INSTITUT FÜR INFORMATIK
der Ludwig-Maximilian-Universität München

Diplomarbeit

Use Cases for Reactivity on the Web: Using ECA Rules for Business Process Modeling

Inna Romanenko

Aufgabesteller:	Prof. Dr. François Bry
Betreuer:	Prof. Dr. François Bry, Dr. Paula-Lavinia Pătrânjan
Abgabetermin:	24. Januar 2006

Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig verfasst habe, und keine anderen als die angegebenen Hilfsmittel verwendet habe.

München, den 24. Januar 2006 Inna Romanenko

Abstract

Nowadays the Web is not only a huge amount of data that can be searched for and viewed upon request. The Web becomes the means for enterprises to deliver their goods and services and for consumers to find and retrieve services that match their needs. *Reactivity on the Web*, the ability to detect situations of interest that occur on the Web and automatically respond to them, becomes a research area of a growing importance. Following a declarative approach to reactivity on the Web, a novel reactive language called *XChange* has been developed at the Institute for Informatics, University of Munich. An XChange program consists of a number of *Event-Condition-Action* rules, that specify the execution of certain actions in case certain situations take place.

A *business process* is a set of logically related activities that are combined to achieve some business goals. *Business process modeling* is a fundamental phase of each business process lifecycle, dealing with the design and implementation of business processes.

This work is dedicated to the investigation and demonstration on examples of the *rule-based* method to model business processes. These are implemented by means of the reactive language XChange, in conformity with the business rules approach – an exciting new technology beginning to have a major impact on the IT industry.

Zusammenfassung

Heutzutage ist das Web nicht mehr nur eine riesige Datenmenge, die man auf der Suche nach Informationen durchforscht. Das Web wird zum Mittel für Unternehmen, ihre Waren und Dienstleistungen an den Kunden zu liefern. *Reaktivität im Web*, die Fähigkeit, bestimmte Situationen im Web zu erkennen und automatisch auf sie zu reagieren, wird zu einem neuen Forschungsgebiet. Folgend dem deklarativen Ansatz, wurde die neuartige reaktive Sprache *XChange* am Institut für Informatik, Universität München entwickelt. Ein XChange Programm besteht aus mehreren *Ereignis-Bedingung-Aktion* Regeln, die beschreiben welche Aktionen in welchen Situationen ausgeführt werden.

Ein *Geschäftsprozess* ist eine Reihe von logisch zugehörigen Aktivitäten, die zusammengefasst dazu dienen, ein Unternehmensziel zu erreichen. Das *Modellieren von Geschäftsprozessen* ist eine grundsätzliche Phase jedes Geschäftsprozess-Lebenszyklus, das aus dem Design und Implementierung von Geschäftsprozessen besteht.

Diese Arbeit untersucht und demonstriert an Beispielen wie Geschäftsprozesse *regelbasierend* modelliert werden können. Dabei werden Geschäftsprozesse mit Hilfe von der reaktiven Sprache XChange implementiert, in Übereinstimmung mit dem Geschäftsregel Ansatz – eine aufregende neue Technologie, die beginnt, einen grossen Einfluss auf IT Industrie zu haben.

Acknowledgments

First of all, I would like to thank my supervisors *Prof. Dr. François Bry* and *Dr. Paula-Lavinia Pătrânjan*, both of the University of Munich, for supporting me in the writing of this work. I am very grateful for the time they have spent discussing research issues of the work and reading my thesis, for the extraordinary encouragement they have given me throughout the work, and faithful concern for my future.

I am also grateful to *Michael Eckert*, a fellow researcher of the University of Munich, for fruitful discussions and helpful suggestions on improving this thesis.

I would like to thank all the other fellow researchers of the *Programming and Modeling Languages research group* at the University of Munich, its technical and administrative staff for providing the pleasant working environment during the work.

I am thankful for the funding from the Project "Reasoning on the Web with Rules and Semantics" that enabled me to attend the European Business Rules Conference 2005 where I had the chance to accumulate a lot of informational material for my thesis.

Last but not least I thank my parents for loving and supporting me and my best friend *Tanja* for continously encouraging me.

CONTENTS

Part I

Introduction

CHAPTER

ONE

Introduction

XChange is a high-level, rule-based language for programming reactive behaviour and distributed applications on the World Wide Web (WWW, or Web for short). An XChange [58] program consists of a number of *Event-Condition-Action* rules, each of which specifies that some *action* is automatically performed in response to an *event*, provided that a *condition* holds. As an action such simple business activities as hotel reservation or issuing a bank order can be executed.

A *business process* is a set of logically related activities that are combined to achieve some business goals. *Business process modeling* is a fundamental phase of each business process lifecycle, dealing with the design and implementation of business processes.

There are different methods and tools for modeling business processes. In this thesis we investigate and demonstrate on examples the *rule-based* method. Through implementation of complex use cases in XChange, we determine whether, how, and to which degree the language (consequently, a rule-based method *in general*) is suitable for modeling (implementing) complex multi-step business processes; determine its potency and shortcomings in this issue; and analyse the rationality of the further language development to support such business applications.

1.1 Motivation

Reactivity, the ability to automatically react to previously defined situations with specified actions, is a growing area of importance in today's World Wide Web. Nowadays the Web is not only a huge amount of data (represented in XML or HTML documents) that can be searched for and viewed upon request. The Web becomes the means for organisations to deliver goods and services and for customers to search and retrieve services that match their needs. Hence, many Web-based systems need the capability to update local and remote data, to exchange information with other systems, and to automatically react to particular changes in the world's state. XChange, as a Web language for programming reactive behaviour and distributed applications, aims at providing the Web (largely a passive system) with such dynamic capabilities.

An XChange program consists of a number of Event-Condition-Action (ECA) rules. An ECA rule specifies events that the rule can detect automatically (*event* part), logical expressions that must be satisfied in order to activate the action part (*condition* part), and activities to be executed (*action* part). Thus, a rule can accomplish a business operation, such as hotel reservation or issuing a bank order, in response to an occurrence of a certain situation (for example, a customer request) provided the given conditions are satisfied. Implementations of such XChange rules are described and exemplified

in the doctoral thesis [58], in the master thesis [27], and in a number of related research articles [15, 13, 5, 16, 4, 12]. Though XChange provides a wide range of constructs and mechanisms, as demonstrated in these implementations, the language practicability is still to be tested. Developing complex application examples (use cases) is the best way to do it. On the one hand, implementing real-life situations proves the practicability of the language itself. On the other hand, it motivates the existing language constructs, discovers missing ones, and gives rise to refinements of the language design.

Many present-day information systems require an automatic support for execution of *business processes*. A business process is a set of logically related activities that are combined to achieve a business goal. For example, handling of insurance applications, allocation of mortgage loans, or business trip authorisation. We consider the modeling of business processes along with the realisation of the *business rules approach* – a new area of a growing importance in today's process management systems.

To the best of our knowledge, the challenge to model (design or implement) complete business processes with a rule-based language exclusively has not been investigated yet. The research point of scientific works on ECA rules is either their using in active databases [55, 60, 67, 37], Web Service applications such as publish/subscribe systems or personalisation [11, 1, 57, 21, 10], or some specific problems within workflow execution such as detection of exceptional situations or activation of alerts [32, 44, 30, 2, 6, 3, 24, 59].

The question whether and how the language XChange supports the mechanisms and constructs essential and typical for the modeling (implementation) of business processes has not been yet investigated. We believe, the capability to support business processes might strengthen the position of XChange as a Web language. On the other hand, if the ability to realise business processes demands a lot of new constructs, extensive changes in the currently compact and easy to understand syntax, and the significant increase of the language semantics complexity, the rationale of the idea to support business process might be reconsidered.

1.2 Research Topic

This work investigates the *business process modeling using ECA rules* by means of the reactive language XChange.

The essential points in our research are:

- to investigate the standards and products used or tend to be used in industrial information systems supporting business processes;
- to investigate the standards and products used or tend to be used in industrial information systems supporting business rules approach;
- to determine the requirements to a programming language for business process modeling;
- to analyse the language XChange in terms of its suitability and expressive power to provide mechanisms and constructs essential for business process modeling;
- determine the potency and shortcomings of XChange for this type of applications;
- make proposals for the further development of XChange, based on the detected shortage of the language to satisfy the requirements for the business processes automatisation;

- make a concluding resume about the rationality of the further development of XChange as a business process modeling language.

1.3 Outline

The remainder of this work is structured into three parts containing six chapters and an appendix. In the first part (called *Fundamentals*) we present the paradigms and design decisions of XChange that provide the basis for working on the language (Chapter 2). In the same part we give extensive overviews of the current standards and products in the business processes management systems (Chapter 3) and in business rules management systems (Chapter 4). In the second part (called *Use Cases in XChange*) we first provide a detailed analysis of XChange based on patterns specific to processes with the objective of determining whether and how XChange supports the business processes functionality (Chapter 5). Then we present a use case called EU-Rent that is well-known in the business rules community. Finally, we illustrate the implementation models of particular EU-Rent business processes and business rules (Chapter 6). We summarise the contributions of this work and conclude this thesis in the third part (called *Conclusion*). This is followed by the appendix containing XChange programs realising EU-Rent business processes and business rules.

Part II

Fundamentals

CHAPTER
TWO

The Reactive Language XChange

XChange is a rule-based declarative language for programming reactive behaviour on the Web; it has been developed as a research project at the Institute for Informatics, University of Munich. *Reactivity* on the Web is the ability to *automatically* respond to certain "events" (happenings or changes in the world's state). In contrast to other reactive languages, XChange supports *complex* update operations and provides the ability to detect and react to *combinations of events*.

This chapter gives an overview of XChange. We first introduce the paradigms of the language and its design principles (Section 2.2). Then we analyse subsequently three main parts of an XChange rule – the event part (Section 2.5), the condition part (Section 2.6), and the action part (Section 2.7).

2.1 Introduction

XChange[1] is a Web reactivity language that has been developed as a research project at the Institute for Informatics, University of Munich. XChange has been the subject of the Ph.D. project of Dr. Paula-Lavinia Pătrânjan [58] written under the supervision of Prof. Dr. François Bry.

Reactivity is the ability to *automatically* react to certain changes in the world's state. Thus, with a reactive program a Web site can detect some happenings it is interested in and respond to them either through updating one or more Web resources (such as HTML or XML documents) or through sending messages to one or more Web sites. For example, an information Web site of a railway station can react automatically to incoming messages about train delays with changes in the timetable; a Web-based personalised organiser might be conceived to respond to certain messages with sending emails to specified persons; etc.

XChange is a language for writing such reactive programs. An XChange program runs locally at a Web site (called an *XChange-aware Web site*), can detect previously specified situations of interest, and respond automatically by executing previously specified actions.

XChange has an advantage over other reactive languages – it supports *complex* update operations and is susceptible to complex *combinations* of incoming events.

2.2 Paradigms

This section introduces the paradigms the language XChange relies upon.

[1] XChange Project, `http:/www.xcerpt.org/related/xchange`

2.2.1 Reactivity to Events

XChange programs are reactive: when an expected event (or combination of events) occurs, they respond automatically by executing some previously specified actions. An *event* is some happening, a change in the state of the world. XChange differentiates between *atomic events*, which occur at a single time point, and *composite events* – combinations of atomic events, occurring over some time period that has a beginning and ending time. Different XChange-aware Web sites communicate with each other by sending atomic events; therefore, we call the representation of an atomic event an *event message*. Which atomic events and which combinations of atomic events are relevant for a Web site is predefined in local reaction programs. These specify which events the Web site reacts upon and what are the reaction activities.

2.2.2 Rule-Based Language

An XChange reactive program is mainly a set of *Event-Condition-Action* (ECA) rules[2] (additionally it may contain deduction rules, see Section 2.6). An ECA rule specifies that some *action* is performed automatically in response to an *event*, provided that a *condition* holds.

A typical XChange rule has the corresponding form *Event query - Web query - Action* (though in another sequence order, see Section 2.4). *Event query* is a query against event data. It is an event message pattern that may contain variables for selecting parts of the events' representation. This pattern is evaluated against each incoming event. If they match[3], the rule is *triggered*: its *Web query* is evaluated. Web query is a query against persistent data (data of Web resources). If the Web query also has an answer (i.e. it was successfully evaluated against data of the specified resources), the rule is *fired*: its *action* is executed. Note that the data extracted by the event query can be used in the condition and in the action part; the data extracted by the Web query can be used in the action part.

2.2.3 Communication Issues

XChange programs run locally at a Web site and can modify directly local resources. XChange programs can communicate with each other by sending representation of events – event messages.

Push Communication of Events XChange programs running at different Web sites communicate through sending and receiving of (atomic) events. XChange uses a *push* approach to forward information. In push communication, event messages are *sent* to the Web sites interested in a certain event (called *observers* or *recipients*) by the Web site where the event is raised (called *emitter* or *sender*).

Alternatively, a *pull* communication is possible: observers poll the emitter Web site periodically to check if a certain event has occurred.

As mentioned in [58], for reactivity on the Web push communication seems to be better suited. It causes less network traffic, allows faster reaction to events, and allows the emitter to initiate communication.

Peer-to-Peer Communication XChange is designed consistently to the *decentralised* manner of the Web: it is based on a *peer-to-peer* model. That means that XChange-aware Web sites communicate with each other without a centralised instance that controls and synchronise the communication. Event

[2] ECA rules are a technology from active databases [71, 56, 26, 9, 55, 60, 67, 37].

[3] The matching of an atomic event query with an incoming event is based on the *simulation unification*, a novel unification method developed for matching query terms with data or construct terms in Xcerpt [65].

messages are exchanged directly between Web sites; all Web sites are equal partners; all of them can initiate the communication.

Asynchronous Communication XChange supports only *asynchronous* type of communication: after sending an event, the XChange program is not-blocked and can continue its work. Alternatively, in case of a synchronous communication, after sending a message the program is blocked till the reply is received. In the Web – a system with unreliable communication – the suspension time, during which other incoming events cannot be reacted, can be quite long. Consequently, an asynchronous communication seems to be more convenient for a Web reactivity language.

2.2.4 Pattern-Based Approach

XChange is a *pattern-based* language: event queries, Web queries, event raising notifications, and updates describe *patterns* for events of interest, Web data, raising event messages, and update operations respectively. All the named specifications are based on data; thus, only one concept – that of data pattern – is needed. This uniform pattern principle allows both easy programming and reading. The principles of data notification will be introduced in the following section.

2.3 Integrated Query Language Xcerpt

In realising reactivity on the Web, data extracted from Web resources play an essential role. For example, to implement reactivity of an information Web site of a railway station that makes automatic changes in the timetable in response to incoming messages about train delays, a corresponding reactive program would probably need some database information about train connections (i.e. data from Web resources). Additionally, in many cases the event data themselves have to be queried to gain some information required in the condition and action parts of the rule.

To be able to query volatile data in events and persistent data in Web resources, XChange embeds Xcerpt, a query language for XML and other semi-structured data. A full introduction to Xcerpt is beyond the scope of this work, but the following should give the reader the general ideas needed to understand XChange and the rest of this thesis. For more detailed information about the language Xcerpt and its use in XChange, we address the reader to the following works: [65, 58, 27].

Data Terms represent data items of a Web document. To understand the term syntax used by Xcerpt, we can compare the following fragment of data in XML

```
<connection>
  <number>4711</number>
  <departure>Munich</departure>
  <destination>Berlin</destination>
</connection>
```

with its equivalent (quite straightforward to understand) representation in Xcerpt:

```
connection {
   number {"4711"},
   departure {"Munich"},
   destination {"Berlin"}
}
```

In this example, the curly braces {} denote an *unordered term specification*. To specify that the order of the sub-terms is relevant we use square brackets `[]`.

Query Terms represent a *pattern* for data terms. Xcerpt differentiates between *total* (complete) and *partial* (incomplete) query pattern. Partial query patterns match data terms that contain the specified content, but may also contain additional elements. Such patterns are denoted with double square `[[ ]]` or curly {{ }} brackets. Total query patterns match only those data terms that contain exactly the specified content without any other elements. They are denoted by single square `[]` or curly {} brackets. Thus, the following three query terms match the above specified data term:

```
connection {{
   destination {"Berlin"},
   number {"4711"}
}}

connection [[
   number {"4711"},
   departure {"Munich"}
]]

connection {{ }}
```

The following query, however, does not match the same data term since the query pattern is *total*, but the sub-term `destination` is not specified:

```
connection {
   number {"4711"},
   departure {"Munich"}
}
```

Note that the used data term itself (as every other data term) is also a query term. Additionally, query terms can contain:

- variables (`var Nr` in the example) serving as placeholder for arbitrary content:

  ```
  connection {{
     number { var Nr }
  }}
  ```

- variable restriction serving as a "restricted" placeholder, i.e., the placeholder for the specified content (in the example, the content that match the pattern `destination {{ }}`):

  ```
  connection {{
     var X -> destination {{ }}
  }}
  ```

- the construct *without*, which allows to specify a query term that matches only if the data term introduced by the keyword `without` cannot be found in the queried data term;

- other constructs, such as *descendant*, *optional*, *position*, etc. Their detailed explanation can be found in [65].

A Query is a number of query terms bind together with the boolean connectives `and` and `or` and the unary `not`. A query in Xcerpt rules is introduced with the keyword `FROM`. It is always associated with one or more resources and is evaluated against the data terms of these resources. For example, the query

```
FROM
   in {resource {"http://www.railway.com/connections.xml", "xml"},
      connections {
         connection [[
            number {var Nr},
            departure {"Munich"},
            destination {"Berlin"}
         ]]
      }
   }
```

is evaluated against the data of the XML file, identified by the URI "http://www.railway.com/connections.xml".

In addition to an XML document, identified by an URI, as in the example above, a resource can be a data "view" created with a *construct-query* Xcerpt rule (cf. example in the following paragraph).

Queries can be restricted by constraints using the `where` clause. For example,

```
FROM
   in {resource {"http://www.railway.com/connections.xml", "xml"},
      connections {
         connection [[
            number {var Nr},
            departure {"Munich"},
            destination {"Berlin"},
            price { var P }
         ]]
      } where var P < 120
   }
```

selects only those train connections from Munich to Berlin whose price is less than 120 units of currency used in the database.

Construct Terms are patterns used to construct new data terms. They may contain variables whose bindings are specified in query terms and *grouping constructs* to collect *some* or *all* instances that result from different variable bindings. Construct terms are used in the construction part of an Xcerpt rule, introduced with the keywords `CONSTRUCT` or `GOAL`. For example, the following construct-query Xcerpt rule relates a construct term `connections-to-berlin` to a query:

```
CONSTRUCT connections-to-berlin {
   all var Connection
}
FROM
```

```
   in {resource {"http://www.railway.com/connections.xml", "xml"},
      connections {
         var Connection -> connection {{
            departure {"Munich"},
            destination {"Berlin"}
         }}
      }
   }
END
```

The result of the query will be a set of substitutions for the variable `Connection` (one per each train connection from Munich to Berlin). When constructing a data term from the construct term, variables are replaced by their value from the substitutions. Thus, the construct `connections-to-berlin` contains all those data terms `connection` from the Web resource "http://www.railway.com/connections.xml" whose sub-terms `departure` and `destination` have values "Munich" and "Berlin" respectively.

The specified construct-query rule can be used as a resource in other Xcerpt rules, for example:

```
CONSTRUCT
   <construct term>
FROM
   connections-to-berlin {{
      connection {{
         number { var Nr }
      }}
   }}
END
```

2.4 XChange Rules

XChange has a compact syntax that is compendious and easy to use by programmers. We introduce the corresponding syntax rules continuously through diverse examples in sections devoted to XChange events (Section 2.5), conditions (Section 2.6), and actions (Section 2.7). Here we would like to show an implementation schema of an XChange rule to give the reader an idea of its structure. For this purpose we use the example from above: *an information Web site of a railway station can react automatically to incoming messages about train delays* `(delay-notification)` *with changes in the timetable*. The corresponding XChange rule would look like:

```
DO
   <!-- update the local XML document: add new delay information -->
ON
   <!-- event query specifying the event message pattern
        for the atomic event labelled "delay-notification" -->
FROM
   <!-- Web query extracting more information about the delayed
        connection, which is to be used in the action part -->
END
```

As we see: an event part of an XChange rule is introduced by the keyword `ON`. The subsequent event query specifies an event message pattern for the event of interest – `delay-notification`. The Web query is introduced with the keyword `FROM`. It is executed after successful evaluation of the event query against one of the incoming events. Provided that the Web query satisfies, the action part, introduced by the keyword `DO`, will be executed.

Note that the notification of the action part given in the doctoral thesis [58] differs from the presented above. In the earlier version of the XChange language, an action part could be initiated either with the keyword `RAISE` or `TRANSACTION` to differentiate between *event-raising* and *transaction* rules. These rules specified respectively either events to be sent or transactions (updates and events which have to be executed in an *all-or-nothing* manner). However, transaction management is currently not considered in XChange and left for future work. In the actual version of the language, the syntax we presented above is used. A single action defined in the the action part is still either a raising event (an event to be sent) or an update of a Web document.

2.5 XChange Events

The event part of an XChange rule specifies an event or a combination of events the rule reacts upon. XChange-aware Web sites communicate with each other through sending and receiving of single (atomic) events, more precisely their representation, which is called in XChange *event message*. An event message is an XML document with a root element labelled `event`, with five prescribed child elements (`raising-time`, `reception-time`, `sender`, `recipient`, and `id`) and with other child elements that can be chosen freely to meet the application's requirements. Being XML documents, XChange event messages represent Xcerpt data terms (as in the code sample below) and thus, incoming event messages can be analogously queried to querying persistent data.

```
xchange:event {
   xchange:sender {"http://railway-station.com"},
   xchange:recipient {"http://railway-info.com"},
   xchange:raising-time {"2006-01-01T18:00"},
   xchange:reception-time {"2006-01-01T18:01"},
   xchange:id {"8214"},
   delay {
      connection { "4711" },
      minutes { "30" }
   }
}
```

2.5.1 Event Queries

The class of events an XChange rule reacts upon is expressed by an *event query* – a pattern for relevant events which trigger the XChange rule. In addition, event queries extract event data that can be used in the condition and action parts of the rule (in form of substitutions for variables). An event query can be atomic or composite. Atomic event queries detect atomic events, while composite event queries specify patterns for a combination of several atomic events.

2.5.2 Atomic Event Queries

An atomic event query is an Xcerpt query term (cf. Section 2.3) that specifies a pattern for a single event message. An atomic event query is evaluated against each incoming event message. If the evaluation is successful, the condition part of the rule is to be checked; if it satisfies, the action part is executed. The data in an XChange rule flow according to this order: the variable substitutions from the event part can be used in the condition part and then in the action part (together with the variable substitutions from the condition part).

The following (more detailed) implementation schema from Section 2.4 shows an atomic event query, specifying the pattern of the event message of interest – `delay-notification`:

```
DO
   <!-- update the local XML document: add new delay information -->
ON
   xchange:event {{
      xchange:sender {"http://railway-station.com"},
      delay-notification {{
         connection { var Nr },
         minutes { var Delay }
      }}
   }}
FROM
   <!-- Web query extracting more information about the delayed
        connection, which is to be used in the action part -->
END
```

The event query would match an event message with a content labelled `delay-notification` if it was sent by the Web site addressed at "http://railway-station.com" and has at least two sub-terms: `connection` and `minutes`.

