Model-Driven Engineering of Web Service Compositions: A Transformation from ISDL to BPEL

Teduh Dirgahayu

Thesis for a Master of Science degree in Telematics from the University of Twente Enschede, The Netherlands

Graduation Committee:
Dr. ir. D.A.C. Quartel
ir. R.M. Dijkman
Dr. ir. M.J. van Sinderen

Enschede, The Netherlands
1 July 2005
Abstract

An approach to system specification defined by the Model-Driven Architecture (MDA) separates the specification of system functionality from the specification of the implementation of that functionality on a specific platform. To this end, the MDA proposes a framework for transforming a model into another model of the same system. The framework can also be used to transform a model into code for creating a system. Model-Driven Engineering, which combines the MDA with process and analysis, puts the MDA into work in software development.

The Interaction Systems Design Language (ISDL) is a modeling language that is aimed at supporting the design of distributed systems. The ISDL allows one to model and to structure a system at higher levels of abstraction. At some level, an ISDL model has to be mapped onto available concrete components to create the system.

Web services allows applications within an enterprise to be accessible through the Web by providing an interface that is described in the Web Service Description Language (WSDL). Web services can be further composed to provide a more sophisticated Web service that performs some business process. A language for defining the behavior of a Web service composition is the Business Process Execution Language for Web Services (BPEL4WS, or BPEL for short). From the perspective of a design process, the WSDL and the BPEL are concrete components for implementing a system being designed.

This thesis provides a transformation from ISDL to BPEL using approaches suggested by the MDA, i.e. marking and metamodel transformation. An ISDL profile for business processes modeling with a mapping to the BPEL is developed. In addition, metamodels for the WSDL and the BPEL are developed. The metamodels are used to define transformation specifications from ISDL to WSDL and from ISDL to BPEL. The transformation is able to generate WSDL and BPEL documents from an ISDL model that complies with the profile.
## Contents

<table>
<thead>
<tr>
<th>Abstract</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>vi</td>
</tr>
</tbody>
</table>

### 1 Introduction  
1.1 Motivation ......................................................... 1  
1.2 Objectives .......................................................... 2  
1.3 Approaches .......................................................... 3  
1.4 Structure ............................................................ 4

### 2 Interaction Systems Design Language  
2.1 Introduction .......................................................... 5  
2.2 Behaviors ............................................................. 6  
2.3 Actions ............................................................... 6  
2.4 Interactions .......................................................... 7  
2.5 Causality Relations .................................................. 8  
2.6 Entry and Exit Points ................................................ 10

### 3 WSDL and BPEL  
3.1 WSDL ................................................................. 12  
3.1.1 Structure of WSDL Documents .................................... 12  
3.1.2 Operations ......................................................... 14  
3.2 BPEL ................................................................. 14  
3.2.1 Structure of BPEL Documents ..................................... 15  
3.2.2 Partnerships ....................................................... 16  
3.2.3 Properties and Correlation Sets .................................. 16  
3.2.4 Activities ......................................................... 16  
3.2.5 Scopes ............................................................. 18

### 4 Approaches and Techniques to Model Transformation  
4.1 Basic Concepts ....................................................... 21  
4.2 Approaches to Model Transformation ................................ 22  
4.2.1 Marking ............................................................ 23
Preface

This thesis presents the result of a final assignment for a Master of Science degree in Telematics. The assignment has been carried out in the Architecture and Services of Network Applications Group at the University of Twente.

I would like to thank my supervisors Dick Quartel and Remco Dijkman whose valuable suggestions helped me carry out this assignment. The discussions that we held have transformed me from a person who thinks at the implementation level to a person who thinks at a higher level of abstraction.

I would like to thank all my friends for making my life as an alien in The Netherlands more pleasant. I would also like to thank all my brothers- and sisters-in-faith for always recharging my spiritual needs. I am very grateful to my family, especially my parents, in Indonesia for their continuous love and support.

I would like to give my deeply thank to my lovely wife, Izzati Muhimmah a.k.a. Emma Nawawi Dirgahayu, for her love, care, support, and cheerfulness that has given me energy to live my life.

I dedicate this thesis to my parents.

Enschede, 1 July 2005

Teduh Dirgahayu

teduh.dirgahayu@fti.uii.ac.id
Chapter 1

Introduction

This chapter presents the motivation, the objective, the approaches, and the structure of this thesis. Section 1.1 gives the motivation of this thesis by identifying a need for an automatic creation of Web service compositions. Section 1.2 presents the objective of this thesis. Section 1.3 describes the approaches and the steps to achieve the objective. Finally, Section 1.4 outlines the structure of this thesis.

1.1 Motivation

Nowadays Web services emerge to be a de facto standard platform for application integration. Application integration refers to efforts to integrate applications within a single enterprise or among enterprises, with the intention to share business processes. These efforts allow the business processes to be executed fully automatically or with little human intervention.

The fact that Web services are based on the Extended Markup Language (XML) makes Web services independent from any computer hardware, networks, and operating systems. This independence makes Web services an ideal platform for integrating different applications that are based on different technologies.

Web services encapsulate one or more applications by providing a unique interface accessible through the Web. The interface, which is described in the Web Service Description Language (WSDL), allows other applications to use the encapsulated applications. Web services can be further composed to provide a more sophisticated Web service that performs some business process.

A language for defining the behavior of a Web service composition is the Business Process Execution Language for Web Services (BPEL4WS, or BPEL for short). Using the BPEL, one can specify a business process that involves interactions with other Web services and provide the business process as another Web service.

An approach to develop a system, that includes a Web service composition, is to first define the service that is required of that system. This service can be represented by a model called a service architecture. Subsequently the service may be decomposed
into a composition of other services. The decomposition is done possibly recursively until the final implementation can be mapped onto available application services or concrete components. In this so-called top-down design, the application service and the concrete components are at the implementation level, while the service architecture and its decompositions are at higher levels of abstraction [39].

At a higher level of abstraction, a model is expressed in a modeling language. The language should be able to represent abstract concepts and be implementation independent. This independence allows the service architecture or its decompositions to be mapped onto different application services or concrete components. Languages at the implementation level, e.g. the BPEL, are not suitable to express the model, because such languages lack ability to represent abstract concepts and make the service architecture depend on a particular concrete component.

The Interaction Systems Design Language (ISDL) [22] is a modeling language that is independent from an implementation. The ISDL aims at supporting the design of distributed systems by providing generic design concepts and a notation, both textual and graphical, to model the structure and behavior of the systems. The ISDL is to be used at higher levels of abstraction.

Traditionally, a mapping from a language independent model onto available concrete components is done manually, e.g. some programmers create the code of the system being modeled. However, this traditional mapping is very costly and cannot follow the rapid changes of service deployment technology. The Object Management Group (OMG) addresses this issue by introducing the Model-Driven Architecture (MDA) [32]. The MDA advocates, among others, an automatic transformation from a model into code.

Model-Driven Engineering (MDE) combines the MDA with process and analysis to put the MDA into work in software development. An MDE may include many tasks, such as specification of modeling languages, models, transformation between models, transformations between languages, and tool support [23].

When an automatic transformation from ISDL models to BPEL code is available, the creation of a Web service composition can be done rapidly and at a reduced cost.

1.2 Objectives

The objective of this research is to develop an automatic transformation from ISDL models that represents business processes to BPEL code. We call the transformation the ISDL-to-BPEL transformation. It aims at rapid creation of Web service compositions, such that one would only need to model a desired business process in the ISDL and the business process implementation would be generated in the BPEL automatically.
1.3 Approaches

To achieve the objectives, we adopt approaches introduced by the MDA \[32\], i.e. marking and metamodel transformation that are described in Chapter \[4\]. The choice of the approaches is based on the following assumptions.

First, our work is the initial step to a fully automatic ISDL-to-BPEL transformation. In this step, the transformation may need to be guided manually. Such a guidance may appear as marks on models.

Second, we want the transformation to be general enough, such that it can be applied in any ISDL models that represent business processes. To achieve such generality, the mapping should be defined in the metamodels of the source and the target languages.

Referring to the approach, we take the following steps.

- **Development of metamodels of the WSDL and the BPEL**
  Since official metamodels of the WSDL and the BPEL are not available, we develop the metamodels by ourselves. Rather than developing the metamodel from scratch, we review proposed metamodels for the WSDL and the BPEL \[5, 7, 8, 12, 27\] and enhance them if necessary.

- **Development of an ISDL Profile for BPEL**
  We consider that the ISDL is more expressive and semantically richer than the BPEL. Therefore, a set of rules should be developed to restrict the use of the ISDL for modeling business processes that will be implemented in the BPEL. The restriction is specified in a so-called ISDL Profile for BPEL. The profile also serves as a guidance for modeling BPEL processes.

- **Development of an ISDL-to-BPEL transformation**
  We develop a transformation specification from the ISDL metamodel to our metamodels for the WSDL and the BPEL by referring to the ISDL Profile for BPEL. We then implement the transformation specification in the YATL \[37\].

- **Application of the ISDL-to-BPEL transformation**
  To show that the ISDL-to-BPEL transformation works, we apply it to a business process. We execute the transformation in a transformation engine YATL4MDR \[10\]. Since the engine transforms a model into another model, we develop code generators to produce WSDL and BPEL documents from WSDL and BPEL models, respectively.
  During the application, we evaluate the profile, the transformation specification, and the generated documents. An iterative refinement on the profile and the transformation specification may be necessary.
1.4 Structure

This thesis is structured in the following chapters.

- Chapter 2 presents the concepts and the notation of the ISDL.
- Chapter 3 presents the concepts of the WSDL and the BPEL.
- Chapter 4 describes approaches and techniques to model transformation that are defined in the context of the MDA.
- Chapter 5 reports on our survey on an existing UML-to-BPEL transformation that is conducted to learn some practical ideas on model transformation.
- Chapter 6 presents the syntax and metamodels of the WSDL and the BPEL.
- Chapter 7 describes our ISDL profile for BPEL. This chapter also explains how the profile is developed.
- Chapter 8 describes the ISDL-to-BPEL transformation with an example.
- Finally, Chapter 9 presents the conclusions and the contributions of our work and indicates topics for future work.
Chapter 2

Interaction Systems Design Language

This chapter presents concepts and a notation defined in the Interaction Systems Design Language (ISDL) \cite{22, 39} in which models as input for the ISDL-to-BPEL transformation are defined. This chapter does not intend to fully describe the ISDL, but to describe only concepts and a notation that are recognized by the ISDL-to-BPEL transformation. Section 2.1 gives an overview of the ISDL. Sections 2.2 to 2.6 describe the concepts and notation.

2.1 Introduction

The ISDL is a modeling language that supports the design the architecture of distributed systems by providing generic design concepts and a notation to model the architecture. The ISDL is aimed at modeling distributed systems at higher abstraction levels \cite{38}. Its generic design concepts allow the ISDL to represent system architectures at different abstraction levels.

An architecture concerns to the structure and functionality of a system. The ISDL models the architecture in three parts:

- **entity model**
  This model identifies system parts and defines how they are interconnected. This model represents the structure of the system.

- **behavior model**
  This model represents the functionality of the system parts and how these parts interact with each other.

- **assignment relation**
  This relation defines the relationship between the entity model and the behavior model.
ISDL models can be expressed in a textual notation or a graphical notation. In this chapter, we describe ISDL behavioral concepts and graphical notation which are used in the ISDL profile for BPEL defined in Chapter 7. The metamodel of the ISDL behavioral concepts can be found in [11].

2.2 Behaviors

A behavior models the functionality of an entity as a collection of activities and the relations between the activities. Each activity can be performed by the system alone or by the system in cooperation with other systems.

The ISDL distinguishes between a behavior type, which is a definition of a behavior, and a behavior instantiation, which creates an instance of a behavior type. When a behavior type instantiates a copy of itself, the instantiation is called a behavior recursion.

A behavior type is graphically expressed as a rounded rectangle that contains its type name. A behavior instantiation is expressed as a rounded rectangle that contains the name of the instantiated type followed by an instantiation name. Figure 2.1 depicts a behavior type System that defines a behavior instantiation namely sub of a behavior type Subsystem.

![Figure 2.1: Behavior type with a behavior instantiation](image)

2.3 Actions

An action is a concept to model the successful completion of an activity performed by a single entity. An action is atomic which means that an action represents an indivisible unit of activity in the abstraction level where the action is defined. Consequently, an action either occurs or does not occur. If an action does not occur, any intermediate result established by the activity cannot be referred to.

An action has three kinds of attributes:

- **information attribute** \( \iota \)
  This attribute models the result of the activity.

- **time attribute** \( \tau \)
  This attribute models the time moment when the result becomes available.
• location attribute $\lambda$

This attribute models the location where the result is available.

An action may specify more than one information attributes but at most one time attribute and one location attribute. An action abstracts from how its results are established. This may be defined at a lower abstraction level by decomposing the action into smaller units of activity.

An action is graphically expressed as an ellipse that contains its name. Its attributes are defined in a text box that is connected to the ellipse. An attribute is declared as a definition of an attribute type followed by an attribute name. Figure 2.2 depicts an action logError whose attributes error and time have attribute types Message and Timestamp, respectively. The kind of each attribute followed by a colon precedes the attribute declaration.

![Figure 2.2: Action with attributes](image)

Attribute declarations of an action may have a list of constraints namely attribute constraints. Such constraints represent constraints on the values that are allowed to be established by the action.

2.4 Interactions

An interaction is a concept to model an activity that is performed by two or more entities in cooperation. The interaction is shared between the entities. The ISDL only considers the occurrence of the interaction if the interaction is performed successfully. Entities involved in an interaction are called participants.

An interaction can only occur if all participants are willing to contribute to the interaction. The contribution of each participant is modeled as an interaction contribution. An interaction contribution has attribute kinds that are as same as attribute kinds of an action.

An interaction contribution is graphically expressed as an ellipse segment that contains its name. An interaction is expressed by connecting the flat sides of the involved interaction contributions through a line. Figure 2.3 (a) depicts an interaction contribution send whose attributes info and receiver have attribute types Message and Address, respectively. Figure 2.3 (b) depicts an interaction performed by two interaction contributions send and receive.
A structured interaction contribution is an interaction contribution that allows one or more interaction contributions in behavior instantiations contribute to an interaction performed by the behavior where the behavior instantiations are defined. A structured interaction contribution is graphically expressed as an interaction contribution that is connected to interaction contributions of the behavior instantiations. Figure 2.4 depicts a structured interaction contribution reserve in a behavior TravelAgent that delegates its interaction to two interaction contributions reserve and book in different behavior instantiations Flight and Hotel, respectively.

2.5 Causality Relations

A causality relation models a causal relationship between actions and/or interaction contributions. A causality relation associates an action or an interaction contribution with a causality condition that must be satisfied to enable the occurrence of the action. The action or the interaction contribution is also called the causality target of the causality relation. There are four basic causality conditions that can be defined for an action $a$ as a causality target in an association with another action $b$: 
• **start condition**
  This condition allows the action $a$ to always occur independently from any other action.

• **enabling condition**
  This condition allows the action $a$ to occur only after the action $b$ has occurred.

• **disabling condition**
  This condition allows the action $a$ to occur if the action $b$ has not occurred previously or simultaneously.

• **synchronization condition**
  This condition makes the action $a$ to occur simultaneously with the occurrence of the action $b$.

Figure 2.5 (a) depicts the start condition, (b) depicts the enabling condition, (c) depicts the disabling condition, and (d) depicts the synchronization condition.

![Figure 2.5: Basic causality conditions](image)

A *conjunction* and a *disjunction* of causality conditions can be used to model a complex causality relation. A conjunction represents that all conditions must be satisfied to enable the occurrence of a causality target. A disjunction represents that any condition must be satisfied to enable the occurrence of a causality target. Conjunctions and disjunctions can be combined.

A conjunction and a disjunction are graphically expressed as a filled square and an unfilled square, respectively. Figure 2.6 (a) depicts a conjunction of two enabling conditions, and (b) depicts a disjunction of two enabling conditions.

![Figure 2.6: Conjunction and disjunction of causality conditions](image)
The ISDL also defines shorthands for complex causality relations that are frequently used. Here we only briefly describe two of them i.e. the disabling relation and the choice relation.

Figure 2.7 (a) shows the definition of a disabling relation. It defines that an action \( b \) is not allowed to occur simultaneously with another action \( a \). If the action \( a \) occurs, the action \( b \) is not allowed to occur anymore. The figure (b) and (c) show two different shorthands to represent the complex causality relation between them.

![Disabling relation and its shorthands](image1)

Figure 2.7: Disabling relation and its shorthands

Figure 2.8 (a) shows the definition of a choice relation. It defines that only one of the actions, either the action \( a \) or the action \( b \), is allowed to occur. The figure (b) and (c) show two different shorthands to represent the complex causality relation between them.

![Choice relation and its shorthands](image2)

Figure 2.8: Choice relation and its shorthands

### 2.6 Entry and Exit Points

A causality condition of an action may involve actions from other behaviors. However, causality conditions may not cross the boundary of a behavior. Therefore, to make the occurrence of an action depend on actions from another behavior, the following notational elements must be used.

- **entry point**
  
  This point represents a causality condition outside the behavior that can be used inside the behavior.
- **exit point**

  This point represents a causality condition inside the behavior that can be used outside the behavior.

  An entry points is graphically expressed as a triangle that contains its *name* and is pointing inside a behavior. An exit points is also expressed as a triangle that contains its *name* but is pointing outside a behavior. Figure 2.9 shows an example of causality conditions that are represented by entry and exit points. An action \( b \) in the behavior \( B1 \) is enabled by an action \( a \) in the behavior \( B2 \). An entry point \( e \) of the behavior \( B1 \) is used to represent this causality condition. The action \( b \) enables an action \( c \) in the behavior \( B2 \). An exit point \( x \) of the behavior \( B1 \) is used to represent this causality condition.

  ![Figure 2.9: Causality conditions with entry and exit points](image)

  An entry point can have parameters to pass values into a behavior. Similarly, an exit point can have parameters to pass values outside a behavior. Parameters of an exit point may have a list of constraints called *parameter constraints*. Such constraints represent constraints on the values that are allowed to be passed through the exit point. Graphically, parameters of entry or exit points are defined in text boxes connected to the entry or exit points. A parameter is declared as a *parameter type* followed by a *parameter name*.

  ![Figure 2.10: Behavior with parameterized entry and exit points](image)

  Figure 2.10 depicts a behavior *Payment* with an entry point \( e \) and an exit point \( x \). The entry point \( e \) is parameterized with *customer* and *order* whose types are *Customer* and *Items*, respectively. The exit point \( x \) is parameterized with *receipt* whose type is *Message*.  

