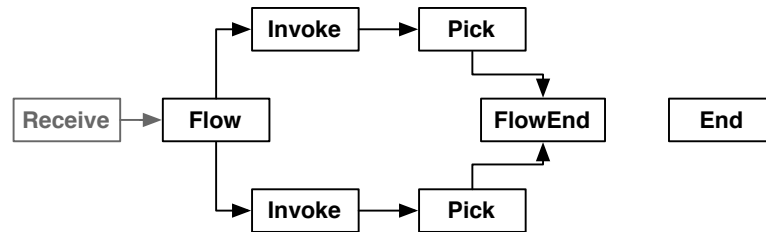


Figure 3.3.: An excerpt from U's current process model.

The next element he adds is an `if` element. Into one branch, he only puts a `reply` element, as he wants to return an error message here. Into the other branch, U places two `invoke` elements, each enclosed in a `loop` element. Around these, he creates a `flow` element so both services are being invoked in parallel. He wants to use the `invoke` elements to save the orders he placed on a machine of his company. Lastly, he lets a `reply` element follow the `loop`, so he can return a success message to the client.

U has created all activities required for his process. From time to time, he has connected them using the transition element, and now connects the last activity, the `if`, to the `end` element (figure 3.4).

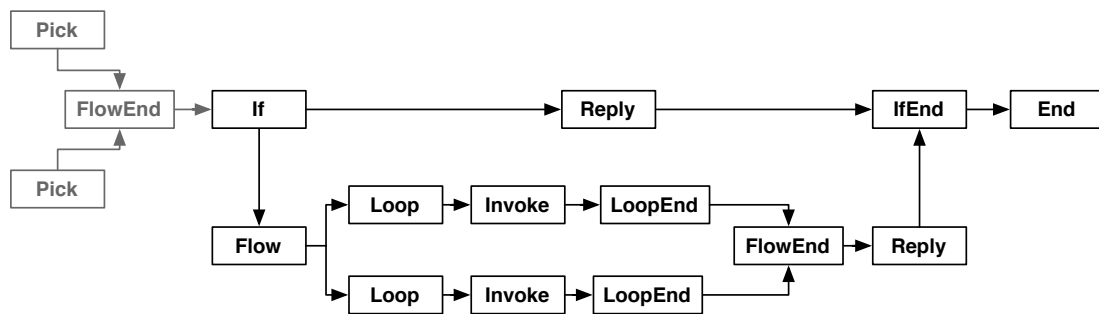


Figure 3.4.: An excerpt from U's current process model.

3.2.2. Process Configuration

U starts the configuration by choosing the initial `receive` element and assigning it the operation name *searchProducts*. He defines the argument of this operation to be an array of strings, which he calls *keywords*.

As there is no configuration to do for the `flow` element, U continues with the two service invocations contained in the `flow`. For each service, he chooses the WSDL file describing it by specifying the path to each on his computer's filesystem. He could also have opted to enter a URL to each file. Now that Composr knows about the operations provided by the services, it displays the available operations for each and lets

3. Approach

U choose one. U chooses the `searchCables` and `searchConnectors` operations, upon which he needs to choose the data input for the parameters accepted by the operations. He chooses to connect the `keywords` array to the `queryTerms` parameter each operation provides. For the `searchCables` operation, he sets the `maxLength` parameter to the literal value of 50.

Now, U configures the `reply` element directly following the `flow`. He assigns it to be a reply to the initial `receive`, choosing that from a list of possible `receive` elements. As there is only one that can reach the `reply`, he chooses the only element contained in the list. Next, U defines the `reply` to return two return attributes. He calls the first one `cableResults` and sets its data type to the `SearchResultArray` type provided by the `CableService`. The second one, he names `connectorResults` and sets its data type to the `Results` type provided by the `ConnectorService`. Now that Composr knows the interface of the `reply` element, U can determine which values are to be returned. He connects the `cableResults` attribute to the `CableService`'s `results` attribute and the `connectorResults` attribute to the `ConnectorService`'s `productsFound` attribute.

As the name of the following `receive`, U chooses `placeOrders`. He defines it to take two parameters. One is called `cableOrders`, which is an array of the `Order` type defined by the `CableService`. The other parameter is called `connectorOrders`, which is an array of the `ProductOrder` type defined by the `ConnectorService`.

U configures the `invoke` elements similar to the previous ones. Only now, he sets the values retrieved by the preceding `retrieve` to be sent to the `orderCables` and `orderConnectors` operations.

He sets the `pick` elements following each `invoke` to receive the two types of callback messages each of the two invoked operations supplies: one for a successful order, and one for an order that has failed. Additionally, U adds a timer event to the picks, set to fire after 15 minutes of waiting for the services' replies.

In the following `if` element, U specifies the conditions of the two branches. He uses an XPath expression to determine whether at least one of the orders returned a success message. If so, the process will pursue the branch containing the `loop` elements. If not, the process will continue with the single `reply` element and then end.

U configures the `loop` elements to each use a counter for looping. He defines the repetitions to be executed in parallel and sets the final counter value to be the size of the respective order arrays, which he does by referencing the values returned by the ordering services and manually fetching the size of the arrays in XPath.

For the `invoke` elements in the loops, U chooses a service already built into Composr: a Commodity Service called `PersistenceService`. For each `invoke`, he selects the `save` operation and connects to it the array of orders returned by each respective ordering service. To be able to address each order item individually, U opts for manually editing the XPath expressions, now pointing at the orders arrays. He appends to the existing expression the simple selection of the n th element of the array, with n being the current value of the surrounding loops' counters — as Composr recognizes that what U is editing is inside a loop with a counter, it makes the counter value available just like the received values before. This ensures that for each part of

3. Approach

the `orders` array, the `PersistenceService` is invoked once.

As the `invoke` elements invoking the `PersistenceService` are contained in `loop` elements, Composr provides arrays of the aggregated values of all invocations therein. U connects these to the `reply` element's `cableOrderParts` and `connectorOrderParts` attributes.

Finally, U specifies the single `reply` message to return an *errorMessage* of the type string, and sets its value to be the literal string "All orders failed. ".

3.2.3. Generation and Deployment

Now that U is done configuring the process, he can start the generation. After activating the `generate` menu option, the system asks U about a directory where to save the generated artifacts and a namespace to use. U enters both and confirms his choice.

Composr's generator now examines the process modeled and configured by U and produces several source files as output: a BPEL process, a WSDL definition for the process and an XML Schema definition to be used by the process and the generated services alike. For the service U specified, the generator creates a service implementation in Java, including classes for the data objects used, as well as a matching WSDL and an XML Schema definition, specifying the messages understood by the service. After the source code generation, the system creates deployable archives of the created service and the process.

U drops the process and service archives into their designated places in his Apache Tomcat¹ directory, which has the ActiveBPEL Engine² and the Axis2³ Web Service engine servlet installed.

Both archives deploy without problems and U begins work on his client application — which concludes the workflow description.

3.3. Graphical User Interface

The approach presented in this thesis aims at simplifying SOA development by supporting a higher abstraction level when creating executable processes. In order to achieve this, solid support from the graphical user interface is required. This section provides details on the requirements for the user interface to achieve this support. In doing so, the separation into three phases introduced in section 3.1 is reused.

The first phase, executed by the role of the Process Modeler, deals with the graphical modeling of processes. As BPEL itself has no implied graphical representation, a suitable notation needs to be chosen. The aim of the notation must not be the exact

¹<http://tomcat.apache.org/>

²<http://active-endpoints.com/active-bpel-engine-overview.htm>

³<http://ws.apache.org/axis2/>

3. Approach

replication of all elements present in BPEL, as the approach tries to abstract from it, enabling a more domain-oriented approach to business process modeling — so a more general notation is needed that can be employed by business users. Also, the notation should at least be *readable* by *all* represented roles, i.e., domain experts as well as developers and deployers should be able to intuitively understand the meaning of a process with little prior training.

The configuration phase requires an already modeled process, so the role dealing with this phase does not need to be able to actually use the notation. Understanding the basic flow of a modeled process is important, though, since specific configuration data must be applied to most elements. The user interface should support this specificity by providing custom configuration dialogs for each element while still maintaining consistency across dialogs.

All dialogs should be context-aware, i.e., they should shield the user from irrelevant options as efficiently as possible — an example would be the `reply` element's configuration dialog, in which the reference to a `receive` element must be set. The available options should only contain those `receive` elements that can actually reach the `reply` being edited.

Where needed, the dialogs must provide additional abstractions for easier configuration. E.g., the configuration of an `invoke` element contained in a `loop` element should display data sources specific to the `loop` element it is contained in. An example for this would be the availability of the current counter value of the `loop` — if and only if it actually employs a counter.

Apart from element-specific dialogs, the system should provide several supporting dialogs not directly targeted at the configuration of a concrete element, but nevertheless required for successful usage of the tool. Below follows a list of these dialogs.

- **Service Library**
To prevent the user from repeatedly providing the path or URL to WSDL descriptions in the `invoke` element's configuration dialog, the system should provide a facility to add, edit, and remove service descriptions. In the configuration dialog for the `invoke` element, the system should provide a selection list of the service descriptions already present in the Service Library as well as a shortcut to manage these service descriptions. The available services include the list of the Commodity Services (see section 3.5) available in the system, which cannot be modified using the user interface. If the user chooses to manually add a service description directly in the `invoke` configuration screen, the system should automatically add the selected WSDL to the Service Library.
- **Types Editor**
To allow for the creation of custom data types to be used in the process, the system should provide an editor for data types which allows the creation, modification, and deletion of user-created data types. This does not imply a complete XML Schema editor, but rather a simple graphical interface that allows for the creation of new types by letting the user choose a name for a new type and as-

3. Approach

sign to it a list of attributes that each also are given a name and a type from the list of the existing types. The list of the existing types includes the types native to XML Schema, types extracted from the WSDL descriptions of the services used in `invoke` elements, as well as custom types previously created by the user.

- Type Chooser

In the `receive` and `reply` configuration dialogs, the user needs to select data types to use in the process's interface. The system should provide a type chooser dialog that provides a selection list of all types known to the tool, i.e., the same types mentioned in the description of the Types Editor.

- Data Mapping Templates

To simplify mappings from one or more source data types to one or more target data types, the system should provide a facility for saving, editing, deleting, and selecting templates for such mappings. This template library should be available from all relevant dialogs, e.g., the dialogs for the configuration of the `receive`, `reply`, and `invoke` elements, providing a shortcut for the instant application of a chosen mapping to previously selected data types. Figure 3.5 shows two examples for such mappings.

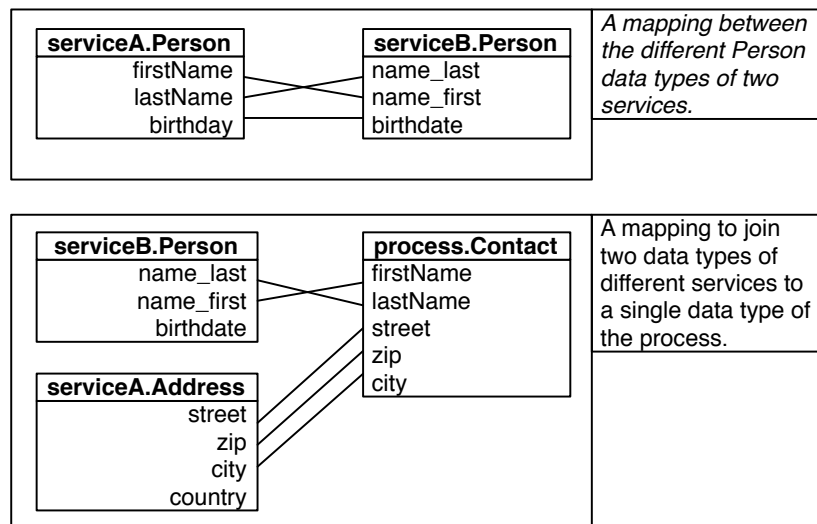


Figure 3.5.: Two examples for data mappings.

In the third phase, artifacts are being generated from the process and those artifacts are deployed by the user. To enable efficient generation, the notation used to model the process should support a block-based structure instead of a graph-based one — similar to most BPEL elements, except for the `<flow>` element. This would avoid expensive calculations checking the activity graph for validity or even restructuring it to achieve compliance with the BPEL specification. E.g., as cycles in `<flow>` graphs are forbidden in BPEL, the usage of an explicit `loop` element is suggested.

3. Approach

Before starting the generation, the user interface must provide means to let the user enter data that might change for each deployment, especially the directory into which to place the generated artifacts, the namespace to use in the XML and a base for the endpoints of Commodity Services used in the process. When generation has finished, the user might be offered to directly go to the directory he specified for generation before.

3.4. Generator Requirements

The generator should be callable from the graphical user interface and take a process model as its input. From this, the following artifacts should be generated.

- A BPEL process representing the process modeled in the graphical editor.
- A WSDL file describing the interface provided by the process.
- An XML Schema file defining the custom types used in the process.
- For each Commodity Service invoked,
 - a WSDL file describing the interface of the service,
 - an XML Schema file defining the messages used in the service,
 - an implementation of the service in a form usable for deployment in a Web Services engine,
 - a set of libraries used by the service.

All of these artifacts will be properly referencing each other, i.e., the BPEL process will reference its own WSDL as well as those of the services it invokes and the WSDL files will reference the XML Schema files defining the data types used.

A metamodel is formally introduced in chapter 4, which will ensure proper alignment between the graphical user interface and the generator.

To stay consistent with the integrated nature of the proposed tool, the result of the generation should be instantly deployable into standard containers. To keep the scope of this thesis reasonable, reference containers are defined: the process archive should be deployable into an Apache Tomcat installation running ActiveEndpoints' ActiveBPEL Engine; the service archives should be deployable into an Apache Tomcat installation running the Apache Axis2⁴ Web Services engine.

To support the continued utilization of the generated artifacts, their source code should be preserved and provided to the user after generation. Should the tool's abstractions prevent sufficient access to implementation details of the artifacts, this will ensure that standard tools can be employed to further develop the process originally modeled

⁴Apache Axis2: <http://ws.apache.org/axis2/>

3. Approach

using the proposed tool. Also, the generated source code should be as readable as reasonably possible — e.g., XML should be indented according to its structure and generated variables should be given names that reflect the semantics of the variables.

The creation of custom generators by third parties should be accounted for in the API of the software. This will allow replacing all technologies used in the generated artifacts, e.g., workflow languages other than BPEL could be supported or Commodity Services could be implemented in a programming language other than Java, using a different Web Services stack. A process once modeled in the proposed tool would thus be technology-agnostic. Although the technology used in this thesis is the most widely used, other technologies might be more suitable for niche industries or for exploration projects.

3.5. Commodity Services

To remove some manual programming work from the development of executable processes, the software application proposed in this thesis employs the concept of Commodity Services. These are services that do not exist yet, but can be generated from the data provided in the `invoke` element's configuration dialog. The name *Commodity Service* was chosen to reflect that several business functions that still need to be programmed manually should be a commodity today, as they present problems that have already been solved generically and could thus be automated.

Of course, not every service is suitable for such a generation — the concept of Commodity Services is targeted at business functions that only vary based on the available configuration data and can thus easily be generated. For each Commodity Service, the software must include a specialized generator.

One example is the persistence of arbitrary data types — depending on the data type passed to the service in the configuration of the `invoke` element, a different service will be generated, providing persistence functions for the respective type. Possible operations provided by the service might be the creation, modification, deletion, and selection of values of the data type persisted by the service. E.g., invoking the persistence service with a value of the type `Person` would result in the generation of a `PersonPersistenceService`. Of course, other types of Commodity Service are imaginable, a selection will be outlined in section 8.2.2.

To allow for the easy modification of the software by third parties wishing to provide their own generators for Commodity Services, the application's API should be structured in a way suitable for this kind of extension.

3.6. Summary

This chapter introduced the general approach to business process modeling and generation pursued by this thesis. The workflow intended to be supported by a software

3. Approach

application has been outlined. Requirements for the graphical user interface and the generator needed to support the proposed workflow were determined. It has been shown that the perspectives of different roles will be supported, enabling each to concentrate on their own field of knowledge. The concept of Commodity Services was introduced, which promises to free developers from repetitive tasks present in many SOA projects, by generating certain services from configuration data.

Figure 3.6 gives a high-level overview of the functionality of the Composr software.

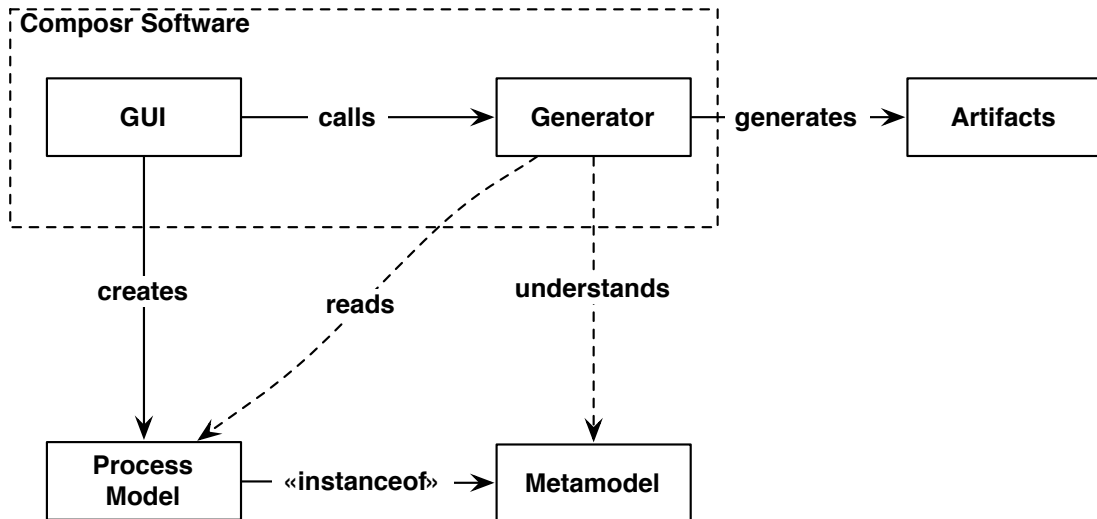


Figure 3.6.: A high-level overview of the Composr software.

4. The Composr UML Profile

To formally define the elements needed to represent the workflow presented in the preceding chapter, this chapter introduces a metamodel for executable processes. The first section defines the elements and their semantics by creating a UML profile and annotating it with OCL constraints, while the second section defines the graphical representation for the elements. Mapping the elements of the metamodel to a BPEL process and associated artifacts will be outlined in chapter 5.

4.1. Metamodel

This section presents the metamodel developed as a formal basis for this thesis' approach. To preserve clarity, several decisions have been made:

- The metamodel has been partitioned into multiple diagrams, each responsible for a particular area.
- The constraints, written in OCL¹, were not included in the diagrams, but appear in the describing text.
- Every class not showing an explicit extension is extending `UML::Classes::Kernel::Class` implicitly, even though this is not being shown in the diagrams.
- The class name `ActivityNode` is used as a shorthand for `UML::Activities::FundamentalActivities::ActivityNode`.
- The class name `ActivityEdge` is used as a shorthand for `UML::Activities::BasicActivities::ActivityEdge`.
- Every unnamed role in an association, composition, or aggregation implicitly has been given the name of the target stereotype, with its first letter being lower-case — e.g., an association from a `StructuredActivity` to an `OpenElement` implicitly can be referenced as *openElement* from the `StructuredActivity`.