Absolute Temporal Restrictions are used to restrict the number of atomic events of interest (specified at least by one atomic event query of a local reactive program) to those that occur within a given time interval or before a given time point. XChange provides two constructs for realisation of absolute temporal restrictions of atomic events: `in` for introducing a time interval and `before` for introducing a time point. For example, since the query

```
xchange:event {{
   example-event {{ }}
}} before 2006-01-02T14:00:00
```

is restricted with the construct `before`, it detects only those `example-events` which occur before 2006-01-02T14:00:00 (information about date and time formats used in XChange can be found in the doctoral thesis [58], page 72-74).

2.5.3 Composite Events Queries

There are cases when a situation that requires a reaction cannot be inferred from a single event. For example, a password change for online-banking does not require any fraud detecting reaction from

the Web site. However, a password change followed by one or more expensive purchases can be conceived as a suspicious situation requiring certain inspection activities. Another scenario is, for example, a situation when a Web site of a travel office initiates a travel destination inspection if a number of customers' claim messages exceeds a certain upper bound.

To accommodate such situations, XChange supports so-called *composite event queries*. Composite event queries are combinations of atomic event queries connected together with *composition operators*. All composition operators supported by XChange and the corresponding event compositions are presented in the following after a short introduction to *temporal restrictions* on composite event queries.

Temporal Restrictions on composite event queries are used not only to restrict the number of events of interest. They also assure that event data are released from the memory sooner or later. Without any temporal restriction this can be guaranteed only if all events of the specified event queries actually occur. The more detailed discussion about the necessity of a bounded time span for composite event queries is postponed to Section 2.5.4; here we present XChange constructs that support temporal restrictions.

Absolute Temporal Restrictions of composite event queries can be specified with the constructs, we have demonstrated for absolute temporal restrictions of atomic event queries: `in` for specifying a finite time interval and `before` for specifying a time point. However, in case of a restricted *composite* event query, both the beginning of the composite event (occurrence time of the first event from the specified event queries) and the ending time (occurrence time of the last event) should take place within the specified time period or before the specified time point.

Relative Temporal Restrictions determine the maximal time interval between the beginning and the ending time of the composite event specified by the composite event query. Note that the beginning time of a composite event is the earliest event occurrence time of all constituent events, while the ending time is the occurrence time of the latest event that is necessary for the successful evaluation of the corresponding event query. The relative temporal restriction allows only composite events that happen *within* the specified length of time (the ending time can be an earlier time point than the latest one denoted by the restriction). Relative temporal restrictions are initiated with the keyword `within` and can specify the time interval as a positive number of years, days, hours, minutes, or seconds.

Event Composition

Temporal and Non-Temporal Event Composition

- **Conjunction** of event queries detects occurrences of all constituent events. The occurrence order of the events is not relevant. A conjunction of event queries is expressed with the composition operator `and`:

```
ON and {
      xchange:event {{
         xchange:sender {"http://example.com/senders/senderA"},
         example-event-A {{ }}
      }},
      xchange:event {{
         xchange:sender {"http://example.com/senders/senderB"},
         example-event-B {{ }}
```

```
      }}
   }
```

- **Temporally Ordered Conjunction** of event queries detects successive occurrences of events. Such event queries have to be *ordered* – denoted with square `[]` brackets – to support the idea of the successive (ordered) receiving of events and are introduced with the keyword `andthen`:

```
ON andthen [
      xchange:event {{
         xchange:sender {"http://example.com/senders/senderA"},
         example-event-A {{ }}
      }},
      xchange:event {{
         xchange:sender {"http://example.com/senders/senderB"},
         example-event-B {{ }}
      }}
   ]
```

- **Inclusive Disjunction** of event queries detects occurrence of *one* from all constituent events. Thus, an occurrence of a single event from the specified ones is sufficient to trigger the corresponding rule. The composition operator for inclusive disjunction is `or`:

```
ON or {
      xchange:event {{
         xchange:sender {"http://example.com/senders/senderA"},
         example-event-A {{ }}
      }},
      xchange:event {{
         xchange:sender {"http://example.com/senders/senderB"},
         example-event-B {{ }}
      }}
   }
```

  Note that in case both events occur, the rule is triggered twice. For the realisation of the *exclusive disjunction* another XChange construct can be used (cf. paragraph **Multiple Selections and Exclusions** in the following).

- **Event Exclusion** detects the *non*-occurrence of a certain event within a given interval, determined either by a finite time interval or by events of a composite event query (in both cases the keyword `during` is to be used)[4]. The event message pattern of the event that is expected *not* to occur is introduced by the keyword `without`:

```
ON without {
      xchange:event {{
         example-event-A {{ }}
      }}
   } during [2006-01-02..2006-01-03]
```

[4]The same definition of an interval is to be implied on other types of composite event queries we discuss.

Note that the specified event query is evaluated once at time point 2006-01-03, that is, when the absolute time interval ends.

Multiple Selections and Exclusions detect a certain number of events within a given interval. Thus, the operator `m of` $\{eq_1, eq_2, \ldots, eq_n\}$ detects occurrences of *exactly* `m` events of the specified event queries (where the eq_i are event queries, `m` and `n` are integers with `m<=n`). Note that in case `m=1`, the construct is equivalent to exclusive disjunction of event queries:

```
ON 1 of {
      xchange:event {{
         xchange:sender {"http://example.com/senders/senderA"},
         example-event-A {{ }}
      }},
      xchange:event {{
         xchange:sender {"http://example.com/senders/senderB"},
         example-event-B {{ }}
      }}
   } during [2006-01-02..2006-01-03]
```

Occurrences

- **Quantifications** of event queries specify that at least, at most, or exactly a given number of events specified in the event query should occur during the given interval. XChange provides following quantification specifications: `times n {eq}`, `times atmost n {eq}` , and `times atleast n {eq}` (where `eq` denotes an event query and `n` is an integer). For example, the event query

  ```
  times atleast 3 {
     xchange:event {
        example-event {{
           subject { var S }
        }}
     }
  } during [2006-01-02..2006-01-03]
  ```

  is evaluated at time point 2006-01-03; the evaluation is successful if in the time period from 2006-01-02 at least three messages `example-event` with the content labelled `subject` has been received. Note that the `subject` of all received messages has to be *the same*, as the variable *S* requires equality. To overcome this, the keyword `any` can be used (it denotes existentially quantified variables – variables that can have different assignments). Corresponding examples and more detailed explanations can be found in [58, 27], pages 81-82, 62-63, respectively.

- **Repetitions** detect every *n*-th occurrence of the event specified in the event query. For example, the rule

  ```
  DO <action>
  ON every 2 {
  ```

```
   xchange:event {{
      xchange:sender {"http://example.com/senders/senderA"},
      example-event-A {{ }}
   }}
} in [2006-01-02..2006-01-03]
FROM <condition>
END
```

is triggered on receiving every other event message `example-event-A` from the specified sender within the given time interval.

- **Ranks** detect an event that matches the specified event query having a given rank (or position) in the incoming stream of events, restricted with a given interval. Such event queries are denoted with `withrank n` {`eq`} or with `last` {`eq`}:

```
ON last {
   xchange:event {
      example-event-B {{ }}
   } during {
      andthen [
         xchange:event {
            example-event-A {{ }}
         },
         xchange:event {
            example-event-C {{ }}
      ]
   }
}
```

For a more detailed discussion on XChange composite events see [58, 27].

2.5.4 Legal Event Queries

XChange is designed in a way that every event can be disposed of after a bounded time. The driving force of this principle is to keep storage requirements constant.

Every incoming event is tested against each event query. Irrespective whether it answers the atomic event query or not, an atomic event can be discarded immediately and consequently does not really need a life-span at all. The situation differs for a composite event query. Assume, a composite event query specifies a conjunction of two atomic events – `eventA` and `eventB`. Once `eventA` has been received, the fact that `eventA` happened and maybe also data contained in `eventA` has to be kept in memory at least until `eventB` happens. However, there is no guarantee that `eventB` will actually happen, so `eventA` might have to be kept in memory forever.

To avoid such situations, all composite event queries in XChange have to explicitly specify a "time-out" that gives a maximum life-span to each event that is potentially part of an answer to a composite event query. This upper bound on the life-span of an event can be, for example, *during an hour*, *within a day*, and so on. XChange captures this restriction on event queries in the notion of *legal* event queries. Thus, every composite event query without time restriction is *not legal*. More about legal event queries can be found in [27], Section 4.8.

2.6 XChange Conditions

The condition of an XChange rule is an Xcerpt query (cf. Section 2.3). The query extracts data from persistent Web resources; it is hence called *Web query* or *data query* in contrast to *event queries* specified in the event part. The Web query is evaluated when the XChange rule is triggered. The Web query can make use of the variable substitutions provided from the event query. If the Web query has no answer (the evaluation is unsuccessful), the rule execution is aborted and the action part will not be executed. Otherwise, it delivers substitutions for the free variables of the Web query, and these can be used in the subsequent execution of the action part.

Web queries can query Web resources directly or by querying data views created by means of *deductive rules* (Xcerpt construct-query rules). Usage of deductive rules provides an elegant solution for complex queries over multiple, heterogeneous data sources. Thereby all views used in XChange rules have to be located in the same XChange program. Examples of a Web query and of a deductive rule can be found in our introduction over the language Xcerpt (Section 2.3).

2.7 XChange Actions

The action part of an XChange rule is executed when both event and Web queries evaluated successfully. As an action the following activities are possible: raising an event or executing an update (modifying local or remote Web data). An action part can also specify a *complex action*, i.e. a combination (such as conjunction) of single actions.

2.7.1 Raising Events

One of the possible actions specified in the action part of an XChange rule is raising an event. It is a specification of an event message that has to be constructed and sent to one or more Web sites as a reaction to the detected situation of interest (occurrence of relevant events plus satisfaction of the condition part).

Analogously to the notification of an event message in the event part, an event message of an event to raise has the root element labelled `xchange:event`. As a parameter only `xchange:recipient` with one or more valid URIs as its children can be specified. Parameters `xchange:sender` and `xchange:raising-time` are provided automatically by the local event manager while sending the message; `xchange:reception-time` and `xchange:id` are determined and inserted by the event manager on the recipient Web site when the event message is received.

For example, a raising event informing about the thirty minutes delay of the train connection with number “4711” can be implemented as following:

```
DO
   xchange:event {
      xchange:recipient {"http://railway-info.com"},
      delay-notification {
         connection { "4711" },
         minutes { "30" }
      }
   }
ON
   <event>
```

```
FROM
   <condition>
END
```

2.7.2 Updates

Elementary updates provide another form of actions realisable with XChange. These can be used for modifying Web resources, either local or remote. XChange defines three types of update operations:

- `insert` *construct-term*
- `delete` *query-term*
- *query-term* `replaceby` *construct-term*

The meaning of update operations is rather obvious: `insert` *construct-term* inserts a new sub-term constructed from the *construct-term*; `delete` *query-term* deletes all sub-terms that match *query-term*; *query-term* `replaceby` *construct-term* replaces all sub-terms that match *query-term* with *construct-term*.

Through combination of query terms with update operations, we can specify *update terms* – patterns for data to be updated. Query terms locate the fragments of data to be modified. Update operations execute the specified modifications. Note that an update term is always associated with a Web resource; can contain more than one update operation; does not allow nesting of update operations. Within an update term update operations can be located on any position where a normal data term can be placed.

The example below shows an elementary update that consists of an update term and a resource specification. Hence, the execution of the action part of this rule results in the insertion a new data term `connection` into the XML file “http://www.railway.com/delays.xml”.

```
DO
   in {resource {"http://www.railway.com/delays.xml", "xml"},
      connections {{
         insert connection {
            number { var Nr },
            delay { var Delay }
         }
      }}
   }
ON
   xchange:event {
      xchange:sender {"http://railway-station.com"},
      delay-notification {{
         connection { var Nr },
         minutes { var Delay }
      }}
   }
END
```

A deeper discussion on updates is provided in [58], pages 95-114.

2.7.3 Complex Actions

Complex Updates Along with elementary updates, XChange supports the notion of *complex updates*. It is a more powerful construct that allows modification of more than one Web resource. A complex update is an ordered or unordered disjunction or conjunction of updates.

A disjunction of updates is used when *exactly* one of specified updates has to be executed; it is initiated with the keyword *or*. In case of the unordered disjunction (denoted with already known curly {} braces) the execution order is irrelevant and can be freely chosen by the runtime system. Specifying the ordered disjunction (using square `[]` brackets), the user gives the execution order explicitly. In both cases, the complex update is finished when *one* elementary update is successfully executed. The complex update failed when *none* of the elementary updates could be executed.

A conjunction of updates demands the execution of all specified elementary updates. It is initiated with the keyword `and`. An ordered conjunction identifies the execution order; in case of an unordered conjunction it is arbitrarily. Note that the result of an unordered conjunction is nondeterministic if the updates modify the same data.

Complex Event Actions and Complex Mixed Actions In real-life applications it is often required to react with raising (quasi simultaneously) multiple events or both with raising events and updating Web data. For example, along with changes on the timetable, an information Web site of a railway station might forward received information about train delays to other train connections or other railway stations (through sending event messages). Analogously to the notion *complex updates*, we capture a combination of different raising events, specified within one and the same action part, in the notion *complex event actions*; a combination of actions of different types (i.e. raising events and updates) can be called *complex mixed actions*. Note that the need of complex event actions and complex mixed actions was recognised in the process of the language development; thus, yet not discussed in the fundamental project works, such as [58, 27].

Analogously with complex updates, we differentiate the following type of complex event actions and complex mixed events:

- unordered disjunction
- ordered disjunction
- unordered conjunction
- ordered conjunction

The semantics and syntax constructs are conform to those we described for complex updates.

2.8 Summary

In this chapter, we have given an overview of the reactive language XChange. An XChange program consists of one or more ECA rules and possibly some deduction rules of the form `CONSTRUCT-FROM`. The latter can be used for evaluating condition parts of XChange rules. An ECA rule specifies that some *action* is performed automatically in response to an *event*, provided that a *condition* holds.

The event part of an XChange ECA rule specifies event queries – patterns for relevant event messages. Once a relevant event (or events' combination) is detected, a rule tests its condition(s), specified with a query against Web data. If the test is successful, the action part is executed. This can

be a single raising event, an update, or (ordered or unordered) conjunction or disjunction of raising events and updates. Both event and Web queries can contain variables whose substitutions can be used in the following rule processing.

XChange is a pattern-based language: event queries, Web queries, event raising specifications, and updates are patterns based on data terms. XChange embeds the Web query language Xcerpt to query volatile data in events and persistent data in Web resources.

The full language specification including operational and declarative semantics can be found in the doctoral thesis [58].

CHAPTER
THREE

Business Process Modeling

During the last years, companies have been experiencing many changes in their business environments. Growing competition, increase in multi-enterprise collaboration, and e-business have created the necessity of technology and tools to ensure efficient and effective process management. A *Business Process Management System* (BPMS), a software to define, manage, execute, and monitor complex business processes [72] provides such a solution. In this chapter we expose the concepts, design, and standard specifications of BPMSs. As the main objective of our work is to investigate the strengths and possible shortcomings of the programming language XChange, particular emphasis is placed on the *Business Process Modeling*, the fundamental phase of each business process lifecycle [73], dealing with design and implementation of business processes.

Section 3.1 provides an introduction into the concepts and design principles of Business Process Management Systems. The underlying software architecture of a BPMS realisation is presented in Section 3.2. Section 3.3 is devoted to the prevalent implementation solution of the architecture – Web services. Section 3.4 overviews the standard business process management stack, including a modeling language, an execution business process language, and a language for incorporating multiple Web services into one business process. In the final section we sum up our discussion with a presentation of a BPMS model.

3.1 Introduction

In an increasingly global marketplace competition grows: if one product or service does not live up to the expectations of customers, they just choose another. Any successful company must adjust its internal activities and resources with the rapidly changing requirements, or to put it differently, business processes must be designed appropriately.

In the literature a number of various definitions of the term *business process* can be found. Here is a representative selection:

"A collection of related, structured activities – a chain of events – that produce a specific service or product for a particular customer or customers."[1]

"An activity or set of activities that are part of a service either to a citizen or to another organisational unit within or outside the particular public administration."[2]

[1] Glossary of IT Investment Terms, `http://www.gao.gov/policy/itguide/glossary.htm`

[2] Information Society Technologies, Glossary,
`http://www.cordis.lu/ist/ka1/administrations/publications/glossary.htm`

"A collection of activities that takes one or more kinds of input and creates an output that is of value to the customer."[3]

"A group of business activities undertaken by an organisation in pursuit of a common goal."[4]

"A structured, measured set of activities designed to produce a specified output for a particular customer of market." [23]

To bring down these interpretations to a common denominator, we define a business process as *a set of logically related activities that are combined to achieve some business goal*[5].

For example, a process of travel authorisation in an office consists of the following activities: filling out a travel application form, verification of the form by a secretary, checking the prices for tickets and hotels, authorisation of the travel expense by the manager, and reservation process by the travel department. Each activity in this process represents a work of a person, however, in other business processes it can be an internal system or a process of a partner company.

Currently, companies have been experiencing many changes in their business environments. One is the internal change caused by the increasingly growing competition. In order to live up to the customer expectations, companies may have to diversify their business activities. Another change is an external one resulting from the increase in multi-enterprise collaboration and e-Business. This change forces a company to become involved in the business processes of other companies. The necessity of technology and tools to ensure efficient and effective process management grows.

The business process approach began to take shape in the 1990s with the development of the technology for *workflow*, "the automation of a business process, in whole or part", and *Workflow Management System* (WfMS), "a system that defines, creates and manages the execution of workflows" [80].

Though WfMS technology helps manage and drive human-based, paper-driven processes within a corporate department [36], the requirements of the contemporary global market, such as integration of different enterprise applications and multi-enterprise complex system interaction exceed the potency of WfMS.

Business Process Management (BPM), on the other hand, copes with such advanced requirements of today's economy. BPM "refers to a set of activities which organisations can perform to either optimise their business processes or adapt them to new organisational needs" [72]. As these activities are usually supported by information technology (IT), the term BPM is synonymously used to refer the software tools supporting BPM themselves [72, 36]. In the context of our work we differentiate the terms BPM and BPMS, using the latter as the notion for the technology and tools for the design, execution, administration, and monitoring of business processes.

BPM can be divided into *Business Process Modeling*[6], *Business Process Automation* and *Business Process Controlling* [74]. A short explanation of these terms follows.

Business Process Modeling refers to the design and implementation of business processes. A description of a process model is provided in [62]:

"Processes of the same nature are classified together into a process model. Thus, a process model is a description of a process at the type level. Since the process model is at the type level, a process is an instantiation of it."

[3] Credit Research Foundation, Glossary, `http://www.crfonline.org/orc/glossary/b.html`

[4] Dream Catchers Inc., Glossary of terms - Application Management, `http://www.dream-catchers-inc.com/White%20Papers/glossary_of_terms-AM.htm`

[5] any administrative goal including

[6] Unfortunately both terms, Business Process Management and Business Process Modeling, have the same acronym – BPM [73]. In the context of our work we don't abbreviate the latter to avoid any misunderstanding.

One possible use of a process model is to prescribe ”how things must/should/could be done” in contrast to the process itself which is really what happens (see Figure 3.1).

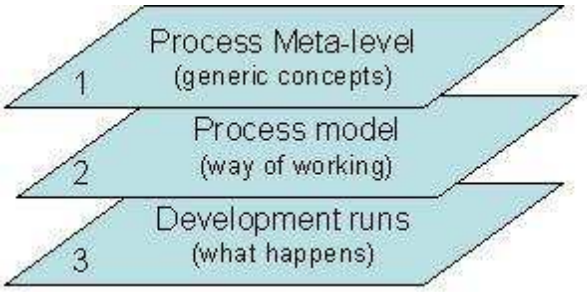

Figure 3.1: Abstraction level for processes proposed in [62]

In Workflow Management, there is a clear distinction between the design of a business process on the meta-level and its implementation. The term itself – ”business process modeling” – is often used in the literature as a notation for discovering, analyse, graphical specification (or documentation), and optimisation of business processes, as mentioned in [66]. The resulting meta-model of a process is mainly used by IT architects as a specification of requirements for their solution. This division between business and IT results in long improvement cycles of business processes and inflexibility of enterprises.

The essential promise of BPMS, combining a design tool and a runtime engine, that can actually execute suitably designed process models, is to eliminate this division and improve agility by speeding-up the process lifecycle. The key to this is to provide a set of modeling standards that are simple and graphically intuitive for business analysts, yet executable on a runtime engine. Though not every BPMS has yet provided such a solution, the boundaries between meta-model design and implementation phase are not that distinct any more and can be merged together into *Business Process Modeling*.

In the Section 3.4 we investigate among others Business Process Modeling Notation (BPMN) and Business Process Execution Language (BPEL). BPMN is a *modeling language*, this means it is used for the design of business processes on the meta-level (see Figure 3.1). The distinguishing feature of modeling languages is their ability to be directly translated in an executable code. A BPMN diagram can be directly translated in BPEL and executed by a runtime engine.

As the main objective of our work is to investigate the strengths and possible shortcomings of the programming language XChange, we focus out attention on the Business Process Modeling, giving below the definitions of Business Process Automation and Business Process Controlling for the sake of the overview’s completeness.

Business Process Automation services for the effective execution of the predefined business processes with a business process engine. A BPM engine, like a computer, loads programs (process models) and runs instances of them (processes). A BPM program specifies a series of steps, and the function of the engine is to run through these steps. A BPM engine is responsible for internal and external interaction with IT systems as well as with human participants (through worklists – lists of pending tasks) of the running processes.

Business Process Controlling is responsible for the control (or monitoring) and administration of the active processes. The ability to control the progress of running processes is important for the detection of exceptions (active process is progressing and not getting stuck), and for the real-time

querying (e.g., finding all active processes for customers). Administration includes the ability to suspend, resume, or terminate running processes. Control and administration requires a management language and a graphical management console.

3.2 Service-Oriented Architecture

In order to work effectively, a BPMS often requires that the underlying software is constructed according to the principles of a *Service-Oriented Architecture* (SOA) [77]. In broad terms, SOA stands for a software model composed of distributed components (services), each of which provides a specific function that can be used by other components.

The distinguishing feature of SOA is loose coupling. Loose coupling means that the service consumer (client) is essentially independent of the service. The way a client (which can be another service) communicates with the service doesn't depend on the implementation of the service. Significantly, this means that the client doesn't have to know very much about the service to use it. For instance, the client doesn't need to know what language the service is coded in or what platform the service runs on. The client communicates with the service according to a specified, well-defined interface. If the implementation of the service changes, for instance, the airline reservations application is altered, the client communicates with it in the same way as before, provided that the interface remains the same.

The concept of SOA is not new. One of its well-known realisations is CORBA [50], which first widely published specification appeared in 1992[7]. However, what is relatively new is the emergence of Web services-based SOAs. A *Web service* is a service that communicates with clients through a set of standard protocols and technologies, such as HTTP[8][7, 8], XML[9] [81], WSDL [84], SOAP [82], and UDDI[10] [49]. UDDI, WSDL, and SOAP standards were developed and submitted to the World Wide Web Consortium (W3C)[11] by IBM, Microsoft, UserLand Software, DevelopMentor, and Ariba. Since then, they are implemented in platforms and products from all the major software vendors, making it possible for clients and services to communicate in a consistent way across a wide spectrum of platforms and operating environments. This universality has made Web services the most prevalent approach to implementing SOA. We will analyse Web services, the standards and technology they are based on more precisely in Section 3.4.