11
Chapter 3

WSDL and BPEL

This chapter presents two languages related to Web services composition description. Section 3.1 describes a language for describing Web service interactions and Section 3.2 describes a language for defining the behaviors of Web service compositions. Both are the target languages of the ISDL-to-BPEL transformation presented in Chapter 8.

3.1 WSDL

Web services describe their interfaces using an XML-based language called the Web Service Description Language (WSDL). This section summarizes parts of WSDL specification 1.1 [42] that are relevant to the ISDL-to-BPEL transformation. The WSDL syntax can be found in Chapter 6.

3.1.1 Structure of WSDL Documents

A WSDL document consists of two main parts i.e. an abstract part and a concrete part [1]. The abstract part is the part of a Web service description that is independent of any particular protocol and message encoding used for interacting with the service. The part is also independent of the location, in term of network address, where the service is available. This makes the abstract part reusable for different protocols, message encoding, and locations. The concrete part of a WSDL document describes a specific protocol binding, message encoding, and location binding of a Web service. Figure 3.1 shows the structure of a WSDL document.

A WSDL document consists of a set of definitions. Each definition is represented by an element of the document. The abstract part of a WSDL document consists of three kinds of elements. These are:

- types
  This element encloses a definition of data types for specifying messages to be exchanged. The WSDL uses the XML Schema [45] to define the data types.
Figure 3.1: Structure of WSDL Documents

- **message**
  This element defines messages to be exchanged. A message is structured into one or more parts that represent logical units within the message. Each part is associated with a simple data type of the XML Schema or a user-defined element. The user-defined element must be defined in the types element.

- **port type**
  This element defines a set of related operations including messages to be exchanged in each operation. Operations are functionalities of a Web service that are provided to its users. Messages can be input messages, output messages, or fault messages. These messages must be defined in the message element.

The concrete part of a WSDL document consists of three kinds of elements. These are:

- **binding**
  This element specifies a protocol and a message encoding for operations and messages of a particular port type. A port type can have multiple bindings.

- **ports**
  This element defines an *endpoint* which is a combination of a binding and a network address. The network address, that is specified by a URI, indicates where the service is available.

- **service**
  This element groups a set of related endpoints.
Elements of a WSDL document can be defined in separated documents. The WSDL document can then be constructed by using an import mechanism. This makes the document modular and leverages the reusability of the constituent elements.

3.1.2 Operations

The WSDL defines four operation types. These are:

- **One-Way Operation**
  When a service user invokes this operation on an endpoint, the user sends a message and the endpoint receives the message.

- **Request-Response Operation**
  When a service user invokes this operation on an endpoint, the user sends a message and the endpoint receives the message. The endpoint then sends a correlated message as a response back to the service user. The response message can be an output message or a message from a set of fault messages.

- **Solicit-Response Operation**
  An endpoint initiates the operation by sending a message to a service user and expects that the user sends a correlated message as a response back to the endpoint. The response message can be an input message or a message from a set of fault messages.

- **Notification Operation**
  An endpoint initiates the operation by sending a message to a service user.

3.2 BPEL

Web services can be composed to provide a more sophisticated Web service. The behavior of the composition can be defined in an XML-based language called the Business Process Execution Language for Web Services (BPEL). This section summarizes parts of the BPEL specification version 1.1 [4] that are relevant to the ISDL-to-BPEL model transformation. The BPEL syntax can be found in Chapter 6.

A Web services provides a stateless client-server service that is not always suitable to perform a business process. A business process may need to be performed in a peer-to-peer relationship that is stateful and long-running. This relationship performs a conversation or message exchanges between the participants in some context.

A BPEL process is aimed to provide such a business process on top of Web services. To do so, the BPEL specifies several types of elements to be used as extensions in WSDL documents. These extensions make the WSDL documents able to support conversational relationships.

We define a BPEL process as a business process that is specified using the BPEL. A BPEL process itself is a Web service that typically comes as a Web
A BPEL process can be described in two ways: abstract or executable process. An abstract process specifies the message exchange behavior of each participant of a business process without revealing their internal behavior. An executable process specifies the actual behavior of a participant of a business process. Here we only deal with BPEL processes as executable processes.

### 3.2.1 Structure of BPEL Documents

A BPEL document consists of a set of definitions i.e. partner links, partners, variables, correlation sets, an activity, a compensation handler, fault handlers, and event handlers. Partner links and partners define relationships between a BPEL process and other Web services that act as partners of the BPEL process. Variables are used to maintain the state of a BPEL process. A correlation set contains a set of properties to correlate messages of a conversation. The activity defines the primary behavior performed by the BPEL process.

A compensation handler defines a behavior to reverse the impact of a transaction between the BPEL process and its partner. Fault handlers are defined to catch and to handle faults within the BPEL process. Event handlers allow events that occur when the BPEL process’ primary activity is executed be to handled concurrently with the activity. Figure 3.2 shows the structure of a BPEL document.

![Figure 3.2: Structure of BPEL Documents](image)
3.2.2 Partnerships

A BPEL process defines a collaboration between multiple business entities called partners. A partnership is typically peer-to-peer with a two-way dependency at the service level. The BPEL represents a partnership by the notion of a partner link.

A partner link is an instance of a partner link type. A partner link type defines roles played by partners involved in a partnership. Each role is associated with a WSDL port type that defines the messages that can be exchanged within the partnership. At most, two roles are defined in a partner link type. One for the BPEL process being defined and another for its partner. At least, one role must be defined. This role can be for the BPEL process being defined or for its partner. Partner link types are BPEL extensions for the WSDL.

A role is assigned operations that is defined in its port type. A BPEL process must implement operations assigned to its role. If a role is defined for a partner, the BPEL process can expect that the partner implements the operations in the partner's role. These operations allow the BPEL process to exchange messages with the partner.

3.2.3 Properties and Correlation Sets

The BPEL defines the notion of a property to provide a unique name. Properties are useful to give some context to a message. The context can be, for instance, the conversation to which the message belongs or the security level of the exchanged messages.

A property is typically mapped to a specific field in a message by using a property alias. This makes the property have a same value as the value of the field. Properties and property aliases are BPEL extensions for the WSDL.

A set of properties can be used to correlates messages in a conversation. Such a set of properties is called a correlation set. To correlate messages, the same correlation sets in different messages are set with a same value.

3.2.4 Activities

A BPEL activity can be a basic activity or a structured activity. A basic activity performs certain functionality. A structured activity controls how other activities must be executed. The BPEL basic activities are:

- **invoke**
  This activity invokes an operation of a partner. A BPEL process immediately continues its execution after invoking a one-way operation. When a BPEL process invokes a request-response operation, the activity blocks the execution of the BPEL process until a response message or a fault message is received.

- **receive**
  This activity is used to receive an invocation from a partner. The activity blocks
the execution of a BPEL process where the activity is defined and waits for a matching message to arrive. When the message arrives, the BPEL process continues its execution.

- **reply**
  This activity is used in combination with a receive activity to provide a request-response operation. This activity sends a reply message or a fault message as a response to a message received by the receive activity.

- **assign**
  This activity updates the values of variables with new data. The new data may already be stored in other variables or be literal values. An assign activity can update any number of variables.

- **throw**
  This activity generates a fault within a BPEL process. The fault is intended to be handled by a fault handler. Otherwise, the fault causes a termination of the BPEL process.

- **wait**
  This activity delays the execution of a BPEL process where the activity is performed for a certain time period or until a certain time moment. When the time period passes or the certain time moment is reached, the BPEL process continues its execution.

- **empty**
  This activity does nothing. It may be useful for synchronizing concurrent activities (see also the flow activity of the BPEL structured activities).

The BPEL structured activities are:

- **sequence**
  This activity orders one or more activities to be performed sequentially.

- **flow**
  This activity allows one or more activities to be performed concurrently. Two concurrent activities can be synchronized by a *link* that specifies the source and the target activities in the synchronization. A target activity can only be performed after a source activity is performed. The source or target activity can be an empty activity.

- **switch**
  This activity allows a BPEL process to have alternative behaviors. The activity specifies a set of conditional activities that are performed if their condition are satisfied. One activity can be specified as the default. This activity is performed in case no condition can be satisfied.
• **while**
  This activity performs another activity repeatedly as long as the condition for the repetition is satisfied.

• **pick**
  This activity defines handlers for events that may occur. An event can be the arrival of a message or the expiration of an alarm. The alarm expires when a certain time period passes or a certain time moment is reached.
  This activity blocks the execution of the BPEL process in which the activity is defined until an event occurs. When an event occurs, its corresponding handler is performed. The activity only executes the handler of the first event that occurs.

The BPEL also specifies activities that are not classified as basic nor as structured activities. These are:

• **terminate**
  This activity terminates the BPEL process in which the activity is performed. All currently running activities are terminated immediately.

• **compensate**
  This activity invokes the compensation handler of a scope (see Subsection 3.2.5). The activity can only be defined within a fault handler or a compensation handler.

### 3.2.5 Scopes

A BPEL scope provides a context within a BPEL process. A context is a unit of related activities. A process can be considered a global scope and is called a **process scope**. A scope consists of an activity called the **primary activity** that defines the normal behavior of the scope. The scope may define variables, correlation sets, fault handlers, a compensation handler, and event handlers.

Another use of a scope is to provide concurrency control to shared variables. Such a scope, that is called a **serializable scope**, orders concurrent access to the shared variables to guarantee their data integrity.

Scopes can be nested. Variables that are defined in a scope is visible for all nested scopes in arbitrary depth.

### Fault Handlers

A fault handler is defined to handle a certain fault that may occur within the scope. A scope may have more than one fault handlers to catch different faults. Possible sources of faults are:

• fault messages as responses to invoke activities,

• faults thrown by throw activities,
• standard faults defined in the BPEL specification, and
• platform-specific faults e.g. communication failures.

A fault handler is executed when its corresponding fault is thrown within the
scope. Although a fault is successfully handled by a fault handler, the scope is not
considered to complete normally. Thus a fault handler is an abnormal behavior of a
scope.

If a fault can not be caught by a fault handler of a scope, the fault is rethrown to
the immediate enclosing scope. If a fault occurs in or is rethrown to the process scope
and there is no fault handler defined for the fault, the process terminates abnormally.

Compensation Handlers

It is not efficient to execute a business process as a single ACID\(^1\) transaction because
the process may need a lock for a long period of time. Therefore a BPEL process
performs a long-running transaction (LRT) that typically involves ACID transactions
with its partners. If a BPEL process fails, any committed ACID transactions must
be compensated. A compensation handler attempts to reverse the effect of the
transactions.

A compensation handler of a scope can be invoked only if the scope completes
normally. The invocation can only be performed within a fault handler or
the compensation handler of a scope that immediately encloses the scope to be
compensated. Since a compensation handler is also for fault handling, it is an abnormal
behavior of a scope.

When a compensation handler is invoked, it sees the scope’s states as they were
when the scope completed. Thus the handler cannot update variables that are used by
a BPEL process. A compensation handler cannot be affected by and does not affect
the states of the process. It can only affect external entities.

Event Handlers

A scope may define handlers for events that occur at unpredictable time moments
alongside its primary activity. An event can be the arrival of a message and the
expiration of an alarm. When its corresponding event occurs, an event handler is
performed concurrently with the primary activity or other event handlers of a scope.
Unlike fault and compensation handlers, event handlers are considered as being part of
the normal behavior of a scope.

An event handler is enabled as long as its scope is active. A scope is active
when it is executing its primary activity. When the normal behavior of the scope is
completed, the event handlers are disabled. However event handlers that are already

\(^1\) Atomicity, Consistency, Isolation, and Durability: properties to guarantee data integrity.
dispatched are allowed to complete. The completion of the scope as a whole is delayed until all dispatched event handlers have completed.

An event handler for an alarm is executed at most once during the time period a scope is active. After its execution, the handler is disabled. An event handler for incoming messages remains enabled after its activity is performed even for concurrent execution. It may occur multiple times during the time period its scope is active.
Chapter 4

Approaches and Techniques to Model Transformation

This chapter describes approaches and techniques to model transformation that are defined in the context of the Model-Driven Architecture (MDA) [32]. Section 4.1 introduces basic concepts to understand MDA model transformation. Section 4.2 describes approaches suggested by the MDA to perform model transformation. Techniques used in the approaches are presented in Sections 4.3, 4.4, and 4.5. Finally, Section 4.6 gives some remarks to the approaches and the techniques with regard to the ISDL-to-BPEL transformation.

4.1 Basic Concepts

The MDA defines model transformation as the process of converting one model to another model of the same system. The basic pattern of model transformation is illustrated in Figure 4.1.

![Figure 4.1: Basic pattern of model transformation](image)
The MDA model transformation is mainly used to convert a platform independent model (PIM) to a platform specific model (PSM). A PIM is a representation of a system independent of the platforms on which the system will be implemented, while a PSM is a representation of a system from a viewpoint of a platform on which the system will be implemented. The transformation needs additional information to be able to transform a PIM into a PSM.

The MDA basic pattern may be applied multiple times. Figure 4.2 depicts an example showing that the pattern is applied twice in sequence. The first transformation produces a PSM from a PIM. The second transformation takes the PSM of the first transformation to produce the second PSM. The PSM produced by the first transformation acts as the PIM for the second transformation. Multiple applications of the MDA basic pattern result in relative notions of PIM and PSM. However, in a single model transformation, the source model is always the PIM and the target model is always the PSM.

![Figure 4.2: Applying the basic pattern twice in sequence](image)

### 4.2 Approaches to Model Transformation

The MDA suggests several approaches to model transformation namely marking, metamodel transformation, model transformation, pattern application, and model
merging. One or more approaches may be used by the MDA basic pattern at a time. In this section, we describe two of them that are used in the ISDL-to-BPEL transformation i.e. marking and metamodel transformation.

### 4.2.1 Marking

The marking approach is illustrated in Figure 4.3. The transformation uses information about the platform to which the PSM is targeted to specify a mapping from the PIM to the PSM. To support the mapping, the designer must add marks to the PIM. These marks are used to guide the transformation. The marked PIM can then be transformed by the mapping to produce the PSM.

Marks can be defined in a profile of the modeling language of the PIM. We describe a language profile including marks in Section 4.3.

![Figure 4.3: Marking](image)

### 4.2.2 Metamodel Transformation

The metamodel transformation approach is illustrated in Figure 4.4. In this approach, the metamodels of the modeling languages that can be used to specify the PIM and the PSM must be available. A metamodel is the model of a model and, therefore, a model is an instance of a metamodel. The metamodels of a PIM and a PSM are called the platform independent metamodel and the platform specific metamodel, respectively. A transformation specification defines a mapping from elements of the platform independent metamodel to elements of the platform specific metamodel. The transformation specification consists of a set of rules to perform the mapping. The
transformation specification is then applied to the PIM to produce the PSM.

To support metamodeling, the OMG provides a framework called the MOF Metadata Architecture [29]. We describe the framework in Section 4.4.

![Figure 4.4: Metamodel transformation](image)

### 4.3 Profiling

To enable model transformation, a model may have to consider the characteristics of the platform to which the transformation is targeted. The characteristics (e.g. the capabilities and the limitations of the platform) can be represented in a profile of the modeling language of the model.

A profile of a modeling language customizes the language for representing concepts within a specific domain, using the concepts and the notation of the language itself [16, 24, 31]. The customization is typically done in two ways.

First, the profile specializes the elements of the modeling language such that the semantics of the elements corresponds to the semantics of concepts within the domain. The specialized elements are represented as stereotypes attached to the elements they specialize. Semantics added by a stereotype should not violate the original semantics of the model element. In this way, a profile is an extension of the base modeling language.

Second, the profile restricts the use of the elements of the modeling language in the domain. Such restriction is necessary, because the domain may not recognize some concepts defined in the language. Thus, only a subset of the elements of the modeling language is allowed to appear in the model for the domain. In this way, a profile is a subset of the base modeling language.

A profile may also prescribe how model elements should be composed to represent a concept that cannot be represented by a single element of the modeling language.
Such compositions typically come from practical experience. They are compositions that are proven to be correct and efficient for modeling the concept. Therefore, the profile acts as a modeling guide.

In the MDA, a profile defines a specialized metamodel for a certain platform. A profile may identify a set of marks to guide a model transformation. A mark can be represented by a stereotype with a list of properties that characterize the stereotype.

4.4 Metamodeling

Metamodeling is the process of defining the metamodel of a concrete language. The OMG provides the MOF Metadata Architecture as a metamodeling framework which consists of four layers of metadata. We concern ourselves with the top three layers of the architecture. They are depicted in Figure 4.5 including examples of instances belong to each layer.

![Figure 4.5: Top three layers of MOF metadata architecture](image)

The M1 layer contains models which are abstract representations of systems or information. The M2 layer contains metamodels which define how to construct M1 models. The M3 layer contains a meta-metamodel which defines how to construct M2 metamodels. The OMG provides the Meta-Object Facility Specification as the meta-metamodel in the architecture. The meta-metamodel is also referred as the MOF model.

The framework can be used to construct different kinds of models and metamodels. The framework supports interchange of models and metamodels that use the same meta-metamodel. To enable the interchange, the OMG provides the XML Metadata Interchange (XMI) Specification. The XMI specifies a standard way to declare models in the XML according to their metamodel.

In the MDA, the MOF model and metamodels play important roles. The MOF
model defines a language for creating metamodels, and the metamodels define languages for creating models [3, 24]. The MOF model and the metamodels do not have a concrete syntax or notation so they are called abstract languages. The metamodels can be visualized using UML notation defined in the UML Profile for MOF [28], which therefore serves as a concrete syntax.

4.5 Mapping

To perform a model transformation, a mapping from a PIM into a PSM must be available. The mapping consists of rules to map elements of the PIM into elements of the PSM. In some approaches, the MDA calls the mapping the transformation specification. A mapping can be a metamodel mapping, a model instance mapping, or a combination of the two.

A metamodel mapping, which is also called model type mapping, specifies a mapping in the metamodel layer (M2) of the MOF metadata architecture. The mapping transforms all instances of a type in the platform independent metamodel into instances of some types in the platform specific metamodel. The transformation specification in the metamodel transformation approach described in Subsection 4.2.2 is a metamodel mapping.

A model instance mapping specifies a mapping in the model layer (M1) of the MOF metadata architecture. The mapping is specified by identifying model elements in the PIM that should be transformed in particular way depending on the platform of the PSM. This approach to mapping defines annotations to be applied to the elements of the PIM such that the elements can represent concepts in the PSM. The marking approach described in Subsection 4.2.1 uses a model instance mapping.