4.1.1. Foundation

Figure 4.1 introduces the basic elements of the Composr metamodel. The `Linkable` stereotype acts as a general interface for all elements that should be connected using

¹Object Constraint Language, <http://omg.org/docs/ptc/03-10-14.pdf>

4. The Composr UML Profile

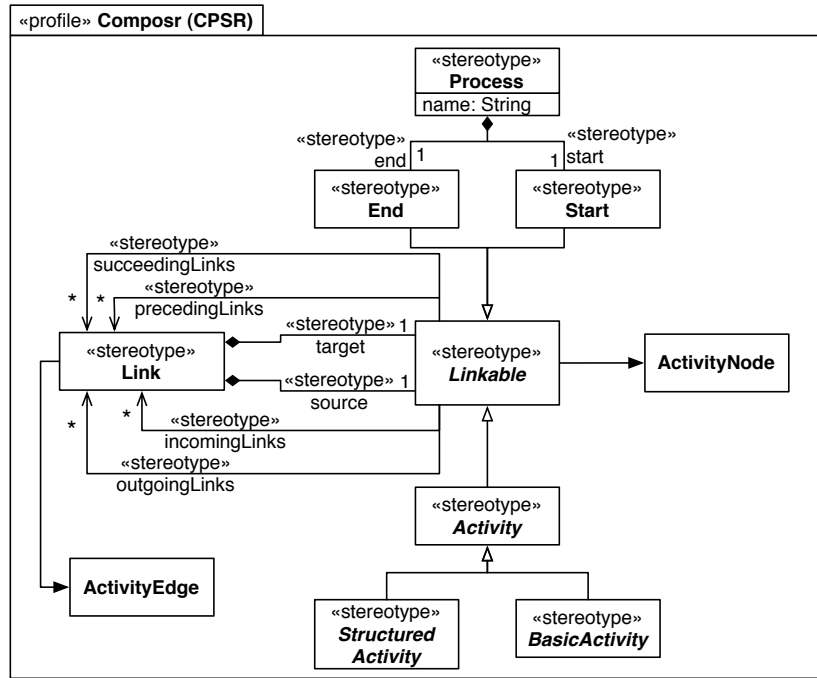


Figure 4.1.: Composr Metamodel: Foundation.

source and target Links; The Start and End stereotypes as well as the stereotypes for the different kinds of activities inherit from it. Linkable extends ActivityNode while Link extends ActivityEdge, both of which stem from the UML Activities package. A Link always has a *target* and a *source* Linkable. Linkable itself has four collections of Links.

The *incomingLinks* and *outgoingLinks* collection have the obvious semantics as defined in listing 4.1.

```

1 context Linkable
2   inv: self.outgoingLinks->forall(link : Link | link.source = self and not(
3     link.target = self))
4   inv: self.incomingLinks->forall(link : Link | not(link.source = self) and
5     link.target = self)

```

Listing 4.1: A part of the OCL constraints for the Linkable stereotype.

The *precedingLinks* — the Links leading up to the Linkable — and *succeedingLinks* — the Links following the Linkable — are required for some constraint definitions regarding the nesting of StructuredActivities. Listing 4.2 formalizes them using two OCL constraints.

```

1 context Linkable
2   inv: self.precedingLinks = self.incomingLinks->union(
3     self.incomingLinks->select(
4       link : Link | link.source oclIsKindOf CloseElement
5     )->collect(
6       link : Link | link.source.openElement.precedingLinks

```

4. The Composr UML Profile

```
7      )->asSet()
8    )->union(
9      self.incomingLinks->select(
10       link : Link | not(link.source oclIsKindOf CloseElement) and not(link.
11         source.type oclIsKindOf StructuredActivity)
12     )->collect(
13       link : Link | link.source.precedingLinks
14     )->asSet()
15   )
16   inv: self.succeedingLinks = self.outgoingLinks->union(
17     self.outgoingLinks->select(
18       link : Link | link.target oclIsKindOf StructuredActivity
19     )->collect(
20       link : Link | link.target.closeElement.succeedingLinks
21     )->asSet()
22   )->union(
23     self.outgoingLinks->select(
24       link : Link | not(link.target oclIsKindOf closeElement) and not(link.
25         target oclIsKindOf StructuredActivity)
26     )->collect(
27       link : Link | link.target.precedingLinks
28     )->asSet()
29   )
30 )
```

Listing 4.2: A part of the OCL constraints for the `Linkable` stereotype.

Listing 4.3 contains the constrains for the `Start` and `End` element, amongst others guaranteeing a `Start` element to always lead to an `End` element across any number of `Links` and the other way around.

```
1 context Start
2   inv: self.succeedingLinks->exists(link : Link | link.target = process.end)
3   inv: self.incomingLinks.size() = 0
4   inv: self.outgoingLinks.size() = 1
5
6 context End
7   inv: self.precedingLinks->exists(link : Link | link.source = process.start
8     )
9   inv: self.incomingLinks.size() = 1
10  inv: self.outgoingLinks.size() = 0
```

Listing 4.3: A part of the OCL constraints for the `Linkable` stereotype.

The constraints shown in listing 4.4 further refine the flow of `Links` between `BasicActivities` and `StructuredActivities`.

```
1 context Link
2   inv: not(self.source = self.target)
3
4 context BasicActivity
5   inv: (inv: self.precedingLinks->exists(L_1 : Link | L_1.source oclIsKindOf
6     StructuredActivity) implies self.succeedingLinks->exists(L_2 : Link |
7     L_2.target = L_1.source.closeElement)
8   inv: self.incomingLinks.size() = 1
9   inv: self.outgoingLinks.size() = 1
```

4. The Composr UML Profile

```

9  context StructuredActivity
10 inv: self.succeedingLinks->exists(link : Link | link.target = self.
    closeElement)
11 inv: self.outgoingLinks.size() = self.closeElement.incomingLinks.size()
12 inv: self.outgoingLinks.size() > 0
13 inv: self.incomingLinks.size() = 1
14 inv: self.closeElement.openElement = self

```

Listing 4.4: Additional OCL constraints for the basic elements of the metamodel.

4.1.2. Basic Activities

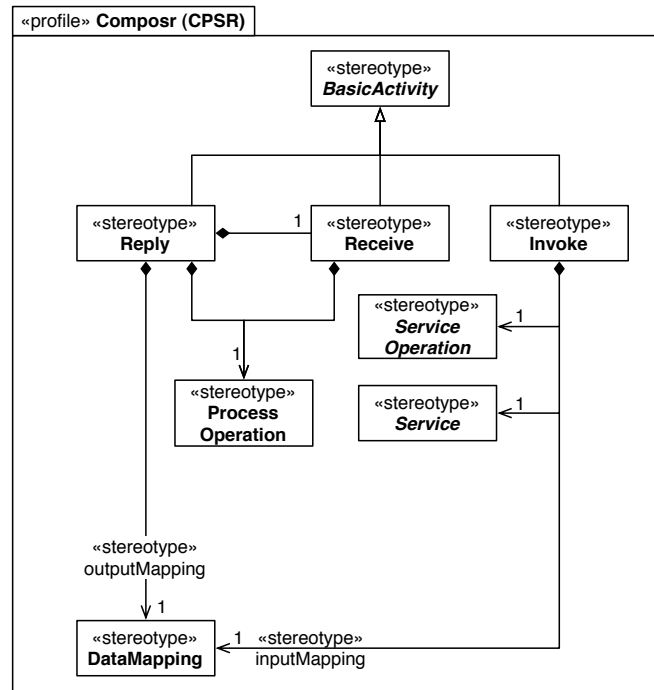


Figure 4.2.: Composr Metamodel: Basic Activities.

The diagram presented in figure 4.2 shows the part of the metamodel mostly concerned with the `BasicActivity`. The `Receive`, `Reply` and `Invoke` stereotypes are all `BasicActivities`. To assign values to parameters passed to remote partners, the `Invoke` and `Reply` stereotypes employ a `DataMapping`, which will be examined further in figure 4.5. Each `Receive` and `Reply` element takes part in a `ProcessOperation`, while an `Invoke` element uses a `ServiceOperation` offered by a `Service`. Relevant constraints are being shown in listing 4.5.

```

1  context Reply
2  inv: self.precedingLinks->exists(link : Link | link.source = self.receive)
3
4  context Invoke

```

```
5 | inv: self.service.operations->exists(o : Operation | o = self.
    serviceOperation)
```

Listing 4.5: OCL constraints for the portion of the metamodel primarily concerned with the descendants of `BasicActivity`.

4.1.3. Services

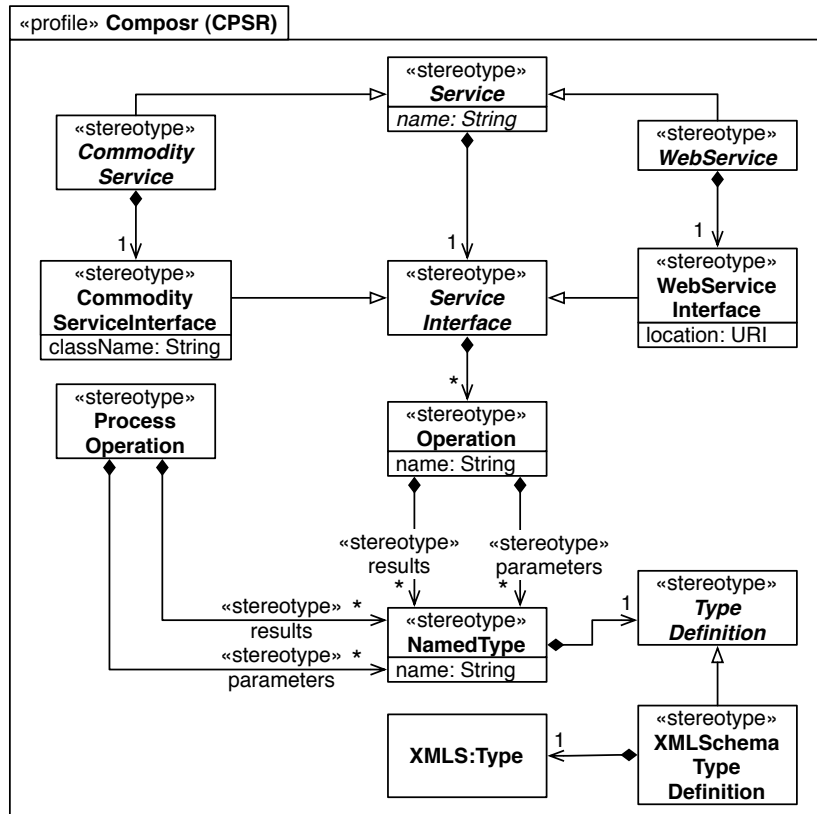


Figure 4.3.: Composr Metamodel: Services.

In Figure 4.3, attention is being paid to services. There are two kinds of services — the `WebService` and the `CommodityService` — and each has a specialized `ServiceInterface`. Each `ServiceInterface` has multiple `Operations`, which, like a `ProcessOperation`, each have two sets of `NamedTypes` — its *parameters* and its *results*. A `NamedType` is a tuple of a *name* and a `TypeDefinition`. The concrete kind of `TypeDefinition` used in this thesis is the `XMLSchemaTypeDefinition`, which has an `XMLNS:Type` as defined by [1]. There are no additional OCL constraints for these stereotypes.

4. The Composr UML Profile

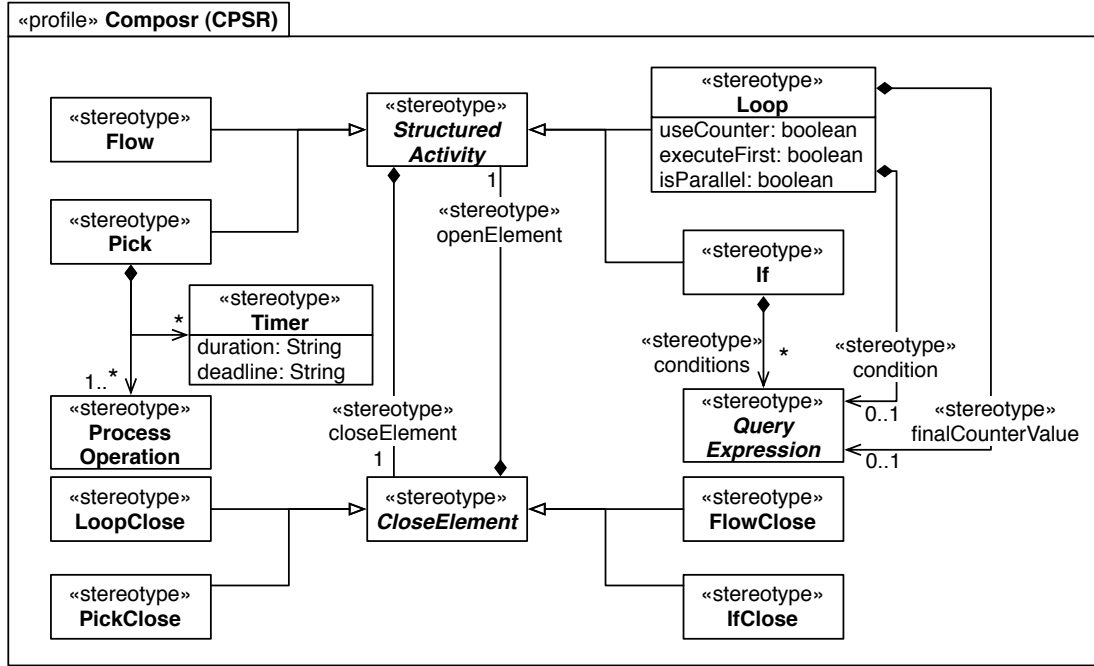


Figure 4.4.: Composr Metamodel: Structured Activities.

4.1.4. Structured Activities

Figure 4.4 shows the `StructuredActivities` present in the metamodel — `Flow`, `Pick`, `If` and `Loop` — as well as their `CloseElements`. The `Timer` element's *timespan* and *dateTime* are constrained according to their respective counterparts “duration-expr” and “deadline-expr”, as defined in section 11.6 in the WS-BPEL 2.0 specification [8]. Listing 4.6 adds some OCL constraints.

```

1 context Pick
2   inv: self.closeElement oclIsKindOf PickClose
3   inv: self.incomingLinks->exists(link : Link | link.source oclIsKindOf
4     Start) implies self.timers.size() = 0
5   inv: self.processOperations.size() > 0
6
7 context PickClose
8   inv: self.openElement oclIsKindOf Pick
9
10 context If
11   inv: self.closeElement oclIsKindOf IfClose
12   inv: self.outgoingLinks.size() == self.conditions.size() or self.
13     outgoingLinks.size() == self.conditions.size() + 1
14
15 context IfClose
16   inv: self.openElement oclIsKindOf If
17   inv: self.conditions->forAll(qe : QueryExpression | qe.resultType =
18     QueryResultType::boolean)
19
20 context Loop

```

4. The Composr UML Profile

```

18  inv: self.outgoingLinks.size() = 1
19  inv: self.closeElement oclIsKindOf LoopClose
20  inv: self.condition.resultType = QueryResultType::boolean
21  inv: self.finalCounterValue.resultType = QueryResultType::unsignedInteger
22  inv: self.useCounter = true implies self.condition = null
23  inv: self.useCounter = true implies executeFirst = false
24  inv: self.useCounter = false implies self.finalCounterValue = null and
    isParallel = false
25
26  context LoopClose
27    inv: self.openElement oclIsKindOf Loop
28
29  context Flow
30    inv: self.closeElement oclIsKindOf FlowClose
31
32  context FlowClose
33    inv: self.openElement oclIsKindOf Flow

```

Listing 4.6: The OCL constraints for the StructuredActivities.

4.1.5. Data

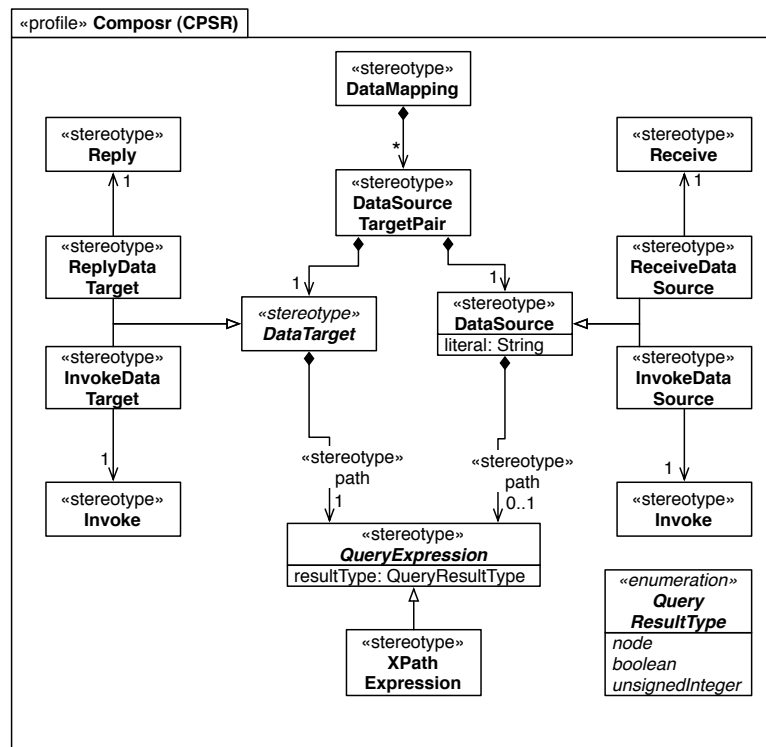


Figure 4.5.: Composr Metamodel: Data.

Figure 4.5 is concerned with how data from `Receive` and `Invoke` elements is used in `Reply` and other `Invoke` elements. Listing 4.7 shows additional OCL constraints

relevant to this portion of the metamodel.

```
1 context DataSource
2   inv: (self.literal = null and not(self.path = null)) or (not(self.literal
3     = null) and self.path = null)
4   inv: self.path.resultType = QueryResultType::node
5 context DataTarget
6   inv: self.path.resultType = QueryResultType::node
```

Listing 4.7: OCL constraints for the portion of the metamodel concerned with data flow.

4.2. Notation

To tailor the presentation of the semantics defined in the metamodel to the needs of this thesis' approach, this section defines a notation for the Composr UML profile. The graphical elements are based on UML Activities — commonly referred to as Activity Diagrams — as defined in [19].

Since there is no standard notation for BPEL processes, existing BPEL editors usually invent their own notation. As the main intent of these notations is the presentation in their respective tool only, the graphical elements easily become rather intricate. This may lead to notations that are unsuitable for simple drawing on paper and process models that appear too complicated for intuitive understanding.

As an example, the notation used in Active Endpoints² ActiveBPEL Designer³ can be seen in section 4.2.2, in which a survey comparing two different notations is presented.

It is a basic assumption of this thesis that a notation for modeling executable business processes should be easily understandable by both domain experts and technology experts, and that these should be able to intuitively grasp processes modeled using that notation — maybe not resulting in a complete understanding, but at least providing a rough overview and orientation, improving communication between roles. The notation presented in this section is aimed at reaching these goals.

4.2.1. Initial Draft

After the examination of existing approaches, two widely tried notations for executable business processes were identified: BPMN [20] and UML's Activity Diagrams [19].

While BPMN is directly targetted at modeling business processes, it has been found to be difficult to map it to BPEL[21]. Also, the relatively high number of different elements present in even simple BPMN models seemed to contradict the requirements formulated before.

²<http://active-endpoints.com/>

³<http://www.active-endpoints.com/active-bpel-designer.htm>

4. The Composr UML Profile

For Activity Diagrams, approaches successfully creating a UML profile with a mapping to BPEL4WS exist [22], [23]. Although the problems found in the BPMN mappings were not encountered here, the profiles pursued a direct mapping between Activity Diagrams and BPEL, effectively requiring the user to recreate in UML the structure found in the XML.