SOA does not have a standardised reference model yet. However, OASIS's[12] SOA Reference Model Technical Committee[13] has started standards work to define SOA reference model. The members of the committee capture the main SOA concepts, combining the principal elements of the nowadays implementations of SOA [48]. Among them: *services*, *service description*, *discovery*.

Services A *service* is a specific function, typically a business function, such as analysing an individual's credit history or processing a purchase order, that can be implemented and provided by a component for use by other components. It can be a single discrete function, such as converting one type of currency into another, or a set of related business functions ("coarse-grained" service [53])

[7] History of CORBA, http://www.omg.org/gettingstarted/history_of_corba.htm

[8] Hypertext Transfer Protocol – HTTP/1.1, http://www.w3.org/Protocols/rfc2616/rfc2616.html

[9] Extensible Markup Language, http://www.w3.org/XML/

[10] Universal Description, Discovery and Integration (UDDI) protocol, http://www.uddi.org/

[11] World Wide Web Consortium, http://www.w3.org/

[12] Organisation for the Advancement of Structured Information Standards, http://www.oasis-open.org/

[13] OASIS SOA Reference Model TC,
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm

such as handling of various operations in an airline reservations system. Multiple services can be used together in a coordinated way.

The term ”services” does not imply Web services, although Web services are the preferred way to realise SOA, as we have already mentioned.

Service descriptions Each service should include a service definition in a standardised format. This enables clients (applications and human actors) to determine what the service does and how they may interact with it. The separation of the service description (*what*) from the service implementation (*how*) is an important aspect of SOA. Having a standard way of communicating with each other, services running on different platforms and systems, written in different languages, can still interact with each other.

Two standards for service descriptions exist today: W3C's Web Services Description Language (WSDL) [84] and ebXML's Collaboration Protocol Profile[14], an initiative sponsored by UN/CEFACT[15] and OASIS. Members of the OASIS project claim ebXML to be more expressive as WSDL, since ”in addition to providing technical components, the Collaboration Protocol Profile was developed to meet the specific needs of electronic business that involve service-oriented interactions between legal enterprises” [48]. In other words, ebXML supports the declaration details about legal terms. For example, if a client invokes a service that places a purchase order to the service provider and the execution is successful, it may result in a financial responsibility to the service provider or some other legal entity.

However, as any legal requirements of e-business go beyond the scope of our work, we do not investigate ebXML Collaboration Protocol Profile any further.

WSDL format, on the other hand, is one of the standards we will deal with describing the Business Process Execution Language (Section 3.4.2). For this reason we will briefly outline WSDL together with other standards for Web services in the next section.

Discovery of services SOA uses the find-bind-execute paradigm as shown in Figure 3.2. In this paradigm, service providers register their service in a public registry. The registry is used by service consumers to find services that match certain criteria. If the registry has such a service, it provides the client with an endpoint address for that service.

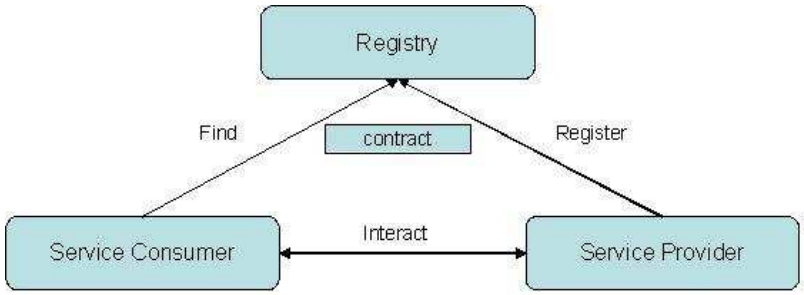

Figure 3.2: Service-Oriented Architecture Model

The most prevalent standard for registry implementation is the OASIS Universal Description and

[14]OASIS ebXML Collaboration Protocol Profile TC,
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ebxml-cppa

[15]United Nations Centre for Trade Facilitation and Electronic Business, http://www.unece.org/cefact/

Discovery Interface (UDDI) Technical Specification. More information on UDDI can be found in Section 3.3.

SOA, as a collection of distributed services communicating with each other, is clearly a suitable architectural model for multi-enterprise business processes: individual services as "activities" can be bound together to one business process irrespective their physical location and implementation peculiarities. The other benefits of SOA are:

- **Reusability** Typically, business applications developed in different companies, even different departments within the same company, have specific features. They run in different operating environments, they are coded in different languages, they use different programming interfaces and protocols. In order to communicate with these applications or reuse them to meet new business requirements, one has to know how and where they run. Such analysis can be very time consuming. In a SOA, the only characteristic of a service that a requesting application needs to know about is the public interface. The functions of an application can be much more easier to access and reuse as a service in a SOA than in some other architecture.

- **Interoperability** Web services provide widespread interoperability. That means, clients and services can communicate and understand each other no matter what platform they run on and what language they are written in. Web services protocols and technologies are platform, system, and language independent. In addition, these protocols and technologies work across firewalls, making it easier for business partners to share services. The more detailed discussion about Web services protocols and technology follows in Section 3.3.

- **Scalability** Services in a Web services-based SOA tend to be coarse-grained. As mentioned earlier, coarse-grained services offer a set of related business functions rather than a single function. Applications using these services require a relatively limited interaction traffic and can scale without putting a heavy communication load on the network.

- **Flexibility** Loosely-coupled services are more flexible than more tightly-coupled applications. In a tightly-coupled architecture, the different components are tightly bound to each other, sharing semantics, libraries, etc. This makes it difficult to change the applications to keep up with changing business requirements. The loosely-coupled nature of services in SOA allows applications to be flexible.

- **Cost Efficiency** Loosely-coupled components of a SOA solution should be less costly to maintain and easier to extend as in a tightly-coupled solution because they are independent from each other. Changes in one component of the integrated solution do not require changes in other components. In addition, a lot of the Web-based infrastructure is already used in many enterprises. Last, but not least, SOA is about reuse of business functions exposed as coarse-grained services. This is a big cost saving as well.

In view of these benefits it is not surprising to read Gartner's report: "By 2008, SOA will be a prevailing software engineering practice, ending the 40-year domination of monolithic software architecture"[16]. Most vendors have already accepted SOA and have an application or platform suite that enables it. More and more enterprises see the potential of SOA – especially in a Web services-based SOA – in providing the flexibility and agility needed to be competitive in the changing market.

[16]Gartner, Service-Oriented Architecture Scenario, `http://www.gartner.com/DisplayDocument?id=391595`

3.3 Web Services

As we have already mentioned, the most widespread implementation way of SOA is by using Web services. Web services are software systems designed to support interoperable machine-to-machine interaction over a network. This interoperability is gained through a set of XML-based open standards, such as WSDL, SOAP, and UDDI. These standards provide a common approach for defining, publishing, and using Web services. Their widespread acceptance makes it possible for clients and services to communicate and understand each other across a wide variety of platforms and across language boundaries. It is the combination of these protocols that make Web services so attractive.

This section briefly describes the protocols and technologies that constitute these Web services standards, namely SOAP, WSDL, and UDDI.

Simple Object Access Protocol (SOAP) XML has become the de facto standard for describing data to be exchanged on the Web. Though this markup language presents an effective way to exchange data, it's not sufficient in today's business world. For instance, the message receiver has to understand what is the main part of the message, and what part contains additional instructions or supplemental content. That's where Simple Object Access Protocol (SOAP) comes in. SOAP is an XML-based protocol for exchanging information in a distributed environment [82].

SOAP was designed in 1998, originally supported by Microsoft. Currently, the SOAP specification is maintained by the XML Protocol Working Group[17] of the W3C.

SOAP uses HTTP as the primary application layer protocol. Because HTTP is the ubiquitous Web protocol with a well-defined port, firewalls are usually configured to allow HTTP traffic, thus enabling SOAP messages to cross firewall boundaries easily. This is a major advantage of SOAP over other distributed protocols like GIOP[18]/IIOP[19] (used in CORBA) or DCOM[20] which are normally filtered by firewalls.

The basic item of transmission in SOAP is a SOAP message. It consists of a mandatory element `<Envelope>`, containing an optional element `<Header>` and a mandatory element `<Body>`.

The root element `<Envelope>` defines the XML document as a SOAP message. It specifies two things: an XML namespace and an encoding style. The XML namespace specifies the names that can be used in the SOAP message. The encoding style identifies the data types recognised by the SOAP message. Typically, the SOAP `<Header>` element contains relevant information about the message. For example, the date the message is sent or authentication information. The `<Body>` contains the main part of the SOAP message.

The slightly modified example from [78] in Figure 3.3 shows a SOAP message requesting product information from a fictional warehouse Web service and a reply SOAP message with the requested product information.

More about SOAP can be read for example in the W3C Recomendation.[21]

Web Services Description Language (WSDL) WSDL is an XML-based specification for describing what a service does and how to communicate with it. A WSDL document resides at a URL location and is "linked" to the actual program module that can be located elsewhere. In addition, a WSDL document informs about the input the service needs and output it responds with.

[17] XML Protocol Working Group, `http://www.w3.org/2000/xp/Group/`
[18] General Inter-ORB Protocol, `http://en.wikipedia.org/wiki/GIOP`
[19] Internet Inter-ORB Protocol, `http://en.wikipedia.org/wiki/IIOP`
[20] Distributed Component Object Model, `http://en.wikipedia.org/wiki/DCOM`
[21] SOAP Version 1.2, W3C Recomendation, `http://www.w3.org/TR/soap12-part1/`

```
<!-- SOAP request message -->
<soap:Envelope xmlns=soap:"http://www.w3.org/2003/05/soap-envelope"
   soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
   <soap:Body>
      <getProductDetails xmlns="http://warehouse.examle.com/ws">
         <productID>555</productID>
      </getProductDetails>
   </soap:Body>
</soap:Envelope>

<!-- SOAP response message -->
<soap:Envelope xmlns=soap:"http://www.w3.org/2003/05/soap-envelope">
   <soap:Body>
      <getProductDetailsResponse xmlns="http://warehouse.examle.com/ws">
         <getProductDetailsResult>
            <productID>555</productID>
            <description>Luggage set. Black polyster.</description>
            <price>99.90</price>
         </getProductDetailsResult>
      </getProductDetailsResponse>
   </soap:Body>
</soap:Envelope>
```

Figure 3.3: Two SOAP messages

A WSDL document describes a service as a set of abstract items called "endpoints" (WSDL element `<portType>`) that operate on messages. The operations and messages are described abstractly and then bound to a concrete network protocol (specifically, SOAP) and message format to define an endpoint. A WSDL element `<binding>` specifies the network address for an abstract endpoint to turn it to a concrete one. Endpoints are combined to a "service".

Figure 3.4 presents a WSDL document for an airline service providing flight availability and ticket price checking. In the example, the WSDL document specifies one operation for the service `FlightAvailability` with the input message `FlightTicketRequestMessage` (we assume that the plane ticket acquirement is an asynchronous operation, hence it has no output message). The `style` attribute of the `<binding>` element specifies that a complete XML document would be exchanged in the call. Alternatively, the value `rpc` of this attribute would specify the input as a Remote Procedure Call. The former type of binding is however more typical of services in SOA [53].

Universal Description and Discovery Interface (UDDI) As mentioned earlier, a SOA solution also includes (or "can include" according to some authors, e.g. [53]) a registry of services. The UDDI specifications define how to publish and discover information about services in an UDDI registry. More detailed, the specifications define a UDDI schema and a UDDI API. The UDDI schema identifies the types of XML data structures that comprise an entry in the registry for a service. Like the Yellow Pages directory for phone numbers, a UDDI registry provides information about services, such as the name of the service, a brief description of what it does, an address where the service can be accessed, and a description of the interface for accessing the service.

Here is an example that shows a part of a complete *BusinessEntity* structure for a hypothetical

```
<definitions>
   <types>
      <element name="TicketRequest">
         <complexType>
            <all>
               <element name="start" type="string"/>
               <element name="destination" type="string">
               <element name="date" type="float">
            </all>
         </complexType>
      </element>
   </types>

   <message name="FlightTicketRequestMessage">
      <part name="body" element="TicketRequest"/>
   </message>

   <portType name="FlightAvailabilityPT">
      <operation name="FlightAvailability">
         <input message="FlightTicketRequestMessage">
      </operation>
   </portType>

   <binding name="FlightAvailabilityBinding" type="FlightAvailabilityPT">
      <soap:binding
      style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="FlightAvailability">
         <soap:operation
          soapAction="http://packtpub.com/service/airline/FlightAvailability">
         <input>
            <soap:body use="literal"/>
         </input>
      </operation>
   </binding>

   <service name="AirlineService">
      <port name="FlightAvailability" binding="FlightAvailabilityBinding">
         <soap:address location="http://packtpub.com/service/airline"/>
      </port>
   </service>
</definitions>
```

Figure 3.4: Airline Web Service WSDL

company named BooksToGo (the example is taken from [53]):

```
<businessEntity
   businessKey="35AF7F00-1419-11D6-A0DC-000C0E00ACDD"
   authorisedName="0100002CAL"
   operator="www-3.ibm.com/services/uddi">
   <name>BooksToGo</name>
   <description xml:lang="en">
      The source for all professional books</description>
   <contacts>
      <contact>
         <personName>Benjamin Boss</personName>
         <phone>(877)1111111</phone>
      </contact>
   </contacts>
</businessEntity>
```

The API describes the SOAP messages that are used to publish an entry in a registry or discover an entry in a registry. Here, for example, is a message that searches for all business entities in a registry whose name begins with the characters `Books` (without the character % used as a wildcard, only those names that exactly match "Books" would be chosen as the answer):

```
<find_business generic="2.0" xmlns=um:uddi-org:api-v2">
   <name>Books%</name>
</find_business>
```

Standards such as XML, SOAP, UDDI, and WSDL address the basics of interoperable services. They ensure that a client can find a needed service and make a request that both the client and service understand, irrespective of where the client and service reside or what language they are coded in. But for a Web services-based SOA to become a mainstream IT in the business world, other standards need to be added and adopted. This is especially true in the areas of Web service security and Web service management. Various standardisation organisations, such as the W3C and OASIS, have drafted standards in these areas that promise to gain universal acceptance [53]. One of them is BPEL. It will be discussed in the Section 3.4, which is devoted to the Business Process Management standards.

To sum up, Web services support interoperable machine-to-machine interaction over a network. They have interfaces described in a standardised format that machines can process (WSDL). Multiple services interact with each other in a manner prescribed by their description using SOAP messages. Web services can be dynamically discovered and invoked through a registry of Web services, UDDI. Web services promise business agility and IT flexibility and become the predominant implementation approach for SOA. However, to program a complex business process consisting of multiple applications' activities, a Web-based BPM should provide a solution to coordinate these applications over the Web. The next section is devoted to the discussion of such standard solutions.

3.4 Standard Business Process Management (BPM) Stack

There are plenty of BPM standards nowadays, each with a different design and feature set. In this chapter we introduce a general BPM application architecture that is comprehensible and meets real-world requirements. It is based on the standards that are mostly extensive supported by the leading

organisations driving the development and adoption of business standards, such as OASIS, Business Process Management Initiative (BPMI)[22], and The Workflow Management Coalition (WfMC)[23].

The architectural model presented here corresponds to the WfMC's reference model[24] (see Figure 3.5) and is based on the standard BPM stack proposed by BPMI (see Figure 3.6).

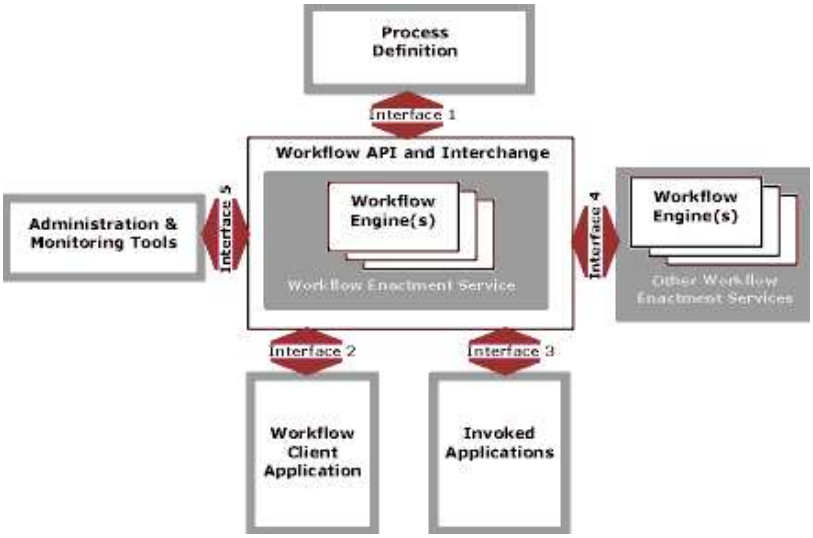

Figure 3.5: WfMC Workflow Reference Model

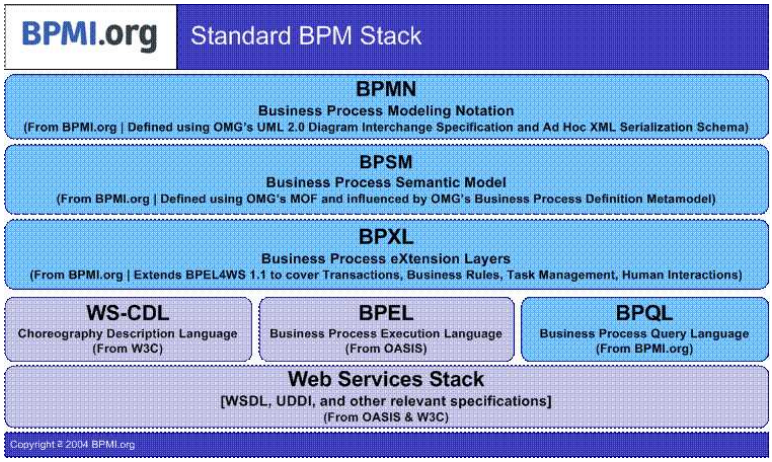

Figure 3.6: Standard Business Process Management Stack proposed by BPMI

Let's take a closer look at each of the major components of the WfMC's model and examine the proposed standard solutions from the BPM stack.

[22] Business Process Management Initiative, http://www.bpmi.org/

[23] The Workflow Management Coalition, http://www.wfmc.org/

[24] *"Reference model is a structure which allows the modules and interfaces of a system to be described in a consistent manner"*, http://www.acq.osd.mil/osjtf/termsdef.html

3.4.1 Business Process Modeling Notation

First of all, a BPMS (and a WfMS as well) should provide a tool for the definition of business processes. Typically, it is a graphical tool. Graphical notation aims at being understandable by all business users, from the business analysts who design the initial draft of the processes, to the IT developers responsible for the implementation and execution of these processes, and finally, to the business people who manage and control them. It is much more easier for non-technical business users to draw a diagram than to compose or read a programming code in Java, C++, or an XML-based language. A standard solution for the graphical representation is even more valuable as it can be understood by a wide audience of business users and IT specialists, irrespective which modeling tool they use or provide.

BPM has two good graphical modeling notations – Business Process Modeling Notation (BPMN) and the UML activity diagram. Though the latter is widely implemented, more vendors tend to support BPMN because it is more expressive and has a mapping to Business Process Execution Language (see Section 3.4.2).

BPMN was developed by the Business Process Management Initiative. The BPMN 1.0 specification was released in May 2004.

All BPMN elements are organised into specific categories. This provides a small set of notation categories so that the user of a diagram can easily recognise the basic types of elements and understand the notation. Within the basic categories of elements, additional variations and informations can be added. The four basic categories of elements are: *Flow Objects*, *Connecting Objects*, *Swimlanes*, and *Artifacts*.

Flow Objects is a set of three core elements presented in Figure 3.7. This and the following tables of BPMN elements represent the extracts from the BPMN Specification[25].

Event	An *Event* is represented by a circle and is something that "happens" during the course of a business process. These Events affect the flow of the process and usually have a cause (trigger) or an impact (result). Events are circles with open centers to allow internal markers to differentiate different triggers or results. There are three types of Events, based on when they affect the flow: *Start*, *Intermediate*, and *End* (see the figures to the right, respectively).	
Activity	An *Activity* is represented by a rounded-corner rectangle (see the figure to the right) and is a generic term for work that company performs. An Activity can be atomic or non-atomic (compound). The types of Activities are: *Task* and *Sub-Process*. The Sub-Process is distinguished by a small plus sign in the bottom center of the shape.	
Gateway	A *Gateway* is represented by the familiar diamond shape (see the figure to the right) and is used to control the divergence and convergence of Sequence Flow. Thus, it will determine traditional decisions, as well as the forking, merging, and joining of paths. Internal Markers will indicate the type of behavior control.	

Figure 3.7: Core BPMN Flow Objects

[25] Business Process Modeling Notation Information, `http://www.bpmn.org/`

Connecting Objects provide connecting of the Flow Objects together in a diagram to create the basic structure of a business process. These connectors are presented in Figure 3.8.

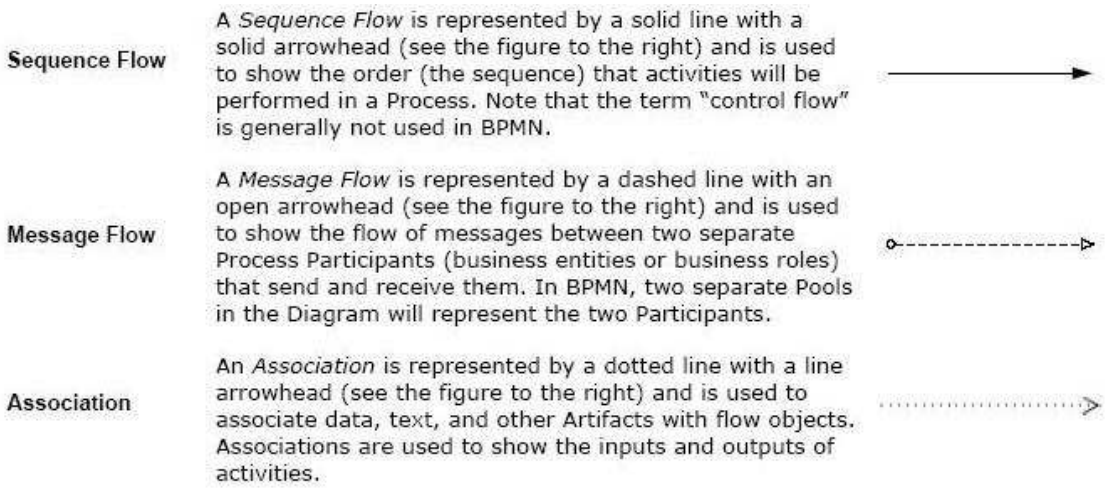

Figure 3.8: BPMN Connecting Elements

Swimlanes express a mechanism to organise activities into separate visual categories. The two types of BPMN swimlane objects are shown in Figure 3.9. Notice that Pools are used to separate different business participants (e.g., Patient versus Doctor's Office). Lanes are used to separate the activities associated with a specific company function or role (e.g., Manager versus Travel Department).