Another approach to mapping combines the mappings described previously. The metamodel mapping relies on the platform independent information alone. However, to produce the PSM correctly, most mappings need platform specific information. The PIM is annotated with marks that provide such information.

A mapping can be specified using natural language, an algorithm, or a model mapping language. The OMG issued the MOF 2.0 Query, Views, and Transformation (QVT) Request for Proposal (RFP) [30] to address issues in the manipulation of MOF models. The RFP includes requirements for a language to define model mappings. Several languages have already been submitted to the OMG to formally answer the RFP [14]. However, no specification for the MOF 2.0 QVT is published yet by the OMG.

Other languages are developed to informally answer the RFP e.g. the ATL [6] and the YATL [37]. In the next subsection, we describe the YATL which we use to implement the mapping of the ISDL-to-BPEL transformation.
4.5.1 YATL: Yet Another Transformation Language

The YATL is a hybrid transformation language which combines both declarative and imperative features. It is designed to express model transformation and to answer the MOF 2.0 QVT RFP. However, the requirements of views are not supported yet.

The YATL uses the Object Constraint Language (OCL) \[33\] to express its queries. A query is evaluated in a given context and returns a value that can be an instance of a MOF primitive type, a model element, a collection, or a tuple.

To perform a transformation, the YATL uses the declarative features of the OCL to select instances in the source model. The YATL then uses its imperative features to create, delete, or manipulate instances in the target model. A YATL transformation consists of YATL transformation rules. Each rule consists of an optional declarative matching part and an imperative part. The imperative part of a rule can invoke other rules. This allows YATL transformations to be defined in a structured way.

The imperative part can contain variable declarations, instructions to create and delete instances in the target model, patterns to create instances in a concise way, conditional statements, loop statements, and native statements. To support traceability between source and target instances, the YATL has a feature to store and retrieve mappings between them at run-time. This feature is expressed by the \texttt{track} operator. Given a relation \( R \) and two model element instances \( X \) and \( Y \), the \texttt{track} operator does the following \[36\].

- \texttt{track (X, R, Y)} stores the mapping between \( X \) and \( Y \) in \( R \).
- \( Y := \texttt{track (X, R, null)} \) retrieves the element instance related to \( X \) in \( R \).
- \( X := \texttt{track (null, R, Y)} \) retrieves the element instance related to \( Y \) in \( R \).

The relation \( R \) is identified by its name. The types of \( X \) and \( Y \) can be OCL 2.0 types, model elements, collections, or tuples.

4.6 Concluding Remarks

We use the approaches described in Section \[4.2\] to develop the ISDL-to-BPEL transformation. The approaches are applied in sequence: first the marking approach, then the metamodel transformation approach, as illustrated in Figure \[4.6\].

The marking approach mentions that a PSM is produced as a result of transforming a marked PIM by a mapping. However, we consider that a marked PIM is already a PSM because marks used to annotate the PIM are defined from a particular platform. The marks are meaningless for other platform, which makes the model specific to the particular platform.
Figure 4.6: MDA approaches used in the ISDL-to-BPEL transformation

The metamodel transformation approach uses annotations given by the marking approach to make sure that the transformation goes correctly or to provide additional information needed by a PSM. The annotations are also useful to indicate what model element must be produced in the PSM when a model element in the PIM has a set of alternative mappings in the PSM. This makes the approach use not only the metamodel mapping but also the model instance mapping.

To enable the transformation, typically a model should comply with some restrictions. Such restrictions are provided in a profile of the modeling language that has a mapping to a target platform. To produce such a profile, we use the profiling technique described in Section 4.3.

The metamodel transformation approach needs the platform independent and the platform specific metamodels to define the transformation specification. Thus, for the ISDL-to-BPEL transformation, we need metamodels for the ISDL and the BPEL. Since the official metamodel for the BPEL is not available, we have to develop the metamodel ourselves. To do so, we use the metamodeling technique described in Section 4.4.

The mapping technique described in Section 4.5 is used to produce (i) a set of marks to be used in the marking approach and (ii) transformation specification to be
used in the metamodel transformation approach. The marks are typically included in the profile of the modeling language of the PIM. The transformation specification is to be implemented using a transformation language to automate the ISDL-to-BPEL transformation
Chapter 5

Survey of A UML-to-BPEL Transformation

This chapter reports on our survey of a transformation from the Unified Modeling Language (UML) models to the Business Process Execution Language (BPEL) documents. In short, we called the transformation the UML-to-BPEL transformation. Section 5.1 describes our motivation to conduct the survey. Section 5.2 gives an overview of the transformation. Section 5.3 briefly describes a UML profile that enables the transformation. Section 5.4 presents a UML model that can be transformed by the transformation. Finally, Section 5.5 lists lessons that we learn from the survey.

5.1 Motivation

In order to learn some practical lessons on how to develop a model transformation and what issues should be considered, we conduct a survey of a transformation that is similar to our objective, i.e. the ISDL-to-BPEL transformation. We choose to survey the UML-to-BPEL transformation for two reasons.

First, it is obvious that the ISDL-to-BPEL and the UML-to-BPEL transformations aim at generating BPEL document. Thus, we can identify issues that should be considered for generating BPEL document from a model.

Second, the UML and the ISDL are usually used to produce models that are independent from any particular technology used to implement the eventual systems. They allow one to model a system without considering the implementation technology. The model may contains elements or compositions of elements that cannot be mapped onto the implementation technology directly. Thus, we can learn how to restrict the use of the languages to enable a model transformation.

For example, the UML can model a system that includes a reuse mechanism namely class inheritance. Since this mechanism cannot be mapped onto the BPEL, the mechanism should not be used for modeling BPEL processes. Such a restriction must be clearly defined in some way.
5.2 Overview of UML-to-BPEL Transformation

A technique and a set of tools have been developed for the UML-to-BPEL transformation [26]. Applying the technique, we conduct the survey by using the following tools.

- **IBM Rational Rose [19]**
  We use this tool to create UML models of business processes that will be transformed into BPEL documents.

- **Unisys XMI Add-in 1.3.6 [18]**
  This is an add-in for IBM Rational Rose such that Rose can convert or export its UML models into XML Metadata Interchange (XMI) format.

- **IBM UML2BPEL plug-in, which is bundled in ETTK 2.1 [17]**
  This is an Eclipse plug-in for generating BPEL documents from UML models in XMI format. This tool does the actual transformation from UML to BPEL.

- **Eclipse 2.1.3 [13]**, with the EMF 1.1.1 and the XSD 1.1.1 plug-ins installed
  We use this tool as a framework for the UML2BPEL plug-in.

The technique consists of two stages. First, we convert UML models into XMI format. Second, we transform the XMI format into BPEL documents. Figure 5.1 illustrates the stages. The figure also shows artifacts produced by the transformation.

![Figure 5.1: Transforming a UML model into artifacts of a BPEL process](image)

The figure assumes that a model has been created by using Rose. The model must conform to a profile [16] which is defined to enable the UML-to-BPEL transformation. By default, Rose saves the model in its proprietary format as a .mdl file. Since
UML2BPEL plug-in only accepts UML models in XMI format as inputs, in the first stage, the model is converted into XMI format. The conversion is performed by the Unisys XMI Add-in.

In the second stage, the UML model in XMI format is fed to the UML2BPEL plug-in to generate web service artifacts. The plug-in that works in Eclipse contains necessary mapping for the transformation. The web service artifacts produced are an XML schema file (denoted as XSD Schema in the figure), one or more web service definition files (denoted as WSDL Definition), and a business process definition file (denoted as BPEL Process).

5.3 UML Profile for BPEL

To facilitate modeling of business processes using the UML, IBM proposed a UML 1.4 Profile for Automated Business Processes with a mapping to the BPEL [16]. The profile is intended to enable an automatic BPEL document generation from UML models. We use the term **UML Profile for BPEL** to refer to the proposed UML profile in the rest of the chapter.

The UML Profile for BPEL identifies stereotypes for representing concepts in BPEL processes and diagrams for viewing the business process. The profile also acts as a guidance by providing a set of composition rules that enable the UML-to-BPEL transformation.

5.3.1 Stereotypes

The UML Profile for BPEL extensively uses stereotypes. The stereotypes add new semantics to existing UML model elements such that the UML is suitable to model concepts within the BPEL. Another benefit of using stereotypes is that a transformation tool knows what BPEL construct must be generated for a model element. This is because stereotypes are explicitly attached to model elements so that they can be processed by the tool.

The stereotypes defined in the profile are summarized as follows.

- **process**
  This is applied to classes for representing business processes. A model of a business process consists of only one class with this stereotype. Attributes of the class represent the state of the business process.

- **role**
  This is applied to classes for representing roles played by parties involved in a business process. It is used to define the structure of a business process and protocols between a business process and its partners.
• **port**  
This is applied to aggregation relationships that represents containments of roles within a business processes.

• **correlation**  
This can be applied to classes for representing correlation sets and their properties. This can also be applied to class attributes for indicating what correlation sets are used by a business process.

• **properties**  
This is applied to classes for representing properties or property aliases.

• **messageContent**  
This is applied to classes for representing messages to be exchanged in business interactions.

• **data**  
This is applied to classes for representing data types.

• **protocol**  
This is applied to packages that contain diagrams specifying protocols of business interactions.

• **external**  
This is applied to packages containing model elements that their corresponding artifacts will not be generated.

• **bpe1 activity**  
Note that the actual stereotypes are types of BPEL activities: receive, reply, invoke, etc. Those are applied to UML activities for representing BPEL activities.

• **catch**  
This is applied to activities that are responsible for catching fault signals and performing operations to handle the faults.

• **compensationHandler**  
This is applied to activities that are responsible for performing operations to compensate impacts of failures.

Other standard stereotypes such as interface, uses, and import may appear in a model. Their semantics are as defined by the UML specification.

5.3.2 **Diagrams**

The UML Profile for BPEL uses class diagrams, activity diagrams, and package diagrams only. The use of the diagrams is summarized as follows.
• **Class diagram**
  This diagram is used to view the static parts of a business process. Those parts are the definition of a business process, protocols, messages, data types, properties, and correlations.

• **Activity diagrams**
  This diagram is used to view the behavior of a business process. The model orders the execution of activities within the business process and specifies partners where interaction activities must be performed.

• **Package diagrams**
  This diagram is used to view dependencies among packages. Packages are also used to specify the partner link type of a protocol.

5.3.3 **Composition Rules**

Using stereotyped model elements and identified diagrams does not guarantee that the corresponding BPEL document is generated properly. To ensure that a model can generate BPEL document, the profile also acts as a modeling guide that provides a set of composition rules. The profile maps BPEL concepts and constructs into UML model elements or compositions of model elements. When a model conforms to the rules, the BPEL document that represents the business process being modeled can be generated properly. Figures 5.2 and 5.3 are examples of the composition rules.

![Figure 5.2: Composition rule of the BPEL interaction protocol](image1)

Figure 5.2: Composition rule of the BPEL interaction protocol

![Figure 5.3: Composition rule of the BPEL process definition](image2)

Figure 5.3: Composition rule of the BPEL process definition
5.4 An Example

To illustrate how a business process is modeled, we describe a model that conforms to the profile. The model is provided as an example in [17]. The model represents a business process of a purchasing service provider. When providing the service to its user, the business process interacts with a shipping service provider, an invoice service provider, and a scheduling service provider.

5.4.1 Data Types and Message Types

Figure 5.4 shows models of data types and message types. Data types *PurchaseOrder*, *CustomerInfo*, *ScheduleInfo*, and *Invoice* are represented by classes stereotyped as <<data>>. Message types *PO*, *ShippingRequest*, and *OrderFault* are also represented by classes but stereotyped as <<messageContent>>.

![Figure 5.4: Data types and message types of the business process](image)

A message part is represented by an aggregation association. The role name of the aggregated class represents the name of the message part. For instance, a message *PO* has a part *purchaseOrder* whose data type is *PurchaseOrder*. Alternatively, if a message part is a primitive data type, the message part can be represented as an attribute. For instance, a message *OrderFault* has a part *problemInfo* whose data type is *String*.

5.4.2 Protocol

Figure 5.5 shows one of the protocols used by the business process to interact with the invoice service provider. The protocol specifies two roles: *InvoiceService* and *InvoiceRequester*. Roles are represented by classes stereotyped as <<role>>. A role may provide a service, use a service provided by the other role, or both. An interface
is used to represent a service provided by a role. The interface lists its contained operations.

For instance, the role InvoiceService provides service ComputePriceService that consists of operations initiatePriceCalculation and sendShippingPrice. At the same time, the role uses service InvoiceCallback provided by the role InvoiceRequester.

Notice that the model of the protocol conforms to the composition rule of the BPEL interaction protocol as depicted in Figure 5.2.

5.4.3 Business Process

Figure 5.6 shows the definition of the business process. The business process is represented as a class stereotyped as <<process>>. The attributes of the class e.g. po, invoice, and poFault represent states of the process.

The business process implements four roles: PurchaseService, InvoiceRequester, ShippingRequester, and SchedulingRequester. Aggregations stereotyped as <<port>> are used to associate the business process with the roles. The role names of the <<role>> classes indicate partners with which the business process interacts. For instance, the business process plays a role InvoiceRequester when it interacts with a partner invoiceProvider.

Notice that the model of the business process definition conforms to the composition rule of the BPEL process definition as depicted in Figure 5.3.

5.4.4 Business Process Behavior

Figure 5.7 shows activities that constitute the behavior of the business process. The business process interacts with four partners represented as swimlanes customer, shippingProvider, invoiceProvider, and schedulingProvider. Activities are stereotyped
as BPEL activities e.g. <<receive>>, <<invoke>>, <<assign>>, and <<reply>>. Arrows indicate the ordering of the activities.

Each activity has one or more operations to be performed when the activity is executed, but they are not shown in the figure. For instance, the activity computePrice in a swimlane invoiceProvider performs an operation initiatePriceCalculation(po).

5.4.5 Package Dependency

Figure 5.8 shows package dependencies within the model. Some of them i.e. Invoicing, Purchasing, Scheduling, and Shipping are stereotyped as <<protocol>> indicating that they contain protocol definitions. Those packages also represent partner link types.

Package PurchaseDatatypes contains definitions of data types and message types as depicted in Figure 5.4. Package Purchase contains definitions of interfaces but it does not contain any diagram. Package PurchaseOrderProcess contains the definition of the business process and its behavior as depicted in Figures 5.6 and 5.7.
Figure 5.7: Behavior of the business process
5.5 Lessons Learned

From our survey of the UML-to-BPEL transformation, we obtain some useful ideas for developing a model transformation, in particular the ISDL-to-BPEL transformation. The ideas are as follows.

5.5.1 To use standardized formats

The use of standardized formats allows a transformation tool to be independent from any modeling editor. Since the transformation tool only deals with files in standard formats, this allows one to use its favorite modeling editor to create models of business processes.

The UML2BPEL plug-in uses the XMI specification as a standard format for inputs. The specification is supported by several UML modeling editors, e.g. Rose, Poseidon for UML [15], XDE [20], and ArcStyler [21]. Theoretically, since we use Rose alone, XMI files produced by those modeling editors can be transformed into artifacts of BPEL processes using the UML2BPEL plug-in.
5.5.2 To generate all artifacts of a business process

A BPEL process needs at least two artifacts. These are a BPEL process definition (BPEL file) and a web service definition (WSDL file). A separated XML schema (XSD file) is optional because its content can be put into the WSDL file.

Although its name leads us to think that the UML2BPEL plug-in generates BPEL files only, the plug-in also generates WSDL files and XSD files. The generation of WSDL and XSD files is necessary since definitions of business processes require information defined in those files. This information consists of available port types, operations of each port type, message structures, and data types. Furthermore, a BPEL process must add extra information to the WSDL file i.e. partner link types, properties, and property aliases.

Generating all artifacts of a business process at once gives us two benefits. First, it minimizes developers involvement because all necessary artifacts are generated. Second, all artifacts will be in synchronization. For instance, when the structure of a message is modified, all generated artifacts will be adjusted appropriately.

5.5.3 To provide a language profile

The UML Profile for BPEL acts as a guide to model business processes that can be automatically transformed into BPEL documents. It consists of stereotypes, identified diagrams, and composition rules. The profile makes the UML suitable for modeling BPEL process because concepts and constructs within BPEL now can be represented by using the UML.

Elements of the modeling language are usually semantically richer than constructs of the implementation technology. A language profile is useful to restrict or to specialize semantics of the modeling elements such that they match the semantics of the implementation constructs. A profile may also provide composition rules. Such rules are necessary when an implementation construct cannot be modeled with a single modeling element. Those rules introduce consistent compositions that make a transformation specification simpler. However, models that conform to a profile are specific for an implementation technology for which the profile is defined.

5.5.4 To map the target technology onto the modeling language

Modeling languages are typically independent from any implementation technology. Such languages view systems being modeled as conceptual systems. Both the UML and the ISDL can be considered as technology independent languages.

It would be difficult to develop mappings from modeling elements of such a language to implementation constructs. This is because the implementation technology may not support some concepts defined in the modeling language. For instance, the ISDL concept of synchronization cannot be mapped directly onto any BPEL construct. It may be mapped to a composition of BPEL constructs, but it will be very complex.
Therefore, it will be more effective to establish mappings from implementation constructs to modeling elements. Based on the mappings, we can develop a language profile for the implementation technology. Furthermore, this approach ensures that all constructs and concepts defined in the implementation technology are accommodated in the profile.

Figure 5.9 illustrates the idea derived from lessons in Subsections 5.5.3 and 5.5.4. A language profile is a specialization of a modeling language that implements concepts within an implementation technology. We can develop a language profile by mapping implementation constructs onto elements or composition of elements of the modeling language. Typically, modeling elements have to be specialized.

Based on the specialized elements or compositions of elements in the profile, we can develop a transformation specification. Specialized elements usually introduce one-to-one mappings that ease us to define transformation rules. Since the profile defines consistent composition rules, the number of mapping rules within the transformation specification is limited.

Figure 5.9: Development of language profile and transformation specification
Chapter 6

WSDL and BPEL Metamodels

This chapter presents metamodels for the WSDL 1.1 [42] and the BPEL 1.1 [4]. Section 6.1 describes our motivation to develop the metamodels. Section 6.2 explains our metamodeling approach. Sections 6.3, 6.4, 6.5, and 6.6 present the produced metamodels together with their informal syntax and a brief explanation. Finally, Section 6.7 points out related work and argues that our work is different and necessary.

6.1 Motivation

Two motivations drive us to develop metamodels for the WSDL and the BPEL. First, we need target metamodels to develop the transformation specification of the ISDL-to-BPEL transformation. The metamodels are used to represent abstract syntax of the WSDL and the BPEL. The specification is then used by a transformation engine, which is a piece of software running on a computer, to transform a source model into a target model. Thus, the metamodels are for machine understanding.