Further discussion of this issue can be found in chapter 7.

Activity Diagrams were chosen as a basis for the notation developed in this thesis, as they define a very concise set of elements and are part of a proven, extendable standard — the UML. While existing approaches created direct mappings between Activity Diagrams and BPEL, this thesis will provide abstractions expected to be suitable for real-world use. Following is a list the elements defined in the Composr metamodel and intended to be visible in a process model, accompanied by their graphical notations chosen during the first draft.

Foundation

The `Process` element will be represented by the canvas on which the elements are being placed.

The `Start` and `End` elements are shown in figure 4.6. The `Start` element is represented by a white circle with a black outline, while the `End` element uses a white circle with a black outline and a cross rotated by 45°.



Figure 4.6.: Graphical representations of the `Start` and `End` stereotypes.

To connect two `Linkable` elements, the `Link` stereotype was introduced. Figure 4.7 shows its graphical representation, a black arrow with a solid line and a solid arrowhead at the *target* end of the `Link`.

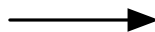


Figure 4.7.: Graphical representations of the `Link` stereotype.

Basic Activities

Figure 4.8 shows, from left to right, the `Receive`, `Reply`, and `Invoke` elements' graphical representations. The `Receive` element's representation is a white circle with a black outline and a black arrow with a solid line and a solid arrowhead, pointing to the circle's center from the upper left. The `Reply` element is represented by a white circle with a black outline and a black arrow with a solid line and a solid arrowhead, originating at the circle's center and pointing to the lower right. Both representations

were chosen to imply data arriving and departing from the elements. The `Invoke` stereotype was decided to be represented by the symbol for the Action node type from Activity Diagrams — a white rectangle with rounded edges, a black outline and a label in its center. The text “operation” is to be replaced with the name of the operation the `Invoke` is referring to.



Figure 4.8.: Graphical representations of the stereotypes extending `BasicActivity`.

Structured Activities

Figure 4.9 depicts the `Loop` element's representation. It consists of two white circles with a black outline and an arrow inside each of them, of which the left one represents the actual `Loop` stereotype and the right one is the `LoopClose` stereotype. The arrow in the `LoopClose` element's circle has been rotated by 180° to clarify start and end of the `Loop`. Both elements are connected by a solid black line for the same reason.

The arrows are not part of the symbol, but have been included to clarify the use of the element. While the left arrow is the incoming `Link` and the right arrow is the outgoing `Link`, the middle arrow represents an empty activity and could be replaced by other basic or structured activities, thus allowing for arbitrary nesting levels.

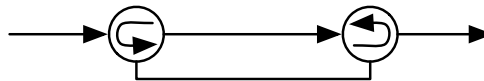


Figure 4.9.: The graphical Representation of the `Loop` stereotype.

In figure 4.10, the `If` stereotype is shown on the left, while its closing element, the `IfClose`, is shown on the right side. Both are white squares with a black outline, rotated by 45°. Each arrow that is outgoing from the `If` element represents either a condition — i.e., a `QueryExpression` of the result type `QueryResultType::boolean` — or the *else* branch of the `If`. Again, as with the `Loop` elements, all arrows shown were included for purely instructional reasons, and are not part of the actual symbols.

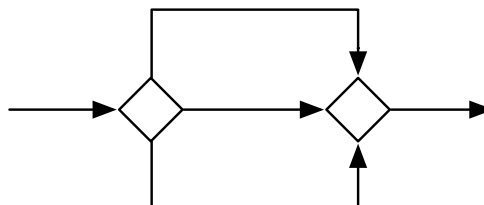


Figure 4.10.: The graphical Representation of the `If` stereotype.

4. The Composr UML Profile

The representation for the `Pick` element is shown on the left side in figure 4.11 alongside its closing element, the `LoopClose`, which is on the right. Both symbols are black squares rotated by 45°. Each arrow outgoing from the `Loop` element represents either a `ProcessOperation` receivable by the `Pick` or a `Timer` that could be fired. The arrows were included for the same reasons as in the description of the `Loop` element.

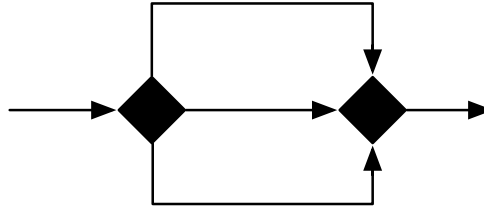


Figure 4.11.: The graphical Representation of the `Pick` stereotype.

Figure 4.12 introduces the symbols for the `Flow` and `FlowClose` elements. None of the five arrows is part of the actual symbol, they have again been included to clarify the usage of the element as in the description of the `Loop` element's representation. Each of the arrows between the opening and closing elements could be replaced by arbitrarily nested activities that would then be executed in parallel by the resulting BPEL process.

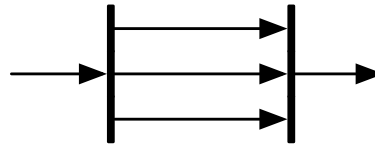


Figure 4.12.: The graphical Representation of the `Flow` stereotype.

4.2.2. Survey

From December 6th to December 16th 2007, a survey was conducted to assess the relative clarity of the notation created in the preceding section. The survey asked the participant to compare two anonymous notations A and B, where notation A was the notation presented here and notation B was the notation used by the ActiveBPEL Designer by ActiveEndpoints mentioned at the beginning of this chapter.

The survey was being taken by filling out forms on a web page. It set a cookie afterwards that would disable participating more than once, thereby at least discouraging multiple entries from the same person. The participants were primarily students of Computer Science at the Leibniz Universität Hannover.

An english translation of the survey can be found in appendix A. Correct answers have been added to the questions where applicable.

The questions of part 1 were asked to find out about the background of the participant, especially if there was any knowledge that would help in understanding the diagrams

4. The Composr UML Profile

from parts 3 and 4. The questions of parts 3 and 4 were asked to check whether each respective notation could be understood without prior introduction of the process shown and without knowing the exact semantics of the notation. The number of correct answers thus was assumed to be a measure for the comprehensibility of a notation. The questions asked in part 4 were supposed to obtain from the participant a subjective opinion concerning certain qualities of the presented notations.

4.2.3. Results of the Survey

19 persons participated in the survey. Table 4.1 shows the results in a condensed form. Percentages have been rounded to the nearest integer value.

The following observations follow from this data:

- Parallel execution is depicted much clearer in the Composr notation than in the ActiveBPEL Designer notation (questions 2.1 and 3.1).
- Except for questions 2.1 and 3.1, the ActiveBPEL Designer notation consistently had more correct answers, implying a lack of comprehensibility for the Composr notation. Supporting this, of all participants, about 70 % thought the Composr notation was clearer. But 70 % also thought the ActiveBPEL Designer notation to be more comprehensible. The Composr notation should thus trade some clarity for better comprehensibility.
- About 25 % of the participants thought that notation A had a fine amount of detail, while about 60 % thought it contained too little. For notation B, about 40 % thought that it contained too much detail, while 50 % thought the detail was just about right. This seems to imply that the Composr notation needs a little more detail, but not as much as the ActiveBPEL Designer notation.

In the informal comments to the survey not represented here it became clear that the `Receive` and `Reply` symbols used by the Composr notation were generally not well understood. Also, the line connecting the `Loop` element with its `LoopClose` element was a source of confusion. The ActiveBPEL Designer notation, however, was being criticized for excessive use of space. Also, its symbol for a service invocation was sometimes thought to be a repetition because of the two arrows surrounding it.

The succeeding section will use these observations to modify the notation to counter some of its apparent downsides.

4.2.4. Final Version

This section introduces the final version of the Composr notation, addressing some of the issues found in the preceding section.

4. The Composr UML Profile

| Survey Results for the Composr Notation | |
|--|--------------------|
| Question | Result |
| Part 1 | |
| Participants familiar with Activity Diagrams | 79 % |
| Participants using Activity Diagrams at least from time to time | 32 % |
| Participants familiar with BPEL | 53 % |
| Participants assuming to understand BPEL syntax | 39 % |
| Part 2 | |
| Correct answers for Question 2.1 | 89 % |
| Correct answers for Question 2.2 | 63 % |
| Correct answers for Question 2.3 | 37 % |
| Correct answers for Question 2.4 | 84 % |
| Correct answers for Question 2.5 | 79 % |
| Correct answers for Question 2.6 | 74 % |
| Correct answers for part A overall | 71 % |
| Part 3 | |
| Correct answers for Question 3.1 | 74 % |
| Correct answers for Question 3.2 | 63 % |
| Correct answers for Question 3.3 | 79 % |
| Correct answers for Question 3.4 | 89 % |
| Correct answers for Question 3.5 | 89 % |
| Correct answers for Question 3.6 | 79 % |
| Correct answers for part B overall | 79 % |
| Part 4 | |
| Thought notation A / B was more comprehensible | 32 % / 68 % |
| Thought notation A / notation B was clearer | 74 % / 26 % |
| Thought notation A contained too much / about right / too little details | 16 % / 26 % / 58 % |
| Thought notation B contained too much / about right / too little details | 37 % / 53 % / 10 % |

Table 4.1.: Survey Results for the Composr Notation

As the percentage of survey participants who knew UML's Activity Diagrams was relatively high (79 %), it is assumed that bringing the notation more in line with similar elements known from Activity Diagrams will improve comprehensibility of the notation.

Also, a main difference between the Composr notation and the notation of the ActiveBPEL Designer is the strikingly complementary text use: while the former uses almost no text at all, the latter allows annotation of virtually every graphical element. It is believed that this is also an area in which the Composr notation should be improved.

The Composr notation will thus be modified as follows:

- The representation of the `start` stereotype will be replaced with the represen-

4. The Composr UML Profile

tation of the `InitialNode` node type of Activity Diagrams.

- The representation of the `End` stereotype will be replaced with the representation of the `ActivityFinal` node type of Activity Diagrams.
- The representation of the `Reply` stereotype will be replaced with the representation of the `SendSignalAction` node type of Activity Diagrams.
- The representation of the `Receive` stereotype will be replaced with the representation of the `AcceptEventAction` node type of Activity Diagrams.
- The black line connecting the `Loop` stereotype with its `LoopEnd` will be removed to bring it more in line with the other stereotypes inheriting from the `StructuredActivity` stereotype.
- The `Pick` and `If` stereotypes' outgoing `Links` will be annotatable using text strings. Since these labels will not bear any semantics relevant to the metamodel whatsoever, they will only be introduced in the notation, while the metamodel will stay untouched.

Expansion regions from UML's Activity Diagrams have been considered as a notation for the `Loop` element, as its semantics allow for the execution of an action for each item in a list. In the end this was found to be more confusing, though, as it would have permitted the `Loop` element for a use similar to the "for each" construct of the Java language, but would have complicated loops based on counters or plain boolean expressions. An abstraction similar to the mentioned "for each" construct could be introduced in a future revision of the notation.

Figure 4.13 shows the modified `Start` and `End` representations. The `Start` element, situated on the left side, now is a black circle, while the `End` element on the right side is a black circle inside a white circle with a black outline.



Figure 4.13.: Updated graphical representations of the `Start` and `End` stereotypes.

Figure 4.14 shows the modified `Receive` and `Reply` representations in context with the unmodified `Invoke` representation. The `Receive` element, which is shown on the left, is now made up of white rectangle with a black outline and a single concavity on its left side. The `Reply` element also is a white rectangle with a black outline, but features a single convexity on its right site.

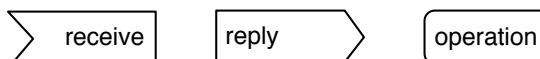


Figure 4.14.: Updated graphical representations of the `Receive` and `Reply` stereotypes.

4. The Composr UML Profile

Figure 4.15 shows the updated `Loop` and `LoopClose` representations, in which the black line formerly connection both has been removed.

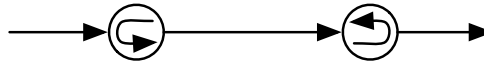


Figure 4.15.: Updated graphical representation of the `Loop` and `LoopClose` stereotypes.

Figure 4.16 shows the `Pick` and `PickClose` representations in context with their updated `Links`, which may now feature a textual label.

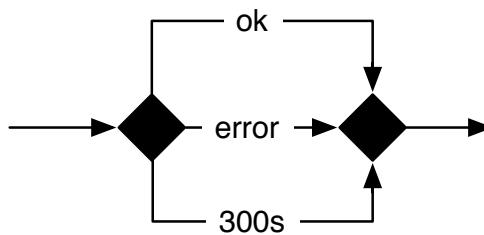


Figure 4.16.: Updated graphical representation of the `Links` of the `Pick` and `PickClose` stereotypes.

Figure 4.17 shows the `If` and `IfClose` representations together with their `Links` that have also been updated to permit textual labels.

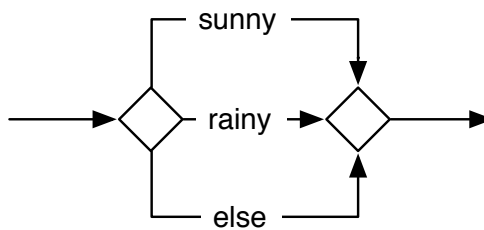


Figure 4.17.: Updated graphical representation of the `If` and `IfClose` stereotypes.

The addition of labels to the `Links` of the `Pick` / `PickClose` and `If` / `IfClose` pairs allows the modeler to informally annotate the `Links`, while leaving the exact XPath expressions to the developer configuring the process.

Table B.1 in appendix B shows all elements of the final version of the Composr notation, listing the available graphical representations and their respective stereotypes.

5. Development

This chapter describes the development of a prototype of the Composr application, covering design and implementation for the generator and the GUI.

5.1. Development Process

As described by [9], when developing a model-driven solution, there are several advantages in beginning the project with a reference implementation, and then developing the models and transformations that are supposed to generate the given reference. This process can be iterated, each time extending the reference implementation and adjusting the models and transformations. Each iteration aims at reaching a state in which the artifacts generated from the models again align with the reference implementation.

This thesis attempted to pursue the described development process. But due to several technological challenges that often hindered development significantly, this could not be achieved practically: Over the course of the thesis, not a single complete iteration was completed. There is a reference implementation, though, which has gradually been approximated by its generating model and the developed transformations, and at the end of the thesis was nearly generated completely. The reference implementation consists of a BPEL process, a commodity service it invokes, as well the the XML Schema definitions and WSDL descriptions required by both.

The process model that was used as input for the transformations is shown in figure 5.1, using the notation developed in chapter 4. The process receives a `Person` object, uses the `PersistenceService` to store it, and then returns the now persisted object it retrieved from the service.

To achieve the generator's degree of completeness that will be presented in this chapter, the priority of the GUI had to be lowered. The user interface is therefore incomplete as well. To proceed with the development of the generator despite the lack of a fully functional GUI, the model shown in figure 5.1 was reconstructed in Java code as a graph of objects. It exhibits the same structure as the model that would have been passed from the graphical editor. Therefore, the process model was suitable for tests of the generator, and alignment of the generated artifacts could be confirmed successfully for most reference artifacts.

As the generation of the XML could not be controlled completely, some aspects of the reference were modified to better match the format of the generated XML documents, so equality could be confirmed in code much easier. The changes that were made

5. Development

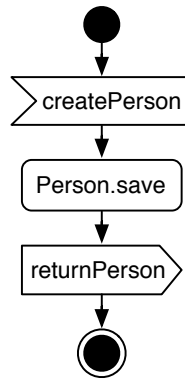


Figure 5.1.: The model used for developing the transformations, shown in the Composr notation.

were completely syntactical and did not change any of the semantics of the documents, so the actual requirements inherent in the reference implementation remained factually unchanged. Examples for modification are element and attribute ordering and distribution of whitespace. Thus, once a generated artifact was aligned with the updated reference artifacts, a successful test would be available that could still be used to constantly verify the continued correctness of the transformations.

5.2. Generator Design

Before the generator could be designed, some aspects had to be clarified: the technologies to be used, the dependencies between the artifacts, and the integration of the generator with the graphical tool. This section explains which decisions were made regarding these questions and provides the reasoning leading to the decisions. Afterwards, the section will explain the actually implemented architecture of the generator.

5.2.1. Technology

Several kinds of artifacts need to be created or modified by the generator. Primarily, these were XML Schema definitions, WSDL descriptions, BPEL code, and a service implementation in Java. Except for the Java code, all of these were generated by using a library providing an object model of the respective XML language. The Java code and some simple changes in build scripts or configuration files were generated using simple string manipulation in Java, as the low quantity of the work to be done in these formats would not have justified more elaborate approaches.

The following paragraphs will be concerned with primarily XML-related technology required for the generation of BPEL processes and the interfaces of the generated services.

To generate XML Schema definitions, the object model provided by the XML Schema

5. Development

Infoset Model¹ of the Eclipse project was chosen, as it was the most complete object model available. Another option was the XMLSchema² provided by the Apache Software Foundation, but some central requirements could not be met as easily as with the one provided by Eclipse.

For WSDL, a formal specification exists that defines an object model as well as a WSDL writer and a reader: JWSDL³ from the Java Community Process. The implementation used was also its reference implementation, WSDL4J⁴, which has been developed by IBM. As the JWSDL has existed for a significant amount of time, development in the area of WSDL object models seems to have settled, so no other viable option was found.

To generate the BPEL code, the object model included in the Eclipse BPEL Project⁵ was chosen. The project covers much more, as it even provides a graphical designer. Only the object model was needed for this thesis though, so it was extracted from the project. Since the BPEL Project is still in active development and remains slightly immature, Java archives of the object model were built from the sources provided and updated regularly. Companies such as IBM, Intel, and Oracle are actively contributing to the Eclipse BPEL Project, so sufficient progress is expected to be sustained for some time. Other BPEL models showed to rely on a lot more dependencies on other libraries and were therefore denied.

The next paragraphs will discuss technologies primarily required for the implementation of the commodity service concept. Since this thesis will provide only a single implemented commodity service, the technologies will in part be related to its special requirements. This commodity service is the PersistenceService, providing operations that enable the persistence of the data objects used in the process.

Many solutions for persistence in Java exist, supporting diverse concepts. To allow for the object-oriented modeling of the data model used by a process, a persistence framework facilitating the use of simple Java beans as data model classes was sought after. The final decision was made in favor of the Java Persistence API (JPA), which is a part of the EJB 3 specification⁶, and for which Hibernate provides an implementation. This permits the use of a standardized API and the unobtrusive annotation of model classes to turn them into persistable entities. Hibernate was chosen as an implementation as it is a proven solution that has already been widely used. Also, prior experience with the framework was available.

To be able to expose the implementation of the PersistenceService as a Web Service, a Web Services engine had to be chosen. A requirement for this was to allow for the generation of a service skeleton based on the generated WSDL description of the service, so the implementations for the services' operations could be included

¹<http://eclipse.org/xsd/>

²<http://ws.apache.org/commons/XMLSchema/>

³<http://jcp.org/en/jsr/detail?id=110>

⁴<http://sourceforge.net/projects/wsdl4j>

⁵<http://www.eclipse.org/bpel/>

⁶<http://jcp.org/en/jsr/detail?id=220>

5. Development

as effortlessly as possible. After the exploration of both Apache Axis2⁷ and Metro⁸ from the GlassFish project, Axis2 was chosen based on features and usability of the respective approaches. The decision was also supported by a comparison of several Web Service stacks⁹.

For the transmission of instances of the data model classes via XML messages over the network, a method for marshalling and unmarshalling the Java objects needed to be decided upon. Axis2 supports several approaches. A short description and the reasoning for or against each method is provided in the list below.