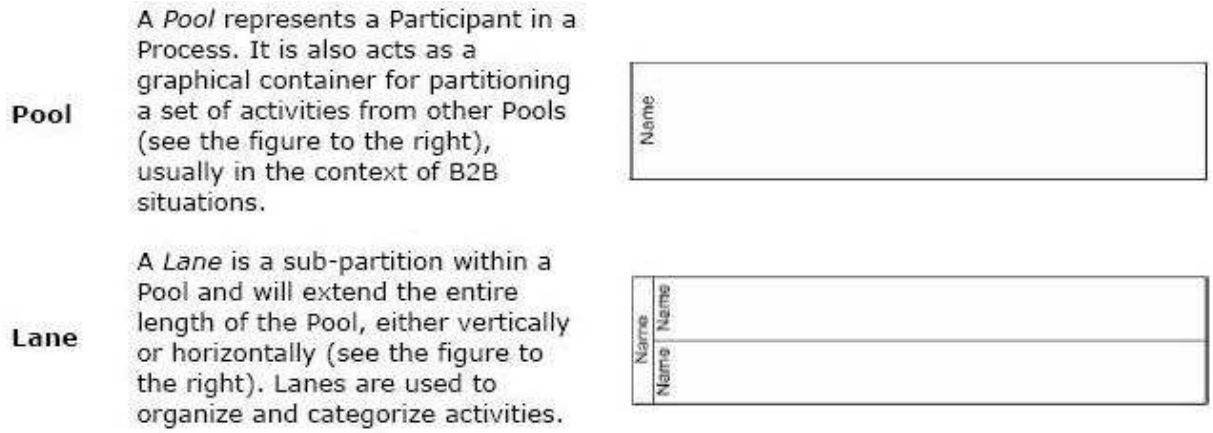

Figure 3.9: BPMN Swimlane Objects

Artifacts add more details about how the process is performed. The three types of BPMN Artifacts represented in Figure 3.10.

The description of other BPMN elements can be found in BPMN Specification we have mentioned above. A compact introduction in BPMN can be found [70].

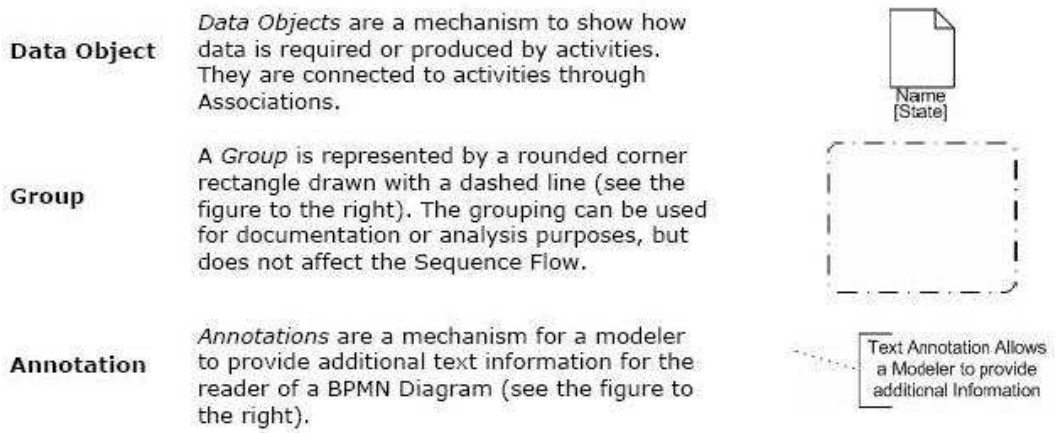

Figure 3.10: BPMN Artifact Elements

Meanwhile, the vast majority of business analysts draw their business process diagrams in Microsoft Visio[26]. The idea of some software companies was therefore to offer Visio add-ins providing the BPMN standards shapes and icons.

One of such products is, for example, Process Modeler for Visio from the Swiss company ITP-Commerce[27]. To create valid BPMN diagrams a business analyst just drags and drops Pools, Activities, Subprocesses, and Events from the BPMN Shape palette and connects them with Sequence Flows and Message Flows (see Figure 3.11).

A complete list "Current Implementations of BPMN" with the names of tools supporting BPMN and their vendors can be found on the official site of BPMN[28]. Here are just some names: JViews (ILOG), Process Modeler for Visio (ITpearls), Oracle BPEL Designer (Oracle), BPMSuite (Pegasystems), System Architect (Popkin), Studio Enterprise Edition (Sun Microsystems), WBI Modeler (IBM).

ITP-Commerce's Process Modeler (and some other tools supporting BPMN) not only allows business analysts to create BPMN diagrams, but also to export them in BPEL (interface 1 in WfMC Reference Model shown in Figure 3.5). However, before describing the mapping process we introduce the BPEL standard itself.

3.4.2 Business Process Execution Language

Business Process Execution Language (BPEL) is a language for the definition and execution of business processes using Web services. BPEL enables the realisation of Service-Oriented Architecture through composition and coordination of Web services. BPEL provides a way to combine several Web services into new, composite services – business processes. BPEL is an XML-based language that supports the Web services standards, including SOAP, WSDL, UDDI.

BPEL was originally drafted by IBM and Microsoft and is now being formalised by a committee at OASIS. Since the BPEL specification was submitted to OASIS in March 2003, it has gained the support of many industry vendors (including Oracle, Microsoft, IBM, SAP, Siebel, BEA, and Sun). A

[26]Microsoft Office Online, `http://office.microsoft.com/en-us/FX010857981033.aspx`
[27]ITP-Commerce, `http://www.itp-commerce.com/`
[28]BPMN Implementors and Quotes, `http://www.bpmn.org/BPMN_Supporters.htm`

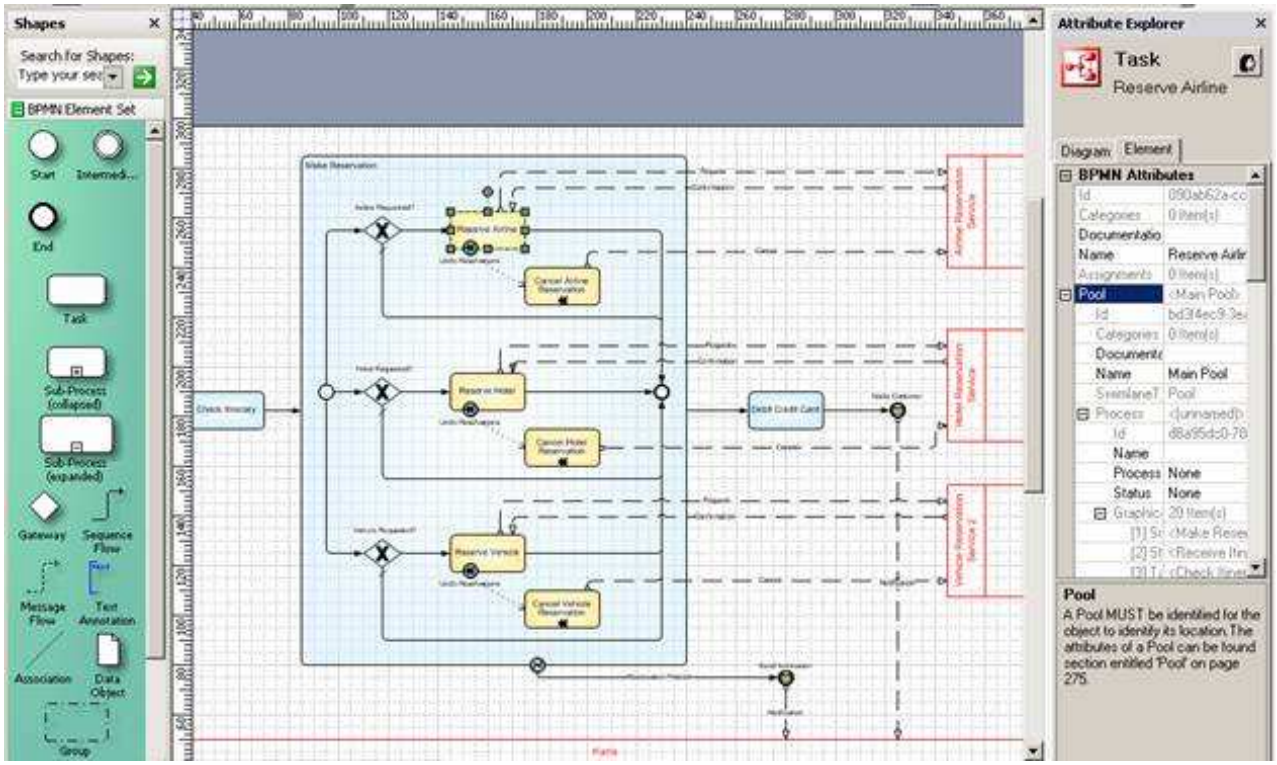

Figure 3.11: ITP-Commerce's BPMN modeling tool with Microsoft Visio

similar proposal for defining executable processes based on Web services – Business Process Modeling Language (BPML) was made by the BPMI. However, now the BPMI has dropped support for this language in favour of BPEL (see the BPM standard stack suggested by BPMI, Figure 3.6).

To understand how business processes are described with BPEL, we will study a simplified business process for employee travel arrangements specified in [43]: The client invokes the business process by specifying the name of the employee, the destination, the departure date, and the return date. The BPEL business process first checks the employee travel status, assuming that a Web service exists through which such checks can be made. The BPEL process will check the price for the flight ticket with two airlines: American Airlines and Delta Airlines. Again, assuming that both airline companies provide a Web service through which such checks can be made. Finally, the BPEL process will select the lower price and return the travel plan to the client.

It is a scenario of a quite typical BPEL process: the process receives a request, invokes Web services to fulfil it, and then responds to the original caller. The BPEL process relies on the WSDL description of the Web services it invokes. Through these descriptions the process learns how to communicate with them. Thus, in our example we would need WSDL documents for American Airlines Web Service, for Delta Airlines Web Service, and for the Employee Travel Status Web Service. In Section 3.3 we have already created a WSDL document for an airline Web service (see Figure 3.4). As well as the author of [43] we assume that both airlines offer identical services (i.e. provide equal port types and operations). A WSDL description of the Employee Travel Status Web Service is omitted to simplify the example, however represented graphically in Figure 3.12.

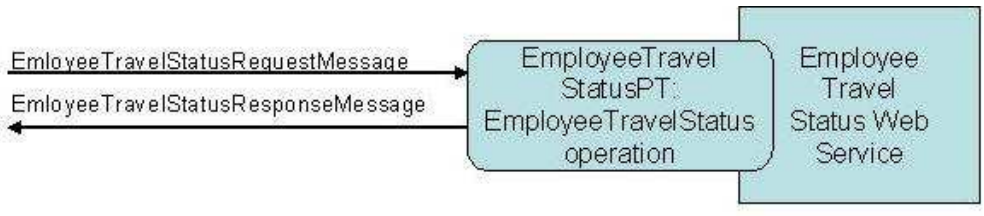

Figure 3.12: The Employee Travel Status Web Service

PartnerLinks The first issue we encounter is how to bind the BPEL process to existing Web services. Such a link can be created with the BPEL construct `<partnerLink>`. This element describes which set of Web service operations – defined by the WSDL `<portType>` construct (see Figure 3.4) – is used in the BPEL process.

To bind the BPEL process to a specific `<portType>` via a `<partnerLink>` we have to add a `<partnerLinkType>` construct into the original WSDL document:

```
<definitions>
   <!-- already defined elements -->
   ...
   <!-- the new element -->
   <partnerLinkType name="flightLT">
      <role name="airlineService">
         <portType name="FlightAvailabilityPT"/>
      </role>
   </partnerLinkType>
</definitions>
```

The BPEL construct `<partnerLink>` itself is to be built in the BPEL process definition document:

```
<partnerLinks>
   <partnerLink name="AmericanAirlines"
                partnerLinkType="flightLT"
                myRole="airlineCustomer"
                partnerRole="airlineService"/>
</partnerLinks>
```

The partner link `AmericanAirline` specifies two attributes: `myRole` (indicates the role of the business process itself) and `partnerRole` (indicates the role of the invoked service). The role of the BPEL process (`myRole`) to the airline Web service is `airlineCustomer`, whereas the role of the airline (`partnerRole`) is `airlineService`. A partner link representing a synchronous request/reply relation would specify only one, `myRole`.

Variables Variables in BPEL processes are used to store and transform messages. For every sent and received message a variable and its corresponding type (`<messageType>`) are to be defined. A variable `FlightDetails` is specified for the request message to an airline Web Service. Its `<messageType>` is therefore conform to the `<message>` construct defined in the WSDL description of an airline Web Service (Figure 3.4):

```
<variables>
   <!-- input for American and Delta Airlines Web services -->
   <variable name="FlightDetails"
             messageType="FlightTicketRequestMessage"/>
</variables>
```

Process Definition consists of a <process> element, a list of <partnerLink>s (Web services) used by the BPEL process, the <variable>s used in the business process, and a list of steps (or activities) to be executed by the business process:

```
<process name="BusinessTravelProcess">
   <partnerLinks>
      <!-- declaration of partner links -->
   </partnerLinks>
   <variables>
      <!-- declaration of variables -->
   </variables>
   <sequence>
      <!-- declaration of the BPEL process main body -->
   </sequence>
</process>
```

Activities Each BPEL process, and our BPEL process in particular, consists of steps, or activities. BPEL supports primitive as well as structured activities. Primitive activities represent basic constructs and are used for common tasks, such as the following:

- Invoking other Web services, using <invoke>;
- Waiting for the client to invoke the business process by sending a message, using <receive> (receiving a request);
- Generating a response for synchronous operations, using <reply>;
- Manipulating data variables, using <assign>;
- Indicating faults and exceptions, using <wait>;
- Terminating the entire process, using <terminate>.

To combine primitive activities, BPEL defines several structured activities. The most important of them are:

- Sequence (<sequence>), which allows to define a set of activities that will be invoked in an ordered sequence;
- Flow (<flow>) for defining a set of activities that will be invoked in parallel;
- Case-switch construct (<switch>) for implementing branches;
- While (<while>) for defining loops;
- The ability to select one of several alternative paths, using <pick>.

Process Main Body specifies the order of the steps to be executed. Usually it starts with a <sequence> that allows to define several activities that will be performed sequentially. Within the sequence, the first thing that has to be specified is the input message that starts the business process. For this purpose BPEL defines the <receive> construct, which waits for the matching message. It is the TravelRequest message in our example. Within the <receive> element the partner link, the port type, the operation name, and optionally the variables for the received message are specified:

```
<sequence>
   <!-- Receive the initial request for business travel from client -->
   <receive partnerLink="client"
            portType="trv:TravelApprovalPT"
            operation="TravelApproval"
            variable="TravelRequest"
            createInstance="yes"/>
   ...
</sequence>
```

The next step is to use the BPEL <assign> construct to transform the input variable TravelRequest into the EmployeeTravelStatusRequest.

```
   <!-- Prepare the input for the Employee Travel Status Web Service -->
   <assign>
      <copy>
         <from variable="TravelRequest" part="employee"/>
         <to variable="EmployeeTravelStatusRequest" part="employee"/>
      </copy>
   </assign>
```

The <invoke> element is used to make a synchronous invocation of the Employee Travel Status:

```
   <!-- Synchronously invoke the Employee Travel Status Web Service -->
   <invoke partnerLink="employeeTravelStatus"
           portType="emp:EmployeeTravelStatusPT"
           operation="EmployeeTravelStatus"
           inputVariable="EmployeeTravelStatusRequest"
           outputVariable="EmployeeTravelStatusResponse"/>
```

The next step is to invoke both airline Web services. We use <flow> element to invoke them concurrently. For each airline Web service two operations – <invoke> and <receive> – are defined to specify the asynchronous communication.

```
   <!-- A concurrent invocation -->
   <flow>
      <sequence>
         <invoke partnerLink="AmericanAirlines"
                 portType="aln:FlightAvailabilityPT"
                 operation="FlightAvailability"
                 inputVariable="FlightDetails"/>
```

```
      <receive partnerLink="AmericanAirlines"
               portType="aln:FlightCallbackPT"
               operation="FlightTicketCallback"
               variable="FlightResponseAA"/>
   </sequence>

   <sequence>
      <invoke partnerLink="DeltaAirlines" .../>
   </sequence>
</flow>
```

In the next step, the construct `<switch>` is applied to select one of two ticket offers:

```
<!-- Select the best offer and construct the TravelResponse -->
<switch>
   <case
    condition="bpws:getVariableData('FlightResponseAA', 'confirmationData',
    '/confirmationData/Price') <= bpws:getVariableData('FlightResponseDA',
    'confirmationData', '/confirmationData/Price')">
      <!-- Select American Airlines -->
      <assign>
         <copy>
            <from variable="FlightResponseAA" />
            <to variable="TravelResponse" />
         </copy>
      </assign>
    </case>
    <otherwise>
       <!-- Select Delta Airlines -->
       <assign>
          <copy>
             <from variable="FlightResponseDA" />
             <to variable="TravelResponse" />
          </copy>
        </assign>
    </otherwise>
 </switch>
```

To locate the price element in the extract above an XPath expression is used. If the American Airlines offer is better than Delta's, the `FlightResponseAA` variable will be copied to the `TravelResponse` variable (which is finally to be returned to the client). Otherwise, the `FlightResponseDA` variable will be copied.

The final step of the BPEL business process is to return a callback to the client using the `<invoke>` activity. For the callback the `ClientCallback` operation on the `ClientCallbackPT` port type is to be invoked. The variable that holds the reply message is `TravelResponse`:

```
   <!-- Make a callback to the client -->
   <invoke partnerLink="client"
```

```
        portType="trv:ClientCallbackPT"
        operation="ClientCallback"
        inputVariable="TravelResponse" />
   </sequence>
</process>
```

With these steps, all the necessary work to connect existing Web services into a fully functional BPEL process is done. Note that business designers do not need to write a raw BPEL code. Rather, highly graphical tools such as Process Modeler for Visio or Oracle BPEL Designer generate much of the code.

However, even a manual mapping of BPMN diagram to BPEL is quite straightforward: the "events" of a BPMN process are BPEL "receive" constructs; "activities" are BPEL "invoke" elements; "splits" are realised with "switch"; and the process flow with the "sequence" element.

3.4.3 Web Services Choreography Description Language

An initial Public Working Draft of the Web Services Choreography Description Language (WS-CDL) Version 1.0 was released by W3C in 2004. In this document WS-CDL is defined as "*an XML-based language that describes peer-to-peer collaborations of parties by defining, from a global viewpoint, their common and complementary observable behaviour, where ordered message exchanges result in accomplishing a common business goal*" [83]. According to the W3C announcements, WS-CDL is "a necessary complement to end point languages such as BPEL and Java. It provides them with the global model they need to ensure that end point behaviour."

WS-CDL provides the global message exchange model, whereas BPEL models the message exchange from the point of view of one participant. WS-CDL is a *choreography* language, BPEL is an *orchestration* language.

Orchestration versus Choreography In orchestration a central process (which can be another Web service) takes control of the involved Web services and coordinates the execution of different operations on the Web services involved in the operation. The involved Web services do not "know" that they are involved in a composition process and that they are taking part in a higher-level business process. Only the central coordinator of the orchestration is aware of this goal, so the orchestration is centralised with explicit definitions of operations and the order of Web services (see Figure 3.13).

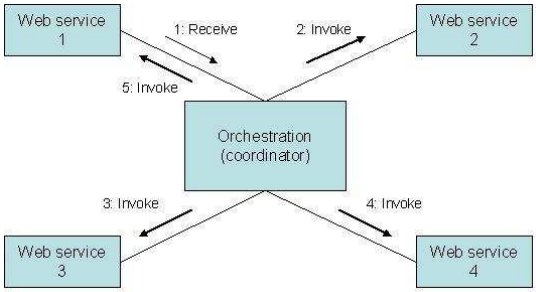

Figure 3.13: Composition of Web services with orchestration

Choreography, in contrast, does not rely on a central coordinator. Rather, each Web service involved in the choreography knows exactly when to execute its operations and with whom to interact.

Choreography is a collaborative effort focusing on the exchange of messages in public business processes. All participants in the choreography need to be aware of the business process, operations to execute, messages to exchange, and the timing of the message exchange (see Figure 3.14).

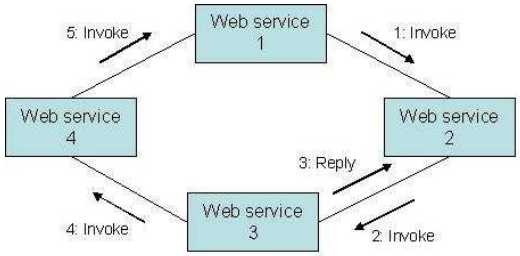

Figure 3.14: Composition of Web services with choreography

From the perspective of composing Web services to execute business processes, orchestration is a more flexible paradigm, as the component processes are centrally managed and can be incorporated in a business process without their being aware of this [43]. On the other hand, it is difficult to imagine a business-to-business participant who would accept orchestrating its services by a business partner. It is sooner possible that the business partners would like to agree about the way they want to coordinate their business processes. For example, with help of a WS-CDL description.

WS-CDL does not depend on a specific platform or implementation language. It provides an interoperable representation of collaborations. The individual parts of this representation could be implemented with completely different mechanisms.

The choreography can be described on the following business process:

- A client sends a purchase to a retailer.
- The retailer sends to the client an acknowledgement that he received the purchase order.
- The retailer forwards the purchase order to a warehouse.
- The warehouse sends a positive or a negative acknowledgement to the retailer. The retailer forwards to the client the result answer of the warehouse.
- If the warehouse accepted the order, it finishes its processing and sends a notification directly to the client.

The first two steps of this process can be represented in WS-CDL code as follows:

```
<interaction name="POProcess"
   operation="handlePO" initiate="true">

   <!-- Indicate message exchange participants -->
   <participate relationshipType="CRRelationship"
      fromRole="Consumer" toRole="Retailer"/>

   <!-- a client sends a request to a retailer -->
   <exchange name="PORequest" informationType="PO"
```

```
      action="request">
      <send variable="getVariable(poC, Consumer)"/>
      <receive variable="getVariable(poR, Retailer)"/>
   </exchange>

   <!-- the retailer responds with an acknowledgement -->
   <exchange name="PORespond" informationType="POAck"
      action="respond">
      <send variable="getVariable(poAckR, Retailer)"/>
      <receive variable="getVariable(poAckC, Consumer)"/>
   </exchange>
</interaction>
```

This code describes an interaction with two exchanges: in the first exchange the client sends (`action="request"`) a purchase order (`informationType="PO"`) to the retailer; in the second, the retailer responds (`action="response"`) with a purchase order acknowledgement (`informationType="POAck"`).

The first step in building the retailer process is to generate a BPMN diagram that satisfies the retailer's role in the choreography. It is a manual step nowadays, as there is no WS-CDL tool on the market that can do it automatically [36]. Figure 3.15 shows the BPMN diagram representing the retailer as a participant in the choreography.

Figure 3.15: Retailer process in BPMN for choreography

Figure 3.16 shows the process with some private steps added (i.e., steps not required by the choreography but driven by internal requirements). `Write PO to DB` saves the purchase order to an internal retailer database; `Update PO Status in DB` updates the database record with the status of the warehouse response; `Sales Followup` is a manual task, assigned to a sales representative to help the client resubmit the order in case it was rejected.