Second, we need to understand concepts contained in the WSDL and the BPEL. Those concepts can be described by metamodels. To be able to define a transformation specification between metamodels, we have to understand well the concepts in the source and the target languages. Thus, the metamodels are for human understanding.

The motivation seems to be contradictory. The first objective requires us to preserve the syntactic details of the languages as much as possible. The second objective suggests us to abstract from those details as long as their semantics are kept. To resolve the contradiction, we prioritize the objectives as follows: the first objective must be achieved; and whenever it is possible, the second objective should be achieved.

6.2 Approaches

We define two approaches to be taken: an approach for producing metamodels and an approach for structuring the metamodels.
6.2.1 Producing Metamodels

The XMI Specification version 2.0 describes three mapping algorithms to produce MOF declarations from different kinds of XML-related documents: XML documents, Document Type Definitions (DTDs), and XML schemas. Since the WSDL and the BPEL are based on XML standards, we can use those algorithms to produce metamodels for the WSDL and the BPEL. The MOF declarations produced can be represented in UML using the UML Profile for MOF.

Our metamodeling approach is based on the algorithm to produce MOF declarations from XML documents. However, instead of using XML documents i.e. WSDL and BPEL documents as input, we use the WSDL and the BPEL informal syntax. The informal syntax that is an integral part of the specifications appears as a document template. As a template, it shows all allowed elements and their occurrence indicators, which allow it to be used to produce a precise metamodel. Rules to build the informal syntax are given in Appendix A.

Before we present our mapping algorithm to produce MOF declarations from the informal syntax, we describe the algorithm to produce MOF declarations from XML documents.

XML to MOF

Algorithms for producing MOF declarations described by OMG consist of two parts: rules and parameterized mappings. The rules extract declarations of classes and attributes from their representations in XML-related documents. Each algorithm has its own rules. The parameterized mappings produce the simplest MOF classes and attributes that can be customized by an implementation. The parameterized mappings are used by all algorithms.

Steps within the algorithm to produce MOF declarations from an XML document are as follows.

1. Parse the XML document into a Document Object Model (DOM) tree.
   A DOM tree presents an XML document as a hierarchy of nodes. Type of nodes that are considered by the algorithm are elements, attributes, texts, character data, and CDATA section. Other types are ignored.

2. Select an existing metamodel or create an empty metamodel.

3. Perform a depth-first traversal of the XML document’s DOM tree.
   At each node, apply the appropriate generalization rule from Table 6.1 based on the type of parent and child nodes encountered. The rule may be customized by setting the parameters in Table 6.2.

The following is an example to give a hint on how to use the rules and the parameterized mappings. Suppose we encounter a node of element E during the
traversal. Rule 1 is applied because the rule does not require any lookup into the
node’s child i.e. child node is none. As a result, a class \( E \) is declared. If the element
has an attribute \( A \), the attribute appear as a child node in the DOM tree. Rule 2 is
then applied. As a result, an attribute \( A \) is declared. Further, the attribute declaration
can be customized with parameterized mappings to specify its type and multiplicity
using the parameters \( \text{AttributeType} \) and \( \text{AttributeMult} \), respectively.

**Informal Syntax to MOF**

Informal syntax appears as a document template that shows all possible elements and
rules to construct a document e.g. alternatives and occurrence indicators. This allows
us to model the languages completely and correctly. On the other hand, if we base on
XML documents, we possibly can not model the language completely, because XML
documents contain only elements that they need.

Since we use informal syntax instead of XML documents, we modify the mapping
algorithm for producing MOF declarations from XML document. The following steps
are identified within our modified algorithm.

1. Parse the informal syntax into a DOM tree.
   * Occurrence indicators are kept in their corresponding nodes.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Parent Node</th>
<th>Child Node</th>
<th>MOF Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Element ( E )</td>
<td>None</td>
<td>Class ( E ) with ( \text{Supertype}(E) )</td>
</tr>
<tr>
<td>2</td>
<td>Element ( E )</td>
<td>Attribute ( A )</td>
<td>Attribute named ( A ) of Class ( E ) with type ( \text{AttributeType}(E, A, \text{&quot;CDATA&quot;}) ) and multiplicity ( \text{AttributeMult}(E, A, \text{&quot;#IMPLIED&quot;}) )</td>
</tr>
<tr>
<td>3</td>
<td>Element ( E )</td>
<td>Element ( F )</td>
<td>( \text{TypedElement}(E,F) ) attribute or association to Class ( F ) and name ( \text{RoleName}(E,F) )</td>
</tr>
<tr>
<td>4</td>
<td>Element ( E )</td>
<td>Text</td>
<td>Attribute named ( \text{Name}(E) ) of type ( \text{AttributeType}(E, \text{Name}(E)) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Supertype}(\text{Element name}) )</td>
<td>Node</td>
</tr>
<tr>
<td>( \text{AttributeType} ) (\text{Element name, Attribute name, Type name})</td>
<td>String for type CDATA; Lookup MOF type for IDREF</td>
</tr>
<tr>
<td>( \text{AttributeMult} ) (\text{Element name, Attribute name, Occurs style})</td>
<td>0..1</td>
</tr>
<tr>
<td>( \text{TypedElement} ) (\text{Element name, TypedElement name})</td>
<td>Association: containment by value, multiplicity 0..<em>, one way navigable; Attribute: multiplicity 0..</em></td>
</tr>
<tr>
<td>( \text{RoleName} ) (\text{Element name, TypedElement name})</td>
<td>LowerCase TypedElement name</td>
</tr>
<tr>
<td>( \text{Name}(\text{Element name}) )</td>
<td>“value”</td>
</tr>
</tbody>
</table>

Table 6.1: Rules for extracting MOF definitions from an XML document

Table 6.2: Parameterized mappings
• Alternatives are all modeled.
• Placeholders for elements are modeled as elements.
• Irrelevance indicators and namespace prefixes are omitted.

2. Select an existing metamodel or create an empty metamodel.

3. Perform a depth-first traversal of the XML document’s DOM tree.
   At each node, apply the appropriate generalization operation from Table 6.3 based on the type of parent and child nodes encountered.

As shown in Table 6.3, the rules are modified to accommodate the difference between informal syntax and XML documents. Rather than using parameterized mappings, we incorporate them into our rules.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Parent Node</th>
<th>Child Node</th>
<th>MOF Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Element E</td>
<td>None</td>
<td>Class E</td>
</tr>
<tr>
<td>2</td>
<td>Element E</td>
<td>Attribute A</td>
<td>Attribute named A of Class E with type attribute value and multiplicity occurrence indicator</td>
</tr>
<tr>
<td>3</td>
<td>Element E</td>
<td>Element F</td>
<td>Aggregation from Class E to Class F with multiplicity occurrence indicator for Class F</td>
</tr>
</tbody>
</table>

Table 6.3: Rules for producing MOF declarations from informal syntax

For example, we are given an informal syntax in Listing 6.1. Its DOM tree is shown in Figure 6.1.

Listing 6.1: Example of informal syntax

```xml
<wsdl:message name="nmtoken">
  <part name="nmtoken" element="qname"? type="qname"?>/ *
</wsdl:message>
```

Figure 6.1: DOM tree of Listing 6.1

45
When we traverse the DOM tree, we first encounter a node of element `message`. Rule 1 is applied that results in a class `Message`. The node has child nodes of attribute `name` and element `part`. Rule 2 is applied for the first child; and the class `Message` has an attribute `name` with type `nmtoken`. Rule 3 is then applied for the next child; and a class `Part` is declared and it is aggregated by the class `Message`. Multiplicity of the class `Part` in the aggregation is zero or more. The traversal is then continued. The result, shown in UML, is given in Figure 6.2.

![UML representation of Listing 6.1](image)

**Figure 6.2: UML representation of Listing 6.1**

**Semantics Consideration**

The informal syntax is merely for constructing a document syntactically. As one of our motivation is to facilitate human understanding, we have to consider semantics of the language constructs and represent it in metamodels produced by our algorithm.

An element may refer to another element. In an XML document, such a reference is represented by values of attributes of the elements. Those values do not appear in the informal syntax, but we can infer which attributes are provided for the values. In a metamodel, the reference should be represented by an association from a class to another class.

If, in informal syntax, an element $E$ uses an attribute $A$ to refer to another element $F$; then, in its metamodel, the attribute $A$ of a class $E$ is removed and a one-way association from the class $E$ to a class $F$ is drawn. The attribute’s name is used to name the role of the class $F$. The attribute’s multiplicity becomes the multiplicity of the class $F$.

For example, Listing 6.2 gives a fragment of the WSDL informal syntax. A metamodel without semantics consideration is shown in Figure 6.3. However, in a WSDL document, attribute `message` of element `input` of an element `operation` is used to refer to an element `message`. Therefore, in its metamodel, the attribute `message` of the class `Input` is removed and an one-way association from the class `Input` to the class `Message` is drawn. This also applies for attributes `element` and `type` of element `part` of element `message`. The modified metamodel is shown in Figure 6.4.
Listing 6.2: Example of implicit association

```xml
<wSDL:message name="nmtoken"> *
  <part name="nmtoken" element="qname"? type="qname"?/> *
</wSDL:message>

<wSDL:operation name="nmtoken">*
  <wSDL:input name="nmtoken" message="qname"?>/?
</wSDL:input>
```

Figure 6.3: Metamodel without semantics consideration

Figure 6.4: Metamodel with semantics consideration

Simplification of Successive Aggregations

We may simplify successive aggregations by eliminating intermediate aggregating classes. The name of an intermediate class becomes the role of the classes it aggregates. The multiplicity of the aggregated class is now a semantic combination
of the multiplicity of the intermediate aggregating class and the multiplicity of the aggregated class. This simplification is also taken in [8, 12].

For example, Listing 6.3 gives a fragment of the BPEL informal syntax. Its metamodel is shown in Figure 6.5. After eliminating the intermediate aggregating class, the new metamodel is shown in Figure 6.6.

Listing 6.3: Example of a collection element

```xml
<bpws:process ...>
    <bpws:partnerLinks>? ...
        <bpws:partnerLink ...>+</bpws:partnerLink>
    </bpws:partnerLinks>
</bpws:process>
```

Figure 6.5: Metamodel with an intermediate aggregating class

Figure 6.6: Metamodel without the intermediate aggregating class

Abstract Classes

We may introduce an abstract class for similar classes. This is to simplify the metamodel, such that the metamodel is more understandable. A complex metamodel typically has many relationships among its classes. They potentially make the metamodel difficult to understand. By introducing abstract classes, some relationships can be handled by the abstract classes. Classes inherited from the abstract classes do not need to handle the relationships individually.

6.2.2 Structuring Metamodels

We structure the metamodels into four packages: schema, WSDL, BPEL Extension, and BPEL. The package schema contains abstract classes and their relationships within an XML schema. We model the schema because the WSDL and the BPEL use XML schemas to define the structure of messages. The packages WSDL and the BPEL contain classes and their relationships in metamodels for the WSDL and the BPEL, respectively.
A WSDL document may contain non-WSDL elements. Such elements are defined as extensibility elements. The BPEL provide some extensibility elements that are necessary to establish a conversation between BPEL processes. We call the elements BPEL extension for WSDL. We model the BPEL extension for WSDL in a separated package called BPEL_Extension.

We model the metamodel dependencies as import relationship between packages. Figure 6.7 shows the package dependencies. In the metamodels, to indicate classes that are imported from other packages, the classes are colored gray and have information below their names that indicate what package the classes are from.

![Figure 6.7: Package dependencies](image)

### 6.3 Schema Metamodel

The WSDL and the BPEL use XML Schemas to define data types and elements used in BPEL processes. However, the schemas are not part of WSDL and BPEL. WSDL and BPEL documents may import schemas. A WSDL document may define its own schema for its messages.

To understand how a schema is structured, we present a metamodel for schemas. However, precise metamodeling of XML schemas is beyond the scope of our work. We only present necessary concepts that are used by our metamodels for the WSDL and the BPEL.

A schema must conform to the XML Schema specification [45]. For such a relationship, we say that a schema is an instance of the XML Schema specification. A schema may be defined for a target namespace to give a globally unique scope of the elements defined in the schema. A schema may consist of types and elements. An element is associated with a type that indicates what can be contained in the element i.e. attributes and contents. A type represents any data type. It can then be specialized into simple types or complex types. A simple type represents a primitive data type such
as int, boolean, string, ncname, and nmtoken. A complex type may define attributes and elements. Figure 6.8 shows a metamodel for schemas.

Figure 6.8: Metamodel for schemas

### 6.4 WSDL Metamodel

The informal syntax of the WSDL’s abstract part is shown in Listing 6.4. Note that we model WSDL 1.1 because BPEL 1.1 uses that version.

Listing 6.4: Informal syntax of the WSDL’s abstract part

```xml
<wsdl:definitions name="nmtoken"? targetNamespace="anyURI"?/>
<wsdl:types>
  <xsd:schema .... />
</wsdl:types>

<wsdl:message name="nmtoken"> *
  <part name="nmtoken" element="qname"? type="qname"/> *
</wsdl:message>

<wsdl:portType name="nmtoken">*
  <wsdl:operation name="nmtoken">*
    <wsdl:input name="nmtoken"? message="qname" />
    <wsdl:output name="nmtoken"? message="qname" />
    <wsdl:fault name="nmtoken" message="qname" /> *
  </wsdl:operation>
</wsdl:portType>
</wsdl:definitions>
```
A WSDL document is a set of definitions. The document may be given a name and associated with a target namespace. The abstract part of a WSDL document may consist of three kinds of definitions. These are types, message, and port type. Although all of them are optional for reuse purposes, a description of a Web service must have at least definitions of a message and a port type.

A message type is given a name that must be unique among all other message types defined within the namespace of a WSDL document. A message type is structured into one or more parts that represent logical units within the message type. Each part is associated with either a data type or an element defined in an XML schema.

A port type definition defines a set of related operations. A port type is given a name that must be unique among all other port types defined within the namespace of a WSDL document. More than one operation can be defined within a port type, but an operation’s name must be unique among all other operations defined within a port type. An operation may have an input message, an output message, and fault messages. Each message may be given a name that is unique among all other messages defined within its port type. A message is associated with a message type.

An operation can be either a one-way, a request-response, a solicit-response, or a notification operation. Table 6.4 lists messages that have to be declared for each operation type. The order of the messages is important.

<table>
<thead>
<tr>
<th>Operation type</th>
<th>Messages to be declared</th>
</tr>
</thead>
<tbody>
<tr>
<td>one-way operation</td>
<td>an input message only</td>
</tr>
<tr>
<td>request-response operation</td>
<td>an input message,</td>
</tr>
<tr>
<td></td>
<td>an output message,</td>
</tr>
<tr>
<td></td>
<td>optional fault messages</td>
</tr>
<tr>
<td>solicit-response operation</td>
<td>an output message,</td>
</tr>
<tr>
<td></td>
<td>an input message,</td>
</tr>
<tr>
<td></td>
<td>optional fault messages</td>
</tr>
<tr>
<td>notification operation</td>
<td>an output message only</td>
</tr>
</tbody>
</table>

Table 6.4: Operation types and their messages that can be defined

The request-response and the solicit-response operations can be used in Remote Procedure Call (RPC)-like bindings where the order of parameters is important to call the operations. For this, a parameter order may be specified to list message part names. Its order reflects the order of parameters in the RPC signature.

Listing 6.5 gives a hint where extensibility elements can be defined within WSDL documents. We only model extensibility elements where BPEL extensions for WSDL should be placed.

Listing 6.5: Informal syntax of non-WSDL elements within WSDL document

```xml
<wSDL:definitions ...>
  <import namespace="anyURI" location="anyURI"/>
  ...
  <!-- extensibility element --> *
</wSDL:definitions>
```
In addition, a WSDL document can import the elements of other WDSL documents. This allows a WSDL document reuse types, messages, port types, and operations that are defined in the other WSDL documents. When importing a WSDL document, its namespace and location must be specified. Figure 6.9 shows aforementioned concepts and syntax in a metamodel.

### 6.5 BPEL Extension Metamodel

As mentioned previously, WSDL can contain any other elements as extensibility elements. BPEL extensions come in place of extensibility elements. The informal syntax of BPEL extensions for WSDL is shown in Listing 6.6.

Listing 6.6: Informal syntax of BPEL extension for WSDL

```xml
<wsdl:definitions ...>
  ...
  <plnk:partnerLinkType name="ncname">
    <plnk:role name="ncname">
      <plnk:portType name="qname"/>
    </plnk:role>
    <plnk:role name="ncname">
      <plnk:portType name="qname"/>
    </plnk:role>
  </plnk:partnerLinkType>

  <bpws:property name="ncname" type="qname"/>
...
</wsdl:definitions>
```

A **BPEL extension** element is also a definition. A BPEL extension can be a **partner link type**, a **property**, or a **property alias** definition.

A partner link type definition represents a conversational relationship. It defines **roles** played by parties involved in a conversation. A partner link type and its roles are given **names**. A role is associated with a **port type** defined in a WSDL document.

A property definition creates a globally unique **name**. A property has a **type** that is a XML Schema simple type to convey information in the property.

A property alias definition maps a property to a field in a message type. For this, a property alias keeps a record of a **property name**, a **message type**, and a **part** of a message type. A **query** points to a field within a message part that is used as a property. Since a property is a simple type, then the field pointed by the query must be a simple type as well. The query is expressed in XPath [40].

Figure 6.10 shows aforementioned concepts and syntax in a metamodel.
Figure 6.9: Metamodel for the WSDL
6.6 BPEL Metamodel

A BPEL process uses information defined in a WSDL document. Hence, our metamodel for the BPEL uses concepts in the metamodels for the WSDL and the BPEL extension defined previously. Since the BPEL is complex, we split the description and the metamodel into several subsections. Concepts and metamodels in the subsections belong to the same package i.e. the BPEL package.

6.6.1 Process Definition

The informal syntax of BPEL process definitions is shown in Listing 6.7. We defer some details to the next subsections.

Listing 6.7: Informal syntax of BPEL process definition

```xml
<bpws:process name="ncname" targetNamespace="anyURI"
  queryLanguage="anyURI"?
  expressionLanguage="anyURI"?
  suppressJoinFailure="yes|no"?
  enableInstanceCompensation="yes|no"?
  abstractProcess="yes|no"?>

<bpws:partnerLinks>?
  <bpws:partnerLink name="ncname" partnerLinkType="qname"
    myRole="ncname"? partnerRole="ncname"?/>
</bpws:partnerLink>
```
Firstly, we introduce an abstract concept namely *scoping element* to handle similarities between processes and scopes. Scopes will be described in Subsection 6.6.3. The scoping element handles the definitions of variables, correlation sets, fault handlers, a compensation handler, event handlers, and activities.