- Selecting none of the provided XML binding libraries will not create any data model classes. Instead, the OMElement class of AXIOM, the Axis Object Model, is used as a fallback. Such a service interface would be hard to use and maintain, as every data type would be representing only by a nested tree of OMElements. So it was decided against this option.
- XMLBeans¹⁰, part of the Apache XML Project, is unsuitable for use in conjunction with Hibernate, which requires all data model classes to have a default constructor without any arguments. As this is not offered in XMLBeans, it could not be used.
- JAXB¹¹, the Java Architecture for XML Binding, is allegedly supported, but in an experimental, undocumented state. Therefore, it was not chosen.
- ADB, the Axis2 Databinding Framework, is a simple schema compiler and Java generator integrated into Axis2. It was not chosen because the Java beans it creates contain much infrastructure code that would be hard to maintain manually should the need arise.
- JiBX¹², which injects its infrastructure code only into the compiled class files, generates completely clean source code for the Java beans of the required types. Due to its consequently low constraints on the usage of the generated beans, it is the solution chosen for this thesis. Another benefit is the availability of the Xsd2Jibx tool¹³, which creates the Java beans from a given XML Schema definition. The central concepts of JiBX will be explained in more detail in section 5.3.3.

5.2.2. Artifact Dependencies

In order to generate all needed artifacts, generation was first divided into stages, each corresponding to a particular class of artifact. As there are several dependencies

⁷<http://ws.apache.org/axis2/>

⁸<https://metro.dev.java.net/>

⁹<http://wiki.apache.org/ws/StackComparison>; last access: 2007-12-17

¹⁰<http://xmlbeans.apache.org/>

¹¹<https://jaxb.dev.java.net/>

¹²<http://jibx.sourceforge.net/>

¹³<http://jibx.sourceforge.net/xsd2jibx/>

5. Development

between the artifact classes, a dependency graph was created. That dependency is shown in figure 5.2, illustrating the order of generation it implies. To preserve clarity, dependencies of a class covered by another dependency have been omitted from the diagram.

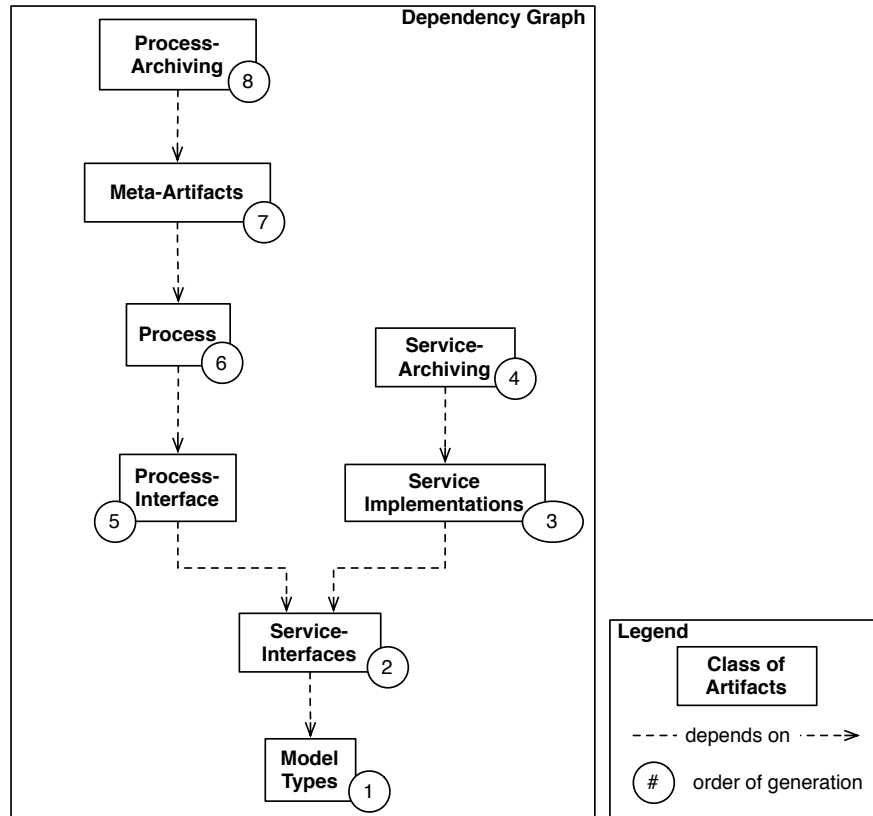


Figure 5.2.: The dependency graph used to model the dependencies between artifact classes.

(1) As can be seen from figure 5.2, the model types have no dependencies at all, they can be generated only using the modeled process. In the class of technologies used in this thesis, this corresponds to the generation of an XML Schema definition for the model types used in the process.

(2) The service interfaces are required by both the service implementations as well as the process interface which must declare the services it uses. This results in the WSDL descriptions of the commodity services being generated next.

Generation can now proceed on either of the two branches of the dependency graph. This thesis continues with the right side, which is concerned with the services, while the left side is about the process. This choice is arbitrary.

5. Development

(3) The service implementations are dependent on the presence of their interfaces as well as the model types and may be generated next. In this case, the skeleton classes for the service are combined with the Java implementations of the operations defined by each commodity service.

(4) To be able to archive the generated services, all of their three components must already exist: the implementation, the interface, and the definition of the model types used by the service. The concrete result is an Axis2 archive (AAR) that can be deployed in the servlet provided by Axis2.

The following artifact classes all concern the process side of generation.

(5) The process interface, a WSDL description of the operations and messages used by the process, can be generated next. It requires the presence of both the model type definitions and the service interface descriptions, as it references them both.

(6) The actual process, in this case the concrete BPEL code, can only be generated if the process interface, the service interfaces, and the model type definitions are in place.

(7) As the meta artifacts are basically references to the files created earlier, all those are required to exist. These artifacts help the process engine to find everything that is referenced by a process. As this thesis targets the ActiveBPEL Engine, a catalog of WSDL and XML Schema files as well as a deployment descriptor for the process are included herein.

(8) Finally, the process, along with its interface, the service interfaces it uses and the model type definitions it needs, can be merged into a process archive. For the ActiveBPEL Engine, the format for the archive is BPR. The resulting file can then be deployed into the engine.

5.2.3. Integration with the GUI

In an exploration phase prior to actual design and development, the exchange format of the process between the graphical user interface and the generator was examined. It was found out that the GUI can provide a Java object graph representing the created process model. When the user activated generation of the process, the GUI would simply pass the Java object containing the process on to the generator without using any intermediary format.

To permit this, both the user interface and the generator were developed to use the same classes to represent the process model, this common set of classes would then be the connection between GUI and generator. Therefore, for each stereotype from the UML profile defined in chapter 4, a Java class was created. As the framework used by the GUI follows strict rules based on the Model-View-Controller design pattern, the model classes only include model-specific behaviour.

5.2.4. Architecture of the Generator

To allow interested parties to develop their own replacements of the generator shown in this thesis, an interface for a generation strategy was created according to the Strategy design pattern. The methods required by this interface were then called in the actual generator class, according to the Template Method design pattern. The methods declared by the interface `GenerationStrategy` are shown below. All of them are `public` and have a `void` return type. They correspond to the generation phases introduced in section 5.2.2.

- `generateTypes()`
- `generateServiceInterfaces()`
- `generateServiceImplementations()`
- `archiveServices()`
- `generateProcessInterface()`
- `generateProcess()`
- `generateMetaArtifacts()`
- `archiveProcess()`

The concrete class implementing the `GenerationStrategy` interface is the `BpelGenerationStrategy`. As it might be desirable to replace only one particular technology used in the generation, the `BpelGenerationStrategy` was divided into smaller generators, each responsible for its own class of artifacts. Interested parties could thus replace the `XsdTypeGenerator` with a `RelaxNgTypeGenerator` to supply schema definitions using the RelaxNG¹⁴ schema language or replace the `BpelProcessGenerator` with a `YawlProcessGenerator` to support the YAWL¹⁵ workflow language.

The interchangeability of the `MetaArtifactsGenerator` is especially interesting, as it currently provides the meta artifacts necessary for the BPEL process to run in the ActiveBPEL Engine. Other engines will most likely require different meta artifacts, which can be implemented by replacing the `MetaArtifactsGenerator` with another generator.

¹⁴<http://relaxng.org/>

¹⁵<http://yawlfoundation.org/>

To allow for the addition of new commodity services, the `CommodityService` interface was introduced, providing sufficient information for the generator to create a service from it. Concrete commodity services need to implement the interface themselves and provide the class name of the implementation to the graphical editor. `Invoke` elements can then be configured to use the new service, and its class name can be used by the GUI to inspect it — to find out about the operations supported and their respective input and output types. The generator can create instances of a concrete `CommodityService` using reflection, thereby gaining access to all details that are required to generate the service and invoke it in the BPEL process.

The `Type` stereotype described in the UML profile created in chapter 4 was mapped to the Java interface `CpsrType`. To permit the distinction between simple and complex types, there are two implementations of the interface: `CpsrSimpleType` and `CpsrComplexType`. These classes are used to represent the data types used in the process. They are being created by the user — e.g., to set the type of a `Receive` element — and are used by the generator to create XML Schema and Java beans representing the data model of the process. As a single type might be used multiple times in the same process, a `public String getSignature()` method was added. It generates a string that is identical for two types if they contain the same sub-types. Overloading the `equals` method of the type classes therefore permits the usage of a simple `java.util.Set` to resolve duplicate types to a single objects, which is required for artifact generation.

5.3. Generator Workflow and Artifacts

For each class of artifact, this section provides a description of how that particular artifact is being generated. The classes of artifacts have been described in section 5.2.2 before. The specified algorithms all use pseudo-code for clarity and assume the existence of several low-level algorithms that would not be of any interest in understanding the generation's concept.

5.3.1. XML Schema

The XML Schema object model from the Eclipse Modeling Project is used to create

1. an XML Schema definition for the data model types used by the services and the process,
2. for each service, an XML Schema definition defining the wrapper types for the messages the service uses, and
3. for the process, an XML Schema definition defining the wrapper types for messages the process uses.

(1) Algorithm 1 describes the strategy used to generate an XML Schema definition containing all the data model types used within the process. It inserts all types used in the `Receive`, `Reply`, and `Invoke` elements into a set, resolving duplicated as has been described in the preceding section. For each of the types found, it then creates an XML Schema representation.

Algorithm 1 XML Schema generation for data model types

```

Set types = new Set()
for all activity in process.getBasicActivities() do
  if activity instanceof Receive then
    types.add(activity.getProcessOperation().getParameters())
5:  else if activity instanceof Reply then
    types.add(activity.getProcessOperation().getResults())
    else if activity instanceof Invoke then
    types.add(activity.getServiceOperation().getParameters())
    types.add(activity.getServiceOperation().getResults())
10:  end if
end for
for all type in types do
  schema.addComplexTypeDefinitionForType(type)
end for
  
```

(2) Algorithm 2 shows how the XML Schema elements for the messages of the invoked services are being generated, effectively creating the basis for a service interface that conforms to the “wrapped” variation described in section 2.2.1. A simplified example for the message types used by the `save` operation of the `PersistenceService` bound to the `Person` type is shown in listing 5.1. A full version is shown in listing C.2 in appendix C.

```

1 <schema
2   targetNamespace="http://composr/tns/PersonPersistenceService">
3   <element name="save">
4     <complexType>
5       <sequence>
6         <element name="person" type="Q1:Person"/>
7       </sequence>
8     </complexType>
9   </element>
10  <element name="saveResponse">
11    <complexType>
12      <sequence>
13        <element name="person" type="Q1:Person"/>
14      </sequence>
15    </complexType>
16  </element>
17 </schema>
  
```

Listing 5.1: The XML Schema elements used in the messages of a `PersistenceService` bound to a `Person` type.

Algorithm 2 XML Schema generation for the service interfaces

```

for all invoke in process.getInvokeActivities() do
  for all operation in invoke.getService().getOperations() do
    XMLSchemaElement requestElement
    requestElement.setName(operation.getName())
5:   for all type in operation.getParameters() do
     requestElement.addElementForType(type)
   end for
    schema.addElement(requestElement)
    XMLSchemaElement responseElement
10:  responseElement.setName(operation.getName() + "Response")
    for all type in operation.getResults() do
     responseElement.addElementForType(type)
    end for
    schema.addElement(responseElement)
15: end for
end for

```

(3) The XML Schema elements for the messages used by the operations of the process are generated very similar to what has been shown in algorithm 2. As these definitions do not reside in their own file, but are generated into the `types` element of the WSDL file of the process, their generation is part of the description of the generation of the process interface in the succeeding section.

5.3.2. Web Services Description Language

Two different kinds of WSDL files need to be generated: those describing the commodity services used in the process (1), and the WSDL description for the process itself (2). The WSDL descriptions of existing service are merely being copied.

(1) Algorithm 3 shows the basic strategy for the generation of the WSDL files for the commodity services invoked in the process. For each invoked commodity service, the XML Schema definitions containing the service's messages and those for the data model types are first being imported. A single PortType is added to the WSDL, and to the PortType, for each operation of the service a WSDL operation and a binding are being added. The WSDL operations use the XML Schema definitions for the message types described in the preceding section. In appendix C, the WSDL description generated for the PersistenceService bound to a `Person` type can be found in listing C.1. As can be seen, it uses the "wrapped" variation described in section 2.2.1 and supported by the XML Schema definitions shown in listing 5.1.

Algorithm 3 WSDL generation for the services

```

for all invoke in process.getCommodityInvokeActivities() do
  wsdl.importMessageElements(invoke)
  wsdl.importModelTypes(process)
  PortType pt = new PortType()
5:  for all operation in invoke.getService().getOperations() do
    WsdlOperation wsdlOperation = new WsdlOperation()
    WsdlMessage messageIn = new WsdlMessage()
    messageIn.setElement(getXsdElementForOperationInput(operation))
    wsdlOperation.setInputMessage(messageIn)
10:  WsdlMessage messageOut = new WsdlMessage()
    messageOut.setElement(getXsdElementForOperationOutput(operation))
    wsdlOperation.setOutputMessage(messageOut)
    pt.addOperation(wsdlOperation)
    wsdl.addBindingForOperation(operation)
15:  end for
  wsdl.addPortType(pt)
end for

```

(2) The generation of the WSDL description for the process is being outlined in algorithm 4. The XML Schema inside the `types` element of the WSDL first imports the WSDL descriptions of all invoked services, and then defines the message types to be used by the process's messages. For this, the generator queries the `Receive` and `Reply` elements for the parameters and results of the `ProcessOperation` implied by each `Receive` element. As multiple `Reply` elements with different return types may be related to the same `Receive` element, these different return types are joined into a new complex type.

Then, similar to the WSDL generation for the commodity services described above, a single port type is added to the WSDL, containing the operations of the process. The messages of these operations use the message types defined in the schema before. To create the `PartnerLinkTypes` for the process, first a set is created containing all distinct services that are invoked by the process. Finally, a `PartnerLinkType` referencing only the process is added to the WSDL description. The `PartnerLinkTypes` generated for the process model used during development are shown in listing 5.2 as an excerpt of the WSDL file for the process. Listing C.3 in appendix C shows the full WSDL description that was generated.

```

1  <partnerLinkType name="Process01PLT">
2    <role name="Process01Role" portType="Process01PortType"/>
3  </partnerLinkType>
4  <partnerLinkType name="PersonPersistenceServicePortTypePLT">
5    <role name="Process01Role" portType="Process01PortType"/>
6    <role name="PersonPersistenceServicePortTypeRole" portType="
    PersonPersistenceServicePortType"/>
7  </partnerLinkType>

```

Listing 5.2: The PartnerLinkTypes generated for the process model used during development.

Algorithm 4 WSDL generation for the services

```

wsdl.setTypes(new Schema())
for all invoke in process.getCommodityInvokeActivities() do
    wsdl.getSchema().addImport(invoke.getService())
end for
5: for all receive in process.getProcessOperations() do
    wsdl.getSchema().addMessageTypeForReceive(receive)
    wsdl.getSchema().addCombinedMessageTypeForReplies(receive.getReplies())
end for
PortType pt = new PortType()
10: wsdl.addPortType(pt)
    for all receive in process.getProcessOperations() do
        WsdOperation wsdlOperation = receive.createWsdOperation()
        pt.addOperation(wsdlOperation)
    end for
15: Set invokedServices
    for all invoke in process.getCommodityInvokeActivities() do
        invokedServices.add(invoke.getService())
    end for
    for all service in invokedServices do
20:     wsdl.addPartnerLinkType(process, service)
    end for
    wsdl.addPartnerLinkType(process, null)

```

5.3.3. Commodity Services

As was outlined in section 5.2.4, the generator uses the `CommodityService` interface common to all commodity services to generate all required artifacts. As the generation of XML Schema definitions and WSDL descriptions has already been covered in sections 5.3.1 and 5.3.2, this section describes how the information provided by the `CommodityService` interface is used to generate the actual implementations for the commodity services. Algorithm 5 outlines the general approach to service generation, implemented in the `generateServiceImplementations` method of the `BpelGenerationStrategy`.

For each invocation of a commodity service, algorithm 5 creates an instance of that commodity service and parameterizes it with the input and output types configured in the `Invoke` element the service is referenced from. This way a commodity service instance can specialize itself: e.g., the `PersistenceService` creates different services depending on the data type that is being passed to it — passing a `Person`, e.g., would

5. Development

create a `PersonPersistenceService`. For this service instance, the `Xsd2Jibx` tool then generates a binding file, mapping the service's message types and the data model types to Java classes. The `Wsd12Java` tool of Axis2 is then being used to generate an Ant build file and all Java classes required for an Axis2 service. Also, a skeleton class containing the service's declared methods will be generated, albeit with empty method bodies.

Algorithm 5 Generation of the implementations of the commodity services

```
for all invoke in process.getCommodityInvokeActivities() do
    Service service = Reflection.instantiate(invoke.getService().getClassName())
    service.setInputTypes(invoke.getServiceOperation().getParameters())
    service.setOutputTypes(invoke.getServiceOperation().getResults())
5: File binding = Xsd2Jibx.createBinding(service)
    File[] serviceFiles = Wsd12Java.generateServiceClasses(service, binding)
    File skeletonClass = serviceFiles.getImplementationSkeleton()
    for all operationName in skeletonClass.getOperationNames() do
        skeletonClass.inject(service.getImplementation(operationName))
10: end for
    for all library in service.getLibraries() do
        library.copyTo(serviceFiles.getLibDirectory())
    end for
    service.processModelClasses(serviceFiles.getModelClasses())
15: for all file in service.getMetaFileNames() do
        file.create()
        file.setContent(service.getMetaContent(file))
    end for
    File buildFile = serviceFiles.getBuildFile()
20: buildFile.injectTarget("jibx.bind")
    buildFile.runTarget("jibx.bind")
end for
```

For each operation name found in the skeleton class, the generator queries the service instance for its implementation and injects it into the class. Then, all libraries and files for the META-INF directory required by the service are being copied into their correct places. Finally, the build file generated by `Wsd12Java` is modified to include an additional build target to compile all classes and inject the JiBX modifications into the bytecode. That build file is then executed, which concludes the generation of the service.

The `archiveServices` method of the `BpelGenerationStrategy` simply executes the Ant task that the Axis2 build file provides for archiving. This creates an AAR file, readily deployable into the Axis2 servlet.

The Persistence Service

The Persistence Service is a commodity service that will allow the process modeler to easily persist data used in the process. To achieve this, it provides the `save` operation, which can be passed any type. Depending on the passed type, a specialized service will be generated, named after the type. E.g., passing a `Person` to the Persistence Service would result in the generation of a `PersonPersistenceService` that is invoked from the process. Other operations to retrieve, update, and delete data are of course also imaginable, but have not been implemented in this thesis.