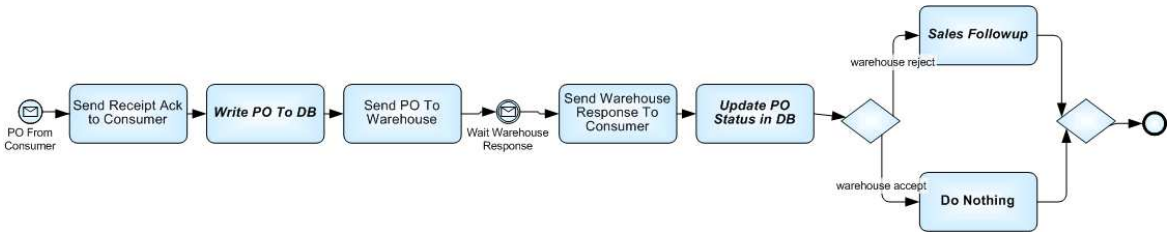

Figure 3.16: Complete retailer process in BPMN

Analogously, a business process of the warehouse can be designed. The internal realisation on each side of the complete business process can be implemented differently, for example with BPEL or Java.

3.5 Business Process Management System (BPMS) Architecture Model

After the introduction of the main principles and today's standards of BPM, we describe a possible BPMS architecture that satisfies them. The objective of this description is to summarise already discussed information rather than to present some new issues.

At the centre of this architecture is a runtime engine that executes program models written in the XML-based BPEL language. These models are designed by business analysts using a graphical editor that supports the modeling language BPMN. The editor includes an exporter that generates BPEL code (which is deployed to the engine) from BPMN diagrams. The runtime engine executes the business processes as a series of steps, each of them involving human or computer interactions. To enable the users to view and execute pending tasks, a typical BPMS provides a worklist graphical console. Internal IT systems are accessed with help of programs written in Java, C++, or integration technologies such as Web services, J2EE [29], or .NET[30]. External interactions are typically Web service-based communications, ordered by the choreography language WS-CDL. A choreography tool can be used to generate a basic BPMN model for each particular participant (a choreography itself describes a global view over a multi-participant business process). BPM administrators use a graphical console to control and administrate the running processes.

The development steps of the system with such an architecture are the following:

- Create a basic BPMN model from a WS-CDL choreography (ignore if the choreography description is not provided);
- Design the BPMN process model (as a diagram);
- Generate BPEL code from the BPMN diagram;
- Deploy the BPEL code to the runtime engine;
- Use the administration console for the tracing of the running processes.

The introduced architecture (illustrated in Figure 3.17) combines the principal elements of the BPMS realisations offered by such vendors as IBM, BEA, Oracle, Tibco, SeeBeyound. The choice of standards is inspired by the Standard Business Process Management Stack proposed by BPMI (see Figure 3.6). Though the support of all the standards together in today's implementations of BPMS is rare, the leading standardisation organisations prophesy their global adoption in the nearest future.

Summary In this chapter we have discussed the concepts and principles of BPM and BPMS, the differences between BPM and WfM, the actual development trends in the BPMS domain, and the technology and protocols standards for these trends. We have introduced an example architecture of BPMS based on the standards promoted by the leading standardisation organisations. In our overview we emphasised standards over vendors to get a better source of concepts and to catch future trends in the development of BPMS, specifically Web-based BPMS. We will base on the obtained awareness in the following analyse of XChange practicability for Business Process Modeling.

[29] Java 2 Platform Enterprise Edition, `http://java.sun.com/j2ee/releases/`

[30] .NET Framework, `http://msdn.microsoft.com/netframework/`

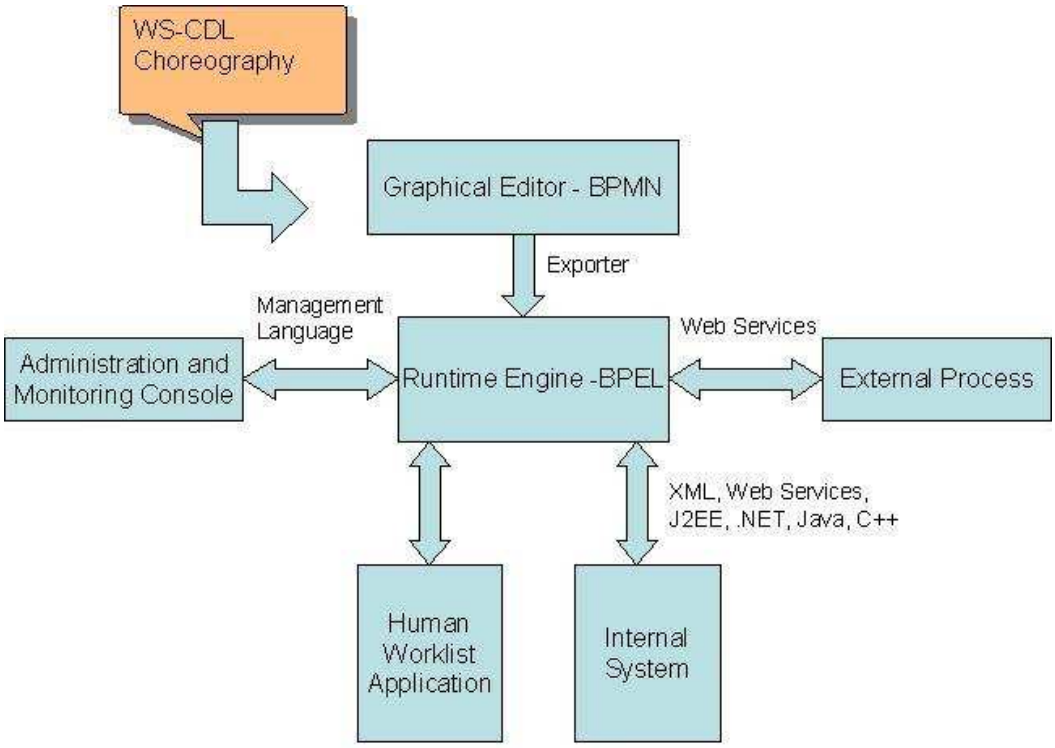

Figure 3.17: BPM Architecture

CHAPTER
FOUR

Business Rules

Most business processes contain multiple decision points. At these decision points, certain criteria are evaluated. Based on these criteria or *business rules*, business processes change their behaviour. In essence, these business rules are the core drivers in the enterprise's business processes. Frequently, these rules are embedded within the business process itself, which makes changing and maintaining business rules difficult and costly. The way to avoid this problem is to separate business processes from business rules. In this chapter we will investigate the business rules approach as one of the BPM aspects ensuring more flexible and cost-effective management of business processes.

The chapter is structured as follows: Section 4.1 provides definitions of a business rule, a business rules approach, and a Business Rules Management System (BRMS). Section 4.1.2 makes an overview of the existing classification schemas of business rules and presents the widely accepted prescriptions for the design and realisation of business rules. Section 4.2 describes the technology background, an example solution of a BRMS, and a rule-based products' overview. Finally, we explain two different methods of the collaboration between a BPMS and a BRMS.

4.1 Introduction

According to the Business Rules Group (BRG)[1], an independent standards group of business and IT professionals, a *business rule* is *"a statement that defines or constraints some aspect of business. It is intended to assert business structure or to control or influence the behaviour of the business."*

Business rules represent the business logic of a company and exist in every enterprise. Typically, a company does not maintain an explicit list of all its business rules. Rather these rules are explicitly stated in documents about marketing strategies, pricing policies, customer relationship management practices, contracts, etc. Sometimes business rules are not even written down and exist as expert knowledge of particular staff members. There are also external business rules, that constrain the way an organisation conducts business but are defined by some other instance, for example legal requirements. Business rules are mostly spread (sometimes redundantly) in many pieces of program code and in databases. This has a highly negative influence on maintainability of the business logic, because business rules tend to change quite frequently. IT departments are being asked to reduce development and maintenance time, increase application performance, and improve application adaptability and flexibility. Business Rules Management (BRM) (also called business rules approach) is becoming the best practice for addressing these challenges.

[1] Business Rules Group, `http://www.businessrulesgroup.org/brghome.htm/`

4.1.1 Extract from the Business Rules Manifesto

Key notions of the business rules approach are outlined in the BRG's *Business Rules Manifesto* [19]. We present below an extract from the Manifesto to give readers a compact overview over the business rules principles from one of the pathfinders in the business rules approach.

- **Separate from Processes, Not Contained in Them** Rules apply *across* processes and procedures. There should be one cohesive body of rules, enforced consistently across all relevant areas of business activity.
- **Deliberate Knowledge, Not a By-Product** Rules build on facts, and facts build on concepts as expressed by terms. Terms express business concepts; facts make assertions about these concepts; rules constrain and support these facts.
- **Declarative, Not Procedural** Rules should be expressed declaratively in natural-language sentences for the business audience. A set of statements is declarative only if the set has no implicit sequencing. A rule is distinct from any enforcement defined for it. A rule and its enforcement are separate concerns. Rules should be defined independently of responsibility for the *who*, *where*, *when*, or *how* of their enforcement.
- **Rule-Based Architecture** A business rules application is intentionally built to accommodate continuous change in business rules. The platform on which the application runs should support such continuous change. Executing rules directly - for example in a rules engine - is a better implementation strategy than transcribing the rules into some procedural form. The relationship between events and rules is generally many-to-many.
- **Of, by and for Business People** Business people should have tools available to help them formulate, validate, and manage rules.
- **Managing Business Logic** Rules, and the ability to change them effectively, are fundamental to improving business adaptability.

4.1.2 Business Rules Types

Currently, there is no existing standard classification scheme for business rules. The Object Management Group (OMG)[2] is working on the *Semantics of Business Vocabulary and Business Rules* (SBVR) [51]. Among others, the objectives of this effort are to define:

- a metamodel for the specification of business rules,
- a metamodel for the capture of vocabularies and definition of the terms used in business rules,
- an XML representation of business rules and vocabularies that permits exchange among software tools that manage, display, use, and translate business rules.

A core idea of business rules formally supported by SBVR is the following from the above quoted Manifesto: "Rules build on facts, and facts build on concepts as expressed by terms. Terms express business concepts; facts make assertions about these concepts; rules constrain and support these facts." Facts and concepts are domain-dependent. A *concept* of in a car rental realm is, for example:

[2]Object Management Group, `http://www.omg.org/`

– "Customer" is someone who has rented a car in the last five years or has a current rental reservation.

A *fact* describes connections between terms. Here are two examples:

– A car group contains one or more car models.

– A car group has a rental rate.

Based on these two concepts, SBVR differentiates three types of business rules: *static constraints*, *dynamic constraints*, and *derivation rules*. A similar classification is given by BRG in [18]: *structural assertions*, *action assertions*, and *derivations*. J. Hall in [34] defines the same types, calling them respectively *fact-based* or *structural*, *dynamic*, and *derivation* rules. However, there are some more or less different classification proposals. Thus, J. A. Bubenko in [42] defines *derivations*, *event-action*, and *constraint* rules; C. J. Date in [22] distinguishes *constraints* and *derivations*; G. Wagner in [69] determines *constraints*, *derivations*, and *reaction* rules. In our work we will follow the classification stated by SBVR and BRG, and use the following denotations: **structural rules**, **dynamic rules**, and **derivations**.

- **Structural rules** define restrictions on business concepts and facts. They detail a specific, *static* aspect of the business. For example:

  – Each customer may be the renter in many contracts.

  – A customer can rent at most one car at a time.

  – Total of money owed to the bank can't be greater than credit limit.

  – The location of each copy of book is unique and only one.

- **Derivation rules** are statements of knowledge derived from other knowledge by using an inference or a mathematical calculation. Derivation rules capture domain knowledge that does not need to be stored explicitly, because it can be derived on demand from existing or other derived information. The following definition of a customer category by an insurance company is an example of a derivation rule:

  – A customer who spends more then $1000 per year in total premiums is a "gold" customer.

  Another example of a derivation rule is a following mathematical equation from the car rentals realm:

  – Rental charge=days*group rental rate*(100-discount%)/100+penalty charges

- **Dynamic rules** concern some dynamic aspect of the business. They specify constrains on the results that actions can produce. For example:

  – A rental reservation must not be accepted if the customer is blacklisted.

  – For an internet request for a rental, quote the price in the currency of the country of residence of the customer.

Note that the dynamic rules are mostly associated with a concrete event (or occurrence), as the rental reservation occurrence in the examples above. Structural rules are actually associated with an event as well, the difference is, however, that structural constraints imply "any relevant event". Thus, a customer can *always* have at most one car at a time, either at the moment of a car reservation, pick-up, or delivery.

When business rules are meant to be directly enforced by an automated system, they are often represented as production rules: a *production rule* is an independent statement of programming

logic that specifies the execution of one or more actions in the case that its conditions are satisfied [25]. Production rules are the most convenient way to represent most business rules, as well as the most widely supported by existing rule engines, as we will see later in this chapter. A transformation of above listed business rules in production rules is rather straightforward for dynamic rules:

– If the customer is blacklisted, reject his/her rental reservation.
– If a rental request is an internet request, quote the price in the currency of the country of residence of the customer.

Some structural and derivation rules can be also modeled in form of production rules:

– If in the result of the actual transaction the total of money owed to the bank would be greater as the credit limit, the transaction has to be aborted.
– If a customer spends more then $1000 per year in total premiums, he/she becomes status "gold".

However, in some cases they are translated to database constraints or are supported (application dependent) in some other way.

In the context of a production rule (or *condition-action* rule), a condition might be either a data-dependent condition (cf. "customer is blacklisted") or an event:

– If a customer confirms the booking of a flight, the hotel booking service starts searching for appropriate rooms for this trip.

As a sequence, our first intuition is that an XChange rule can be an adequate representation for business rules, either in form of an *event-(condition)-action* or *condition-action* rule. However, possible XChange specification forms for business rules is one of topics of the next chapter. Let's go back to the business rules types.

The above presented business rules types – *structural*, *derivation*, and *dynamic* rules – are specified in conformity with the rules types given in [14] within the scope of logic languages for the Semantic Web: *normative*, *constructive*, and *reactive* rules[3].

From another point of view, dealing with the functional role of business rules, they can be divided into two types:

- **core business rules**
- **work practice rules**

While the rules of the first type are directed at achieving the business goals, the latter concern the operations of the organisation. As J. Hall, the co-chair of the European Business Rules Conference (EBRC)[4], mentioned on the EBRC 2005 in his overview over the business rules, it is useful to distinguish them for the following reasons:

– Core business rules tend to be changed gradually. Reorganisations happen relatively infrequently but have a major impact on the work practice rules.

– For the same core business rules there may be different work practice rules in different types of location.

[3] the fourth type – *descriptive specification* – corresponds to a conceptual *schema* of facts (fact structure) that declares the relevant *fact types* [51]

[4] European Business Rules Conference, `http://www.eurobizrules.org/`

– Core and work business rules may be implemented on different technologies (for example, a rule engine and a workflow management middleware).

4.1.3 Business Rules Applications

Rules technology is used in applications to replace and manage some of the business logic. Such a technology is best used in applications where the business logic is too dynamic to be managed at the source code level – that is, where a change in a business policy needs to be immediately reflected in the application. Applications in domains such as insurance (for example, insurance rating), financial services (loans, fraud detection, claims routing and management), government (application process and tax calculations), telecom customer care and billing, e-commerce (personalising the user's experience), media companies (rights, contracts and royalties management), and so on benefit greatly from using rules technology. Let's have a look at a selection of typical business rules applications in the areas of e-commerce personalisation, loan underwriting, and customer relationship management. The business rule examples used below are defined in [47].

E-Commerce Personalisation More and more companies extend their sales and services from traditional channels to the Web. The ability to personalise each customer's experience becomes essential for these companies to attract, satisfy, and retain customers. The personalisation based on rules enables a precise update of the displayed content for each individual. By analysing the requests and specific information of the customers visiting a Web site and applying business rules against the gained information, visitors can be provided with information and recommendations that can be relevant to them. Web traders can inform visitors about interesting promotions, offer them discounts based on their purchasing history, and take advantage of the cross-sell. Here are some typical rules:

– If the shopping cart contains more than $100 worth of CDs, give $5 off the next purchase.

– If the customer purchased a Pilot III last week, show all Pilot accessories.

– If the customer travelled to Europe within the last year, send notification of European travel promotions.

By externalising the business rules, Web traders can create a more flexible personalisation solution. Business rules can be changed by the marketing staff, enabling them to immediately implement new promotions or offer new products. Vendors such as BEA and IBM are offering personalisation solutions that allow the trader to define and modify business rules in WebLogic Commerce Server[5] and WebSphere Commerce Suite[6] respectively.

Mortgage Loan Underwriting Mortgage industry is one of the first business domains that apply rule-based applications. Rules are used, for example, to simulate the decision-making process of a human underwriter. To stay competitive, lenders have to develop more types of mortgages and mortgage finance products to offer flexible alternatives to their consumers. At the same time, in order to guarantee that the risk for these new products is managed, the credit policy organisations within lending institutions are defining new risk rules. In addition, lenders must follow the regulations

[5] BEA WebLogic Portal, http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic/portal

[6] WebSphere Commerce, http://www-306.ibm.com/software/de/websphere/commerce/

established by the federal government and the policies of the secondary mortgage institutions. The use of business rules applications for the definition of new types of products and new risk management norms allows many lenders to implement and maintain the large number of rules required in the underwriting applications. For example:

– If the borrower is a first-time homebuyer, offer the no-down-payment option.

– If the lien type is a second mortgage, the borrower's occupancy status must be principal residence.

– If the transaction is a cash-out refinance, the loan-to-value ratio must be less than or equal to 85%.

Telecom Customer Care and Billing The telecom industry is a world of rapid deregulations and innovations. Its services providers must be able to react immediately to market changes, customer demands, or technological progress. In order to stand out against the competitors, telecom companies have to offer the customers new service packages and attractive prices continually. For example:

– If the destination is a preferred number, give 50% discount on call.

– If the customer is a member of nickel nights plan and call time is after 5 p.m., billing rate is $0.05 per minute.

To be effective, these promotions must be incorporated into the billing system. The business rules approach offers time-saving and cost-effective ways to do it. Externalising the business rules in the billing system and allowing the marketing department to define and modify them is the key to that.

Adopting a rule-based approach is useful in virtually every business domain and has the following advantages:

– Rules that represent policies are easily communicated and understood.

– Rules retain a higher level of independence than conventional programming languages.

– Rules separate knowledge from its implementation logic.

– Rules can be changed without changing source code.

These benefits, however, are not without cost. As with any tool, the decision to integrate a rule technology into the application system should be based on cost versus benefits. The cost includes not only the learning process and the efforts to create an interface between the existing system and the new software. In addition, different solutions use different formats and syntaxes for defining rules. Therefore, if an organisation decides to move from one tool to another, business analysts and developers must learn and understand the operation of the new one. The lack of standards may be a major cause of deterring businesses from using rule-based applications [46] and one of the explanations to the palette of different solutions in the domain of the rules technology.

4.2 Business Rules Technology

4.2.1 Background

The earliest rules technologies had their roots in artificial intelligence (AI) methodologies. The first AI system using *rule-based* knowledge representation was MYCIN [75, 20, 17], developed at Stanford University in the 1970s. Used to diagnose blood diseases and recommend treatments, MYCIN's core concept was the separation of knowledge and control. Knowledge, represented in rules (that's why *rule-based*), was separated from the control logic – *reasoning*, or *inference engine*, responsible for evaluating and executing the rules. This architectural principle applies equally to other rule-based, sometimes called *expert* or *knowledge-based* systems. A lot of them appeared in the 1980s, during the AI boom.

The widely used knowledge representation in expert systems is the *production* or *condition-action rule*. Such a rule consists of an `IF` part and a `THEN` part. The `IF` part lists a set of conditions in some logical combination. If the `IF` part of the rule is satisfied, the `THEN` part is executed.

By reasoning in a rule-based system two different *problem-solving models* or *paradigms* can be used. If the chaining of an `IF-THEN` rule starts from a set of conditions and moves towards some conclusion, the method is called *forward chaining*. If the conclusion is known but not the path to it, then reasoning backwards is called for, and the method is *backward chaining*. These problem-solving methods are built into program modules (inference engine) that manipulate and use knowledge bases.

Although the interest surrounding rule-based programming and expert systems reduced in the late 1980's (the knowledge acquisition methods suffered on performance), the technology continued to be used for certain types of applications, for example in such rules-heavy industries such as finance and insurance. The current trend is to use rule-based technology again, though more effective and more efficient. Nowadays the major vendors offer such a technology as a complete *Business Rules Management System*.

4.2.2 Business Rules Management System (BRMS)

Business Rules Management System (BRMS) is a set of software for modeling, writing, testing, deploying, and maintaining business rules. As shown in Figure 4.1, a typical BRMS provides a modeling tool for the design and management of business rules, a rule repository where business rules are stored, and a rule engine for the execution and deployment.

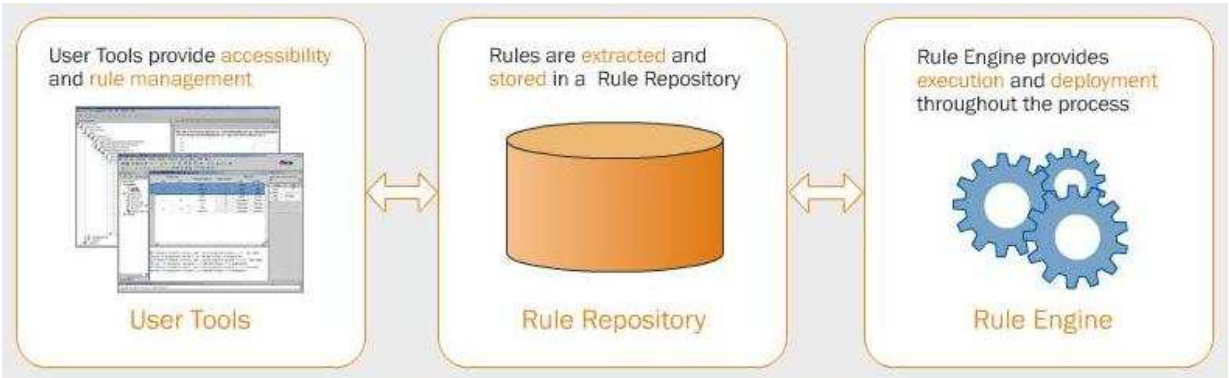

Figure 4.1: BRMS Architecture

A modeling tool provides a graphical user interface for writing business rules. Mostly, such a tool supports a natural language syntax to be easily used by non-technical business personnel.

A repository provides the facilities required to store, organise, and manage business rules. It offers services for versioning, history, permission management, etc.

A rule engine is a software component designed to evaluate and execute rules, mostly represented as `IF-THEN` statements. A rule engine implements all of the logic necessary to perform rule evaluation and execution. As a result, the rules can be coded as stand-alone atomic units, separate from and independent of the rest of the application logic. This makes them easier to develop and maintain.

Rule engines have a proprietary rule language for writing rules. To implement an application using a rule engine, it is necessary to write the rules in the rule language and embed the rule engine into the application. In the next two sections we will see how it works on two example implementations.

The use of a rule engine improves application performance since rule engines implement an algorithm to optimise rule evaluation. The most widely used one is the RETE algorithm. In RETE rules are compiled into a discrimination network that enables conditions shared by multiple rules to be evaluated just once [76].