A process extends a scoping element. The process is given a name and a target namespace. A BPEL process may need languages for querying nodes of XML documents and for making boolean and time-related expressions. The BPEL assumes such languages are available. The attributes *query language* and *expression language* specify the languages to be used. The default language is XPath 1.0 [40].

A standard fault called *join failure* may be thrown during a synchronization of two activities in a BPEL process. However, this fault can be suppressed. Instead of suppressing it in each activity, the suppression can be applied globally for all activities within the process. The attribute *suppress join failure* is used to indicate that.

To maintain data integrity, a BPEL process can compensate a failed transaction. The compensation can be defined in a BPEL process or be handled by platform-specific means. The attribute *enable instance compensation* determines whether the process instance is compensated by platform-specific means.

A BPEL process can be defined as an abstract process or an executable process. To indicate this, the attribute *abstract process* is used.
A BPEL process specifies partner links to model conversational relationships with its partners. A partner link is given a name and is associated with a partner link type that is defined as a BPEL extension in a WSDL document. A partner link specifies roles played by entities in a partnership. A role specified in the attribute my role indicates the role of the BPEL process being defined, while the attribute partner role indicates the role of the partner.

A process may have partners to represent a logical relationship that consists of one or more conversational relationships. A partner is given a name and is associated with one or more partner links. A partner introduce a constraint to the partner links such that a partner link that is associated to the partner should not be associated to another partner. However, a partner links does not need to be associated to any partners.

Within a scoping element, variables can be defined. A variable has a name that must be unique within its scoping element. A variable is associated with either an element defined in a schema, a type of XML Schema’s simple type, or a message defined in a WSDL document.

A scoping element may have correlation sets. A correlation set is identified by a name and is associated with one or more properties defined as BPEL extensions in a WSDL document.

Descriptions of the other elements i.e. a compensation handler, event handlers, fault handlers, and activities are deferred to next subsections.

Figure 6.11 shows aforementioned concepts and syntax in a metamodel. Note that we model the enumeration of yes and no as boolean because they have a same semantics.

6.6.2 Basic Activities

An activity may have standard attributes and standard elements. The standard attributes are shown in Listing 6.8, whilst the informal syntax of the standard elements is shown in Listing 6.9.

Listing 6.8: Standard attributes of BPEL activities

```xml
name="ncname"?
joinCondition="bool-expr"?
suppressJoinFailure="yes|no"?
```

The attribute name gives a name to an activity. The attribute join condition specifies how concurrent activities should be joined when they end up on a same target activity. For instance, the expression of a join condition indicates that a target activity can only be performed if all concurrent activities are completed. The attribute suppress join failure indicates whether a join failure fault should be suppressed if it occurs.
Figure 6.11: Metamodel for BPEL process definition
A link to synchronize two activities is identified by its *link name*. The element *source* is defined in the activity that becomes the synchronization source, and the element *target* is defined in the activity that becomes the synchronization target. An activity may be a synchronization source of more than one target activity, and an activity may be a synchronization target of more than one source activity. However, a link has exactly one source activity and one target activity. Both activities have the same link name.

A synchronization may be conditional. The condition is expressed as a *transition condition*. For instance, if the condition is not satisfied, then the synchronization is not performed. When the condition is omitted, the transition will always be performed.

The informal syntax of BPEL basic activities is shown in Listing 6.10. We describe the BPEL structured activities in Subsection 6.6.3.

Listing 6.10: Informal syntax of BPEL basic activities

```xml
<invoke partnerLink="ncname" portType="qname" operation="ncname"
         inputVariable="ncname"? outputVariable="ncname"?
         standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="ncname" initiate="yes|no"?
                pattern="in|out|out-in"/>?
  </correlations>
  <catch faultName="qname" faultVariable="ncname"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
  <compensationHandler>?
    activity
  </compensationHandler>
</invoke>

<receive partnerLink="ncname" portType="qname" operation="ncname"
         variable="ncname"? createInstance="yes|no"?
         standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="ncname" initiate="yes|no"/>?
  </correlations>
</receive>

<reply partnerLink="ncname" portType="qname" operation="ncname"
        variable="ncname"? faultName="qname"?
        standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="ncname" initiate="yes|no"/>?
  </correlations>
</reply>
```
Firstly, we model the abstract concept of *activity* where all BPEL activities are derived from. Its purpose is to define the standard attributes and the standard elements. We also introduce an abstract concept named *interaction activity* to handle similarities among *invoke*, *receive*, and *reply* activities.

An *interaction activity* is associated with a *partner link* in which its interaction is carried out. The activity also specifies the *port type* and the *operation* of the partner link that are to be performed by the activity. The activity may contain one or more *correlations*. Each correlation identifies a logical part within a conversation. A correlation is associated with one or more *correlation sets*. When a BPEL process wants to initialize its correlation, it sets the attribute *initiate* to *yes*.

We include the attribute *pattern* within the correlation, but it is only applicable for the *invoke* activity. Thus, we make it optional. When performing a request-response operation, an *invoke* activity sends and receives messages. The attribute pattern is used to determine in which direction the correlation is applied.

All activities derived from the *interaction activity* are associated with one or more variables in different ways. Since an *invoke* activity can send and receive messages, it can have an *input variable* and an *output variable*. A *receive* activity is only for receiving an input message, thus it only has a *variable* for receiving a message. A *reply* activity sends messages that can be output messages or fault messages. To handle a message to be sent, the activity has a *variable*. When it sends a fault message, a *fault name* is specified and its *variable* contains the fault message.
Figure 6.12: Metamodel for BPEL basic activities
A BPEL process can only be instantiated when a message is received. However, an incoming message does not always lead to a process instantiation. The attribute `create instance` indicates whether a receive activity can instantiate a process instance.

An invoke activity may have `fault handlers` and a `compensation handler` to handle fault messages that may be received. The handlers are useful, for instance, to abort an ongoing transaction when a partner indicates unavailability to proceed the transaction.

An `assign` activity updates the values of variables by making a `copy` of a value specified or contained in the attribute `from` to the target variable in the attribute `to`. A `throw` activity generates a service-level fault. The fault is given a `fault name` and may be associated with a `fault variable` that carries information necessary for handling the fault. A `wait` activity waits either `for` a given time period or `until` a certain time has passed. An `empty` activity does nothing.

A `terminate` activity immediately terminates the process instance where the activity is performed. Actually this activity is not classified as a basic activity in the BPEL specification. We include it here because it performs a certain action.

Figure 6.12 shows aforementioned concepts and syntax in a metamodel.

### 6.6.3 Structured Activities

The informal syntax of BPEL structured activities is shown in Listing 6.11.

Listing 6.11: Informal syntax of BPEL structured activities

```xml
<sequence standard-attributes>
  standard-elements
  activity +
</sequence>

<flow standard-attributes>
  standard-elements
  <links>? 
    <link name="ncname">+
  </links>
  activity +
</flow>

<switch standard-attributes>
  standard-elements
  <case condition="bool-expr">+
    activity
  </case>
  <otherwise>? 
    activity
  </otherwise>
</switch>

<while condition="bool-expr" standard-attributes>
  standard-elements
  activity
</while>

<pick createInstance="yes|no"? standard-attributes>
```

61
Structured activities are also derived from the abstract concept of activity. A sequence activity orders one or more activities to be performed sequentially, whereas a flow activity specifies one or more activities to be performed concurrently. Concurrent activities are synchronized using links as described in Subsection 6.6.2.

A switch activity allows a business process to have conditional behaviors. A conditional behavior is specified as a case with a condition. A default branch otherwise can be specified to handle an undefined condition.

A while activity performs one or more activities repeatedly as long as a certain condition is satisfied.

A pick activity waits for an event that can be the arrival of a message or the expiration of an alarm. They are handled by an activity defined in onMessage and onAlarm, respectively. A pick activity can be used for instantiating a process instance by setting the attribute create instance to yes. When a pick activity is allowed to create a process instance, no alarm is permitted to be defined. At least one onMessage event handler must be defined.
Figure 6.13: Metamodel for BPEL structured activities
A scope activity gives a context for one or more activities. Since it is derived from the scoping element, it has variables, correlation sets, event handlers, fault handlers, and a compensation handler. Activities within a scope are performed sequentially.

A scope can be a serializable scope by setting the attribute variable access serializable to yes. In the BPEL specification, the scope activity actually is not classified as a structured activity. We include it here because it applies some controls to other activities.

A compensate activity is used to invoke a compensation handler. The handler to be invoked is indicated by the name of the scope that defines the handler. In case the handler is defined within an invoke activity, the name of the invoke activity is used to indicate the handler. In the BPEL specification, this activity is also not classified as a structured activity.

Figure 6.13 shows aforementioned concepts and syntax in a metamodel.

6.6.4 Event, Fault, and Compensation Handlers

Events are handled by event handlers called onMessage and onAlarm. The informal syntax of BPEL event handlers is shown in Listing 6.12.

Listing 6.12: Informal syntax of BPEL event handlers

```
<bpws: onMessage partnerLink="ncname" portType="qname" 
operation="ncname" variable="ncname" ?>
  <bpws:correlations>?
    <bpws:correlation set="ncname" initiate="yes|no">+
  </bpws:correlations>
  activity
</bpws: onMessage>

<bpws: onAlarm for="duration - expr"? until="deadline - expr" ?>
  activity
</bpws: onAlarm>
```

We introduce an abstract concept named event handler to represent both onMessage and onAlarm event handlers. It is then specialized into onMessage and onAlarm. An onMessage event handler is semantically equivalent to a receive activity. Syntactically they are quite similar.

The condition for an onAlarm event to occur is either after waiting for a time period or until a certain time moment has elapsed. An event must exactly have one of these conditions.

Fault handlers and compensation handlers enclose an activity to be performed when its corresponding abnormal condition happens. The informal syntax of BPEL fault and compensation handlers is shown in Listing 6.13.
Listing 6.13: Informal syntax of BPEL fault and compensation handlers

```
<bpws:catch faultName="qname"? faultVariable="ncname"?>* 
  activity 
</bpws:catch>
<bpws:catchAll>? 
  activity 
</bpws:catchAll>
<bpws:compensationHandler>
  activity 
</bpws:compensationHandler>
```

We introduce an abstract concept named fault handler to represent both catch and catchAll. A catch can only handle a fault whose name matches with the fault name specified in the catch. A fault variable can be specified to receive information brought by the fault. A general catch can be defined to catch all faults that cannot be caught individually. Such a catch can only be defined once within its scope.

Figure 6.14 shows aforementioned concepts and syntax in a metamodel.

![Metamodel for BPEL event, fault, and compensation handlers](image)

Figure 6.14: Metamodel for BPEL event, fault, and compensation handlers
6.7 Related Work

This chapter describes two things: (i) our approach to produce metamodels from XML-related documents and (ii) metamodels for the WSDL and the BPEL. We are aware that work has been done on those things. Here we argue that our work is different and necessary.

Approach to produce metamodels

The WSDL 1.1 and the BPEL 1.1 specifications include their XML schemas. However, instead of using the algorithm to produce metamodels from XML schemas introduced by the OMG [34], we propose an algorithm to produce metamodels from the informal syntax of the specifications.

An XML schema is intended to be used by XML parsers for validating its corresponding XML documents. Thus, the XML schema contains all details that are needed for machine understanding. Our second motivation mentioned in Section 6.1 brings an idea of abstracting syntactic details such that we are able to understand concepts in the languages without being too distracted by the syntactic details. We found that such an idea is already provided in the informal syntax used by the WSDL and the BPEL specifications. We consider that informal syntax is intuitively more understandable, without losing syntactic details, than XML schemas.

Furthermore, metamodels to be used for defining the transformation specification of a model transformation should represent the semantics and the syntax of the source and the target languages of the transformation. This leads us to extend the algorithm by some consideration on the semantics of the language being modeled. Such consideration cannot be found in the algorithms to produce metamodels from XML-related documents introduced by the OMG.

Metamodels for the WSDL and the BPEL

Work has been done to develop the metamodels of WSDL 1.1 [5, 7, 27] and BPEL 1.1 [8, 12]. All the metamodels are produced without semantic consideration as in our approach. Moreover, they do not represent the language’s syntax completely, e.g. data types for attributes and multiplicity of attributes or classes. Therefore, we cannot use them to define a transformation specification accurately.

A tool called hyperModel [35] can generates UML models from XML schemas. We can use the tool to generate UML models from the XML schemas of the WSDL and the BPEL. However, some manual refinements are still necessary before one may accept the UML models as the metamodels of the languages represented by the XML schemas [7]. Moreover, the tool does not consider the semantics of a language being modeled.
Chapter 7

ISDL Profile for BPEL 1.1

This chapter describes our ISDL profile for BPEL 1.1. Section 7.1 describes our motivation for developing the profile. Section 7.2 proposes a profiling mechanism for the ISDL. Sections 7.3 to 7.7 describe how the profile is developed and present notation and shorthands introduced by the profile. The profile does not yet include the BPEL properties and correlation sets.

7.1 Motivation

We indicate in Chapters 4 and 5 that a language profile is useful to enable model transformation. This motivates us to develop an ISDL profile that should be used to model business processes to be implemented in the BPEL. The profile enables a model transformation from the ISDL to the BPEL by applying two general restrictions. First, an ISDL model should not contain elements or compositions of elements that cannot be mapped onto the BPEL elements. Second, an ISDL model should be structured in a standard way such that a mapping of ISDL elements onto BPEL constructs is possible. The mapping is preferably as one-to-one mapping because such a mapping eases us to define transformation rules from ISDL elements to the BPEL constructs.

To accommodate the restrictions, the profile provides rules for composing ISDL model elements. Those rules ensure that BPEL concepts are modeled consistently. A benefit of having consistent compositions is that we need to define only a limited set of rules in the transformation specification. Hence, the profile reduces the complexity of the transformation specification.

7.2 ISDL Profiling Mechanism

At the moment, the ISDL does not have any profiling mechanism. Therefore, we introduce a profiling mechanism that uses stereotypes and properties. A stereotype adds extra semantics to ISDL model elements. The additional semantics allows the ISDL model elements to represent concepts within a specific domain. A stereotype
may have properties to add information about itself.

We propose a notation for ISDL stereotypes and properties by adding text fields to the attribute box attached to an ISDL model element. A stereotype is represented by a stereotype name. Properties are represented as pairs of a property name and a property value. The metamodel for the profiling mechanism is given in Appendix B.

Within the attribute box, the stereotype name is written in the first line as <<stereotype name>> and is immediately followed by its properties, if any. A separation line is used to isolate the stereotype and its properties from the name, attributes, and constraints or assignments of the attribute of the ISDL model element. Figure 7.1 shows the use of a stereotype and its properties in an ISDL action.

![Stereotypes attached to an ISDL action](image)

We choose a textual notation for representing stereotypes in order to allow one to write and recognize the notation easily, especially when a large number of stereotypes must be defined. The notation which is similar to the UML stereotypes and tags lets one that is already familiar to stereotyping in the UML use the same notation in the ISDL.

A concept within a domain may have to be represented by a composition of ISDL model elements. A complex concept may lead to a complex composition of ISDL model elements that makes it inefficient to be drawn repeatedly. For such a concept, we introduce a shorthand to abstract from the actual model of the concept. The shorthand which appears as a stereotyped ISDL element provides a simpler notation. In this way, one can model the concepts easily and produce a neat model.

In this profile, we define that any enabling conditions has a must uncertainty attribute.

### 7.3 BPEL Process Stereotype

A BPEL process defines the behavior of a business process. It defines and orders activities to be performed when an instance of the business process is executed. This profile models a BPEL process as an ISDL behavior stereotyped as <<process>>.
The stereotype has a property `targetNamespace` to specify the target namespace of the BPEL process definition. The properties `schemaNamespace` and `schemaLocation` are explained later in Section 7.4. A `<<process>>` behavior may not contain a behavior recursion. The behavior has one entry point and no exit point. Figure 7.2 shows a BPEL process named `Purchase`.

![Figure 7.2: Notation of a BPEL process](image)

### 7.4 BPEL Data Definition and Manipulation

A BPEL process maintains its processing states in variables that are defined for certain data types or message types. The variables use data types defined in the XML Schema [45]. The message types are defined in WSDL documents that can be the WSDL document of the process or the WSDL documents of its partners. A BPEL process is responsible to define the message types in its WSDL document; while the process has no control to define the message types of its partners’ WSDL documents.

This profile assumes that the necessary message types of a BPEL process being defined are already defined in an XML schema. The process just needs to specify the namespace and the location of the XML schema. They are specified in the properties `schemaNamespace` and `schemaLocation` depicted in Figure 7.2.

This profile uses information attributes to abstract BPEL variables since the ISDL only deals with information that is established in actions or interaction contributions. The information attributes can be referred by other actions or interaction contributions to establish their information. When information has to be passed between behaviors that represents BPEL structured activities, scopes, and handlers, the information is passed as parameters of the behaviors’ entry or exit points. Figure 7.3 shows an information attribute `shippingInfo` as an instance of a message type `ShippingInfo`.

All attributes in this profile are information attributes. Therefore, the profile does not explicitly mention the attribute kinds $\iota$, $\tau$, and $\lambda$. Later, in Subsection 7.5.3, we describe how the profile handles time attributes.

The ISDL models data manipulation of an information attribute in two ways:
(i) using a constraint on the information attribute itself, or (ii) using an attribute constraint on an enabling condition that refers to the attribute. Figure 7.4 shows the use of both ways of data manipulation.

![Diagram](image)

Figure 7.4: Two ways for data manipulation

In Figure 7.4 (a), data manipulation is represented as a constraint on an information attribute \( dst \) of an action \( b \). The constraint refers to an information attribute \( src \) of an action \( a \). Figure 7.4 (b) models the same data handling by using a attribute constraint on the enabling condition between the actions.

Attribute constraints are expressed in a modified XPath expression. The modification allows an information attribute in an action or interaction contribution refer to another information attribute in another action or interaction contribution. To refer to an action or interaction contribution, the name of the action or interaction contribution followed by a dot is written before the information attribute being referred.

To manipulate data, we may have to deal with elements of a variable. This profile models the structure of the message type of the information attribute by a modified XPath location path. The modification is to accommodate \( parts \) of the message type because they are not recognized in a regular XPath location path.

In a modified XPath location path, this profile defines a message part between a double slash (//) after the information attribute name and a slash (/) starting a regular XPath location path. If a structure does not have any part, an information attribute is simply followed by a slash (/) and then a regular XPath location path.