The `save` operation takes an “anytype” from the XML Schema recommendation¹⁶ and returns the data that was passed along with an automatically generated id. The database configuration to be used can be retrieved by the service using the `Invoke` element’s `options` association, which must have been set by the user in the configuration dialog.

The reference implementation for the Persistence Service uses Hibernate for persistence. The resulting commodity service therefore executes the following modifications on the generated service.

- Copy the Hibernate and JPA libraries to the service’s library directory.
- Annotate the classes of the data model with `@Entity` and `@Id`.
- Create a `persistence.xml` file in the service’s META-INF directory for configuration.
- Inject the operation implementations into the method bodies of the skeleton class.

5.3.4. Business Process Execution Language

The BPEL implementation of the business process given by the process model is generated according to algorithm 6, which sets up the BPEL process. It does not add the actual activities modeled in the process, which is done in the `addSuccessors` method. It is initially called from algorithm 6, but then recursively calls itself until all model elements have been added to the BPEL process.

Algorithm 6 begins by importing into the BPEL process the WSDL description of the process itself as well as those of the invoked services. For all `PartnerLinkTypes` defined in the process WSDL, the generator adds a `PartnerLink` to the process. It creates an empty `<variables>` element and sets an empty `<flow>` as the process’s activity. Then, it calls the `addSuccessors` method, which will recursively walk the process graph to add all elements in the process model to the BPEL process.

The `addSuccessors` method takes as its input a collection of model elements, a flow element that these elements are contained in, and a link element. The link element is

¹⁶<http://www.w3.org/TR/xmlschema-0/#anyType>

Algorithm 6 The algorithm used for BPEL generation

```

WsdIFile processWsdI = process.getWsdI()
bpel.addImport(processWsdI)
for all invoke in process.getInvokeActivities() do
    bpel.addImport(invoke.getService().getWsdI())
5: end for
    for all partnerLinkType in processWsdI.getPartnerLinkTypes do
        PartnerLink pl = new PartnerLink()
        pl.setType(partnerLinkType)
        pl.setMyRole(partnerLinkType.getProcessRole())
10: pl.setPartnerRole(partnerLinkType.getServiceRole())
        bpel.addPartnerLink(pl)
    end for
    bpel.setVariables(new Variables())
    Flow rootFlow = new Flow()
15: bpel.setActivity(rootFlow)
    addSuccessors(process.getStart().getTargetElements(), rootFlow, null)

```

required to connect the source element of the passed model elements to each of its successors.

For each model element, the method will add its corresponding BPEL elements to the process, and then call itself recursively, passing as model elements the successors of the current element.

Each kind of model element needs to be treated differently. Implementations exist for the `Receive`, `Invoke`, and `Reply` elements, but are incomplete — their `<assign>` elements are not generated correctly yet.

Listing C.5 in appendix C shows the BPEL generated from the model used during development. The generation of the deployment artifacts and the archiving of the process could not be implemented in the time frame set for this thesis.

5.4. Graphical User Interface

As explained in section 5.1, the priority for the GUI was lowered in favor of a more complete generator. Therefore, this section presents only a portion of what was planned for the tool. It first shows some sketches of the interface envisioned for the full tool, and then shows how the graphical editor was implemented, leaving out the configuration dialogs from the sketches.

5.4.1. Sketches

This section presents some sketches for user interface dialogs related to the dialogs discussed in section 3.3. Even though they could not be integrated into the final software, they are believed to give valuable clues to the envisioned workflow.

Types Editor

Figure 5.3 shows the Types Editor while being used to edit the custom “Address” data type. On the left side, a selection box of all available types is being shown. These include the built-in types from the XML Schema recommendation, types that were imported from services that are used in the process, as well as custom types created by the user. On the right hand side, the name and the properties of the data type can be edited.

| Types Editor | | | | | | | | | | | | | |
|---|--|--------------------------------------|------|--|---------------------------------------|-------------------------------------|-----|---------------------------------------|-----------------------------------|-----|---|--------------------------------------|-----|
| Built-in Types integer double string Types from Services de.u...se.pws.Person de.u...se.pws.Subject de.u...se.pws.Thesis Custom Types Person User Address Create new Type | Name: <input type="text" value="Address"/> Delete Type | | | | | | | | | | | | |
| | <table border="1"> <thead> <tr> <th>Type</th> <th>Name</th> <th></th> </tr> </thead> <tbody> <tr> <td><input type="text" value="string"/> ></td> <td><input type="text" value="street"/></td> <td>(X)</td> </tr> <tr> <td><input type="text" value="string"/> ></td> <td><input type="text" value="city"/></td> <td>(X)</td> </tr> <tr> <td><input type="text" value="com.se...Country"/> ></td> <td><input type="text" value="country"/></td> <td>(X)</td> </tr> </tbody> </table> | Type | Name | | <input type="text" value="string"/> > | <input type="text" value="street"/> | (X) | <input type="text" value="string"/> > | <input type="text" value="city"/> | (X) | <input type="text" value="com.se...Country"/> > | <input type="text" value="country"/> | (X) |
| | Type | Name | | | | | | | | | | | |
| | <input type="text" value="string"/> > | <input type="text" value="street"/> | (X) | | | | | | | | | | |
| | <input type="text" value="string"/> > | <input type="text" value="city"/> | (X) | | | | | | | | | | |
| | <input type="text" value="com.se...Country"/> > | <input type="text" value="country"/> | (X) | | | | | | | | | | |
| | Add Property | | | | | | | | | | | | |

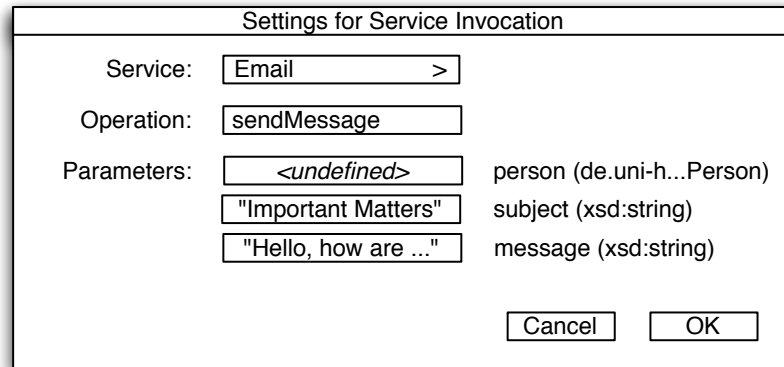
Figure 5.3.: A sketch for the Types Editor, currently being used to edit the custom “Address” data type.

Data Mapper

Figure 5.4 presents the Settings dialog for the `Invoke` element. Currently, an invocation of an Email service is being configured: an operation has been chosen and parameters are just being populated with values to be passed. While the two bottom parameters have already been filled with literal values, the “person” parameter is still undefined. As the user clicks the box missing a value, the Data Mapper dialog is being shown.

The Data Mapper is depicted in figure 5.5. Having opened from the Settings dialog shown in figure 5.4, it is being used to choose values for the “de.uni-hann.Person” type required by the Email service. This target type is shown on the left, while the data for

5. Development



A dialog box titled "Settings for Service Invocation". It contains the following fields:

- Service: >
- Operation:
- Parameters:

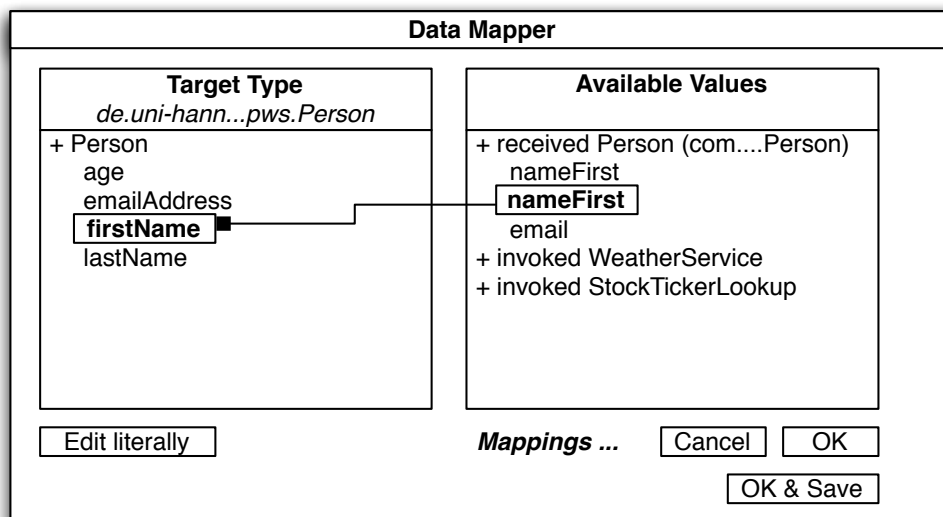
| | |
|--|----------------------------|
| <input type="text" value="<undefined>"/> | person (de.uni-h...Person) |
| <input important="" matters\""="" type="text" value="\"/> | subject (xsd:string) |
| <input ...\""="" are="" hello,="" how="" type="text" value="\"/> | message (xsd:string) |

At the bottom right are "Cancel" and "OK" buttons.

Figure 5.4.: A sketch for the Settings dialog of the `Invoke` element, being used to configure the invocation of an Email service.

the parameters of the service can be chosen on the right side. Here, for each `receive` and each `invoke` preceding the invocation currently being configured, are shown with their respective result types. In the sketch, the user chooses the “nameFirst” attribute from a prior `receive` element and drags it onto the “firstName” attribute of the target type. This is being repeated for each attribute of the target type.

Because the user regularly needs this exact mapping between the different Person types, he uses the “OK Save” button to not only apply the mapping to the service invocation, but to also save the mapping as a template. Should the user later encounter the same target type and source types again, the tool would propose to apply the saved mapping.



A dialog box titled "Data Mapper". It is divided into two main panels:

- Target Type**: `de.uni-hann...pws.Person`
 - + Person
 - age
 - emailAddress
 - firstName** (highlighted with a small square)
 - lastName
- Available Values**
 - + received Person (com....Person)
 - nameFirst (highlighted with a box)
 - email
 - + invoked WeatherService
 - + invoked StockTickerLookup

A line connects the **firstName** attribute in the Target Type to the **nameFirst** attribute in the Available Values.

At the bottom left is an "Edit literally" button. At the bottom right are "Mappings ...", "Cancel", "OK", and "OK & Save" buttons.

Figure 5.5.: A sketch for the Data Mapper, shown while choosing a mapping to the target “de.uni-hann.Person” type on the left.

5.4.2. Implementation

ProFlow is a framework for developing Eclipse plug-ins that employ a graphical editor for the creation of graph models. It was developed at the Software Engineering group of Leibniz Universität Hannover and was used as the basis for the GUI of the Composr application.

To graphically represent the model elements on its canvas, ProFlow requires a model class and a view class for each element. These were implemented for several of the elements present in the UML profile introduced in chapter 4. This resulted in an Eclipse plug-in that is able to create diagrams using the defined notation.

In order to implement the dialogs presented in the preceding chapter, it was planned to use the JFace framework included in Eclipse. As has been mentioned before, this was abandoned.

6. A real-world Example

To assess the value of the approach presented in this thesis, this chapter presents a process implemented during the student project “Entwicklung einer Webservice-basierten Anwendung” (“Development of a Web Service-based Application”) at the Leibniz Universität Hannover during the winter term of 2006.

The process and the implementation by the students as well as the same process modeled using the Composr notation will be introduced in the following section. Later sections define metrics on aspects of the development of service compositions and, based on the example process, are being evaluated in context of this thesis.

6.1. The Thesis Process

The implemented process was a real-world process used at the Software Engineering Group of Leibniz Universität Hannover. It describes the activities necessary to write a thesis at said group. Below is a textual description of the process.

1. A tutor proposes a subject for a thesis.
2. The examiner, usually the professor, either accepts or rejects the subject.
 - a) If the subject is rejected, the tutor can make adjustments to the thesis and have it reviewed by the examiner again.
3. Once the subject is approved, it is published, so students can apply for it. Also, a second examiner is chosen.
4. Once a student and a second examiner have been found, the thesis is registered with the university to officially start it.
5. A thesis can be written externally, i.e., not at the university, but at another organization.
 - a) If the thesis is external, the introductory presentation is planned.
6. Then, the intermediate and final presentations are planned.
7. Once the thesis has been delivered, both examiners give it a grade.

6.1.1. Student Implementation

The process implemented by the students consisted of a collection of BPEL processes and several Web Services. The BPEL processes were organized hierarchically with a main process that would call sub-processes. Also, a client application to access the main BPEL process was created. For the BPEL development, the ActiveBPEL Designer by ActiveEndpoints was used, which has been mentioned before. The Web Services were implemented in Java using Hibernate¹ for persistence and used Apache Axis² as their Web Services engine.

Tables 6.1 and 6.2 present data retrieved from the student project, showing the numbers for lines of code, classes, and methods for several software components involved — all concerning the Java code created during the project. The data was gathered using the Metrics plug-in for Eclipse³.

Table 6.1.: Data collected for the Student Project (Lines of Code)

| SW Component | Lines of Code | | | |
|----------------|---------------|--------|-------|-----------|
| | generated | manual | total | generated |
| Persistence WS | 1221 | 394 | 1615 | 75.60% |
| Print WS | 0 | 314 | 314 | 0.00% |
| Mail WS | 0 | 309 | 309 | 0.00% |
| Auth. WS | 0 | 434 | 434 | 0.00% |
| Client | 5544 | 11496 | 17040 | 32.54% |
| Total | 6765 | 12947 | 19712 | 21.63% |

Table 6.2.: Data collected for the Student Project (Classes and Methods)

| SW Component | Classes | Methods | |
|----------------|---------|---------|-----------|
| | total | total | per class |
| Persistence WS | 7 | 147 | 21.00 |
| Print WS | 3 | 7 | 2.33 |
| Mail WS | 4 | 38 | 9.50 |
| Auth. WS | 8 | 15 | 1.88 |
| Client | 222 | 1592 | 28.78 |
| Total | 244 | 1799 | 12.70 |

For illustration and comparison, figure 6.1 shows the main process developed during the student project in ActiveBPEL Designer. Since the unscaled image takes about 3500 x 8800 pixels of space, the figure depicts a scaled-down version of the complete

¹<http://hibernate.org>

²<http://ws.apache.org/axis/>

³<http://metrics.sourceforge.net/>

6. A real-world Example

main process on the left and an unscaled portion of the process to the right. In the middle, a part of the process is presented using a medium scale. The arrows indicate the approximate place the area right of each arrow is situated in the respective larger-scale depiction. The sub-processes used by this process are not shown.

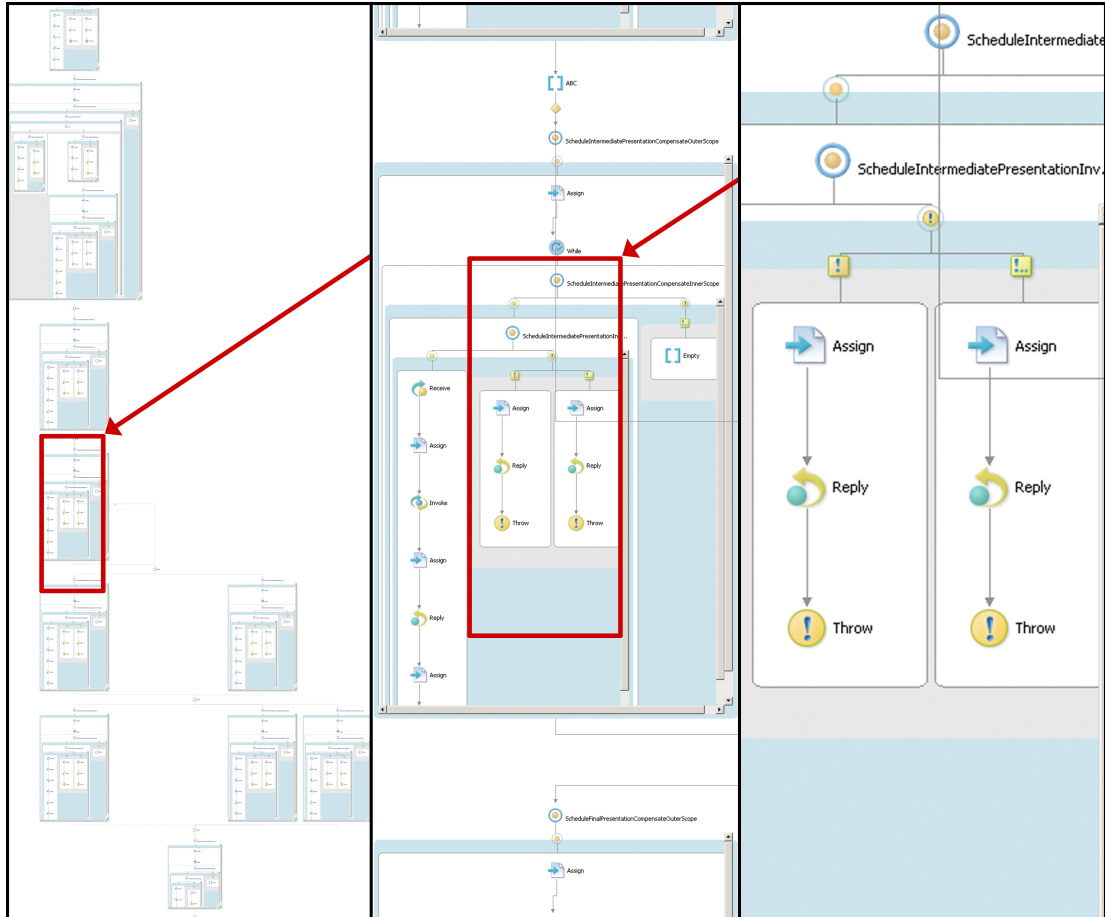


Figure 6.1.: The BPEL process developed in the student project in three different scales.

6.1.2. Composr Model

Figure 6.2 presents the same process modeled using the Composr notation, also scaled down a bit. Viewed on a computer screen in a scale appropriate for editing, it would occupy about 300 x 1100 pixels.

6. A real-world Example

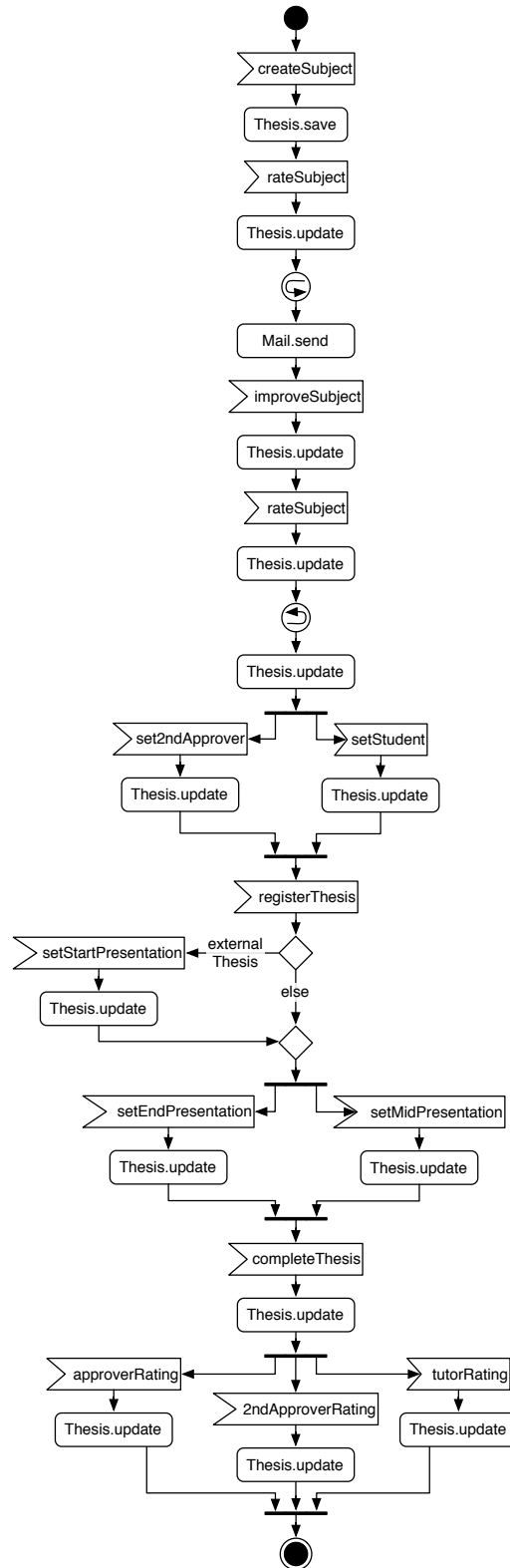


Figure 6.2.: The thesis process modeled using the Compoor notation.