Typically, existing business rule engines do not support all the different rule types as described above (*structural*, *derivation*, and *dynamic* rules). More precisely, business rules have to be modeled appropriately in form of production rules. Below we present an example of possible rules for the telecommunication sector from [63]:

```
<ruleset name="CellPhoneRegistration">
   <parameter name="cust" type="Customer"/>
   <parameter name="service" type="ServiceDescription"/>
   <rule>
      <condition>
         cust.registrationDate >= 1.8.2005
      </condition>
      <condition>
         cust.registrationDate <= 1.9.2005
      </condition>
      <action>
         service.rate = rate* 0.9
         service.freeSMS += 100
      </action>
   </rule>
   <rule>
      <condition> cust.sex == female </condition>
      <condition>
         hasBirthDayInNextTooWeeksAfterRegistration(cust)
      </condition>
      <action> sendFlowersToCustomer(cust) </action>
   </rule>
</ruleset>
```

In this example, the rules are specified in a high level domain-specific language (which is supported, for example, by Drools[7]). These rules can be invoked by a business application when, for

[7] Java Rule Engine Drools, `http://www.drools.org`

example, a new customer is registered. The first rule implements a typical rule for a marketing campaign. When the customer registers between the 1^{st} August and the 1^{st} September, he or she gets 10% discount and 100 free SMS. The second rule states, if the customer is female and has her birthday in the next two weeks, flowers should be sent to her home[8].

4.2.3 A BRMS Solution

In this subsection we illustrate the way a rule engine works from the developer's standpoint. We introduce JRules, a BRMS solution of ILOG, one of the leaders in Gartner's Magic Quadrant for Business Rule Engines[9] for the last three years. It is an example of a widely-used product with rather extensive and clear product documentation and not to be taken as our preference against other products.

ILOG JRules is a complete BRMS for the Java environment. It includes all the tools necessary for modeling, writing, testing, deploying, and managing business rules throughout the enterprise. This BRMS provides a modeling tool for definition and maintaining of business rules (called Rule Builder), a repository for organising and storing business rules, and a rule engine for executing them. According to the product documentation the rule engine "can accommodate thousands of rules and has the ability to fire thousands of them per second" [39].

JRules provides for business rules application development a Java library. Because the rule engine is a part of it, embedding it into an application is like adding any other Java class. The application instantiate namely an `IlrContext` object:

```
<!-- create an ILOG rule engine -->
IlrContext myContext = new IlrContext();
```

Let's see what are the constituents of the instantiated object.

Context, Working Memory, and Application Objects In their logic business rules typically reference application objects (e.g. customer, shopping cart, etc). *Working memory* is where ILOG JRules stores references to all of the objects with which it is currently working.

In ILOG JRules, rules are grouped into sets, called "rulesets"[10]. Rules in a ruleset (henceforth, we use the notation common for business rules community – "ruleset", written in one word) and the application objects that they reference are associated by what is called a *context*. That is, a context associates rules with the working memory and implements the rule engine that controls the relationship between them.

The constructor of the above instantiated object `IlrContext` created a ruleset container, a working memory, a rule engine, and a context to control it all.

In the next step, rules can be added to the ruleset container. Rules in the form of XML, a text file, stream, or URL can be added or removed from the engine.

```
<!-- get the IlrRuleset associated with myContext -->
IlrRuleset myRuleset = myContext.getRuleset();
```

[8] According to the example, a succession of condition or action specifications within the same rule denotes their conjunction. We have not investigated if other combinations of conditions and/or actions are possible since it is outside the scope of our work.

[9] classified positioning of vendors from one of the world's leading provider of research and analysis about the global information technology industry, `http://mediaproducts.gartner.com/reprints/isaac/125811.html`

[10] *"A ruleset is a collection of business rules grouped together for some purpose." [52]*

```
<!-- add rules to myRuleset -->
myRuleset.parseFileName("ruleset1");
```

In this example, the rules in a text file "ruleset1" are read into an `IlrRuleset` container, called `myRuleset`, and associated with `myContext`. The rule engine is now ready to examine the working memory, matching objects referenced in the working memory against patterns found in the rule conditions, and placing rule instances on an *agenda*, a container that stores rule instances that are ready to be executed. These collaborating elements are represented in Figure 4.2.

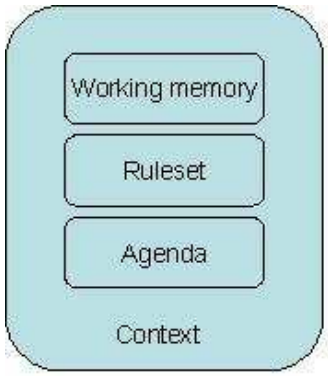

Figure 4.2: ILOG rule engine

Writing Rules Before the rules can be added to a ruleset, they have to be written. A standard syntax of rules in JRules is the `IF-THEN` construct. The examples below are given in a natural language syntax. In JRules business rules can be written either in Business Action Language (BAL) (with a syntax close to natural language) or Technical Rule Language (TRL), which are then translated to ILOG Rule Language (IRL) that can be directly executed by the rule engine.

Rule: GoldCategory

```
IF      the purchase value is greater or equal to $100
THEN    change the customer category to "gold"
```

Rule: FreeSample

```
IF      the item is fish
THEN    add free food sample to shopping cart
```

Execution of Example Application Let's assume the working memory of an application contains five objects as shown in Figure 4.3 and the ruleset possesses two rules defined above. Once the objects are in the working memory, the rule engine determines which rules are eligible for execution (comparing the contents of the working memory with the `IF` parts of the available rules) and places an instance of these rules on the agenda. In the right part of Figure 4.3, representing the agenda, four rules' instances are shown. It should be noted that three instances of the `FreeSample` rule are placed on the agenda since there are three instances of fish items in the working memory.

Rules are fired once the `fireAllRules` method is called on the `IlrContext` object:

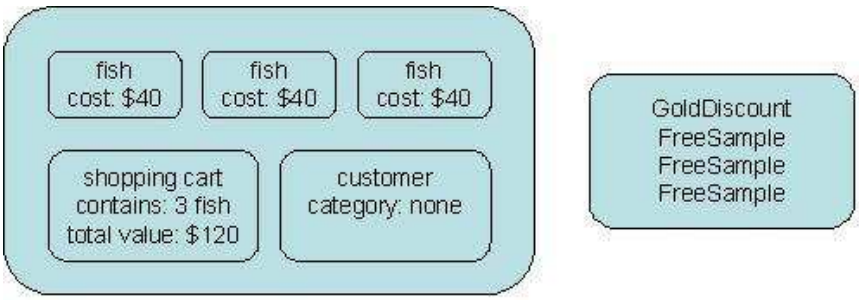

Figure 4.3: Working memory and agenda before the rules' execution

```
<!-- execute the rules on the agenda -->
myContext.fireAllRules();
```

Executing the `GoldCategory` will update the customer object in the working memory, changing the category to "gold". Once fired, the rule is removed from the agenda and will not be fired again (even though the rule condition is still true). The snapshot of the working memory and the agenda after the `GoldCategory` is fired is shown in Figure 4.4.

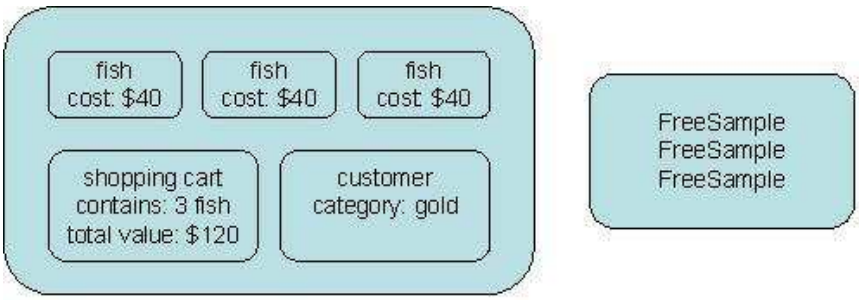

Figure 4.4: Working memory and agenda after the execution of the `GoldCategory` rule

The rule engine fires the rules from the agenda, till it is empty. At this point, the business application can proceed its further actions, having satisfied the customer with some free fish samples.

4.2.4 Product Profiles

In the following we present a short overview of other products supporting business rules management. Limited in time and financial resources to test all the products on market (provided by about 80 vendors currently active [34]) we focus on the particular vendors presented in the Gartner's Magic Quadrant for Business Rule Engines 2005, two mostly well-known open source solutions, and a complimentary product for verifying and validating business rules. The overview is based on the report [33], the vendors' presentations at EBRC 2005, and their product documentations [41, 39, 40, 38, 28, 29].

There are two broad categories of rule engines: event-driven and inference engines [34]. Event-driven rule engines handle rules as statements that must always be true (for example, *a car out on rental must not be assigned to another rental*) and invoke rules in relation to update events no matter where or when these events occur. If a change is detected that would cause a rule to be broken, the system rejects the change. Inference engines handle rules as production rules and are invoked by

applications when they need a decision or result. Garnter's six "Leaders" (positioned in the Leader Quadrant[11]) – Fair Isaac, Computer Associates, ILOG, Pegasystems, Gensym, Corticon – are all but one (Corticon) inference-driven (and 80% of the "Visionaries"). Let's analyse some of these products.

HaleyAuthority from Haley Systems[12] is a rule authoring and knowledge management system that generates code for the rule engine HaleyRules. HaleyAuthority is a Window application, but it does have a Web services components as well. HaleyRules is available for both .NET and Java environments. It offers natural language authoring and automatic code generation for .NET and Java applications.

Blaze Advisor from Fair Isaac provides the same BRMS on 2 main platforms, Java and .NET. The former requires a JVM to be installed, and the latter requires Microsoft's .NET Framework to be installed. Blaze Advisor offers: multiple views of the same rule (English-like, decision tables[13], etc.), code deployment for various installations, version control, debugging tools, an English-like rule-building language. According to the author in [54], Blaze Advisor doesn't match JRules in performance, but offers far more tools and views for design, analysis, and debugging.

PegaRULES[14] from Pegasystems Inc. provides HTML rule forms to build, manage, and configure rules; offers a variety of rule specification types; provides both forward and backward chaining; uses Microsoft Visio as a graphical front-end to the rulebase; can be included with other systems via Web Services, Enterprise Java Beans, and other protocols.

CleverPath Aion Business Rules Expert from Computer Associates[15] is RETE-based and offers a component-based development environment. However, the rule engine is not offered as an embeddable component.

Corticon[16] is a non-RETE-based rule engine. It generates Web service, Java and J2EE applications on top of existing applications but does not offer a set of components for embedding in them.

RulesPower is a product of the RulesPower company[17] co-founded by Ch. Forgy, the inventor of the RETE algorithmic concept. RulesPower's business rules engine is the only known to have implemented RETE III – the most sophisticated "tuning" of the original algorithms. In autumn 2005 Fair Isaac announced that has acquired RulesPower Inc. to complement its Blaze Advisor rules management system. Since Fair Isaac's RETE implementation has struggled in scenarios with large number of rules, the new RETE III engine would give Fair Isaac a high-performance alternative. In literature the RulesPower was mentioned as having focused on business process modeling, using rules for the decision-making capabilities as part of a workflow based solution [33, 64]. Most likely, this will be changed due to the fact that the product is to be integrated into some others.

Jess is an open source solution. It is a rule engine and scripting environment written in Java at Sandia National Laboratories in Livermore[18]. Jess allows to build Java software that has the capacity to perform inference on declarative rules. "Jess is small, light, and one of the fastest rule engines available" [33]. However, Jess is not a BRMS, that means it does not provide any authoring and management tools.

---

[11] Gartner's Magic Quadrant for Business Rule Engines, `http://mediaproducts.gartner.com/reprints/isaac/125811.html`

[12] Haley Systems, `http://www.haley.com/`

[13] a powerful way to visualise and edit chains of dependent rules in the form of a spreadsheet (see the representation of discounting rules in Section 6.1.2)

[14] PegaRULES, `http://pegasystems.com/Products/RulesTechnology.asp`

[15] Computer Associates, `http://www.ca.com/`

[16] Corticon, `http://www.coricon.com/`

[17] Rules Power, `http://www.rulepower.com`

[18] Jess, `http://www.herzberg.ca.sandia.gov/jess/`

Drools[19] is another open source solution. As well as Jess it uses RETE algorithmus and is implemented in Java. It processes RETE graphs and is therefore a purely forward chaining system. Drools does not provide management functions.

Versata from the eponymously named company[20] is a BRMS product that focus on database-oriented applications.

LibRT[21] offers a complimentary product to any BRMS. LibRT VALENS is the first independent product targeted at verifying and validating business rules created in third-party BRMSs.

4.3 Business Rules in a BPM Context

After discussing the main concepts of the business rules approach, we consider the ways it can be used in BPMSs. According to our literature investigations, there are two methods: the collaboration of a BPMS with a BRMS on the one hand, and their merge on the other. We present both methods without judging about their advantages or week points.

There are two alternative approaches to include business rules in business applications. One is an embeddable rule engine such as that provided by ILOG (has been introduces in Section 4.2.3). The ILOG system provides a business rule engine such that any specific business application can be linked to a rules-based reasoning system. A business application allows the rule engine to reason about business objects and update them by triggering events or invoking specific processes based on the outcome of the rules.

An alternative approach to including rules in an application is to use rules for the decision-making capabilities as part of a process. One example of this is a system provided by RulesPower. RulesPower uses a workflow model to describe a business process and has embedded in the workflow the ability to include rule-based decision making points that can operate over predefined business objects.

4.3.1 Embeddable Rule Engines

The following example (from the Oracle Technology Network's article [31]) illustrates the integration of Oracle BPEL Process Manager with ILOG's BRMS – JRules.

Separating Rules from Processes As illustrated in Figure 4.5 business logic of this IT infrastructure is spread across three different layers: business processes, Web services, and business rules.

Business Process Layer is responsible for the entire execution of the business process. Web service Layer exposes the existing application layer functionality as services. Rules Layer is typically the center of complex business logic. Extracting business rules as a separate entity from business process leads to better decoupling of the system, which, in consequence, increases maintainability. As rules are exposed as services in the Web services layer, they can be reused across all inter-enterprise application, making the development of new applications and intergrations easier.

Development and Maintenance To illustrate the development process, the authors use the example of a business process called *Eligibility Process*. This process determines the eligibility of a family for a specific healthcare program. A family can be assigned to Healthcare Program 1 or Healthcare

[19] Drools, `http://drools.org/`
[20] Versata, `http://www.versata.com/`
[21] LibRT, `http://www.LibRT.com/`

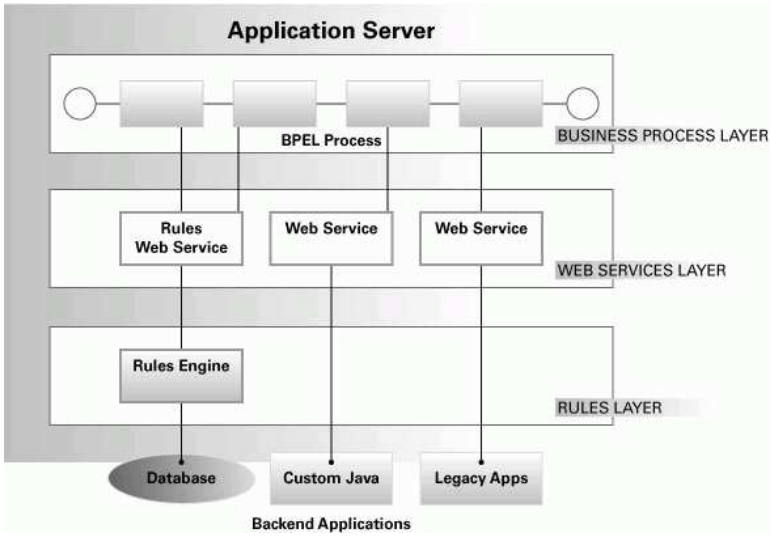

Figure 4.5: Business logic layers

Program 2, depending on the specific attributes (income, number of children, etc.). The typical process of the development of the relevant business rules consists of three steps:

1. Create rules in a ruleset.
2. Expose the ruleset as a Web service.
3. Invoke the ruleset Web service from BPEL.

The ILOG rule editor, Rule Builder, supports the definition of rules through its Business Application Language. It has a default template and allows a developer to create conditions and action using the `IF-THEN` construct (see Figure 4.6).

Wenn rule editing is complete, the analyst will export the rule package into a ruleset.

For the second step (exposing the ruleset as a Web service) JRules provides a tool to wrap a newly developed ruleset.

In the next step[22] the ruleset Web service can be integrated with the BPEL engine. It is as simple (or as complex) as to invoke any Web service. This process can be modeled on a BPEL graphical editor (for example Oracle BPEL Process Manager) or implemented using the code below.

```
<assign name="setAccount">
   <copy>
      <from variable="BPELInput"
      query="EligibilityProcessRequest/Account">
      </from>
      <to variable="webservice_account"/>
   </copy>
</assign>
```

[22] the detailed implementation of the ruleset and the `Eligibility` Web Service can be found in [31]

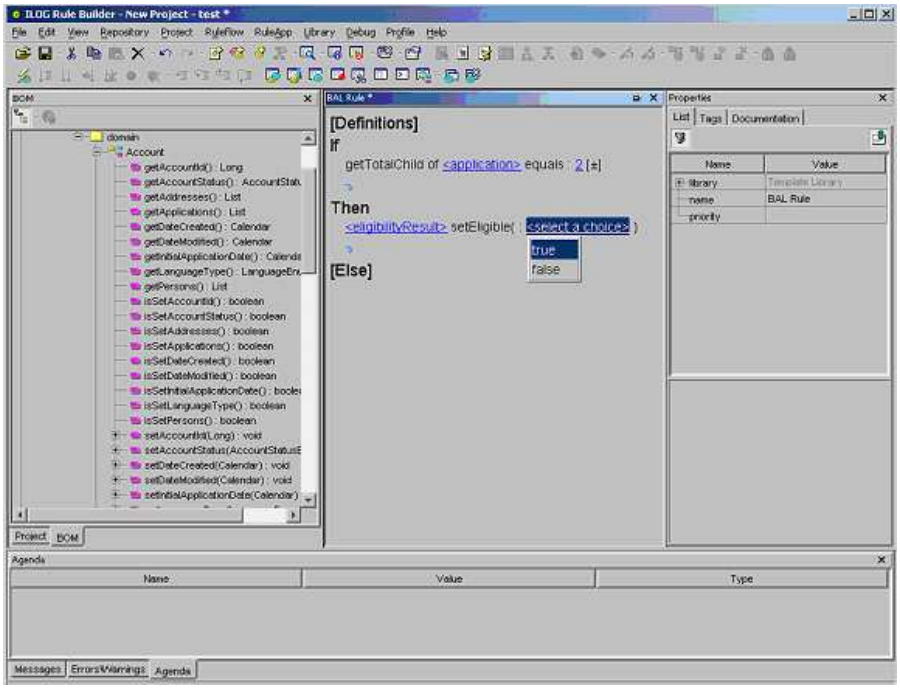

Figure 4.6: Creating a rule with the ILOG Rule Builder

```
<invoke name="CallEligibilityWebservice"
   partnerLink="EligibilityWebservice"
   portType="EligibilityService"
   operation="assessEligibility"
   inputVariable="webservice_account"
   outputVariable="webservice_eligibilityResults"/>
```

As we see, the design and development of rules, Web services, and BPEL processes involve multiple different technologies. The rule engine invocation occurs across three tiers: BPEL (`Eligibility Process`) invoking the rules Web service (`Eligibility`, rule engine application code receiving input and returning results). Based on these results, the BPEL `Eligibility Process` will invoke different branches. Program 1 and 2 each execute different steps, and these steps can be easily modified using a BPEL designer.

In this section we have discussed how BPM and BRM technologies work together. Although this example is based on Oracle BPEL Process Manager and ILOG JRules specifically, it is applicable to many other environments.

4.3.2 Rules-driven BRMS

As we have already mentioned in the beginning of this section, RulesPower is an example of an alternative approach for including rules in an application. The idea of this product is to use rules for the decision-making capabilities as a part of a workflow based solution. A workflow model was used in this product to describe a business process and has embedded in the workflow the ability to include business rules based decision making points that can operate over predefined business objects. For example, “if credit risk is > 50% then refuse loan” or “if data collection is complete then email data

owner". However, in autumn 2005 the product has been acquired by Fair Isaac to complement its rules management technology. As a consequence, it is hardly possible that the RulesPower will be further used in its original form.

Summary Business rules approach becomes increasingly popular since it implicates more flexible and adaptable applications. In this chapter we have discussed business rules and business rule engines, and how to integrate a rule engine into a business process application. We also examined how business rules applications are developed using a rule engine. We summaries the issues of the investigation that are important for our future work:

1. Business rules are atomic, highly specific, and structured statements that constrain some aspect of a business, control or influence its behaviour.
2. The ability to change business rules effectively is a fundamental to improve business adaptability.
3. Business rules should be formulated, validated, and managed by business analysts with help of graphical tools. In a native language formulated rules (mostly as production rules) have to be automatically translated into executable rules. One-to-one relationship between the rule specification and its implementation ensures more flexibility and adaptability of the underlying system.
4. The platform on which the application runs should support continuous changes effectively. Therefore, business rules should be separated from other application code, particularly from business processes.
5. Rules are typically clustered into manageably-sized logically related rulesets.
6. Mostly, a business rules engine is a component within a larger application, for example embedded into a BPMS. In such a case, business rules are all encapsulated in rules tasks in a business process. An embedded rules engine (when invoked) processes over the current state of the business objects to see if the conditions of any rules are satisfied and subsequently executes resulting actions.

Part III

Use Cases in XChange

CHAPTER
FIVE

XChange Rules for Business Process Modeling

Having discussed the main principles and requirements of BPM and BRM systems in the previous chapter, we now analyse the suitability of the language XChange and its expressive power for modeling business processes. We examine the expressiveness of the language based on the workflow patterns – useful routing constructs systematically addressing workflow requirements. Due to this pattern-based analysis we establish to what extent the business process requirements are addressed in the current state of the language XChange. We also describe the mechanism of the business rules approach realisation in XChange, as business rules are a growing area of importance in BPM systems.

In Section 5.1 we introduce the workflow patterns stepwise, describe the implementation mechanism of these patterns in XChange, partially give the example implementations in other modeling languages, especially in BPEL, and propose new constructs in some cases to extend the expressiveness of the language XChange. In Section 5.2 we present the solution for the realisation of the business rules approach using XChange.

5.1 Pattern-Based Analysis of XChange

In order not to start the investigation of the usability of XChange for business process modeling from scratch, we base our analysis on existing practices in this domain. The BPM community has taken an approach to identify *patterns*[1] specific to business processes. In their article "Workflow Patterns" van der Aalst, ter Hofstede, Kiepuszewski, and Barrios describe twenty patterns addressing "comprehensive workflow functionality" [68]. The researchers have developed a Web site[2] that contains descriptions and samples of these patterns describing the behaviour of business processes, as well as evaluations of how workflow products and existing standards (among others UML, BPML, and BPEL) support them. These workflow patterns are arguably suitable for analysing Web languages, such as XChange, since the situations they capture are also relevant in this domain. We assume, the mentioned patterns and existing evaluations deliver a good basis for an in-depth analysis of the suitability and expressiveness of XChange as a business process modeling language.

The patterns are grouped into six main categories:

- Basic patterns
- Advanced branch and join patterns

[1] *"A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary context"* [61]
[2] Workflow Patterns, `http://www.workflowpatterns.com/`

- Structural patterns
- Multiple instances patterns
- State-based patterns
- Cancellation patterns

In the following we discuss each pattern's intent, examples, and possible XChange implementations.