Figure 7.5 shows an information attribute \( ShippingInfo \) that has a part \( info \) with an element \( price \). The element is given a value that refers to the value of an information attribute \( price \) of an action \( getPrice \).
7.5 BPEL Basic Activity Stereotypes

A BPEL basic activity performs a certain functionality. The functionality can be: invoking a service, providing a service, waiting for an alarm, or terminating a process instance.

7.5.1 Service Invocation

A service invocation by a BPEL process on one of its partners constitutes a message passing from the BPEL process to the partner through a middleware that delivers the message. The BPEL process interacts with the middleware for sending the message and then the partner interacts with the middleware for receiving the message. Thus, a message passing consists of two ISDL interactions. Figure 7.6 shows interactions within a message passing from a BPEL process to its partner.

![Figure 7.6: Two interactions within a message passing](image)

This profile stereotypes the ISDL interaction contribution of the BPEL process as <<invoke>> to indicate that the interaction contribution is responsible to send a message for invoking a service provided by the partner. The stereotype has properties partnerLink, portType, and operation to represent attributes of the BPEL invoke activity.

In a request-response operation, two message passings are performed. One is for sending a request message from a BPEL process to a partner; another is for receiving a response message from the partner. Therefore, two ISDL interaction contributions are needed in the BPEL process to model the invocation of the operation. In addition to the interaction contribution stereotyped as <<invoke>> that sends the request message, this profile stereotypes the ISDL interaction contribution that is responsible for receiving the response message as <<result>>. The stereotype has the same properties as
its corresponding <<invoke>> interaction contribution. The <<result>> interaction contribution is enabled by its corresponding <<invoke>> interaction contribution.

Messages to be sent and received by the interaction contributions are modeled as information attributes. Since only one message can be sent for invoking a service, then only one information attribute can be defined for the interaction contribution. The same also applies for receiving a response message in a request-response operation. Figures 7.7 and 7.8 show BPEL invocations of one-way and request-response operations, respectively.

![Figure 7.7: Notation of a BPEL invocation of a one-way operation](image)

![Figure 7.8: Notation of a BPEL invocation of a request-response operation](image)

7.5.2 Service Provision

A BPEL process may provide a service to be invoked by its partner. Now a partner sends a request message and the BPEL process must be ready to receive the message. Again, this message passing is done through a middleware. This profile models a BPEL service provision as an ISDL interaction contribution stereotyped as <<receive>> to indicate that the interaction contribution is responsible for receiving a request message.
The stereotypes have properties `partnerLink`, `portType`, and `operation` to represent attributes of the BPEL receive activity.

If the service provided by the BPEL process is a request-response operation, another message passing is needed to send back the response. This profile models the contribution of the process to the interaction as an ISDL interaction contribution stereotyped as `<<reply>>`. The stereotypes have properties `partnerLink`, `portType`, and `operation` to represent attributes of the BPEL reply activity. The interaction contribution exists only in combination with a `<<receive>>` interaction contribution. This means that the `<<reply>>` interaction contribution causally depends on the `<<receive>>` interaction contribution.

Messages to be received and sent by the interaction contributions are modeled as information attributes of the interaction contributions. Figure 7.9 shows a request-response service provision. For a one-way service provision, the `<<reply>>` interaction contribution is simply not used.

![Figure 7.9: Notation of a BPEL service provision](image)

In Figure 7.9, a dashed line is a placeholder for sub-behaviors, actions, or other interaction contribution. The placeholder must not contain other `<<receive>>` interaction contributions from the same partner, port type and operation. This is to comply with a constraint defined in the BPEL specification that “more than one synchronous request from a specific partner link for a particular port type, operation, and correlation sets must not be outstanding simultaneously”.

73
7.5.3 Waiting

A BPEL wait activity allows a BPEL process to temporarily stop its execution. The BPEL defines two types of the activity, i.e. waiting for a certain time period and waiting until a certain time moment to expire. The first one allows a BPEL process to execute a successive activity only after the time period elapses after the previous activity has completed. Thus, the time moment the next activity is performed depends on the time moment the previous activity completes.

The second one allows a BPEL process to execute the next activity only after an absolute time moment specified by a global clock expires. Thus, the time moment the next activity is performed does not depend on the time moment the previous activity completes. Since the ISDL has no concept of absolute time moment, this profile cannot model the second type of the wait activity.

The ISDL models the first type of the wait activity by using time attributes of two consecutive ISDL actions or interaction contributions. Each action or interaction contribution specifies a time attribute. The time attribute of the second action or interaction contributions refers to the time attribute of the first action or interaction contribution and specifies some time period. The time period is specified as an XML Schema expression of duration [45].

Figure 7.10 shows an action b that is allowed to complete only after one hour from the time moment an action a completes. The constraint for the time attribute of action b is expressed by using a larger than (>) notation, because we must take into account the time spent by the action b to perform.

![Figure 7.10: Action b waits for a duration of an hour after action a](image)

**Shorthands**

An action or interaction contribution occurs at a certain time moment. Thus, an action or interaction contribution always has a time attribute, although the attribute is not explicitly declared. When time attributes are unconstrained, typically they are omitted.

Regarding time attributes, this profile considers only constraints that are applied to the attributes. The declaration of the attributes may be omitted. The profile introduces a keyword Time to represent the time moments of actions or interaction contributions. As a result, the profile does not need to mention the attribute kind \(\tau\) explicitly. Figure 7.11 shows the model in Figure 7.10 using the shorthand.

74
7.5.4 Terminating Process Instance

When a BPEL terminate activity is performed by a BPEL process, it disables all other activities to occur by terminating the process immediately. This profile models the termination by one or more disabling relations that originate from an ISDL action or interaction contribution, which disables all other actions or interaction contributions within the <<process>> behavior.

Figure 7.12 gives an example of how to model the BPEL terminate activity. If action $d$ occurs, it disables all other actions within the behavior. Since all actions now cannot occur, we can consider that the <<process>> behavior terminates. Thus we can say that if action $d$ occurs, the <<process>> behavior terminates.

Shorthand

This profile introduces a shorthand to abstract from all the disabling relations. The shorthand is an action stereotyped as <<terminate>>. Figure 7.13 shows the shorthand. The action is responsible only for terminating the process instance. A <<terminate>> action must follow an action or interaction contribution whose occurrence disables all other action or interaction contribution. Figure 7.14 shows the use of the shorthand in the example depicted in Figure 7.12.
7.6 BPEL Structured Activity Stereotypes

A BPEL structured activity controls how other activities are performed. Such a control executes activities sequentially or concurrently, executes an activity repeatedly, executes an activity according to a decision that is made, or executes an activity to handle an event.

7.6.1 Coordinating Activities

The BPEL defines two structured activities, i.e. sequence and flow, for coordinating other activities. A sequence activity orders the execution of other activities such that an activity can only be executed after the execution of its preceding activity completes. Thus a completion of an activity enables the next activity to occur. This profile models a sequence activity by one or more enabling conditions that sequentially enable ISDL actions, interaction contributions, or sub-behaviors. The sequence of enabling conditions originates at an entry point. Figure 7.15 shows an ISDL model of a sequence activity of an action, a sub-behavior, and another action.

Figure 7.13: Shorthand of the BPEL terminate activity

Figure 7.14: Action d terminates the process by using the shorthand

Figure 7.15: Model for a BPEL sequence activity
A flow activity contains two or more activities to be performed concurrently. The flow activity completes when all of its concurrent activities complete. This profile models the behavior of the flow activity as an ISDL behavior stereotyped as <<flow>>. The behavior has one entry point and one exit point. The entry point enables activities to be performed concurrently. The exit point is enabled by a conjunction of all those activities.

A flow may contain links to synchronize activities within it. This profile represents a link by an enabling condition stereotyped as <<link>>. The stereotype has a property name to name the link. A link originates at an action, an interaction contribution, or the exit point of a sub-behavior. The link ends at an action, an interaction contribution, or a conjunction preceding a sub-behavior. Figure 7.16 shows the ISDL model of a flow activity with links. From the figure, we know that the flow activity contains two sequence activities because two enabling conditions originate at the entry point of the <<flow>> behavior.

Figure 7.16: Model for a BPEL flow activity

An action, interaction contribution, or a sub-behavior in a model of a sequence activity can enable only a single action, interaction contribution, or sub-behavior. If the sequence is defined within a flow, then each of them may enable more than one action, interaction contribution, or sub-behavior. One of the enabling condition is un stereotype and the others must be <<link>> enabling conditions to other sequences. Figure 7.16 shows two examples of this. Action d enables action e and sub-behavior B. Sub-behavior B enables action c and action f.

7.6.2 Repeating Activity

A BPEL while activity contains an activity to be performed repeatedly while its condition is satisfied. Instead of modeling its behavior and proposing a shorthand by ourselves, this profile uses a shorthand that is already defined in the ISDL namely repetitive behavior.
Shorthand

This profile extends the already defined shorthand by stereotyping it as \texttt{<<while>>}. The stereotype does not add any semantics. This profile introduces it to make the shorthand be stereotyped so it is similar to other shorthands. The condition is expressed as a repetition constraint of the behavior in an XPath boolean expression \cite{10}. Variables to be evaluated by the condition must be passed as parameters to the entry point of the behavior.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{shorthand.png}
\caption{Shorthand of BPEL while activity}
\end{figure}

7.6.3 Decision Making

A BPEL switch activity contains one or more conditional cases. The switch activity evaluates conditions of the cases, then executes a case whose condition is satisfied. Each case has its own behavior. The activity may contain a default to be performed when all the conditions specified are not satisfied.

We give an example of the behavior of the switch activity in Figure 7.18. The figure shows a switch activity with two cases \textit{Gold} and \textit{Silver} and a default \textit{Default}. Since only one case can be executed, the exit point is enabled by a disjunction of all cases.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{switch.png}
\caption{Behavior of a BPEL switch activity}
\end{figure}
Shorthand

This profile introduces a shorthand for the BPEL switch activity as a behavior stereotyped as `<switch>`.

The behavior abstracts the evaluation of conditions of the cases and the disjunction before the exit point.

The behavior of a case is modeled as a behavior stereotyped as `<case>`, and the behavior of the default is modeled as a behavior stereotyped as `<otherwise>`.

Each stereotyped behavior has one entry point and one exit point.

Since the evaluation of the condition of each case is abstracted by the `<switch>` behavior, this profile models the condition as a constraint of the `<case>` behavior.

The constraint is expressed as an XPath boolean expression.

No condition is specified for the `<otherwise>` behavior.

Figure 7.19 shows the use of the stereotypes to model the behavior depicted in Figure 7.18.

![Figure 7.19: Shorthands of the BPEL switch activity](image)

7.6.4 Picking Event

A BPEL pick activity consists of one or more event handlers. When its corresponding event occurs, a handler is executed. An event can be the arrival of a message or the expiration of an alarm.

The pick activity executes only one of the handlers although two or more events can occur simultaneously.

Figure 7.20 shows the behavior of the BPEL pick activity. We use a previously defined stereotype `<receive>` and a time attribute to model the kinds of events.

A dashed line is a placeholder for the handler that is to be executed when its corresponding event occurs. The action `first` represents the first action or interaction contribution of the handler of an alarm event.

The time moment the alarm expires is modeled as the time attribute constraint of the action `first`. 

Figure 7.20: Behavior of the BPEL pick activity

![Figure 7.20](image)
Shorthand

This profile introduces a shorthand to abstract some behavior of the BPEL pick activity. The shorthand is represented as an ISDL behavior stereotyped as <<pick>>. The stereotyped behavior abstracts the selection of handlers and the disjunction before the exit point. Since each handler has its own behavior, this profile represents the handler by an ISDL behavior stereotyped as <<onMessage>> or <<onAlarm>>. Each stereotyped behavior has one entry point and one exit point. Figure 7.21 shows the use of the shorthand for representing the behavior of a pick activity depicted in Figure 7.20.
The entry point of the <<onMessage>> behavior enables a <<receive>> interaction contribution, while the entry point of the <<onAlarm>> behavior enables the action first. Since the <<onMessage>> behavior is defined as a sub-behavior of the <<pick>> behavior, a structured interaction contribution is needed to delegate the <<receive>> interaction contribution of the <<onMessage>> sub-behavior to the <<pick>> behavior.

7.7 BPEL Scope and Handlers

A BPEL scope is a sub-behavior within a BPEL process to provide a context within the process. A scope may handle faults, compensate a completed nested scope, and handle events.

7.7.1 Throwing and Handling Faults

Fault handlers of a scope are ready to be executed when the scope starts to execute. During the execution of the scope, a fault can be thrown explicitly by a BPEL throw activity or implicitly by the BPEL engine where the BPEL process is executed. The fault terminates the normal behavior of a scope and can be used to invoke a handler assigned for the fault. The scope completes either when its normal behavior or its fault handler completes. In Figure 7.22, we show an example of a scope that throws and handles a fault.

In the figure, the normal behavior of scope AScope is modeled in a sub-behavior ANormal. One of its actions i.e. action d may throw a fault. The action d has an information attribute fault with a data type Fault. The information attribute fault is given a value error. After the fault is thrown, action d disables all other actions within the sub-behavior ANormal and enables the exit point f. Since no more actions can occur in the sub-behavior, the sub-behavior terminates.

The sub-behavior FaultHandlers, which represents a collection of fault handlers, is enabled when the scope starts and a fault is thrown in the sub-behavior ANormal. The parameter passed to the sub-behavior is evaluated to determine which handler should be executed. In our case, the fault is handled by sub-behavior FaultHandler1. A fault handler FaultHandler2 is defined to handle any other faults.

Shorthand

This profile introduces shorthands to abstract the behavior of throwing and handling faults. But firstly, the profile introduces behaviors stereotyped as <<scope>> and <<normal>> to represent a scope behavior and the normal behavior of the scope, respectively. The <<scope>> behavior abstracts any relations among its sub-behaviors (not only for handling faults, but also for scope compensation and event handling, as described later in SubSections 7.7.2 and 7.7.3). Both stereotyped behaviors abstract any
The BPEL throw activity is represented by an action stereotyped as <<throw>>. The stereotypes abstract from all disabling relations that originates from the action and the enabling relation to an exit point that passes the fault. The stereotype has a property fault to specify the fault name. No other action or interaction contributions should be defined after the action because it would never be performed. Figure 7.23 shows the shorthand.

The behavior of a collection of fault handlers and how they are enabled are abstracted by the <<scope>> behavior. Each handler for a particular fault is represented by an ISDL behavior stereotyped as <<faultHandler>> that performs an
activity to handle the fault. The stereotype has a property fault whose value is a fault name to be caught. This profile introduces a specific fault name otherFaults to represent all other possible faults that cannot be handled by fault handlers defined within the scope. Figure 7.24 applies the shorthands for the behavior depicted in Figure 7.22.

![Figure 7.24: Shorthands for throwing and handling faults](image)

### 7.7.2 Compensating Scope

The compensation handler of a scope can be invoked only if the normal behavior of a scope completes. A BPEL compensate activity is used to invoke the compensation handler. The invocation can only be done from a fault handler or another compensation handler of the immediately enclosing scope. Figure 7.25 shows an example of a scope that compensates its enclosing scope when a fault is thrown. We use already defined stereotypes to model the example.

In the figure, a scope BScope defines its normal behavior BNormal and its compensation handler BCompensation. The scope BScope is defined within another scope AScope. If, after a successful completion of the scope BScope, there is a fault thrown within the scope AScope, then a fault handler AFaultHandler is executed to enable the compensation handler BCompensation.

**Shorthand**

This profile introduces shorthands to abstract the behavior of a scope compensation. This profile represents the compensation handler of a scope by an ISDL behavior stereotyped as <<compensationHandler>>. To abstract how the compensation handler
Figure 7.25: Behavior of a scope compensation

is enabled from within a fault handler or another compensation handler, this profile uses an action stereotyped as \texttt{<<compensate>>}. The stereotype has a property \textit{scope} whose value identifies a scope to be compensated. Figure 7.26 shows the shorthand of the compensate activity. Figure 7.27 applies the shorthands for the same behavior depicted in Figure 7.27.

Figure 7.26: Shorthand of the BPEL compensate activity
7.7.3 Event Handling

Event handlers are behaviors that may be executed concurrently with the normal behavior of a scope when their corresponding events occur. Event handlers are ready to handle events as its scope starts. When the normal behavior of the scope completes, it disables the event handlers. The completion of the scope must wait until all event handlers that have already been dispatched have completed. An event handler cannot be dispatched after the completion of the normal behavior, although the scope does not complete yet when it waits for another event handler to complete. Figure 7.28 shows this behavior by using already defined stereotypes.

The figure shows the behavior of the event handlers. A dashed line is a placeholder for the actual activities that handle the event. The behavior of event handlers may complete when no handler is executed, some handlers are executed, or all handlers are executed. The handler onAlarm can only be executed once during its life time. The handler onMessage can be executed as many times as its corresponding event occurs. The execution of an instance of the handler onMessage can be performed concurrently with other instances of handlers for the same message.
Shorthand

This profile introduces shorthands to abstract behavior of the event handling. A handler for incoming messages is represented by an ISDL behavior stereotyped as <<onMessage>>. A handler for an alarm is represented by an ISDL behavior stereotyped as <<onAlarm>>. The behavior of a collection of event handlers, how the <<onMessage>> behavior is replicated, and how the event handlers are disabled are abstracted by the <<scope>> behavior. These give the <<onMessage>> and <<onAlarm>> behaviors same semantics as the same stereotypes defined in the BPEL pick activity (see Section 7.6.4). Figure 7.29 shows the shorthands for event handling.
Figure 7.29: Shorthands of event handling
Chapter 8

ISDL-to-BPEL Transformation

This chapter describes the ISDL-to-BPEL transformation as a proof of concept. Section 8.1 gives the scope of the transformation that we define and implement. Section 8.2 provides an overview of the transformation specifications from ISDL to WSDL and from ISDL to BPEL. Section 8.3 describes a procedure to perform the transformation. Section 8.4 shows an example of the transformation. Finally, we give some implementation notes in Section 8.5.

8.1 Scope

We define and implement the transformation for an ISDL model that contains only a single behavior type. The behavior type does not contain any behavior instantiations. We do not yet define the transformation rules from ISDL to BPEL structured activities, except the BPEL sequence activity, and the transformation rules from ISDL to BPEL scopes and handlers.

We define the transformation rules from ISDL to any BPEL basic activities that are defined in the profile. Thus we can model a BPEL process that provides and invokes services whether they are one-way or request-response operations, assigns values to variables, waits for a certain time period, and terminates itself using the BPEL terminate activity.