6.2. Metrics

A comparison between the generated BPEL by Composr and the BPEL manually created in the BPEL Designer would not add much value to this discussion, since, as has been seen in chapter 5, the BPEL generated by Composr is still incomplete. Therefore, the metrics to be defined in this section will concentrate on aspects concerning the usage of the tools, assuming both were complete.

Number of Elements (NoE) The NoE in a business process model is hereby being defined as the number of activities observable in the diagram, explicitly excluding links between activities. For semantically equal process models, a lower NoE is considered beneficial.

The NoE for the process shown in the BPEL Designer is about 290. The NoE for the process shown in the Composr notation is about 40.

Size of Diagram (SoD) The number of million pixels needed to display a process diagram on a 72 dpi screen will be called SoD. The scaling of the diagram must be appropriate for viewing and editing. For the size of the Composr diagram, a scale has been chosen that lets the size of a single activity be about the same size as an activity in a diagram displayed using the ActiveBPEL Designer. For semantically equal process diagrams, a lower SoD is considered beneficial.

The SoD for the process shown in the BPEL designer is about 30.8. The SoD for the process shown in the Composr notation is about 0.33.

Number of Artifacts to be maintained (NoAm) The NoAm for an executable business process is hereby being defined as the number of files that need to be changed to maintain the executable process. For semantically equal process models, a lower NoAm is considered favorable.

The NoAm for the process shown in the BPEL designer is 37 (13 BPEL files, 6 WSDL files, 4 XML Schema files, 13 deployment descriptors, 1 build file). The NoAm for the process shown in the Composr notation is 1 (the Composr process file).

Complexity (CFC^{BPEL} , CFC^{CPSR}) In [24], a metric for the complexity of BPEL processes is presented, the CFC metric — measuring the control flow complexity. Since the Composr notation contains basically the same activities as BPEL, it might be insightful to apply the CFC metric originally targeted at BPEL to a process modeled using Composr. Therefore, $CFC^{BPEL} := CFC^{CPSR}$ is defined. Of course, the metric was intended for the comparison of BPEL processes, so this should only be taken as a supportive indicator in conjunction with others, not as a crucial one. For semantically equal process models, a lower CFC is considered beneficial.

6. A real-world Example

The process modeled in Composr has a CFC^{CPSR} of 66. Since the process modeled in the ActiveBPEL Designer has at least 126 basic activities alone, it is safe to say that it has a $CFC^{BPEL} \geq 126$.

Expressiveness The expressiveness of a tool for BPEL development is hereby being defined as the percentage of 65 distinct BPEL elements as defined in [8] that can be influenced by the tool. A higher expressiveness is considered advantageous.

This is a complete listing of all 65 elements considered when calculating expressiveness: *assign, branches, catch, catchAll, compensate, compensateScope, compensationHandler, completionCondition, condition, copy, correlation, correlations, correlationSet, correlationSets, else, elseif, empty, eventHandlers, exit, extension, extension-Activity, extensionAssignOperation, extensions, faultHandlers, finalCounterValue, flow, for, forEach, fromPart, fromParts, if, import, invoke, joinCondition, link, links, messageExchange, messageExchanges, onAlarm, onEvent, onMessage, partnerLink, partnerLinks, pick, process, receive, repeatEvery, repeatUntil, reply, rethrow, scope, sequence, source, sources, startCounterValue, target, targets, terminationHandler, throw, toPart, toParts, transitionCondition, until, validate, variable, variables, wait, while.*

Composr supports influence on 36 BPEL elements, which equals 55 % of all considered elements. Even though not all of them can be edited or configured directly, it is still possible to influence them. E.g., a user of Composr will never see an `<assign>` element, but still the choices made for the data sources and data targets for the `<receive>`, `<reply>` and `<invoke>` activities still alter the quantity and quality of `<assign>` elements found in the generated BPEL source. The ActiveBPEL Designer, as it provides direct access to the source code and through its GUI basically provides direct graphical representations and configurable options for the elements, can influence 100% of all considered elements.

Cost In the following section, cost is used as a relative amount of resources — e.g., money, time, employees, knowledge — necessary to accomplish a goal.

6.3. Comparison

This section uses the metrics defined in the preceding section for a comparison of the Composr notation with the notation of ActiveBPEL Designer. It discusses several aspects by first showing why a specific aspect is relevant in the context of the development of service compositions and then comparing what is achieved by both tools regarding this aspect.

A central aspect of service-oriented architecture is the flexibility aimed for. To be able to speak about flexibility, a definition of the term is required. This thesis uses the definition of [25]:

6. A real-world Example

The ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed.

I.e., a key advantage of a SOA is that it can easily be adapted for uses and scenarios not envisioned at its inception. This observation will be important in the following paragraphs.

Maintainability The IEEE defines maintainability [25] as follows:

The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.

As can be seen, the definition of maintainability is relatively similar to that of flexibility, as they both consider the ease of adaption to changed environments. Therefore, maintainability is a crucial quality for executable business processes.

Metrics relevant for this aspect are the NoE, the SoD, and the NoAm of an executable business process. For each metric, a lower value translates to better maintainability, because

- a lower number of elements in a process means there are fewer potential sources for defects,
- a clearer diagram improves navigability of the process and therefore the quality of orientation in it, and,
- a lower number of artifacts to be maintained means there are fewer chances for side-effects.

As has been shown in the previous section, the values for the ActiveBPEL Designer are consistently and significantly higher than those for the Composr tool. If these assumptions hold true, business processes created using the Composr tool exhibit a much better level of maintainability than those developed using the ActiveBPEL Designer.

As can be seen from tables 6.1 and 6.2, this advantage would be even higher if generated service had been taken into account, since the portion of manual code would be 0 % for the Composr tool.

Complexity The IEEE defines complexity [25] as follows:

The degree to which a system or component has a design or implementation that is difficult to understand and verify.

6. A real-world Example

To modify a system or component safely, it is necessary to understand it. Thus, maintainability, and therefore the flexibility advantage of a SOA, is directly dependent on the complexity of its executable business processes.

While both notations represent the same business process, but use different abstractions, the semantics of the process models are the same in a business sense, but their complexities differ. As has been shown in the preceding section, the $CF C^{BPEL}$ of the process modeled in the ActiveBPEL Designer is significantly higher than the $CF C^{PSR}$ of the process modeled using the Composr notation. Also, the sheer NoE as defined before can be taken as an indicator for complexity, which supports the preceding observation. Therefore, because the Composr notation lowers complexity, it has an advantage over the ActiveBPEL Designer notation when supporting the flexibility essential to SOA.

Knowledge required If less knowledge is required to use a software tool, it can be used in more scenarios, since more roles are able to operate it. This can be seen as an advantage in flexibility and maintainability, since changing the process becomes less expensive and, therefore, easier to accomplish for an organization.

The ActiveBPEL Designer's notation is a direct mapping to the underlying BPEL. To understand its constructs, the user must have detailed knowledge of BPEL. The notation used by Composr, however, provides more abstractions, requiring the user only to have a basic understanding of control flow and the receive, reply, and invoke concepts primarily present in the Web Services context. Therefore, the Composr notation supports flexible SOAs, as it demands less knowledge from the user.

Expressiveness The improved flexibility of the Composr notation observed in the preceding paragraphs comes at the cost of expressiveness. According to the definition of expressiveness from the preceding section, ActiveBPEL Designer with an expressiveness of 100 % is more expressive than Composr, which has an expressiveness of 55 %.

Higher abstractions always result in fewer ways possible available for expression. With features like faultHandlers, eventHandlers and correlationSets not included, Composr's scope of application is different from that of a complete BPEL editor.

Nevertheless, the usage of Composr is appropriate in several scenarios in which a higher level of abstraction is more important than function completeness. E.g., organizations initiating SOA development might especially profit from its use, as the provided abstractions lower the entry-barrier and the generated artifacts can still be modified using full-featured tools like the ActiveBPEL Designer. This would enable users to first generate a high-level process that is already functional, and to later add handling of exceptional events and requirements. Therefore, users might profit from the higher abstractions in the beginning, while still being able to leverage more powerful tools once the need arises.

Table 6.3 provides a summary of the values found for each tool regarding the metrics

6. A real-world Example

defined before. In combination with the aspects compared in this section, it shows that the usage of higher abstractions as they are provided by Composr has significant advantages.

Table 6.3.: Summary of the values for each tool

| Metric | ActiveBPEL Designer | Composr |
|--------------------------------------|----------------------------|----------------|
| Number of Elements | ~ 290 | ~ 40 |
| Size of Diagram | 30.8 | 0.33 |
| Number of Artifacts to be maintained | 37 | 1 |
| Complexity | ≥ 126 | 66 |
| Expressiveness | 100 % | 55 % |

7. Related Work

This chapter discusses works found to be related to this thesis' approach, shows similarities and draws distinctions to the scope of the thesis. The first section examines work related to the model-driven development of executable processes, while the second section mentions products that are already available.

7.1. Model-Driven Development of executable Processes

This section identifies work concerned with the generation of executable processes from a given model.

(1) Gardner [22], and later Ambühler [23], each proposed UML profiles based on Activity Diagrams with direct mappings to BPEL, thus allowing the creation of UML models that could directly be translated into an executable process. Ambühler provided a partial implementation of the UML profile in the form of an Eclipse plug-in targeted for use with the Rational Software Modeler, which is a UML 2.0 modeling tool built on top of Eclipse.

Both approaches lack any kind of abstraction as it can be found in this thesis; models would always be a direct representation of the underlying BPEL constructs. As the notation used by Ambühler stayed very close to the Activity Diagrams notation, several elements lacked obvious distinctive features — e.g., the representations of the `receive` and `reply` elements are both represented like a regular UML Activity with only a small decorating arrow indicating the direction of data flow. Also, archives for deployment or the generation of services were out of scope.

(2) Decker et al. [26] proposed an extension layer to BPEL which allows the generation of abstract BPEL processes from models of choreographies while decoupling behaviour from actual endpoints. This research has spawned a tool¹ that offers graphical modeling of choreographies using BPMN and can export to abstract BPEL processes.

While similar in approach, on business process modeling level, the addressed issues are different from those addressed in this thesis. While the latter is concerned with service orchestrations, the concept of Decker et al. provides a solution for service choreographies. As choreographies are formal contracts used to govern orchestrations, both approaches rather complement each other.

¹<http://bpel4chor.org/editor/>

7. Related Work

(3) Based on the work presented by Ouyang et al. in [27], Giner et al. have created a tool [28] suitable for creating BPMN models for processes and then translating the models to BPEL.

As Ouyang et al. point out, transformations from BPMN to BPEL are impossible to cover all of BPMN, since some constructs available in BPMN cannot be mapped to BPEL. Therefore, the actual elements used in the tool created by Giner et al. must always be a subset of BPMN, possibly leading to confusion in users as to which constructs are allowed and which are not. This thesis assumes that, to avoid such ambiguities, it is best to provide a notation that allows all of its elements to be used, which is achieved with the Composr notation. Also, neither deployment nor service generation were an aspect of either work.

(4) In [29], Talib et al. propose a different approach to BPEL generation. Instead of a graphical notation, they propose a set of consecutive dialogs requesting the necessary information from the user to model the process internally.

This thesis assumes that for successful and efficient business process modeling, a graphical overview is essential for orientation. The dialog-based approach by Talib et al. is therefore assumed to significantly lack in these aspects. Also, there is no generation of services and several important technical details are left to the user, e.g., data transformations and namespaces.

(5) The generation of BPEL processes from EPC models has been described by Ziemann & Mendling et al. [30], and from Nautilus EPC to BPEL by Kopp et al. These approaches allow the development organisation to translate the control flow of the business processes to the service compositions. However, definitions of services and data is missing and have to be added manually to the generated compositions.

In order to further refine these approaches, Schmelzle [31] proposed the annotation of business processes with the required technical details: Service definitions and the data types of business objects are added in a pure development view. A generator can create a fully functional service composition. The developers then need to fine-tune it and add error handling.

These approaches use a simplistic notation (event-driven process chains) combined with the annotation of the graphical elements with technical details, possibly by another role. This is very similar to the approach presented in this thesis. Nevertheless, deployment and service generation have not been included, which limits the potential for integration of tasks into a single workflow.

(6) The approach pursued by Lübke [1] also builds upon the aforementioned strategies using EPCs, but extends it by also covering the transformation of use cases to EPCs and then integrating extended support for tests and the generation of graphical client applications for generated processes.

While there are intersecting elements between Lübke's approach and this thesis, the former is much broader in scope. Some of these aspects are being addressed in section 8.2. Nevertheless, the approach misses some concepts introduced in this thesis, namely the generation of services used by the modeled process.

7.2. Available Products

This section presents existing products pursuing similar or related approaches to what is being presented in this thesis.

(1) NetBeans² is an IDE by Sun Microsystems primarily targeted at Java development, but also supporting many other technologies using a plug-in system. One of those supported technologies is BPEL, which can be created using a BPEL editor that provides graphical elements for the constructs found in BPEL.

While this approach has been criticized in this thesis for its lack of abstraction from technical details as provided by Composr, the NetBeans BPEL editor includes the so-called BPEL Mapper, improving the configuration of `assign` elements. Its approach is quite similar to the data mappings found in this thesis, as can be seen in the screenshot shown in figure 7.1.

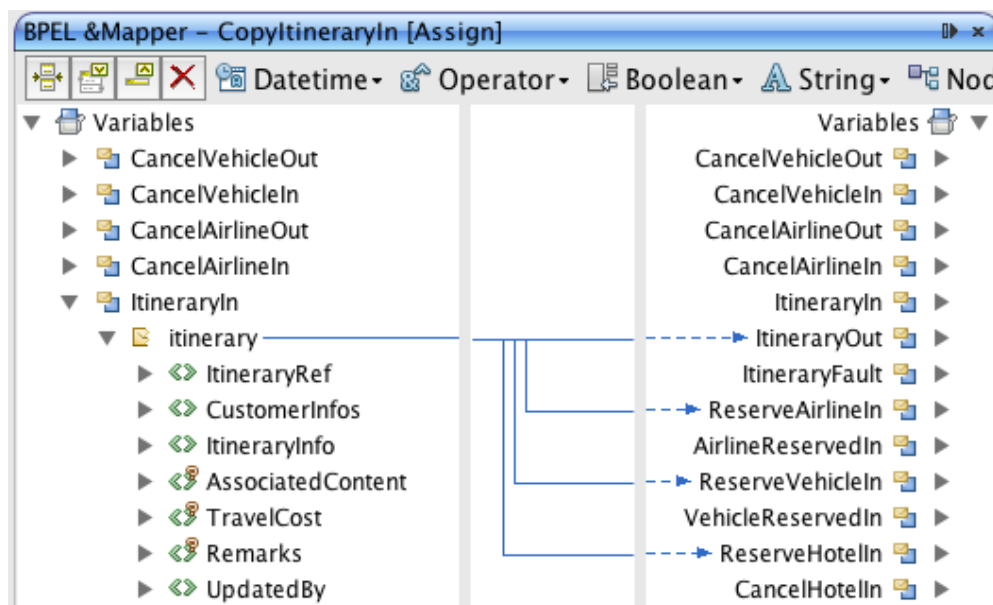


Figure 7.1.: A screenshot showing the BPEL Mapper of the NetBeans IDE.

²<http://netbeans.org/>

7. Related Work

(2) objectiF³ by microTOOL GmbH is a tool covering the whole process of model-driven development. It uses BPMN for process models and UML for all other models. All aspects of an application can be modeled in an integrated environment, including the data and presentation layers. Services are being implemented using EJB 3 SessionBeans, while business processes use WS-BPEL. By reflecting changes in code in the model and the other way around, the application supports round-trip engineering.

While being significantly larger in scope and targeted at a different, more technical audience, the concept of generating WS-BPEL from a process model found in objectiF is also covered by Composr. Despite its solid support for the creation of Web Services, objectiF does not provide predefined services in a way similar to Composr. But as objectiF provides means for users to specify their own transformations from models to code, the commodity service concept could probably be introduced by users themselves.

(3) MagicDraw⁴ is a graphical UML editor by No Magic, Inc., featuring the ability to export BPMN diagrams to BPEL4WS (BPEL 1.1).

While a mapping from BPMN to BPEL has been shown to be problematic before [27], the generated BPEL is restricted for deployment inside a BEA WebLogic Server⁵.

(4) Intalio BPMS⁶ is a tool suite targeted at business process management, based on Eclipse. It provides a designer application, used to create BPMN models, and a server runtime, used to run the WS-BPEL processes generated from the BPMN models.

The designer application shares Composr's concept of supplying different views for different roles. While the business process designer will most likely only see the process diagram, a developer can switch to a technical view to complete the process model, e.g. by providing data transformations. For this, the Intalio Designer provides a data mapping GUI similar to that introduced in this thesis, as well as the BPEL Mapper found in the NetBeans IDE which was mentioned at the beginning of this section. Figure 7.2 shows a screenshot of Intalio Designer containing the business process modeling canvas in the upper half and the data mapper in the lower half.

Similar to the objectiF application presented before, the Intalio Designer also support round-trip engineering, something that Composr is not aimed at.

Additionally, the Intalio BPMS enables human tasks in a business process by supporting BPEL4People on the process side and using XForms[32] for the user interface side. For the creation of the latter, a forms editor is included. The actual services used by the modeled process are expected to exist or to be developed manually, though, as in the Intalio BPMS, there is nothing similar to the commodity service concept presented in this thesis.