5.1.1 Basic Patterns

Basic patterns handle with fundamental business process capabilities, such as sequence, branching, joining, parallel execution. The five basic patterns are: Sequence, Parallel Split, Synchronisation, Exclusive Choice, and Simple Merge.

Sequence is defined as an ordered series of activities, with one activity starting after the completion of the previous one. The WfMC[3] defines this behaviour as *Sequential Routing*.

Sequence is obviously one of the fundamental process patterns: almost every process has at least two activities to be performed sequentially. For example, a travel application form can be approved only after it has been filled; a car reservation will be executed after a credit card has been checked. In Figure 5.1 an example from [35] is illustrated: *The process of opening a bank account requires getting a manager to approve the account application, updating the account database, and sending a welcome package to the customer.*

Figure 5.1: Sequence Pattern

As one of the essential patterns in a business process, Sequence is directly supported by all BPM vendors and specifications (here and in the following discussion on the patterns we base on the product and standard evaluation from the authors of the workflow patterns[4], on the pattern-based analysis of BPEL in [79], and on the discourse about BPEL, BPMN, and BPML in [35]). In BPEL, for example, the implementation segment of the bank account process can be implemented using the `<sequence>` element (or `sequence` *activity* according to the BPEL terminology):

```
<receive name="account-request"... />
<sequence>
  <invoke name="get-approval"
          inputVariable="..."
          outputVariable="..." ... />
  <invoke name="update-account-db"
          inputVariable="..." ... />
```

[3] Workflow Management Coalition Terminology & Glossary (1999), http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf

[4] Product and Standard Evaluation, http://is.tm.tue.nl/research/patterns/products.htm

```
  <invoke name="send-welcome-package"
          inputVariable="..." .../>
</sequence>
```

The <invoke> operations can be either a one-way or a synchronous request/reply. The difference in implementation of these cases is the specified variables, associated with the invoke activity. For a synchronous operation, both input and output variables are to be specified (cf. the get-approval activity in the code above); for an asynchronous (one-way) operation only an input variable is required.

Notation For simplifying the understanding of examples, an XChange-like syntax is used in this chapter. Thus, the XChange keywords DO, ON, FROM, and END remain unchanged, whereas to denote an event, an update, and a condition, we use the following notations respectively: event {{ label }}, update {{ label }}, {{ label }} (where label stands for an informal explanation of the event/action content, update or condition specification). In case the differentiation between an event and an update is irrelevant, we use the action {{ label }} component. We tried to use self-explanatory labels that correspond to the textual explanations of the examples and their graphical representations.

In XChange we can model a sequence of actions using the linking of rules. This may be achieved by raising an event when the previous action completes and using the same event as a triggering one in the subsequent XChange rule (cf. XChange event get-approval in the schema below).

```
DO                      DO                      DO
  event {{                event {{                action {{
    get-approval            approval-granted        update-account-db
  }}                      }}                      }}
ON                      ON                      ON
  event {{                event {{                event {{
    account-request         get-approval            approval-granted
  }}                      }}                      }}
                        FROM
                          {{ approved }}
END                     END                     END
```

The key idea of the Sequence pattern is to start the execution of an activity *after* the previous one is complete. Given, the sequential activities have to be taken at different Web nodes or by different actors (execution entities), as in the example above, we need multiple XChange rules for the pattern's realisation. We cannot trigger get-approval and update-account-db events within the same rule (even using an ordered conjunction) since that would mean we *trigger* two activities *sequentially*, but not *execute* them that way. We have to wait till the activity triggered by the get-approval event is completed, before we can start the following update-account-db activity. Without a mechanism for synchronous communication, the only way to forward the information about the completeness of an activity is an additional message reply, such as the approval-granted in the example.

Discussion Linking of rules as the implementation approach for the Sequence pattern is a workaround solution, but the only possible one since in XChange sending of events is performed exclusively *asynchronously*. That is, the execution of an XChange program continues immediately after the send-operation, without waiting for a reply to the event message. The design decision to support only asynchronous communication in XChange is not haphazardly. As mentioned in [27] (Section 2.3.2, page 11): "Asynchronous communication seems to be the most favorable for a reactive

Web site. Using synchronous message-passing, execution of a program has to be suspended until a sent message is transmitted correctly. In a setting like the Web with unreliable communication, the suspension time can be quite long and has no clearly established upper bound. During this time the Web site cannot react to other incoming events." The realisation of the Sequence pattern requires, however, that a program has to be suspended until the process step triggered by the sent message is complete, i.e. until the sent message is transmitted and replied. To simulate such a *synchronous* communication model, we need an additional message exchange: a *recipient* of an event replies after the reaction activities are executed. The reply is to be evaluated with an event query of an XChange rule on the the *sender*-site. In the literature, for example in [45], this implementation approach – the linking of multiple rules – is stated to be a typical realisation of a sequence using ECA rules.

However, XChange supports the Sequence pattern directly if sequential activities are data updates. In this case we need only one XChange rule, having an ordered conjunction of the sequential updates in its action part:

```
DO and [
  update {{ update1 }},
  update {{ update2 }}
  ]
ON
  event {{ eventA }}
END
```

This solution does not require additional messaging. However, we can easily imagine, that a sub-activity (for example, `update2`) can be used in different business processes do not necessarily following the same order of activities. Thus, specifying `update1` and `update2` in the same XChange rule would obviously restrict the possible order of activities and contradict the principle of loosely-coupled components that the BPM community strives for. The possible solution to this problem can be a new structural construct, which abstracts a part of a program as a unit and can be invoked from different XChange rules. Such a construct will be discussed later in this section while analysing the Exclusive Choice pattern.

Conclusion The Sequence pattern is directly supported in XChange for data updates through a complex update, namely through an ordered conjunction of elementary updates, expressed by the keyword `and` and square brackets `[]`. *Workaround:* For the realisation of the sequential execution of activities performed by different actors (execution entities), multiple XChange rules with additional event messages are necessary. This seems to be a great disadvantage of an ECA-based language over procedural modeling languages and it can be overcome only with additional message exchange.

Parallel Split is a starting point of a branch that allows multiple activities to be performed concurrently. A single path in the process is split when two or more streams of work need to be executed at roughly the same time. The WfMC defines this behaviour as *Parallel Routing* and its starting point as *AND-Split*. For example: *After the customer's itinerary is received, the reservation process spawns into two different activities: handling an airline and a hotel booking* (see Figure 5.2).

The Parallel Split is supported by all BPM vendors and standards. In BPEL, for example, by defining activities that have to be performed quasi parallel as components of an activity of type `flow`:

```
<flow>
  activityA
```

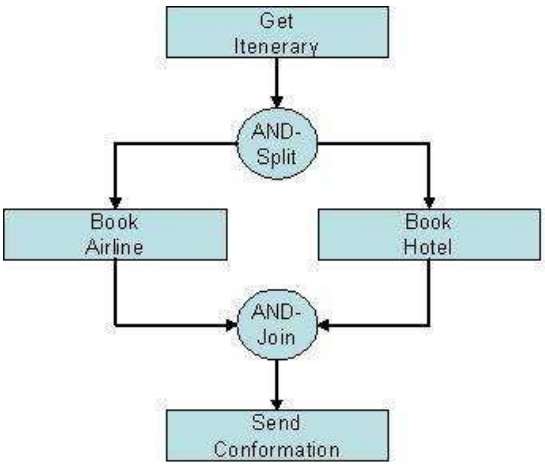

Figure 5.2: Parallel Split and Synchronisation Patterns

```
  activityB
</flow>
```

In XChange parallel flows of actions can be modeled in two different ways. One way is referencing the same event in different rules (cf. `book` in the example below):

```
DO                    DO                    DO
  event {{              action {{             action {{
    book                  book-airline          book-hotel
  }}                    }}                    }}
ON                    ON                    ON
  event {{              event {{              event {{
    get-itinerary         book                  book
  }}                    }}                    }}
END                   END                   END
```

Another way is to raise several events in one XChange rule using an unordered conjunction. These events trigger the subsequent actions in parallel (`book-airline` and `book-hotel`):

```
DO and {              DO                    DO
    event {{            action {{             action {{
      book-airline,       book-airline          book-hotel
      book-hotel        }}                    }}
    }}
  }
ON                    ON                    ON
  event {{            event {{              event {{
    get-itinerary       book-airline          book-hotel
  }}                    }}                    }}
END                   END                   END
```

The Parallel Split pattern for data update is not supported in XChange. The elementary updates `update1` and `update2` of the complex update

```
and {
  update {{ update1 }},
  update {{ update2 }}
  }
```

will be executed sequentially in some arbitrary order (see [58], page 121).

Conclusion The Parallel Split pattern is directly supported in XChange through an unordered conjunction of raising events expressed by the keyword `and` and curly braces {} in the action part of a rule. *Problem:* The parallel execution of elementary updates is not supported by XChange.

Synchronisation indicates the concluding point of Parallel Routing and is also known as *AND-Join*. At this point several paths converge into a single activity; it is a point of synchronising multiple threads. To continue the previous example of a travel reservation, we determine `send-confirmation` activity as a synchronising point of two booking proceedings: `book-hotel` and `book-airline` (see Figure 5.2).

In BPML and BPEL an element for parallel processing (`<all>` and `<flow>` respectively) manages the synchronisation itself. Therefore, when the `flow` in BPEL is completed, its sub-activities are guaranteed to have completed.

In XChange the merging of multiple parallel subprocesses can be realised using a conjunction event query (cf. the conjunction of `book-airline-ready` and `book-hotel-ready`):

```
DO
  action {{ send-confirmation }}
ON and {
  event {{ book-airline-ready }},
  event {{ book-hotel-ready }}
  }
END
```

Note that the use of an ordered or unordered conjunction is application dependent. An example of the Synchronisation pattern, realised with the construct `andthen` (ordered conjunction), is given in the following chapter (Section 6.2.3).

Discussion An event query, as it is presented in the code above, is not *legal* at the current version of XChange. The reason is that no any bounded life-span for events (to avoid increasing demand in event storage requirements) is specified (see Section 2.5.4 of this thesis and Section 4.8 in [27]). To make the event query legal, we would have to restrict it temporally either to a finite absolute time interval with `in`, to happen before some point in time with `before`, or to a finite duration (finite relative time interval) with `within`. Such a restriction cannot be combined with a business process modeling, as business processes are repeatable and can take place at any point in time. We believe that the life-span restrictions for event queries in the business process modeling are desirable, however, restrictions determined by the life-span of the business process and not by a finite time point or duration. In the following examples we will still use event queries with unbounded life-spans, keeping in mind, that the temporally restriction of legal event queries in XChange limits the language in terms of the business process modeling.

Note that in the example above, the actual booking activities are realised in other XChange rules (since we assume them to be external services).

Synchronisation of multiple updates (in the sample below it is the start of update `C`) can be realised in a single XChange rule:

```
DO and [
  and {
    update {{ A }},
    update {{ B }}
  },
  update {{ C }}
  ]
ON
  event {{ event1 }}
END
```

Conclusion The Synchronisation pattern is directly supported in XChange through a conjunction: in the action part of a rule for data updates and in the event part for expected event messages. Both an ordered and unordered conjunction can be used for the realisation of the pattern. The choice is application dependent. *Problem:* In the current version of XChange, every composite event query (used in our specification for synchronisation of, other than local updates, activities) has to be temporally restricted to a finite time point or interval. This approach limits the ability of the language to model the flow of activities in business processes, which are repeatable and mostly time irrespective.

Exclusive Choice is defined as a location where a process branches into multiple alternative paths, whereas exactly one of them can be chosen. The synonym notion of WfMC is *XOR-Split*.

The usage of such a pattern is commonplace. Here is a typical example: *If a customer's request to open a bank account is approved, he or she is sent an information package; otherwise, the customer is sent a rejection* (see Figure 5.3).

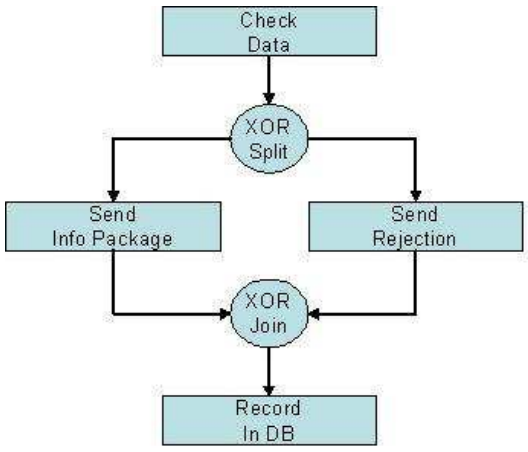

Figure 5.3: Exclusive Choice and Simple Merge Patterns

An natural solution for this pattern in a rule-based language is presented in [45]: different action parts of an ECAA (Event-Condition-Action-Alternative Action) rule raise different events.

This implementation approach (though restricted to a single condition and two possible decisions) is not supported in XChange. The only way to realise alternative actions in XChange is to use multiple XChange rules, one for each alternative, based on the evaluation of a condition:

```
DO                    DO
  event {{              event {{
    send-package          send-rejection
  }}                    }}
ON                    ON
  event {{              event {{
    check-data            check-data
  }}                    }}
FROM                  FROM
  {{ approved }}        {{ NOT approved }}
END                   END
```

As shown in the schema above, we need two XChange rules which are triggered by the same event and having such condition parts, that one of them is the negation of the another. This solution – evidently redundant coding of, for the most part identical, XChange rules – provokes time costs and error-proneness.

To compare, the standard business process modeling language BPEL provides a solution using a complex activity `<switch>`:

```
<switch>
   <case condition="condition1">
      activityA
   </case>
   <case condition="condition2">
      activityB
   </case>
   <otherwise>
      activityC
   </otherwise>
</switch>
```

According to the BPEL specification[5]:

"The `<switch>` activity consists of an ordered list of one or more conditional branches defined by `case` elements, followed optionally by an `otherwise` branch. The `case` branches of the `switch` are considered in the order in which they appear. The first branch whose condition holds true is taken and provides the activity performed for the switch. If no branch with a condition is taken, then the `otherwise` branch is taken. If the `otherwise` branch is not explicitly specified, then an `otherwise` branch with an `empty`[6] activity is deemed to be present. The `switch` activity is complete when the activity of the selected branch completes."

BPML used (the language is not under development any more, see Section 3.4.2) an analogue element:

[5] Business Process Execution Language for Web Services. Version 1.1, `http://www-128.ibm.com/developerworks/library/specification/ws-bpel/`

[6] a construct that allows to insert a "no-op" instruction into a business process.

```
<switch>
   <case>
      <condition> condition1 </condition>
      <action name="actionA"> ... </action>
   </case>
   <case>
      <condition> condition2 </condition>
      <action name="actionB"> ... </action>
   </case>
   <default>
      <action name="actionC"> ... </action>
   </default>
</switch>
```

The Exclusive Choice pattern is as well supported by the majority of workflow products: 14 of 15 products analysed in the mentioned product evaluation[7] provide a construct that enables a direct implementation of the pattern.

We believe, the direct support of the Exclusive Choice pattern would enrich the expressiveness of XChange without making the language unessential complicated. There are two essential points allowing such a support. First of all, a construct providing a disjunction. As already mentioned in Section 2.7, XChange has such a construct for *complex updates*. A complex update introduced by the keyword *or*, followed by the constituent updates enclosed in square brackets `[ ]` implements a *strict or*. Roughly speaking, we could specify the Exclusive Choice for raising events the same way. The second essentiality of the Exclusive Choice pattern is the ability to link a condition with the corresponding reaction. Unfortunately, the actual syntax of XChange doesn't offer such a possibility: a condition part is linked together with the whole action part of a rule rather than with a fragment of it. Thus, our attempt to support the Exclusive Choice directly results in the modification of the XChange rule syntax structure in a way that would allow the correlation of different conditions with different reaction activities. Before we describe the syntax and semantics of the rule's structure we propose, we make two further suggestions:

- to use the keyword `IF` in the condition part of an XChange rule instead of `FROM` to underline the conditional meaning of the corresponding rule part,
- to change the order of event, condition, and action parts in an XChange rule. Using the event part (introduced with the keyword `ON`) at the beginning of a rule would be more conform to the abstract structure of an ECA rule.

Thus, a simple XChange rule, consisting of a single event, condition, and action part, would look like:

```
ON
   <event>
IF
   <condition>
DO
```

[7] Product Evaluation, `http://is.tm.tue.nl/research/patterns/products.htm`

```
   <action>
END
```

The Web query can be left out:

```
ON
   <event>
DO
   <action>
END
```

The Exclusive Choice Pattern could be specified using an ordered disjunction of condition-action groupings in the part introduced by the keyword `DO` (such a complex combination is called a *complex reaction part* henceforth):

```
ON
   <event>
DO or [
   IF <condition1>
   DO <action>,

   IF <condition2>
   DO <actionB>,

   <actionC>
   ]
END
```

The informal semantics of such a rule is: the ordered disjunction in a complex reaction part consists of an ordered list of one or more condition-action groupings. Each condition is defined by `IF` keyword, each action by `DO`. The condition-action groupings are considered in the order in which they appear. The first grouping in the ordered disjunction whose condition holds true is taken and provides the activity performed for the complex reaction part. The following condition-action groupings will not be considered any more. If no grouping with a condition is taken, then the last specified action without the condition part is performed. This one (since not related to any peculiar condition specification) is linked with the condition defined for the whole complex reaction part. Hence, it is introduced by the keyword `DO` of the complex reaction part and does not need the same keyword written directly for its specification. If no action without a condition is specified, then an empty condition-action part must be deemed to be present to guarantee that the exclusive disjunction of actions completes even if none of conditions hold. Such an empty grouping can be simulated, for example, by means of a data query that always has an answer. The query `true { }` will be always successfully evaluated if the following Xcerpt rule is specified:

```
CONSTRUCT
   true { }
END
```

Such a query can be used in a complex reaction part, for instance:

```
ON
   <event>
DO or [
   IF <condition1>
   DO <actionA>,

   IF <condition2>
   DO <actionB>,

   true { }
   ]
END
```

The example about the bank account from the beginning of this paragraph can be implemented as following:

```
ON
   event {{ check-data }}
DO or [
   IF {{ approved }}
   DO event {{ send-package }},

   event {{ send-rejection }}
   ]
END
```

After the detection of the event `check-data`, the rule tests the condition `approved`. If this condition holds, the event `send-package` will be sent and the subsequent condition-action part will not be considered (in our example, the condition part is empty). Otherwise, the event `send-rejection` will be performed (the condition part of this grouping is empty, hence is always evaluated to true).

Keeping a condition and an action parts of a rule integrated together is perhaps less declarative, but enables creating a linkage between them. The same design approach is suggested in [10], where active rules for XML in the E-services domain are investigated. The authors note that "in the industrial development of trigger systems the condition and action parts of an active rule have been kept integrated" and suggest an "XML active document language" that consists of an event part and of a condition-action part. We haven't proved the statement about the trigger systems in today's industry, have noticed however independently, that the possibility to integrate different conditions with different reaction activities is an essential requirement for a process modeling language.

The language XChange has a compact human-readable syntax. It was one of the driving forces in the development of the language – to make it as compendious as possible and easy to use by programmers. The syntax changes proposed in this section can make the language more complicated and less plain. As we will see in the following chapter while implementing business processes, an XChange rule with a complex reaction part consisting of multiple condition-action groupings is not that clearly arranged as an XChange rule written in the syntax of the actual language version. The solution to the problem is to provide a possibility to implement a condition-action grouping separately as a named unit. Such units can be invoked from any action part of an XChange program (either an XChange rule or another unit). This has two advantages. First of all, an XChange rule stays easy

to read. Secondly, a condition-action part can be accessed repeatedly without the code having been written more than once.

An appropriate name for such a component (grouping condition and action parts together) seems to be a "procedure". Here some definitions of this term:

"A part of a program that is abstracted as a unit and can be called from multiple places, often with parameters"[8].

"A sequence of code which performs a specific task, as part of a larger program, and is grouped as one or more statement blocks"[9].

"A subprogram which is invoked by a procedure call statement (an instruction to carry out some action; a single step within a program)"[10].

"The detailed elements of a process used to produce a specified result"[11].

The schematic representation of the procedure construct is exemplified in the following:

```
PROCEDURE procedureA {{ }}        PROCEDURE procedureB {{ }}
IF <conditionA>                   IF <conditionB>
DO <actionA>                      DO <actionB>
END                               END
```

An XChange procedure can be invoked from an action part of an XChange rule or from an action part of another procedure:

```
ON                              PROCEDURE
  event {{ eventC }}               procedureC {{ }}
DO or [                         DO or [
      procedureA {{ }},            procedureA {{ }},
      procedureB {{ }}             procedureB {{ }}
  ]                             ]
END                             END
```

The return value of a procedure is *true* (corresponding to a successful execution) or *false* (corresponding to a failed execution). Thus, if the procedureA is successful, the procedureB will not be invoked. Otherwise, the procedureB will be executed. Note that the usage of unordered disjunction in the same rule is also possible, the result would be, however, nondeterministic: falls both `conditionA` and `conditionB` can be successfully evaluated, either `actionA` or `actionB` might be executed (depending on the runtime system "choice").

Conclusion XChange does not support the Exclusive Choice pattern directly. *Proposal:* For the future language development we propose to restructure an XChange rule in order to have a possibility to integrate different conditions with the corresponding reaction activities. Using the (ordered or unordered) disjunction of condition-action groupings in the part introduced by the keyword `DO` (the complex reaction part) would allow to realise the Exclusive Choice pattern both for data updates and raising events. To guarantee that *exactly one* activity of a complex reaction part will be performed, an action without any linking condition or a condition that is always satisfied should be specified within a complex reaction part. Next, we propose to replace the keyword `FROM` with `IF`, as the latter one

[8] The University of Texas, CS 307 Vocabulary, `http://www.cs.utexas.edu/users/novak/cs307vocab.html`

[9] From Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Procedure_(programming)`

[10] University of Brighton, Glossary, `http://www.it.bton.ac.uk/staff/je/adacraft/glossary.htm`

[11] Cooper Development Association, Glossary of Technical Terms,
`http://www.copper.org/applications/plumbing/techcorner/glossary_technical_terms.html`

seems to denote the conditional meaning of the rule part more expressively. Further on, to conform to the typical ECA rule form, the event part can be specified at the beginning of an XChange rule. In this section we follow, however, the syntax of the actual XChange version (or define explicit if not) to avoid any confusions in the context of the actual XChange syntax and our proposals.

Simple Merge is defined as being a point in a process where two or more alternative branches are joined together into a single path without synchronisation. It is an endpoint of a process split started by an Exclusive Choice. For example: *After an information package or a rejection has been sent to a customer, an appropriate database record will be made* (see Figure 5.3). It is an assumption of this pattern that none of the foregoing alternative branches are ever executed in parallel. In XChange we can realise the Simple Merge in a rule with a complex event query specifying an unordered disjunction, as shown in the scheme below. The risk that `record-in-db` activity will be executed twice is eliminated a priori due to the assumption mentioned above.

```
DO
  action {{ record-in-db }}
ON or {
  event {{ send-package-ready }},
  event {{ send-rejection-ready }}
  }
END
```

It is important to point out, that the single parts in the composite event query of the rule above (`send-package-ready` and `send-rejection-ready`) are actually event queries for confirmation messages that inform about the completeness of `send-package` or `send-rejection` activity. Again, we come across the same restriction of a rule-based language we have discussed for the Sequence pattern: we have to trigger a special event to model a reply message.