The transformation implementation does not yet deal with complex expressions of variable assignments. An implementation note in Section 8.5 explains this later.

8.2 Overview of Transformation Specifications

We define two transformation specifications i.e. from ISDL to WSDL and from ISDL to BPEL. These specifications provide mappings from the elements of the ISDL metamodel to the elements of our metamodels for the WSDL and the BPEL. They produce WSDL and BPEL models instead of WSDL and BPEL documents. A subsequent step is necessary to transform the models into WSDL and BPEL documents, respectively.
A subset of the ISDL metamodel that we use for defining the transformation specification is given in Appendix B. The metamodels for the WSDL and the BPEL are already given in Chapter 6. In this section, we provide an overview of the transformation specifications. The rules of the specifications are given in Appendix C.

The rules of the transformation specifications are defined in a natural language instead of in a particular transformation language, although we implement the rules in the YATL. Our intention is to allow the rules to be implemented in different transformation languages.

The transformation specifications are independent of each other such that they can be executed in any order. However, the rules of each individual transformation specification must be executed in order because the result of a rule may be used by its following rules. To make the specifications independent, some rules are redundant. The redundancy is described in Subsection 8.2.2.

8.2.1 ISDL to WSDL

To transform an ISDL model into a WSDL model, the specification looks for an ISDL Behaviour Type stereotyped as «process» and creates a WSDL Definition. This WSDL Definition must be created first because the WSDL Definition acts as a container of other WSDL model elements. To create those elements, the specification only deals with ISDL Interaction Contribution Instantiations. No rule of the specification deals with any ISDL Action Instantiations, because a WSDL document describes the interfaces of a Web service without revealing its internal activities.

The WSDL document of a BPEL process is used in a partnership that is established by the process itself and the partner that initiates the process. The specification introduces the notion of the initiating partner link for a partner link that represents the partnership.

The specification creates the elements of the WSDL model, i.e. messages, port types, operations, inputs, and outputs, only when it encounters ISDL Interaction Contribution Instantiations defined for the initiating partner link. Therefore, the rules must identify the initiating partner link in the ISDL model. The initiating partner link is indicated by the property partnerLink of the ISDL Interaction Contribution Instantiation stereotyped as «receive» that is enabled by the entry point of the ISDL Behaviour Type.

The specification also consists of rules to create model elements that represent the BPEL extensions for WSDL i.e. Partner Link Type and Role. The elements are created by referring to the properties of ISDL Interaction Contribution Instantiations defined for the initiating partner link.

The properties of stereotypes that are attached to ISDL Interaction Contribution Instantiations supply necessary information for creating and naming WSDL model elements. Some rules combine the property values with other words to make the values
suitable for representing the created elements and to avoid name conflicts. For instance, to name a WSDL Partner Link Type, Rule 10 (see Appendix C.1) concatenates the value of the property \textit{partnerLink} with \textit{_Type}. The concatenation makes the property value suitable for representing the Partner Link Type. At the same time, it avoids a name conflict between a WSDL Partner Link and its type.

\subsection{ISDL to BPEL}

The specification for transforming a ISDL model into a BPEL model first creates a BPEL Process that acts as a container for other BPEL model elements. The specification then identifies the initiating partner link.

Before creating other BPEL model elements, the specification creates a set of BPEL Messages, a set of BPEL Partner Link Types including their Roles, a set of BPEL Port Types including their Operations, and the relationships between them. The elements in those sets are generated in the same way as the transformation specification from ISDL to WSDL generates Messages, Partner Link Types, Roles, Port Types, and Operations. The difference is that the transformation specification from ISDL to WSDL only deals with ISDL Interaction Contribution Instantiations defined for the initiating partner link, while the transformation specification from ISDL to BPEL deals with all ISDL Interaction Contribution Instantiations in the ISDL model. Consequently, some of the elements are redundant.

The elements in those sets are used as references in the BPEL model, but are not to be generated in the BPEL document. The BPEL document refer to the corresponding elements in the WSDL document of the ISDL model being transformed.

To create BPEL activities, the rules look for ISDL Interaction Contribution Instantiations and Action Instantiations whose types are stereotyped. The ISDL Information Attributes and Time Attributes that are attached to ISDL Alternative Attribute Constraints also trigger the specification to create BPEL activities i.e. Assign and Wait, respectively. The activities are created without considering how they must be ordered. Some rules concatenate attribute or property values with other words to make the values suitable to represent the elements and to avoid name conflicts.

When all activities have been created, the specification looks for the entry point of the ISDL Behavior Type and creates a BPEL Sequence to be the primary activity of the BPEL Process. Then, starting from the entry point, the specification traverses all enabling conditions in the model. Each time an ISDL Causality Target is encountered, its corresponding BPEL activities are put into the BPEL Sequence. The specification first checks whether the ISDL Causality Target also corresponds to a BPEL Assign, a BPEL Wait activity, or both. If so, these BPEL model elements are put into the ISDL Sequence prior to the main activity that correspond to the ISDL Causality Target.
8.3 Transformation Procedure

The transformation procedure consists of five stages. Figure 8.1 illustrates the stages and artifacts needed and produced by each stage. It also shows tools used in each stage. Complete lists of standard specifications and tools used in the procedure are given in Subsections 8.3.1 and 8.3.2

![Figure 8.1: Transformation procedure](image)

Initially, we assume that an ISDL model has been created using Grizzle. The model must conform to the restrictions and composition rules defined in the ISDL
Profile for BPEL. Grizzle by default saves models in its own proprietary format as 
.grizzle file. To enable the transformation, in the first stage, we export the model to
the ISDL XML format using Grizzle’s export feature.

Grizzle does not yet support the profiling mechanism introduced in Chapter 7.
Therefore, in the second stage, we add stereotypes and properties to the model that is
in the ISDL XML format. We use an XML editor for doing this, i.e. Butterfly XML
IDE. In the third stage, we convert the stereotyped ISDL model into an XMI format
that conforms to the metamodel of the ISDL and its profiling mechanism. To do so,
we define an XSLT transformation from the ISDL XML format to the XMI format and
use Xalan to execute the transformation.

The fourth stage splits the procedure into two branches: (i) for producing a
WSDL document and (ii) for producing a BPEL document. For each branch, we
implements transformation specification, which is defined in Section 8.2 in the YATL.
The transformations are executed in the YATL4MDR and produce WSDL and BPEL
models in XMI formats.

At the last stage, we use Xalan to convert the WSDL and BPEL models into
WSDL and BPEL documents. To do so, we define two XSLT transformations: (i) from
XMI format that represents the WSDL model to a WSDL document, and (ii) from
XMI format that represents the BPEL model to a BPEL document.

Relationships with MDA Approaches

The ISDL-to-BPEL transformation uses two model transformation approaches
suggested by the MDA, i.e. marking and metamodel transformation. The approaches
are applied in sequence. The marking approach is performed in the second stage to
mark an ISDL model with stereotypes and properties defined in the ISDL Profile for
BPEL. The metamodel transformation approach is performed in the fourth stage, where
we transform a stereotyped ISDL model to WSDL and BPEL models.

The procedure also consists of conversions in the third and the fifth stages. The
conversions are not considered as MDA model transformations, since they convert a
model from one format to another, while the model itself remains the same. Therefore,
the conversions are better considered bridges between technological spaces [25] i.e.
between the XML and the MDA spaces.

8.3.1 Specifications

To support the transformation, we use the following specifications.

- XMI 2.0 [34]

  The XMI (XML Metadata Interchange) is a specification to define a serialization
  format of a model in XML.
8.3.2 Tools

To support and to execute the transformation, we use the following tools.

- Grizzle 0.58.10 [22]
  This is an editor for the graphical notation of the ISDL. It can export its models into the ISDL XML format which is defined in the XML schema for ISDL specifications [22]. We use Grizzle to create ISDL models and to save them in the ISDL XML format.

- YATL4MDR 0.20 [10]
  This is a transformation engine that accepts the YATL as a transformation language. We use YATL4MDR tool to execute the ISDL-to-BPEL transformation.

- Xalan-Java 2.6.0 [2]
  This is an XSLT processor that fully implements the XSLT 1.0. We use this tool to execute XSLT transformations from XML-based formats to other XML-based formats. Any XSLT processors can be used instead.

- Butterfly XML IDE 1.1 [9]
  This is an XML editor that we use to add stereotypes on ISDL models that are already saved in the ISDL XML format. Any XML or text editors can be used instead.

8.4 An Example

To demonstrate that the transformation works, we give an example of a business process for handling a purchase order. First, we describe and model the business process in the ISDL, then we describe marks that are added to the model to enable the ISDL-to-BPEL transformation and, lastly, we describe the transformation itself.

8.4.1 Business Logic

Figure 8.2 depicts an ISDL model of the business logic of purchase handling. The model is a platform independent model.

The business process starts when it receives a purchase order from a customer. The reception of the purchase order is modeled as an interaction contribution receiveOrder.
To calculate the total price of the items listed in the purchase order, the process forwards the purchase order to its partner, i.e. a selling partner, which sells items listed in the purchase order. In return, the selling partner sends back an invoice for the purchase order. The selling partner provides its service as a request-response operation. The operation is modeled as two interaction contributions: `calculatePrice` for forwarding the purchase order to the selling partner and `returnPrice` for receiving the invoice from the selling partner.

On receiving the invoice, the process sends the invoice to the customer. The process provides this invoicing service to the customer as a request-response operation. The sending of the invoice is modeled as an interaction contribution `sendInvoice`.

To schedule the delivery of the items ordered, the process then makes a request to another partner, i.e. a shipping partner, which provides a delivery service. The purchase order is again forwarded, but now to the shipping partner. For some reason, the process forwards the purchase order only twelve hours after the invoice has been sent to the customer. The shipping partner provides its service as two one-way operations: one for receiving a request, another for sending the result. The operations are modeled as interaction contributions `requestShipping` and `receiveSchedule`, respectively.

When the shipping partner sends its delivery schedule as a result to the process, the process forwards the schedule to the customer. The customer sees this scheduling service as a callback invocation. This forwarding is modeled as an interaction contribution `sendSchedule`. The process then terminates.

Figure 8.2: Business logic of the purchase handling
8.4.2 Stereotyped ISDL Model

To be able to be transformed by the ISDL-to-BPEL transformation, the model must be made to conform to the ISDL Profile for BPEL. To do so, we add stereotypes and properties to the model as shown in Figure 8.3.

Figure 8.3: Model ready to be transformed

We stereotype the behavior type PurchaseHandling as <<process>> to indicate that the behavior type represents a BPEL process. The interaction contribution receiveOrder is stereotyped as <<receive>> to indicate that the interaction contribution is for receiving messages. Since the invoicing service is a request-response operation, the interaction contribution sendInvoice is stereotyped as <<reply>>. The interaction contribution sendSchedule is stereotyped as <<invoke>> to indicate that the BPEL process uses this interaction contribution to make a callback invocation to the customer.

Since the selling partner provides its service as a request-response operation, the interaction contributions calculatePrice and returnPrice are stereotyped as <<invoke>>.
and \texttt{\textless result\textgreater}, respectively. The shipping partner provides its service as two one-way operations, thus both interaction contributions \texttt{requestShipping} and \texttt{receiveSchedule} are stereotyped as \texttt{\textless invoke\textgreater}.

The attributes of the interaction contributions and their constraints are now expressed in the way Grizzle expresses attributes and constraints. Notice that the declarations of the time attributes of the interaction contributions \texttt{sendInvoice} and \texttt{requestShipping} are removed, but the time constraint in the interaction contribution \texttt{receiveSchedule} remains.

We add an action stereotyped as \texttt{\textless terminate\textgreater}. Actually, this action is not necessary to terminate a BPEL process, but we include it here to show that the action is recognized by the ISDL-to-BPEL transformation and is transformed to its corresponding activity.

### 8.4.3 Model Transformation

The ISDL-to-BPEL transformation takes the stereotyped ISDL model as input and produces its corresponding WSDL and BPEL documents. These documents are presented in Appendix D.

### 8.5 Implementation Notes

During the implementation of the transformation specifications and the transformation procedure, we took some notes that are discussed here.

The ISDL Profile for BPEL models an assignment of a BPEL Variable as the expression of an ISDL Alternative Attribute Constraint. Thus, the transformation must specify rules that involve a parsing mechanism for such expressions. However, the YATL lacks the capabilities to parse complex expressions or sentences because the YATL is a transformation language rather than a traditional programming language. Hence we implement two variable assignments only, i.e. from a variable to another variable and from a character string to a variable.

XML namespaces are not handled properly yet. We deal with XML namespaces in the fifth stage of the transformation procedure. In this stage, we convert WSDL and BPEL models in their XMI formats into WSDL and BPEL documents, respectively. The conversion includes creating namespaces for the generated WSDL and BPEL models by referring to the value of the property \texttt{targetNamespace} of the stereotype \texttt{\textless process\textgreater} that is attached to an ISDL Behaviour Type. In the implementation, the XSLT 1.0 specification, which cannot create a namespace definition dynamically, prevents us to create the namespaces properly. This limitation hopefully is resolved in the XSLT 2.0.

The transformation rules accept a model under the assumption that the model complies with the ISDL Profile for BPEL. The assumption makes the transformation
specification less complex, because no checking is made on the model to evaluate whether the model complies with the profile. Models that do not comply with the profile can be transformed, but they produce meaningless documents. Although such checking is beyond the scope of a transformation specification, it is necessary.
Chapter 9

Conclusions

This chapter presents the conclusions and the contributions of our work and identifies areas where further investigation is necessary. Section 9.1 presents general conclusions, Sections 9.2 presents the contributions of our work, and Section 9.3 identifies some potential future work.

9.1 General Conclusions

We have developed an ISDL-to-BPEL transformation. The transformation includes a transformation from ISDL to WSDL, because BPEL processes need WSDL definitions to work. Our efforts include: (i) the development of metamodels for WSDL and BPEL, (ii) the development of an ISDL profile for BPEL, (iii) the development of transformation specifications from ISDL to WSDL and from ISDL to BPEL, and (iv) an implementation of the model transformation. Using the transformation, we can automatically generate WSDL and BPEL documents from an ISDL model that represents a business process.

In the metamodel transformation approach, the metamodel of the source and the target languages play two roles: (i) as abstract syntax of the model to be transformed or to be produced, and (ii) as a source or target for defining a transformation specification. The first role requires the metamodel to be syntactically complete and the second role requires that the metamodel is developed by considering the semantics of the language being modeled. Metamodels produced by current tools only satisfy the first requirement. In Chapter 6, we develop metamodels for the WSDL and the BPEL by considering the semantics of the language being modeled to satisfy the second requirement.

In a design methodology that uses a step-wise refinement or top-down approach, an ISDL model that complies with the ISDL Profile for BPEL would be the final implementation of a service architecture. A mapping from the model onto the WSDL and the BPEL as concrete components would then be performed automatically by the ISDL-to-BPEL transformation. Furthermore, the profile eases conformance assessment
of the final implementation to the service architecture since both are defined in the same language i.e. the ISDL.

The ISDL Profile for BPEL introduces shorthands for representing BPEL code. Some shorthands appear to be visual representations of the BPEL code. A benefit of this is that the shorthands give us one-to-one mappings to the BPEL code. Such a mapping eases the development of a transformation specification.

There is a trade-off between the number of restrictions introduced by a language profile and the number of rules in a transformation specification. The more restrictions in the profile, the less rules in the specification; the less restrictions in the profile, the more rules in the specification. When determining how restrictive the profile should be, one should remember that a restrictive profile prevents a model designer from using the profile at higher levels of abstraction.

9.2 Contributions

From the development of the ISDL-to-BPEL transformation, we give some contributions to a more general context. The contributions are listed as follows.

- Metamodels for the WSDL and the BPEL
  In Chapter 6, we develop the metamodels by considering the semantics of the languages and by being aware of syntactic details of the languages.

- An ISDL profiling mechanism
  In Chapter 7, we introduce a profiling mechanism for the ISDL. The mechanism is based on stereotypes and properties.

- An ISDL profile for BPEL 1.1
  In Chapter 7, we develop an ISDL profile to enable ISDL models to be mapped into the BPEL 1.1.

Our contributions at the implementation level are the following.

- An XSLT transformation that converts the format of an ISDL model, i.e. from the ISDL XML format into an XMI format that conforms to the ISDL metamodel.

- Transformation specifications from ISDL to WSDL and from ISDL to BPEL. The specifications are defined in a natural language and the YATL.

- XSLT transformations from a WSDL model to a WSDL document and from a BPEL model to a BPEL document. The models are in XMI formats that conform to our metamodels for the WSDL and the BPEL, respectively.
9.3 Future Work

We consider our work the initial step in the development of an ISDL-to-BPEL model transformation. Many issues need to be investigated further. They are indicated as follows.

In Chapter 8, we limit the scope of the transformation implementation. There is still a portion of the ISDL Profile for BPEL, for which its transformation rules need to be defined and be implemented. Having a complete implementation, the profile then could be evaluated and be improved with regard to efficiency and user-friendliness.

An improvement for the ISDL Profile for BPEL would be to make the profile less restrictive. Such an improvement would allow a model designer to use the ISDL in a more natural way. In turn, it would result in less steps in the design refinement process, because the model could be transformed into a BPEL document earlier in the design process. Another improvement would be to represent BPEL concepts by available ISDL elements, rather than stereotypes and properties.

The ISDL Profile for BPEL defined in Chapter 7 relies on an external information model. This is because the ISDL does not have a mechanism to represent information structure. The only element that can be used for this purpose is the information attribute, which is not suitable for modeling complex information structure. Implementing a mechanism to represent information structure would make the ISDL more suitable for representing BPEL processes.

In Chapter 8, we indicate that models are assumed to comply with the ISDL Profile for BPEL when they are transformed. The assumption may lead to a “garbage in, garbage out” phenomenon. A check for profile compliance on the models could prevent the phenomenon.
References


[33] OMG. *UML 2.0 OCL Specification.*

[34] OMG. *XML Metadata Interchange (XMI) Specification Version 2.0.*

[35] ONTOGENICS. *hyperModel.*


[37] PATRASCOIU, O. *YATL: Yet Another Transformation Language.* In *Proceedings of the 1st European MDA Workshop, MDA-IA* (January 2004), University of Twente, the Nederlands, pp. 83–90.


[40] W3C. *XML Path Language (XPath) Version 1.0.*


Appendix A

Informal Syntax

The specifications of the WSDL 1.1 and the BPEL 1.1 use an informal syntax to describe the XML grammar of WSDL and BPEL documents. The informal syntax is as follows.