³<http://microtool.de/objectif/>

⁴<http://magicdraw.com/>

⁵<http://bea.com/weblogic/>

⁶<http://intalio.com/>

7. Related Work

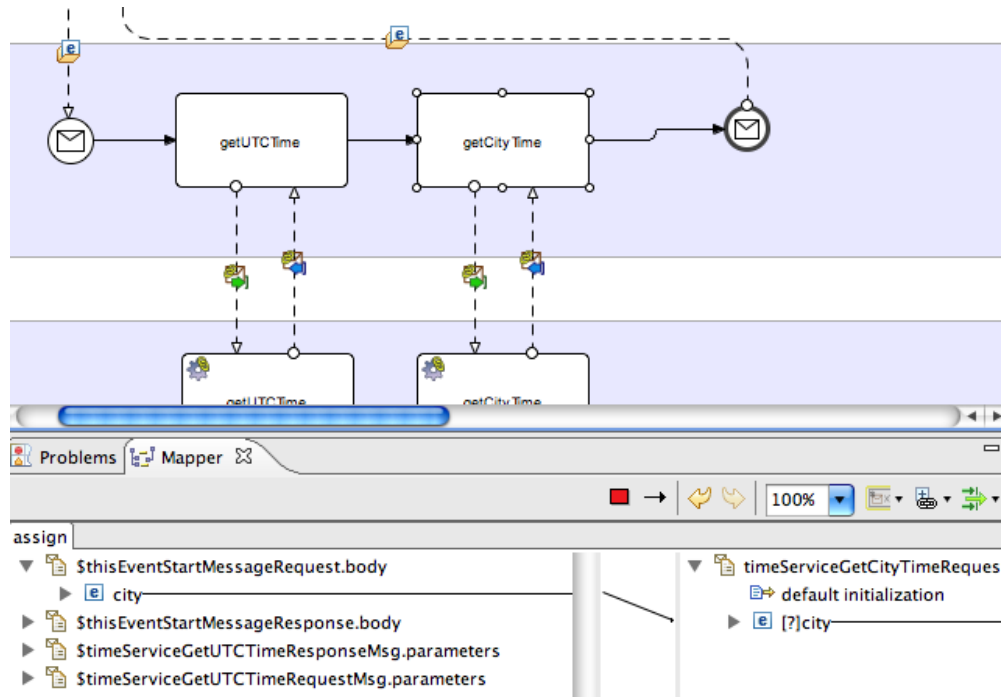


Figure 7.2.: A screenshot showing the process modeling canvas and the data mapper of the Intalio Designer.

(5) The WSO₂ Web Services Application Server⁷ (WSAS) is an application server that federates several Web Services-related technologies into a single middleware server, primarily targeted at the exposure of functionality as Web Services.

WSAS includes the ability to create new data services using its web frontend. This approach has some similarities to the PersistenceService presented in this thesis and acknowledges the need for easy exposure of data as services. Nevertheless, there are significant differences in the approaches, as WSAS data services require the user to manually enter an SQL query for each operation to be exposed, while the PersistenceService will always provide a predefined set of operations. This is a trade-off between the level of abstraction achievable and the desired versatility of the solution. The addition of such an option to Composr might be desirable for scenarios in which more diverse views on the data are required than can be provided by the PersistenceService.

⁷<http://wso2.org/projects/wsas/java>

8. Conclusions and Outlook

This chapter first provides a critical appraisal of the thesis' contributions, while afterwards, an outlook is being given on possible future work that might relate to or stem from this thesis. The last section concludes the thesis with a summary of its accomplishments.

8.1. Critical Appraisal

The approach to the model-driven development of service compositions presented in this thesis provides several benefits that improve the conventional creation of executable processes and associated services significantly. This section presents some of the main benefits and discusses the execution of the thesis.

By providing a concept and the partial implementation of a software tool, this thesis introduces significant abstractions for the development of BPEL processes, lowering the amount of initial manual work required.

BPEL employs a “whiteboard” approach for reading and writing data used throughout the process, i.e., all data is stored in global variables accessible by every activity contained in the process. This may lead to side effects when multiple activities are writing to a variable, as this approach makes it hard for developers to think of all the places a variable is used in, when only introducing a small, seemingly local change. Although the visibility of variables can be restricted to some extent using the `scope` construct, the problem remains on a slightly smaller scale. The situation is worsened by BPEL requiring multiple `assign` statements — even for simple tasks — which are cumbersome to create.

Implementing the data flow graph approach to data management as an abstraction on top of BPEL, this thesis overcomes these problems by allowing process developers to create data mappings for each data sink, with the availability of any data that has been received so far in the process. This approach increases the orientation a developer can achieve within a process and removes the mentioned side effects altogether. By providing quick access to already used data mappings, repetitive tasks are reduced further.

The generator has been designed to be replaceable with new generation strategies implemented by interested parties, thereby allowing for the generation of artifacts not yet accounted for in the tool and providing a means for supporting different technologies. Independently from issues specific to a particular process language, the approach presented in this thesis simplifies the overall workflow required for the development of

8. Conclusions and Outlook

executable business processes.

Supporting the participation of multiple roles in process modeling, different target audiences are provided with different views upon the same process. This allows for the usage of the same process model in different areas of development: while a business analyst might initially create the process, it can be refined by a technical developer and finally be generated and deployed by an administrator. This approach reduces losses occurring during the transition of information into different media, since the number of such transitions is reduced drastically.

The simple notation helps comprehension by members of different roles by hiding technical details in the configuration of its elements. The abstractions it provides unite multiple technical concepts into a single one in the graphical view — e.g., the `Loop` element of the notation represents three concepts present in BPEL: the `while`, `repeatUntil`, and `forEach` elements.

By introducing the commodity service concept, the approach presented in this thesis relieves developers of the repetitive implementation of several similar services required for data persistence. By providing an interface common to all commodity services, interested parties may introduce their own service generator into the tool, thereby further increasing its flexibility.

The restricted expressiveness of the presented approach might render it inappropriate for complex SOA projects. Nevertheless, target audiences that require a fast introduction to process development or immediate results that may still be refined later on could benefit significantly from the abstractions developed in this thesis.

While the premise of the thesis — the exploration and evaluation of a concept for model-driven development of service compositions — can be regarded as being successful, the execution of the thesis was not optimal at all times.

Due to the time constraints imposed on a Master's thesis, both the concept and the development had to be restricted in scope. Therefore, several concepts available in BPEL were planned not to be supported by the approach, e.g. correlation sets and fault handlers.

Many technologies and concepts were required during the course of this thesis, as it incorporates and tries to unite two vast fields of ongoing research — model-driven development and service-oriented architecture. Because the effort required for acquiring the knowledge and experience needed to cover all aspects that were planned to be included was initially underestimated, the scope of the thesis had to be restricted further after about half the allocated time.

Namely, the priority for the user interface was lowered, while the priority for the generator was raised. Instead of using process models supplied by the user interface, the development of the generator used process models manually created in Java. As the same structures were passed as they would be provided by the Eclipse plug-in, this resulted in an appropriate decoupling of both aspects.

Still, the generator has not been developed as completely as was planned. Since the technology involved in the generation is partly still very immature, development

was significantly hindered not by conceptual, but by very low-level, technical problems. E.g., the only documentation available on the use of the BPEL object model from the Eclipse project was the source code itself, which was only sparsely commented.

Nevertheless, this thesis has achieved a proof of concept by showing a prototypical and partial implementation of the envisioned application, supported by a formally defined UML profile for the metamodel used therein and a notation that has been refined based on feedback from a survey.

8.2. Outlook

This section provides an outlook on further work that could be based on the results of this thesis. It is divided into two sections: the first describes ideas that use the tool as their basis, i.e., the graphical editor and the generator. The second one suggests ideas for exploring the commodity service concept.

8.2.1. Graphical Editor and Generator

To ensure only processes resulting in valid BPEL code can be passed on to the generator, static checks could be implemented, denying the generation of processes not consistent with the static constraints defined by [8]. Errors resulting from these checks could then be incorporated into the “Problems View” of the Eclipse IDE. This would also provide a list of missing configuration details, improving the overview the developer has over the process.

To visually map activities to partners participating in an executable business process, the concept of swimlanes as known from Activity Diagrams could be incorporated into the metamodel and the notation. This would perhaps allow for models that are even better to understand.

In order to extend the range of supported BPEL constructs, e.g., concepts for eventHandlers, faultHandlers, or correlationSets could be introduced into the abstraction. An implicit correlationSet generated into all processes could even be easily implemented by assigning a process an identification number and using this in all communication with clients. Of course this would alter the interface of the process defined by the user, so this should be an optional choice.

To allow for interaction with humans, support for BPEL4People could be integrated, possibly introducing a new abstraction or a new kind of commodity service. This could be combined with the addition of the generation of client applications either tailored to the specific process or the usage of generic clients, supporting access to a wide range of processes.

The addition of solutions for non-functional requirements such as security or transactions could add further value to the approach. This could be achieved by integrating the process with an aspect-based BPEL container as presented in [33].

8. Conclusions and Outlook

Due to its extendable architecture, the generator could be modified to export to other languages using specialized generation strategies. E.g., executable processes could be generated using YAWL¹ while service generation could be implemented using the .NET framework².

To add further support for common software development processes, the generation of test stubs or test skeletons could be introduced, e.g. using the BPELUnit³ testing framework for BPEL processes. Also, the generation of complete tests for the generated services could be feasible.

Similar to the concept of commodity services, predefined abstractions could be conceived and integrated into the tool, improving comfort even further. A simple example might be a construct similar to “for each” in Java, allowing the iteration over all elements of a collection. The usage of more complex modeling patterns could be explored to create a pattern language available in modeling, further abstracting the created model.

8.2.2. Commodity Services

This thesis includes one commodity service, the persistence service. On a technical level, there are already several opportunities for extension. The generation could be extended for Web Service engines other than Apache Axis2 to support a richer set of target engines. The annotations provided by JSR 181 [34] — e.g., `@WebMethod`, `@WebService`, `@SOAPBinding` — could be leveraged, leading to a leaner, more readable and thus more maintainable service implementation.

But stepping back, services can generally be understood as data sinks and sources. From this view, several other possible commodity services that could be of much use when developing business processes come to mind. Following is a small selection of some such ideas.

REST Services

What could be considered as a backlash reaction to the heavy-weight Web Services standards, more light-weight approaches are gaining traction fast amongst developers. The prime example for this movement is the increasing usage of the REST (Representational State Transfer, [35]) architectural style for developing weakly-typed, non-self-describing services commonly using HTTP for transport and method invocation.

The approach presented in this thesis would benefit from support for these light-weight alternatives, as process modelers would instantly gain access to many services already in existence. A new commodity service could be provided, wrapping “RESTful” services to make them accessible to the generated BPEL process. In contrast to Web Services that use WSDL, these services are generally not self-describing. Thus, the

¹<http://www.yawl-system.com/>

²<http://msdn.microsoft.com/netframework/>

³<http://bpelunit.org/>

user would need to specify the interface of the service manually. A method to describe such light-weight services has been proposed [36] and the HTTP binding of WSDL 2.0 [37] has brought some improvements in this area [38], but neither option has yet entered general use.

Feeds

Feeds, best known for their news feeds incarnation, commonly use the RSS [39] or Atom [40] formats. Using only a URL as a parameter given by the modeler, a commodity service could be generated, providing access to the items in the specified feed. The development of this commodity service should be trivial, as the formats clearly specify the defined elements and attributes. The BPEL process could thus use items and their properties.

While this would not appear very useful when being limited to news feeds, it must be noted that the mentioned formats do not have to be used for news — the simplicity and the general nature of the vocabulary used (e.g. “entry” and “item”) makes it easy to adopt the formats for new uses. E.g., a service registry supporting the Atom Publishing Protocol [41] has been developed [42] in the context of light-weight approaches to services.

8.3. Conclusions

This thesis has introduced several important concepts supporting a model-driven approach to the development of service compositions. This section concludes the thesis by summarizing it and highlighting its central achievements.

The first chapter gave an introduction to the central problems relevant to this thesis and provided an outline of how these problems were to be addressed. To provide the technical background necessary for understanding the presented solution, chapter 2 gave an overview of the central concepts employed in the following chapters.

The approach pursued by this thesis was presented in chapter 3, describing the intended workflow and supporting this by showing a possible scenario for the application of the thesis’ results. Based on the requirements implied by this workflow, a metamodel for executable business processes was developed as a UML profile in chapter 4. A notation for the profile was drafted and eventually finalized based on the results of a survey conducted among students.

Facilitating the workflow concept and the associated metamodel created before, chapter 5 showed how the development of a prototypical implementation of the envisioned tool was accomplished. It outlined the decisions made for the design and the utilized technologies, explained the encountered challenges and the strategies employed to overcome them.

An assessment of the value of the approach developed in this thesis was provided

8. Conclusions and Outlook

by chapter 6. Based on a real-world example and the definition of some metrics, the thesis' concept was contrasted with an existing application for the modeling of BPEL processes, thereby providing the basic indicators for the subsequent evaluation of the concept's fitness for SOA development.

Chapter 7 gave a presentation of some of the approaches, works, and products related to this thesis' concept, providing an overview for each as well as a discussion of similarities and differences. It is followed by this chapter, which first gave a critical view on the thesis' accomplishments, then outlined possible future work that could be based upon its achievements, and finally concludes the thesis.

As has been shown in chapter 7, the approach developed in this thesis distinguishes itself from existing works primarily by proposing an integrated development process for service compositions that includes the generation of commonly needed services. Therefore, this thesis can serve as a basis for further research on integrating the diverse tasks occurring in SOA development into a single workflow.

Bibliography

- [1] Daniel Lübke. *An Integrated Approach for Generation in Service-Oriented Architecture Projects*. PhD thesis, Gottfried Wilhelm Leibniz Universität Hannover, Sep 2007.
- [2] W M P van der Aalst, A H M ter Hofstede, B Kiepuszewski, and A P Barros. *Workflow patterns*, 7 2003.
- [3] The Web Services-Interoperability Organization. *Basic Profile Version 2.0*, 10 2007.
- [4] Anne Thomas Manes. The “wrapped” document/literal convention. <http://atmanes.blogspot.com/2005/03/wrapped-documentliteral-convention.html>; last access: 2007-12-11, 2005.
- [5] David C. Fallside and Priscilla Walmsley. *XML Schema Part 0: Primer Second Edition*. <http://www.w3.org/TR/xmlschema-0/>; last access: 2007-12-12, 2004.
- [6] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) 1.1*. W3C, 3 2001.
- [7] BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems. *Business Process Execution Language for Web Services version 1.1*, 5 2003.
- [8] OASIS. *Web Services Business Process Execution Language Version 2.0*, 4 2007.
- [9] Thomas Stahl and Markus Völter. *Modellgetriebene Softwareentwicklung*. dpunkt.verlag GmbH, 2005.
- [10] Sun Microsystems, Inc. *JavaTM Platform, Enterprise Edition 5 (Java EE 5) Specification*, 5 2006.
- [11] David Heinemeier Hansson et al. *Ruby on RailsTM 2.0*. <http://www.rubyonrails.org/>; last access: 2007-12-12, 6 2007.
- [12] Sun Microsystems, Inc. *Enterprise JavaBeansTM 3.0*, 11 2007.
- [13] OMG Model Driven Architecture. <http://www.omg.org/mda/>; last access: 2007-12-12, 11 2007.
- [14] Object Management Group. *Unified Modeling Language: Superstructure*, 2.1.1 edition, 2 2007.

Bibliography

- [15] Object Management Group, Inc. *Meta Object Facility (MOF) Core Specification 2.0*, 1 2006.
- [16] Object Management Group, Inc. *Object Constraint Language 2.0*, 5 2006.
- [17] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, 7 2007.
- [18] Object Management Group, Inc. *MOF 2.0/XMI Mapping, Version 2.1.1*, 12 2007.
- [19] *Unified Modeling Language: Superstructure; Activities*, pages 295–417. Object Management Group, 2.1.1 edition, 2 2007.
- [20] Object Management Group, Inc. *Business Process Modeling Notation (BPMN) Specification*, 2 2006.
- [21] Chun Ouyang, Marlon Dumas, Stephan Breutel, and Arthur ter Hofstede. Translating Standard Process Models to BPEL. In *Advanced Information Systems Engineering*, Lecture Notes in Computer Science, pages 417–432, 2006.
- [22] Tracy Gardner. UML Modelling of Automated Business Processes with a Mapping to BPEL4WS. In *First European Workshop on Object Orientation and Web Service (EOOWS)*, 2003.
- [23] Thomas Ambühler. UML 2.0 Profile for WS-BPEL with Mapping to WS-BPEL. Master's thesis, Institut für Architektur von Anwendungssystemen, Universität Stuttgart, 2005.
- [24] Jorge Cardoso. Complexity analysis of BPEL Web processes. *Software Process: Improvement and Practice*, 12(1):35–49, 2007.
- [25] The Institute of Electrical and Electronics Engineers. *IEEE Standard Glossary of Software Engineering Terminology*, 2 2006.
- [26] Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. Bpel4chor: Extending bpel for modeling choreographies. In *ICWS 2007*, pages 296–303. IEEE Computer Society, 2007.
- [27] C Ouyang, W van der Aalst, M Dumas, and et al. Translating BPMN to BPEL. In *BPM Center Report BPM-06-02*. BPMcenter.org, 1 2006.
- [28] P Giner, V Torres, and V Pelechano. Bridging the Gap between BPMN and WS-BPEL. M2M Transformations in Practice. In *Proceedings of the 3rd International Workshop on Model-Driven Web Engineering*, 2007.
- [29] Muhammad Adeel Talib, Zongkai Yang, and Qazi Mudassir Ilyas. A framework towards web services composition modeling and execution. In *BSN '05: Proceedings of the IEEE EEE05 international workshop on Business services networks*, 2005.

Bibliography

- [30] Oliver Kopp, Tobias Unger, and Frank Leymann. Nautilus Event-driven Process Chains: Syntax, Semantics, and their mapping to BPEL. In Markus Nüttgens, Frank J. Rump, and Jan Mendling, editors, *EPK 2006 – Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten*, pages 85–104. Gesellschaft für Informatik e.V., Arbeitskreis Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (WI-EPK), 2006.
- [31] Oleg Schmelzle. Transformation von annotierten Geschäftsprozessen nach BPEL. Master's thesis, Gottfried Wilhelm Leibniz Universität Hannover, May 2007.
- [32] John M. Boyer. XForms 1.0 (Third Edition). <http://www.w3.org/TR/xforms/>; last access: 2007-12-17, 2007.
- [33] Anis Charfi and Mira Mezini. An aspect-based process container for bpel. In *AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development*, 2005.
- [34] Java Community Process. *JSR-181: Web Services Metadata for the Java™ Platform*, 2005.
- [35] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
- [36] Marc J. Hadley. Web Application Description Language (WADL). Technical Report TR-2006-153, Sun Microsystems Laboratories, 2006.
- [37] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0. <http://www.w3.org/TR/wsdl20/>; last access: 2007-12-12, 2007.
- [38] Eran Chinthaka. Enable REST with Web Services, Part 1: REST and Web Services in WSDL 2.0. <http://www.ibm.com/developerworks/webservices/library/ws-rest1/>; last access: 2007-12-12, 2007.
- [39] RSS Advisory Board. *RSS 2.0 Specification*, 6 2007.
- [40] M. Nottingham and R. Sayre. The Atom Syndication Format. <http://tools.ietf.org/html/rfc4287>; last access: 2007-12-12, 2005.
- [41] J. Gregorio and B. de hOra. The Atom Publishing Protocol. <http://tools.ietf.org/html/rfc5023>; last access: 2007-12-12, 2007.
- [42] Paul Fremantle. A new kind of (SOA) Registry. <http://pzf.fremantle.org/2007/12/new-kind-of-soa-registry.html>; last access: 2007-12-12, 2007.

A. Survey

Thank you very much for taking the time to take part in this survey. Answering the questions will take about 10 minutes. Your participation will help me in improving the quality of my Master's thesis.

The aim of this survey is the comparison of two notations for modeling business processes.

A business process is a directed graph whose nodes are activities. Possible activities might be

- the reception of a request from a client by the process ("Take this order!")
- the calling of a service ("Please get me these ordered books!")
- the return of an answer to a client ("Received your order!")

These activities can be structured, e.g., by

- conditional branching into alternative activities,
- the repetition of activities,
- parallel branching into activities to be executed at the same time.

A single process can be running for a long time (order received, order products, commission shipper, mail invoice, check receipt of money) and serve different clients (customers, suppliers, logistics, accounting).

Potential users of the notation are technical as well as non-technical users. Therefore, it is okay if you do not understand some of the terms — to the contrary, that would increase the variance of the data available to me.

Part 1

Question 1.1 — Do you know the Activity Diagrams of the UML?

Possible answers: yes / no

Question 1.2 — If yes, do you use Activity Diagrams?

Possible answers: often / from time to time / rarely / tried it once / never

Question 1.3 — Do you know BPEL, i.e., would you understand its XML syntax?

Possible answers: yes / somewhat / no

Part 2

In figure A.1, you see an example process, modeled using notation A.

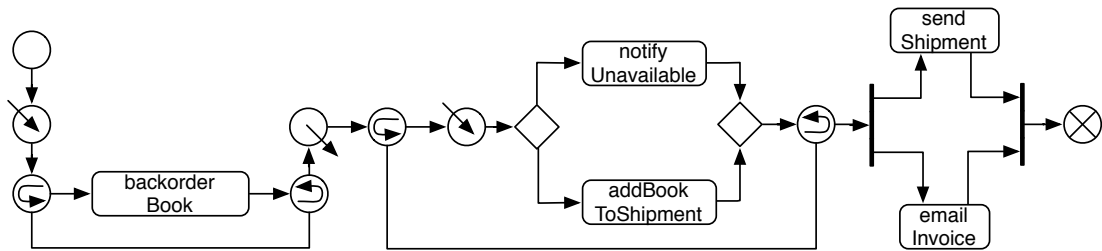


Figure A.1.: An example process modeled using notation A.

Please try answering the following questions as good as possible. As I also try to understand how intuitively comprehensible the notations are, I will not give any more clues regarding the process or the notation. You can informally describe activities lacking a distinct name.

Question 2.1 — Which activities are being executed in parallel?

Possible answers: free text entry.

Correct answer: send Shipment, email Invoice.

Question 2.2 — Which activities are being executed if a book could not be backordered?

Possible answers: free text entry.

Correct answer: notify Unavailable.

Question 2.3 — What happens to a book that could successfully be backordered?

Possible answers: free text entry.

Correct answer: it is being added to the shipment.

A. Survey

Question 2.4 — Which activities are being executed multiple times?

Possible answers: free text entry.

Correct answer: backorder Book and the one including notifyUnavailable and addBookToShipment.

Question 2.5 — How many different requests does the process accept?

Possible answers: free text entry.

Correct answer: 2.

Question 2.6 — How many different responses does the process return?

Possible answers: free text entry.

Correct answer: 1.

Part 3

In figure A.2, you see another example process, this time modeled using notation B.

Please try answering the following questions as good as possible. Again, I will not give any more clues regarding the process or the notation. Again, you can informally describe activities lacking a distinct name.

Question 3.1 — Which activities are being executed in parallel?

Possible answers: free text entry.

Correct answer: emailManager, emailRequester.

Question 3.2 — Which activities are being executed if no date can be negotiated?

Possible answers: free text entry.

Correct answer: returnRejection, emailRejection.

Question 3.3 — Which activities are being executed multiple times?

Possible answers: free text entry.

Correct answer: requestAlternativeDate.

A. Survey

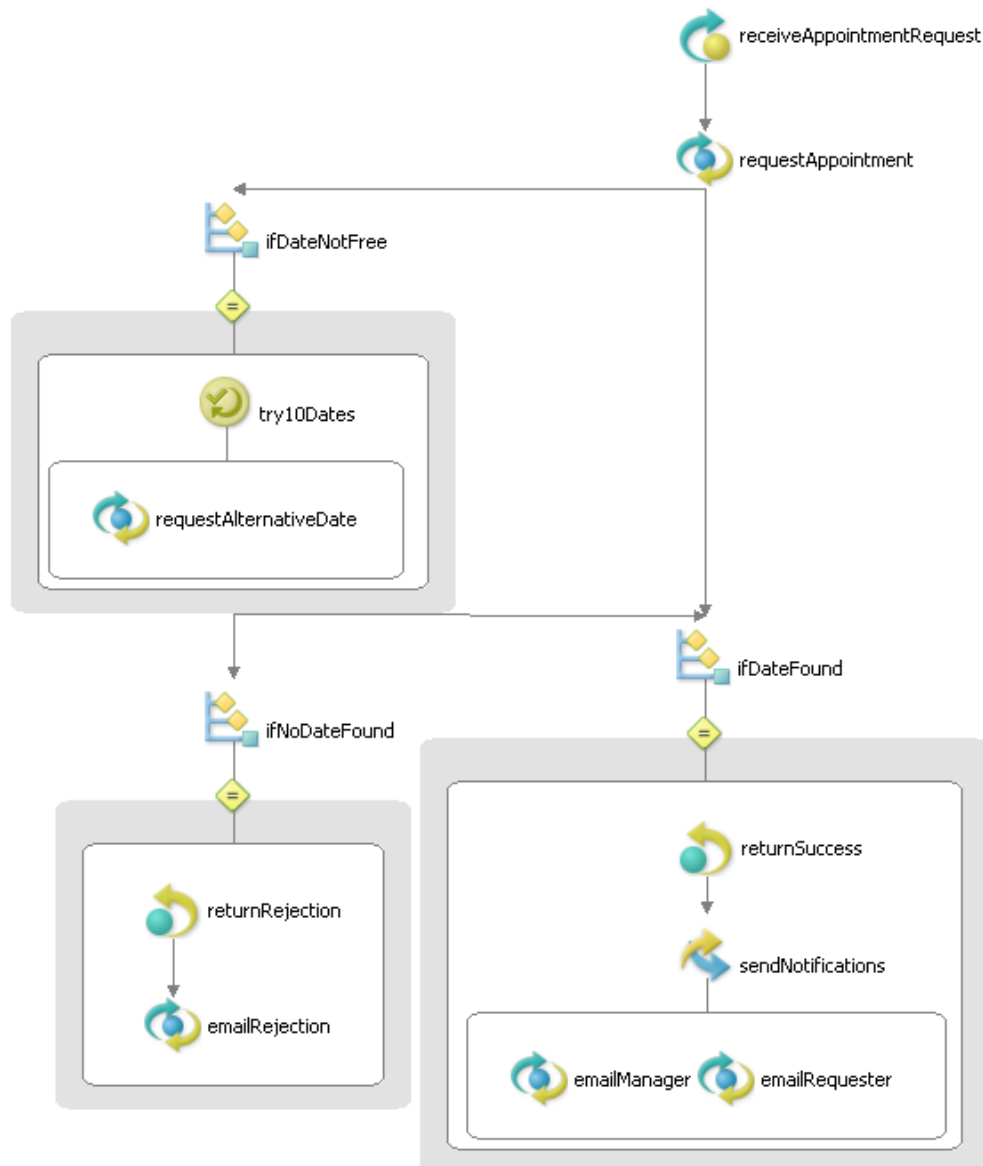


Figure A.2.: An example process modeled using notation B.

A. Survey

Question 3.4 — What happens if the date proposed initially is not available?

Possible answers: free text entry.

Correct answer: 10 different dates will be tried.

Question 3.5 — How many different requests does the process accept?

Possible answers: free text entry.

Correct answer: 1.

Question 3.6 — How many different responses does the process return?

Possible answers: free text entry.

Correct answer: 2.

Part 4

Question 4.1 — Which of both notations do you feel is more comprehensible?

Possible answers: notation A / notation B

Question 4.2 — Which of both notations do you feel is more clear?

Possible answers: notation A / notation B

Question 4.3 — Does notation A include enough information to understand what the process is about?

Possible answers: rather too much detail / just about right / rather too little detail

Question 4.4 — Does notation B include enough information to understand what the process is about?

Possible answers: rather too much detail / just about right / rather too little detail

Question 4.5 — Do you want to add anything else regarding the notations? Did something stand out for being positive or negative?

Possible answers: free text entry.

B. Final Notation

Table B.1 shows the final version of the Composr notation, listing the available graphical representations and their respective stereotypes.




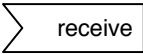
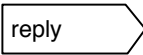
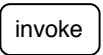





| | |
|---|--|
|  | The representation for the Link stereotype. |
|  | The representation for the Start stereotype. |
|  | The representation for the End stereotype. |
|  | The representation for the Receive stereotype. |
|  | The representation for the Reply stereotype. |
|  | The representation for the Invoke stereotype. |
|  | The representation for the Loop stereotype. |
|  | The representation for the LoopEnd stereotype. |
|  | The representation for the If and IfEnd stereotypes. |
|  | The representation for the Pick and PickEnd stereotypes. |
|  | The representation for the Flow and FlowEnd stereotypes. |

Figure B.1.: The final version of the Composr notation.

C. Generated Artifacts

This appendix contains listings for some of the artifacts generated from the process model used during development.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <wsdl:definitions xmlns:tns="http://composr/tns/PersonPersistenceService"
   xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:wSDL="http://schemas.xmlsoap.org/wsdl/" targetNamespace="http://composr/tns/PersonPersistenceService">
3   <wsdl:types>
4     <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
       attributeFormDefault="qualified" elementFormDefault="qualified">
5       <xs:import namespace="http://composr/tns" schemaLocation="Process01.xsd"/>
6       <xs:import namespace="http://composr/tns/PersonPersistenceService"
       schemaLocation="PersonPersistenceService.xsd"/>
7     </xs:schema>
8   </wsdl:types>
9   <wsdl:message name="saveResponse">
10     <wsdl:part name="parameters" element="tns:saveResponse">
11     </wsdl:part>
12   </wsdl:message>
13   <wsdl:message name="saveRequest">
14     <wsdl:part name="parameters" element="tns:save">
15     </wsdl:part>
16   </wsdl:message>
17   <wsdl:portType name="PersonPersistenceServicePortType">
18     <wsdl:operation name="save">
19       <wsdl:input message="tns:saveRequest" wsaw:Action="urn:save">
20       </wsdl:input>
21       <wsdl:output message="tns:saveResponse" wsaw:Action="urn:saveResponse">
22       </wsdl:output>
23     </wsdl:operation>
24   </wsdl:portType>
25   <wsdl:binding name="PersonPersistenceServiceBinding" type="
     tns:PersonPersistenceServicePortType">
26     <soap:binding style="document" transport="http://schemas.xmlsoap.org/
       wsdl/soap/http"/>
27     <wsdl:operation name="save">
28       <soap:operation soapAction="urn:save" style="document"/>
29       <wsdl:input>
30         <soap:body use="literal"/>
31       </wsdl:input>
32       <wsdl:output>
33         <soap:body use="literal"/>
34       </wsdl:output>
35     </wsdl:operation>
36   </wsdl:binding>
```

C. Generated Artifacts

```
37 <wsdl:service name="PersonPersistenceService">
38   <wsdl:port name="PersonPersistenceServicePort" binding="
      tns:PersonPersistenceServiceBinding">
39     <soap:address location="http://localhost:8080/axis2/services/
        PersonPersistenceService"/>
40   </wsdl:port>
41 </wsdl:service>
42 </wsdl:definitions>
```

Listing C.1: The WSDL description generated for the PersistenceService bound to a Person type.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema xmlns:Q1="http://composr/tns" xmlns:xs="http://www.w3.org/2001/
  XMLSchema" attributeFormDefault="qualified" elementFormDefault="
  qualified" targetNamespace="http://composr/tns/PersonPersistenceService"
  >
3 <xs:import namespace="http://composr/tns" schemaLocation="Process01.xsd"/>
4 <xs:element name="save">
5   <xs:complexType>
6     <xs:sequence>
7       <xs:element minOccurs="0" name="person" nillable="true" type="
        Q1:Person"/>
8     </xs:sequence>
9   </xs:complexType>
10 </xs:element>
11 <xs:element name="saveResponse">
12   <xs:complexType>
13     <xs:sequence>
14       <xs:element minOccurs="0" name="person" nillable="true" type="
        Q1:Person"/>
15     </xs:sequence>
16   </xs:complexType>
17 </xs:element>
18 </xs:schema>
```

Listing C.2: The XML Schema definition generated for the PersistenceService bound to a Person type.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <wsdl:definitions xmlns:plnk2="http://docs.oasis-open.org/wsbpel/2.0/
  plnktype" xmlns:tns="http://composr/tns" xmlns:wsdl="http://schemas.
  xmlsoap.org/wsdl/" name="Process01" targetNamespace="http://composr/tns"
  >
3 <plnk2:partnerLinkType name="Process01PLT">
4   <plnk2:role name="Process01Role" portType="Process01PortType"/>
5 </plnk2:partnerLinkType>
6 <plnk2:partnerLinkType name="PersonPersistenceServicePortTypePLT">
7   <plnk2:role name="Process01Role" portType="Process01PortType"/>
8   <plnk2:role name="PersonPersistenceServicePortTypeRole" portType="
    PersonPersistenceServicePortType"/>
9 </plnk2:partnerLinkType>
10 <wsdl:import location="PersonPersistenceService.wsdl" namespace="http://
    composr/tns/PersonPersistenceService"/>
11 <wsdl:types>
12   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

C. Generated Artifacts

```
13      <xs:import namespace="http://composr/tns" schemaLocation="Process01.
        xsd"/>
14    </xs:schema>
15  </wsdl:types>
16  <wsdl:message name="persistPersonMessage">
17    <wsdl:part name="parameters" type="tns:Person"/>
18  </wsdl:message>
19  <wsdl:message name="persistPersonResponseMessage">
20    <wsdl:part name="parameters" type="boolean"/>
21  </wsdl:message>
22  <wsdl:portType name="Process01PortType">
23    <wsdl:operation name="persistPerson">
24      <wsdl:input message="persistPersonMessage"/>
25      <wsdl:output message="persistPersonResponseMessage"/>
26    </wsdl:operation>
27  </wsdl:portType>
28 </wsdl:definitions>
```

Listing C.3: The WSDL description generated for the process model used during development.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" attributeFormDefault=
  "qualified" elementFormDefault="qualified" targetNamespace="http://
  composr/tns">
3   <xs:complexType name="Person">
4     <xs:sequence>
5       <xs:element minOccurs="0" name="id" type="xs:long"/>
6       <xs:element minOccurs="0" name="adult" type="xs:boolean"/>
7       <xs:element minOccurs="0" name="age" type="xs:int"/>
8       <xs:element minOccurs="0" name="birthday" nillable="true" type="
        xs:dateTime"/>
9       <xs:element minOccurs="0" name="firstName" nillable="true" type="
        xs:string"/>
10      <xs:element minOccurs="0" name="lastName" nillable="true" type="
        xs:string"/>
11    </xs:sequence>
12  </xs:complexType>
13 </xs:schema>
```

Listing C.4: The XML Schema definition generated for the process model used during development.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <bpel:process xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/
  executable" xmlns:ns="http://composr/tns" xmlns:ns0="http://composr/tns/
  PersonPersistenceService" xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exitOnStandardFault="yes" name="Process01" suppressJoinFailure="yes"
  targetNamespace="http://composr/tns">
3   <bpel:import importType="http://schemas.xmlsoap.org/wsdl/" location="
    Process01.wsdl" namespace="http://composr/tns"/>
4   <bpel:import importType="http://schemas.xmlsoap.org/wsdl/" location="
    PersonPersistenceService.wsdl" namespace="http://composr/tns/
    PersonPersistenceService"/>
5   <bpel:partnerLinks>
6     <bpel:partnerLink myRole="Process01Role" name="Process01PLTPartnerLink"
      partnerLinkType="Process01PLT"/>
  </bpel:partnerLinks>
  </bpel:process>
```

C. Generated Artifacts

```
7      <bpel:partnerLink myRole="Process01Role" name="
      PersonPersistenceServicePortTypePLTPartnerLink" partnerLinkType="
      PersonPersistenceServicePortTypePLT" partnerRole="
      PersonPersistenceServicePortTypeRole"/>
8    </bpel:partnerLinks>
9    <bpel:variables>
10     <bpel:variable messageType="ns:persistPersonMessage" name="
      persistPersonInput1"/>
11     <bpel:variable messageType="ns0:saveRequest" name="
      PersonPersistenceServicePortTypesaveInput2"/>
12     <bpel:variable messageType="ns0:saveResponse" name="
      PersonPersistenceServicePortTypesaveOutput3"/>
13     <bpel:variable messageType="ns:persistPersonResponseMessage" name="
      persistPersonOutput4"/>
14   </bpel:variables>
15   <bpel:flow>
16     <bpel:links>
17       <bpel:link name="L0"/>
18       <bpel:link name="L1"/>
19       <bpel:link name="L2"/>
20       <bpel:link name="L3"/>
21       <bpel:link name="L4"/>
22     </bpel:links>
23     <bpel:receive createInstance="yes" operation="persistPerson" partnerLink
      ="Process01PartnerLink" portType="Process01PortType" variable="
      persistPersonInput1">
24       <bpel:sources>
25         <bpel:source linkName="L0"/>
26       </bpel:sources>
27     </bpel:receive>
28     <bpel:assign validate="no">
29       <bpel:targets>
30         <bpel:target linkName="L0"/>
31       </bpel:targets>
32       <bpel:sources>
33         <bpel:source linkName="L1"/>
34       </bpel:sources>
35       <bpel:copy>
36         <bpel:from />
37         <bpel:to />
38       </bpel:copy>
39       <bpel:copy>
40         <bpel:from />
41         <bpel:to>
42           <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0
              :sublang:xpath1.0"/>
43         </bpel:to>
44       </bpel:copy>
45     </bpel:assign>
46     <bpel:assign validate="no">
47       <bpel:targets>
48         <bpel:target linkName="L1"/>
49       </bpel:targets>
50       <bpel:sources>
51         <bpel:source linkName="L2"/>
52       </bpel:sources>
```

C. Generated Artifacts

```
53     <bpel:copy>
54         <bpel:from />
55         <bpel:to />
56     </bpel:copy>
57 </bpel:assign>
58 <bpel:invoke inputVariable="PersonPersistenceServicePortTypesaveInput2"
    operation="save" outputVariable="
    PersonPersistenceServicePortTypesaveOutput3" partnerLink="
    PersonPersistenceServicePortTypePLTPartnerLink" portType="
    ns0:PersonPersistenceServicePortType">
59     <bpel:targets>
60         <bpel:target linkName="L2"/>
61     </bpel:targets>
62     <bpel:sources>
63         <bpel:source linkName="L3"/>
64     </bpel:sources>
65 </bpel:invoke>
66 <bpel:assign validate="no">
67     <bpel:targets>
68         <bpel:target linkName="L3"/>
69     </bpel:targets>
70     <bpel:sources>
71         <bpel:source linkName="L4"/>
72     </bpel:sources>
73     <bpel:copy>
74         <bpel:from>
75             <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0
                :sublang:xpath1.0"/>
76         </bpel:from>
77         <bpel:to>
78             <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0
                :sublang:xpath1.0"/>
79         </bpel:to>
80     </bpel:copy>
81 </bpel:assign>
82 <bpel:reply operation="persistPerson" partnerLink="Process01PartnerLink"
    portType="Process01PortType" variable="persistPersonOutput4">
83     <bpel:targets>
84         <bpel:target linkName="L4"/>
85     </bpel:targets>
86 </bpel:reply>
87 </bpel:flow>
88 </bpel:process>
```

Listing C.5: The BPEL generated for the process model used during development.

D. Compact Disc

The Compact Disc accompanying the hard copy of this thesis provides the following elements.

- This thesis as a PDF file.
- The Eclipse project of the Composr application.
- The reference implementation that development was oriented at.
- The artifacts generated from the reference model presented in section 5.1.

Erklärung der Selbständigkeit

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 21. Dezember 2007

Leif Singer