- The syntax appears as an XML instance, but the values indicate the data types instead of values.

- Characters are appended to elements and attributes as follows: “?” (0 or 1), “*” (0 or more), “+” (1 or more). The characters “[” and “]” are used to indicate that contained items are to be treated as a group with respect to the “?”,”“*”, or “+” characters. We call them occurrence indicators.

- Elements names ending in “...” (such as <element ... /> or <element ... >) indicate that elements or attributes irrelevant to the context are being omitted. We call them irrelevance indicators.

- Elements and attributes separated by “|” and grouped by “(” and “)” are meant to be syntactic alternatives.

- <-- element --> is a placeholder for elements from some “other” namespace (like ##other in XSD).

- The XML namespace prefixes are used to indicate the namespace of the element being defined.
Appendix B

Metamodel of ISDL Behavioural Concepts

A subset of the metamodel of the ISDL behavioral concepts \[1\] is used in the ISDL-to-BPEL transformation presented in Chapter 8. The subset is identified by Figures B.1 and B.2. Notice that we introduce two abstract concepts i.e. Behaviour Member and Conjunctive Causality Condition for a practical reason.

Figure B.1: Metamodel of ISDL action, interaction contribution, and attribute

106
Figure B.2: Metamodel of ISDL behavior, causality relation, and entry point
We extend the metamodel to accommodate the ISDL profiling mechanism introduced in Chapter 7. Figure B.3 depicts the extension.

Figure B.3: Metamodel of ISDL profiling mechanism
Appendix C

Transformation Specifications

The rules of the transformation specifications from ISDL to WSDL and from ISDL to BPEL are the following.

C.1 From ISDL to WSDL

The transformation rules from ISDL to WSDL are the following.

1. For each instance of ISDL Behaviour Type stereotyped as \langle\langle\text{process}\rangle\rangle, create an instance of WSDL Definitions whose
   - attribute \textit{name} is set to the value of the attribute \textit{name} of this instance.
   - attribute \textit{targetNamespace} is set to the value of the property \textit{targetNamespace} of the stereotype.

2. For each instance of ISDL Behaviour Type stereotyped as \langle\langle\text{process}\rangle\rangle, create an instance of WSDL Import whose
   - attribute \textit{namespace} is set to the value of the property \textit{schemaNamespace} of the stereotype.
   - attribute \textit{location} is set to the value of the property \textit{schemaLocation} of the stereotype.
   - attribute \textit{owner} is set to an instance of WSDL Definition that corresponds to this instance (see Rule 1).

3. Use the instance of ISDL Entry Point of an instance of ISDL Behaviour Type stereotyped as \langle\langle\text{process}\rangle\rangle to identify the initiating partner link. This partner link is indicated by the value of the property \textit{partnerLink} of the stereotype \langle\langle\text{receive}\rangle\rangle that is attached to an ISDL Interaction Contribution Type whose instantiation is enabled by this Entry Point.

4. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as \langle\langle\text{receive}\rangle\rangle, \langle\langle\text{reply}\rangle\rangle, or \langle\langle\text{invoke}\rangle\rangle in the initiating partner link, create an instance of WSDL Port Type whose
   - attribute \textit{name} is set to the value of the property \textit{portType} of the stereotype.
attribute owner is set to an instance of WSDL Definition that corresponds to the instance of ISDL Behaviour Type where this instance is defined (see Rule 1).

5. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as <<receive>>, <<reply>>, or <<invoke>> in the initiating partner link, create an instance of WSDL Operation whose

* attribute name is set to the value of the property operation of the stereotype.
* attribute owner is set to an instance of WSDL Port Type that corresponds to this instance (see Rule 4).

6. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as <<receive>>, <<reply>>, or <<invoke>> in the initiating partner link and has ISDL Information Attribute, create an instance of WSDL Message whose

* attribute name is set to the value of the attribute type of the ISDL Information Attribute.
* attribute owner is set to an instance of WSDL Definition that corresponds to the instance of ISDL Behaviour Type where this instance is defined (see Rule 1).

7. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as <<receive>>, <<reply>>, or <<invoke>> in the initiating partner link and has ISDL Information Attribute, create an instance of WSDL Part whose

* attribute name is set to payload.
* attribute element is unknown.
* attribute owner is set to an instance of WSDL Message that corresponds to this instance (see Rule 6).

8. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as <<receive>> or <<invoke>> in the initiating partner link and has ISDL Information Attribute, create an instance of WSDL Input whose

* attribute name is set to the value of the attribute name of the ISDL Information Attribute.
* attribute type is set to an instance of WSDL Message that corresponds to this instance (see Rule 6).
* attribute owner is set to an instance of WSDL Operation that corresponds to this instance (see Rule 5).

9. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as <<reply>> in the initiating partner link and has ISDL Information Attribute, create an instance of WSDL Output whose

* attribute name is set to the value of the attribute name of the ISDL Information Attribute.
• attribute type is set to an instance of WSDL Message that corresponds to this instance (see Rule 6).
• attribute owner is set to an instance of WSDL Operation that corresponds to this instance (see Rule 5).

10. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as <<receive>>, <<reply>>, or <<invoke>> in the initiating partner link, create an instance of WSDL Partner Link Type whose

• attribute name is set to the value of the property partnerLink of the stereotype concatenated with _Type.
• attribute owner is set to an instance of WSDL Definition that corresponds to the instance of ISDL Behaviour Type where this instance is defined (see Rule 1).

11. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as <<receive>>, <<reply>>, or <<invoke>> in the initiating partner link, create an instance of WSDL Role whose

• attribute name is set to the value of the property portType of the stereotype concatenated with _Role.
• attribute portType is set to an instance of WSDL Port Type that corresponds to this instance (see Rule 4).
• attribute owner is set to an instance of WSDL Partner Link Type that corresponds to this instance (see Rule 10).

Rules 4 to 11 only creates a WSDL element instance that corresponds to an ISDL element instance if the same WSDL element instance is not created yet; otherwise the rule makes the ISDL element instance to correspond to the already created WSDL element instance.

C.2 From ISDL to BPEL

The transformation rules from ISDL to BPEL are the following.

1. For each instance of ISDL Behaviour Type stereotyped as <<process>>, create an instance of BPEL Process whose

• attribute name is set to the value of the attribute name of this instance.
• attribute targetNamespace is set to the value of the property targetNamespace of the stereotype.

2. Use the instance of ISDL Entry Point of an instance of ISDL Behaviour Type stereotyped as <<process>> to identify the initiating partner link. This partner link is indicated by the value of the property partnerLink of the stereotype <<receive>> that is attached to an ISDL Interaction Contribution Type whose instantiation is enabled by this Entry Point.
3. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as <<receive>>, <<reply>>, <<invoke>>, or <<result>>, create an instance of BPEL Partner Link Type whose
   - attribute name is set to the value of the property partnerLink of the stereotype concatenated with _Type, if this instance is in the initiating partner link;
   - attribute name is unknown, otherwise.

4. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as <<receive>>, <<reply>>, <<invoke>>, or <<result>>, create an instance of BPEL Port Type whose
   - attribute name is set to the value of the property portType of the stereotype.

5. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as <<receive>>, <<reply>>, <<invoke>>, or <<result>>, create an instance of BPEL Role whose
   - attribute name is set to the value of the property portType of the stereotype concatenated with _Role, if this instance is in the initiating partner link;
   - attribute name is unknown, otherwise.
   - attribute portType is set to an instance of BPEL Port Type that corresponds to this instance (see Rule 4).
   - attribute owner is set to an instance of BPEL Partner Link Type that corresponds to this instance (see Rule 3).

6. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as <<receive>>, <<reply>>, <<invoke>>, or <<result>>, create an instance of BPEL Operation whose
   - attribute name is set to the value of the property operation of the stereotype.
   - attribute owner is set to an instance of BPEL Port Type that corresponds to this instance (see Rule 4).

7. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as <<receive>>, <<reply>>, <<invoke>>, or <<result>> that has ISDL Information Attribute, create an instance of BPEL Message whose
   - attribute name is set to the value of the attribute type of the ISDL Information Attribute.

8. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as <<receive>>, <<reply>>, <<invoke>>, or <<result>>, create an instance of BPEL Partner Link whose
   - attribute name is set to the value of the property partnerLink of the stereotype.
   - attribute myRole is set to an instance of BPEL Role that corresponds to this instance, if this instance is stereotyped as <<receive>> or <<reply>> (see Rule 5).
• attribute `partnerRole` is set to an instance of BPEL Role that corresponds to this instance, if this instance is stereotyped as `<<invoke>>` or `<<result>>` (see Rule 5).

• attribute `partnerLinkType` is set to an instance of BPEL Partner Link Type that corresponds to this instance (see Rule 3).

• attribute `owner` is set to an instance of BPEL Process that corresponds to the instance of ISDL Behaviour Type where this instance is defined (see Rule 1).

9. For each instance of ISDL Information Attribute attached to an instance of ISDL Interaction Contribution Type stereotyped as `<<receive>>`, `<<reply>>`, `<<invoke>>`, or `<<result>>`, create an instance of BPEL Variable whose

• attribute `name` is set to the value of the attribute `name` of the instantiation of the ISDL Interaction Contribution Type concatenated with the value of the attribute `name` of this instance.

• attribute `messageType` is set to an instance of BPEL Message that corresponds to the instantiation of the ISDL Interaction Contribution Type (see Rule 7).

• attribute `owner` is set to an instance of BPEL Process that corresponds to the instance of ISDL Behaviour Type where the instantiation of the ISDL Interaction Contribution Type is defined (see Rule 1).

10. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as `<<receive>>`, create an instance of BPEL Receive whose

• attribute `name` is set to the value of the attribute `name` of this instance.

• attribute `createInstance` is set to `yes`, if the instance is enabled by the instance of ISDL Entry Point of the instance of ISDL Behaviour Type where this instance is defined.

• attribute `variable` is set to an instance of BPEL Variable that corresponds to the instance of ISDL Information Attribute attached to the ISDL Interaction Contribution Type of this instance (see Rule 9).

• attribute `partnerLink` is set to an instance of BPEL Partner Link that corresponds to this instance (see Rule 8).

• attribute `portType` is set to an instance of BPEL Port Type that corresponds to this instance (see Rule 4).

• attribute `operation` is set to an instance of BPEL Operation that corresponds to this instance (see Rule 6).

11. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as `<<reply>>`, create an instance of BPEL Reply whose

• attribute `name` is set to the value of the attribute `name` of this instance.

• attribute `variable` is set to an instance of BPEL Variable that corresponds to the instance of ISDL Information Attribute attached to the ISDL Interaction Contribution Type of this instance (see Rule 9).
• attribute `partnerLink` is set to an instance of BPEL Partner Link that corresponds to this instance (see Rule 8).
• attribute `portType` is set to an instance of BPEL Port Type that corresponds to this instance (see Rule 4).
• attribute `operation` is set to an instance of BPEL Operation that corresponds to this instance (see Rule 6).

12. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as `<<invoke>>`, create an instance of BPEL Invoke whose

• attribute `name` is set to the value of the attribute `name` of this instance.
• attribute `inputVariable` is set to an instance of BPEL Variable that corresponds to the instance of ISDL Information Attribute attached to the ISDL Interaction Contribution Type of this instance (see Rule 9).
• attribute `partnerLink` is set to an instance of BPEL Partner Link that corresponds to this instance (see Rule 8).
• attribute `portType` is set to an instance of BPEL Port Type that corresponds to this instance (see Rule 4).
• attribute `operation` is set to an instance of BPEL Operation that corresponds to this instance (see Rule 6).

13. For each instance of ISDL Interaction Contribution Instantiation whose type is stereotyped as `<<result>>`, set the attribute `outputVariable` of the instance of BPEL Invoke, that corresponds to the instance of ISDL Interaction Contribution Instantiation that enables this instance, to an instance of BPEL Variable that corresponds to this instance (see Rule 12).

14. For each instance of ISDL Action Instantiation whose type is stereotyped as `<<terminate>>`, create an instance of BPEL Terminate whose

• attribute `name` is set to the value of the attribute `name` of this instance.

15. For each instance of ISDL Time Attribute, create an instance of BPEL Wait whose

• attribute `name` is set to `wait` concatenated with the value of the attribute `name` of the instantiation of the ISDL Interaction Contribution Type or the ISDL Action Type where this instance is attached.
• attribute `for` is set to the duration expression in the attribute `expression` of an instance of the ISDL Alternative Attribute Constraint attached to this instance.

16. For each instance of ISDL Information Attribute that has ISDL Alternative Attribute Constraint attached, create an instance of BPEL Assign whose

• attribute `name` is set to `assign` concatenated with the value of the attribute `name` of the instantiation of the ISDL Interaction Contribution Type or ISDL Action Type where this instance is attached.
and then create an instance of BPEL Copy whose

- attribute `owner` is set to the instance of BPEL Assign that is just created.

and then create an instance of BPEL From whose

- attribute `variable` is set to an instance of BPEL Variable that has a same name with the right-hand-side part of the expression of the ISDL Alternative Attribute Constraint, or
  attribute `expression` is set to an XPath expression that represents the attribute `expression` of the ISDL Alternative Attribute Constraint.
- attribute `owner` is set to the instance of BPEL Copy that is just created.

and then create an instance of BPEL To whose

- attribute `variable` is set to an instance of BPEL Variable that corresponds to this instance (see Rule 9).
- attribute `owner` is set to the instance of BPEL Copy that is just created.

17. For the instance of ISDL Entry Point of an instance of ISDL Behaviour Type stereotyped as `<<process>>`, create an instance of BPEL Sequence whose

- attribute `owner` is set to an instance of BPEL Process that corresponds to the instance of ISDL Behaviour Type (see Rule 1).

and then traverse all instance of ISDL Causality Relations started from the instance of ISDL Entry Point:

- Each time an instance of ISDL Interaction Contribution Instantiation or ISDL Action Instantiation is encountered, the attribute `sequence` of its corresponding BPEL element instance is set to the instance of BPEL Sequence that is just created (see Rules 15 and 16).
- If the instance of ISDL Interaction Contribution Instantiation or ISDL Action Instantiation also corresponds to instances of BPEL Wait and/or BPEL Assign, then their the attributes `sequence` are set prior to setting the attribute `sequence` of its corresponding BPEL element instance.

Rules 3 to 9 only creates a BPEL element instance that corresponds to an ISDL element instance if the same BPEL element instance is not created yet; otherwise the rule makes the ISDL element instance to correspond to the already created BPEL element instance.
Appendix D

Examples of Generated
WSDL and BPEL Documents

The following are documents generated by the ISDL-to-BPEL transformation for an example given in Chapter 8.

D.1 WSDL Document

Listing D.1: WSDL document of the BPEL process PurchaseHandling

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/
xmlns:tns="http://example.nl/process"
targetNamespace="http://example.nl/process"
name="PurchaseHandling">

    <import namespace="http://example.nl/schema"
            location="http://example.nl/schema/schema.xsd"/>

    <message name="PurchaseOrder">
        <part name="payload" element="xxx:XXX"/>
    </message>

    <message name="Schedule">
        <part name="payload" element="xxx:XXX"/>
    </message>

    <message name="Invoice">
        <part name="payload" element="xxx:XXX"/>
    </message>

    <portType name="requester">
        <operation name="callbackSchedule">
            <input name="schedule" message="tns:Schedule"/>
        </operation>
    </portType>

    <portType name="provider">
        <operation name="sendPO">
            <input name="PO" message="tns:PurchaseOrder"/>
            <output name="invoice" message="tns:Invoice"/>
        </operation>
    </portType>
</definitions>
```
D.2 BPEL Document

Listing D.2: BPEL document of the BPEL process PurchaseHandling

```xml
<?xml version="1.0" encoding="UTF-8"?>
<process xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://example.nl/process"
  suppressJoinFailure="yes"
  targetNamespace="http://example.nl/process"
  name="PurchaseHandling">

  <partnerLinks>
    <partnerLink name="shipper"
      partnerLinkType="shipper:XXX"
      myRole="invoker_XXX" partnerRole="provider_XXX"/>
    <partnerLink name="customer"
      partnerLinkType="tns:customer_Type"
      myRole="provider_Role" partnerRole="requester_Role"/>
    <partnerLink name="seller"
      partnerLinkType="seller:XXX"
      partnerRole="provider_XXX"/>
  </partnerLinks>

  <variables>
    <variable name="sendSchedule.schedule"
      messageType="tns:Schedule"/>
    <variable name="calculatePrice.PO"
      messageType="seller:PurchaseOrder"/>
    <variable name="receiveOrder.PO"
      messageType="tns:PurchaseOrder"/>
    <variable name="receiveSchedule.schedule"
      messageType="shipper:Schedule"/>
    <variable name="requestShipping.shippingReq"
      messageType="shipper:ShippingReq"/>
  </variables>
</process>
```
<variable name="returnPrice.priceList" messageType="seller:PriceList"/>

<variable name="sendInvoice.invoice" messageType="tns:Invoice"/>
</variables>

<sequence>
<receive name="receiveOrder" createInstance="yes" partnerlink="customer" portType="tns:provider" operation="sendPO" variable="receiveOrder.PO"/>

<assign name="assign.calculatePrice">
<copy>
  <from variable="receiveOrder.PO"/>
  <to variable="calculatePrice.PO"/>
</copy>
</assign>

<invoke name="calculatePrice" partnerlink="seller" portType="seller:provider" operation="sendPO" inputVariable="calculatePrice.PO" outputVariable="returnPrice.priceList"/>

<assign name="assign.sendInvoice">
<copy>
  <from variable="returnPrice.priceList"/>
  <to variable="sendInvoice.invoice"/>
</copy>
</assign>

<reply name="sendInvoice" partnerlink="customer" portType="tns:provider" operation="sendPO" variable="sendInvoice.invoice"/>

<wait for="PT12H" name="wait.requestShipping"/>

<assign name="assign.requestShipping">
<copy>
  <from variable="receiveOrder.PO"/>
  <to variable="requestShipping.shippingReq"/>
</copy>
</assign>

<invoke name="requestShipping" partnerlink="shipper" portType="shipper:provider" operation="requestShipping" inputVariable="requestShipping.shippingReq"/>

<receive name="receiveSchedule" partnerlink="shipper" portType="shipper:invoker"/>
<assign name="assign.sendSchedule">
  <copy>
    <from variable="receiveSchedule.schedule"/>
    <to variable="sendSchedule.schedule"/>
  </copy>
</assign>

<invoke name="sendSchedule"
  partnerlink="customer"
  portType="tns:requester"
  operation="callbackSchedule"
  inputVariable="sendSchedule.schedule"/>

<terminate name="end"/>
</sequence>