However, we can again differentiate the case where alternative activities that have to be merged, as well as the merge point itself, represent some data updates. Such a situation can be solved within a single XChange rule. For example, in the schema sample below update `C` can be executed only if (and after!) one of the disjunct updates (`A` or `B`) succeeds:

```
DO and [
      or {
         update {{ A }},
         update {{ B }}
      },
      update {{ C }}
  ]
ON
  <event>
FROM
  <condition>
END
```

Conclusion Simple Merge of data updates is directly supported in XChange through an unordered disjunction within a complex update. In order to merge a number of event messages, XChange uses

a complex event query specifying inclusive disjunction. Both cases are indicated with the keyword `or` and curly braces {}. *Workaround:* The drawback of the XChange implementation of the Simple Merge is, as in the case of the Sequence pattern, the fact that we need an additional event message exchange to model synchronous interactions. *Problem:* Besides, the inclusive disjunction event query, as it is presented here, is not legal (as in the case of Synchronisation) without a temporally restriction.

5.1.2 Advanced Branch and Join Patterns

Equally important as the basic patterns discussed in the previous section, but less well supported by BPM products and specifications, are the advanced branch and join patterns. There are five of them: Multi-Choice, Synchronising Merge, Multi-Merge, Discriminator, and N-out-of-M Join.

Multi-Choice is a point where a process makes a decision which alternative branch to take and allows more than one path to be chosen (to remain, the Exclusive Choice pattern, considered earlier, selects exactly one alternative). For instance, an adapted example from the underlying research work, "Workflow Patterns" [68]: *Within an investigation process either the insurance company or the fire department, none of them*[12]*, or both, must be informed* (see Figure 5.4).

Figure 5.4: Multi-Choice and Synchronising Merge Patterns

Such non-exclusive alternative actions can be specified using several XChange rules triggered by the same event but specifying different conditions and reactions:

```
DO
  event{{
    inform-fire-depart
  }}
ON
  event{{
    evaluate-damage
  }}
FROM
  {{ structural-damage }}
END
```

```
DO
  event{{
    inform-insurance
  }}
ON
  event{{
    evaluate-damage
  }}
FROM
  {{ damage >= $1000 }}
END
```

In the example above, only the `inform-fire-depart` event, only the `inform-insurance` event, none of them, or both can be triggered.

Again, if XChange would allow the integration of a condition with an action, we could implement the Multi-Choice pattern in a single rule, through a combination of the Exclusive Choice and the

[12] though not mentioned by the authors, the case is surely also possible.

Parallel Split. In the code schema below we use the the earlier proposed syntax with condition-action groupings. Each possible branch is preceded by an XOR-Split (exclusive disjunction expressed in XChange with `or` and `[]`) which decides, based on the condition evaluation, either to activate the branch or to bypass it.

```
ON
  event {{ eventA }}
DO and {
  or [
    IF {{ condition1 }}
    DO action {{ actionA }},

    <!-- otherwise -->
    true { }
  ],
  or [
    IF  {{ condition2 }}
    DO action {{ actionB }},

    <!-- otherwise -->
    true { }
  ] }
END

CONSTRUCT
   true { }
END
```

The same approach was used in BPML: multiple exclusive ORs (`<switch>` elements) run in parallel (`<all>` element), as shown below.

```
<all>
  <switch>
    <case>
      <condition> condition1 </condition>
      <action name="actionA"> ... </action>
    </case>
  </switch>
  <switch>
    <case>
      <condition> condition2 </condition>
      <action name="actionB"> ... </action>
    </case>
  </switch>
</all>
```

The informal semantics of the XChange rule presented above is defined as following: in response to the event `eventA` either both actions `actionA` and `actionB` (provided the conditions `condition1`

and `condition2` hold), `actionA` (provided `condition1` holds and `condition2` fails), `actionB` (if `condition2` holds and `condition1` fails), or none of them (none of the conditions hold) will be performed. Without the disjunctions between `actionA` and the query `true { }` (which is always satisfied) and between `actionB` and the same query, the realisation of the Multi-Choice pattern would be incomplete. Assume, the first tested condition (`condition1` or `condition2`) failed. The second one would be not considered any more since the conjunction (denoted with the keyword `and`) demands successful evaluation of all segments it combines.

Conclusion In the actual version of XChange, one has to specify separate rules for each alternative branch in a Multi-Choice. *Proposal:* A more elegant solution would be to use the proposed syntax of integrated condition-action parts. Through a combination of an exclusive choice of condition-action groupings within each alternative branch and a parallel split between them, one would be able to realise the Multi-Choice pattern within a single XChange rule.

Synchronising Merge is defined as a point where multiple branches converge together. Some of these branches are executed at runtime (are "active"), some are not. If only one activity of the alternative branches is active, the subsequent action is triggered as soon as this one is completed. If multiple branches are active (i.e. multiple activities have been started), the synchronisation of all active paths needs to take place before the following activity can be triggered. For instance, we extend the example introduced for the Multi-Choice pattern (Figure 5.4): *After the fire department and/or the insurance company have been contacted (if necessary), a report submission should be made*.

How this behaviour can be specified in a language based on ECAA rules has been described in [45]. The solution is to use several ECAA rules triggered by the same event but having different condition and action parts. The `ELSE`-branches of these rules do not result in a real action but serve as "dummies" to assure that the alternative branches can be merged by a conjunction. We exemplify this idea with XChange, using condition-action groupings. Both rules below are triggered by `evaluate-damage`; they fire `inform-fire-depart` or `fire-depart-ready` (`inform-insurance` or `insurance-ready`) depending on the condition `structural-damage` (`damage>=$1000`):

```
ON
  event {{
    evaluate-damage
  }}
DO or [
  IF {{ structural-damage }}
  DO event {{
    inform-fire-depart
  }},

  <!-- otherwise -->
  event {{
    fire-depart-ready
  }}
  ]
END
```

```
ON
  event {{
    evaluate-damage
  }}
DO or [
  IF {{ damage>=$1000 }}
  DO action {{
    inform-insurance
   }},

  <!-- otherwise -->
  event {{
    insurance-ready
  }}
]
END
```

The rules, triggered by `inform-fire-depart` and `inform-insurance` events, fire `fire-depart-ready` or `insurance-ready` as well:

```
ON
  event {{
    inform-fire-depart
  }}
DO and [
  action {{ actionA }},
  event {{ fire-depart-ready }}
  ]
END
```

```
ON
  event {{
    inform-insurance
  }}
DO and [
  action {{ actionB }},
  event {{ insurance-ready }}
]
END
```

Both alternative (but not exclusive!) branches will be merged by a conjunction in an event query. The following activity (`submit-report` in our example) will be performed only after each of the started foregoing activities has been completed:

```
ON and {
  event {{ fire-depart-ready }},
  event {{ insurance-ready }}
  }
DO
  <action-part>
END
```

The suggested schema can be simplified in XChange if we combine two rules triggered by `evaluate-damage`. This is however only possible if we use the earlier proposed syntax:

```
ON
  event {{ evaluate-damage }}
DO and {
    or [
      IF {{ structural damage }}
      DO event {{ inform-fire-depart }},

      <!-- otherwise -->
      event {{ fire-depart-ready }}
    ],
    or [
      IF {{ damage>=$1000 }}
      DO event {{ inform-insurance }},

      <!-- otherwise -->
      event {{ insurance-ready }}
    ]
  }
END
```

In comparison, the BPEL language directly supports the Synchronising Merge pattern using the principle of *dead path elimination*. This principle states that the value of an *incoming link* (start point of an activity) is propagated to its *outgoing link* (start point of the subsequent activity). In the following

code sample, adapted from [79], if condition `condition1` (`condition2`) evaluates to true, activity `activity1` (`activity2`)[13] receives a positive value and it is therefore executed. On the other hand, if condition `condition1` (`condition2`) evaluates to false, activity `activity1` (`activity2`) receives a negative value, and it is not executed but still propagates the negative value through its outgoing link `link1` (`link2`). In particular, if the two conditions `condition1` and `condition2` evaluate to true, `activity3` is triggered only after both `activity1` and `activity2` are completed. Note that the dead path elimination semantics is used if the attribute `suppressJoinFailure` is set to "yes".

```
<flow suppressJoinFailure="yes">
  <links>
    <link name="link1"/>
    <link name="link2"/>
    <link name="link3"/>
    <link name="link4"/>
  </links>
  <empty>
    <source linkName="link1"
      transitionCondition="condition1" />
    <source linkName="link2"
      transitionCondition="condition2" />
  </empty>
  activity1
    <target linkName="link1">
    <source linkName="link3">
  activity2
    <target linkName="link2">
    <source linkName="link4">
  activity3
    <target linkName="link3">
    <target linkName="link4">
</flow>
```

However, the Synchronising Merge pattern is not supported by BPML, and there are only two workflow engines known to the authors of the workflow patterns, that provide a straightforward construct for the realisation of this pattern.

Conclusion *Workaround:* Raising additional messages allows to realise the Synchronising Merge pattern. This solution, though not straightforward, is conform to the typical realisation of processes with ECA rules [45].

Multi-Merge is a point within a process where multiple branches merge and the following activity is instantiated for every active alternative branch. Thus, this pattern differs from the Synchronisation or Simple Merge patterns since for these patterns the target activity will be executed only once.

According to the authors of the regarding patterns, Multi-Merge is useful in cases where "two or more parallel branches share the same ending". For example: *In a process that permits simultaneous auditing and processing of an application, each independent action should be followed by a close case activity* (see Figure 5.5).

[13] here `activity` denotes "each possible activity in BPEL", see Section 3.4.2

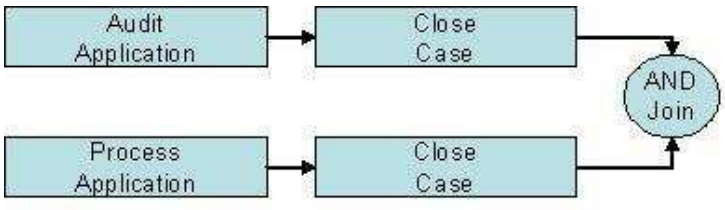

Figure 5.5: Multi-Merge Pattern

Neither BPEL nor BPML support directly this pattern. In XChange we would use a complex event query specifying inclusive disjunction:

```
DO
  action {{ close-case }}
ON or {
  event {{ audit-application-ready }},
  event {{ process-application-ready }}
  }
END
```

The event query will be evaluated both after the auditing application and the process application are finished. As a result, the action `close-case` will be performed twice, for each evaluation of the complete event query.

The Multi-Merge of different updates is however not directly supported by XChange. The appropriate realisation would be to implement `close-case` update twice, after `audit-application` and `process-application` respectively.

```
DO and {
  and [
    update {{ audit-application }}
    update {{ close-case }}
  ],
  and [
    update {{ process-application }},
    update {{ close-case }}
  ]
  }
ON
  <event>
END
```

Conclusion XChange supports the Multi-Merge pattern directly using complex event queries specifying inclusive disjunction. For data updates, however, a multiple implementation of the activity that follows a Multi-Merging point is necessary.

Discriminator pattern is a point where parallel branches converge and the subsequent activity is executed exactly after the first of parallel branches arrives. All remaining branches will eventually

arrive at the Discriminator point, but they will be ignored. As an example, the authors of [68] (page 14) give the following situation: *To improve query response time a complex search is sent to two different databases over the Internet. The first one that comes up with the result should proceed the flow. The second result is ignored.*

This pattern is not directly supported in XChange. Neither an *inclusive disjunction* of an event query expressed by `or` and {}, nor *multiple inclusions and exclusions*, expressed by the keyword `of` preceded by the occurrence specification construct `at least n` {`eq`} or just `n` {`eq`}, can be used for capturing it. In the first case, the reactive rule would fire each time the disjunction event query evaluates, not only once after the first answer is found. The reason for not being able to use the `at least 1 of` {`eq`} construct, is the fact that the event query (`eq`) specifies a certain event message but not a set of event messages. The semantics of the construct `1 of` $\{eq_1, \ldots, eq_n\}$ is not convenient with the pattern as well. This construct represents the exclusive disjunction, that means it is successfully evaluated only if *exactly one* event query from $\{eq_1, \ldots, eq_n\}$ is successfully evaluated.

In the following we propose a construct that can be used for the direct implementation of the Discriminator pattern. It can be expressed by the grammar rule shown below (where `ceq` denotes a complex event query).

```
ceq ::=   "first" n "of" "{" eq  ("," eq)* "}" "during" "{" ceq "}"
        | "first" n "of" "{" eq  ("," eq)* "}" "during" FiniteTimeInterval

n                  ::= [1-9][0-9]*
FiniteTimeInterval ::= "[" TimePoint ".." TimePoint "]"
TimePoint          ::= ISO_8601_format
```

The keyword `first` introduces a quantification event query of a rule, which is to fire after the first `n` occurrences of instances of a given set of event queries (following the keyword `of`) take place.

Discussion We recall that event queries restricted by means of the construct `during` are evaluated at each successful evaluation of the composite event query or at the end of the given finite time interval. To support the Discriminator pattern it should be possible to react earlier if the first `n` events have been received.

Conclusion XChange does not support the Discriminator pattern directly. *Proposal:* We propose to extract the actual event queries patterns with a construct introduced by the keyword `first` for the direct implementation of the pattern.

N-Out-of-M Join is defined as a point where *M* parallel paths converge into one; the subsequent activity should be activated once *N* paths have completed. All remaining paths should be ignored. For example: *A paper can be processed after receiving two of three possible reviews. The third one can be ignored.*

The construct presented for the Discriminator pattern can be as well used for the N-Out-of-M Join, with the only difference in the input of `n` (n instead of 1):

```
DO
  action {{ process-paper }}
ON
  first 2 of { event {{ review }} }
END
```

Conclusion In its actual version, XChange does not support the N-Out-of-M pattern directly. *Proposal:* A new construct, presented for the Discriminator pattern, can be as well used for the N-Out-of-M Join.

5.1.3 Structural Patterns

The two patterns in this group cover looping and independence of separate process branches. These patterns are: Arbitrary Cycles and Implicit Termination.

Arbitrary Cycles is a mechanism for allowing segments of a process to be repeated. It is an unstructured or non-block structured loop. That is, the looping section of the process may allow more than one entry, exit, and jump-back point. This approach is looser as block-structured loops such as `WHILE-DO` or `DO-WHILE` and can be important to model a complex looping situation. For example: *In the loan process*, as designed in [35] and shown in Figure 5.6, *the activity, dealing with the validation of the application* (`validate-appl`)*, can be retried not only when validation fails, but also when something in the application prevents approval*. Though this logic is difficult to realise with a structured loop, this approach is used to model Arbitrary Cycles in BPEL, BPML, and in the majority of the analysed workflow products[14].

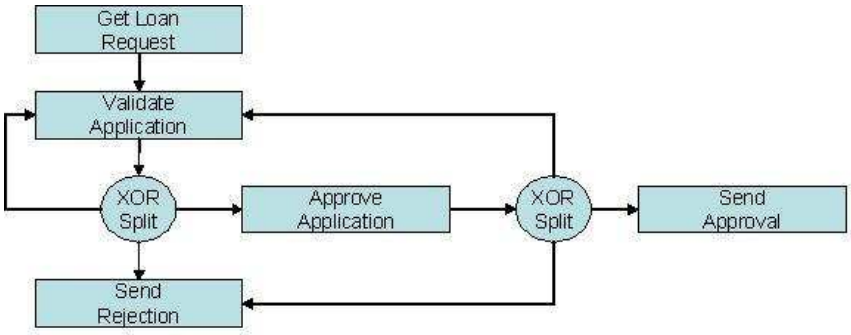

Figure 5.6: Arbitrary Cycles Pattern

Due to the modular rule-based principle of the language, it is easy to model the Arbitrary Cycles pattern in XChange, however, only using the *condition-action groupings* syntax.

```
ON
  event {{
    get-loan-req
  }}
DO
  event {{
    validate-appl
  }}
END
```

```
ON
  event {{
    validate-appl
  }}
DO and [
    action {{
      validate
    }},
    or [
      IF {{ appl-ok }}
      DO event {{
```

```
ON
  event {{
    approve-appl
  }}
DO and [
    action {{
      approve
    }},
    or [
      IF {{ approved }}
      DO event {{
```

[14] Standard and Product Evaluation, `http://workflowpatterns.com/`

```
        approve-appl                    send-approval
      }},                             }},

      IF {{ appl-failed }}            IF {{ failed }}
      DO event {{                     DO event {{
        send-rejection                  send-rejection
      }},                             }},

      IF {{ appl-recheck }}           IF {{ recheck }}
      DO event {{                     DO event {{
        validate-appl                   validate-appl
      }}                              }}
    ]                               ]
]                                 ]
END                               END
```

Conclusion *Proposal:* The Arbitrary Cycle pattern can be easily modeled in XChange due to the modular design of XChange rules using the *condition-action grouping* syntax.

Implicit Termination means that a given business process completes when there is nothing else to be done. In other words, there is no activity related to the process that is active or can be made active.

BPEL, for example, supports the Implicit Termination pattern by the `flow` construct. A structured activity (without `flow` and `links`) completes when its outermost activity completes and therefore corresponds to explicit termination. On the other hand, within the `flow` construct it is possible to implement multiple activities that are not a source of any link. In this case there is no need for one unique termination activity; the `flow` activity is complete when none of its sub-activity are active or can be made active. Some BPM systems do not support this pattern. Their typical solution is to transform the model to an equivalent model that has only one terminating node.

Discussion In the actual version of XChange this pattern cannot be realised as the language does not support the idea of a "process instance". Though XChange rules can be evaluated in an order determined by a business process and execute the process due to the designed specification, they can not be, however, combined (surely temporally) explicitly to one process.

Conclusion *Problem:* The implicit termination of a process is not provided in XChange since the language does not support the idea of a "process instance".

5.1.4 Multiple Instances Patterns

The following four patterns belong to the Multiple Instances group and describe how multiple instances of the same activity can run concurrently. A modeling language that provides a support for these patterns should have the ability to spawn multiple instances and, for three patterns, the ability to perform their synchronising merge. The four Multiple Instances (MI) patterns are: MI Without Synchronisation, MI With a Priory Design Time Knowledge, MI With a Priory Runtime Knowledge, and MI Without a Priory Runtime Knowledge.

MI Without Synchronisation allows to generate multiple instances of one activity without synchronising them afterwards. For example: *When booking a trip, the same activity handling a flight*

booking can be started multiple times depending on the number of involved flights. This scenario is represented in BPMN in Figure 5.7.

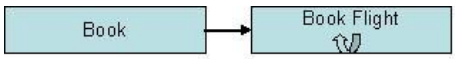

Figure 5.7: Multiple Instances Without Synchronisation Pattern

The modeling languages such as BPEL and BPML implement this patterns through the use of a loop construct:

```
<process name="book">
  <while cond="more-flights">
    <invoke name="book-flight"/>
  </while>
</process>
```

XChange does not provide a construct for this pattern. Though it is possible to fire multiple instances of the same event using the construct `all` related to the attribute `recipient`, this implementation is not conform with the pattern whose key idea is to start *the same* activity (i.e. having, among others, the same recipient) multiple times.

As a possible solution we propose to use the construct `all` in the event term related to some of the parameters (`flight` in the example):

```
DO
  event {{ book-flight { flight { all var Flight } } }}
ON
  event {{ book }}
FROM
  {{ var Flight }}
END
```

The idea behind this implementation is the following: as a response to the event message `book` the rule binds the variable `Flight` to the appropriate terms and depending on their number fires `book-flight` possibly multiple times. The actual airline booking is to be executed by the XChange rule, the one triggered by the event `book-flight`.

Conclusion The MI Without Synchronisation pattern is not directly supported in XChange. *Proposal:* Using the Xcerpt construct `all` related to some of the parameters in the event term is a possible XChange extension that would provide a support for the pattern.

MI With a Priori Design Time Knowledge is a pattern where multiple instances of the same activity are started and have to be synchronised before continuing with the remainder of the process. The number of instances that have to be started/synchronised is known at the design time. For instance: *The requisition of hazardous material requires three authorisations*. This example from [79] (page 21) is illustrated in Figure 5.8.

The XChange realisation of this pattern is rather simple. To implement the starting point, we place three `authorise` raising events in the action part of a rule triggered by `get-autorisation` event:

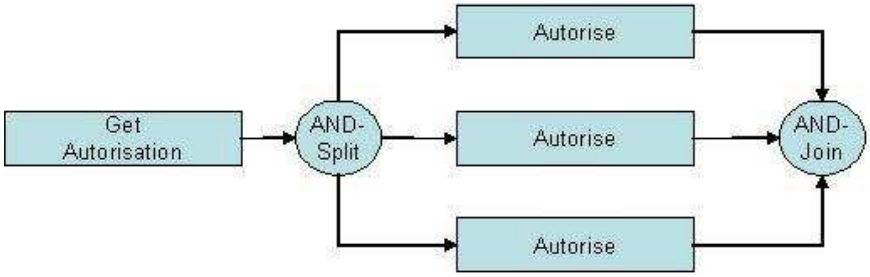

Figure 5.8: Multiple Instances With Design Time Knowledge Pattern

```
DO and {
  event {{ authorise }},
  event {{ authorise }},
  event {{ authorise }}
  }
ON
  event {{ get-authorisation }}
END
```

To assure the synchronisation we use additional event messages to confirm the completion of each `authorise` activity – `authorise-ready`:

```
DO and [
  action {{ authorise }},
  event {{ authorise-ready }}
  ]
ON
  event {{ authorise }}
END
```

In the rule specifying the next activity of the process, these messages are joined together by means of a conjunction event query:

```
DO
  <action-part>
ON and {
  event {{ authorise-ready }},
  event {{ authorise-ready }},
  event {{ authorise-ready }}
  }
END
```

Another approach to guarantee the synchronisation of the started activities is to use the quantification construct `times`. In both cases, the event query should be restricted to be legal.

Conclusion The starting point of the MI With Runtime Knowledge pattern is directly supported in XChange through the conjunction of multiple but equal event terms. *Workaround:* To guarantee the

synchronisation at the final point of the pattern, additional messages confirming the completeness of the started activities are necessary. *Problem:* Another handicap of XChange is the required temporal restriction of the composite event query used for the synchronisation – the time points of the life-span cannot be determined.

MI With a Priori Runtime Knowledge is a pattern where multiple instances of the same activity are generated and synchronised before the following activity can be performed. In contrast to the previous pattern, the number of instances to be started/synchronised is not known at the design time, but can be determined at the runtime (before the actual start of their execution). To adapt our example for the MI Without Synchronising (Figure 5.7), we state that *only after all bookings are made (which number is unknown at the design time) an invoice can be sent to the client*.

The implementation of this pattern is rather challenging. We can use neither `times` nor an *ordered conjunction* in the event part of the merging rule (the rule providing the synchronisation), as the number of expected events is unknown at the design time. To be able to merge an unknown number of parallel instances of the same activity, we need something like a counter increasing with each new started instance. The simplest way to simulate a counter in XChange is to use a temporary file where the number of started instances can be saved and then dynamically decreased each time when the execution of an instance is completed. The activity following the merging point of the pattern can be started when this number is null.

To implement our example, we modify an XChange rule implemented for the pattern MI Without Synchronisation: in response to the event `book` not only the `book-flight` events have to be raised but the update in a temporary file has to be made in addition:

```
DO and {
    event {{ book-flight { flight { all var Flight } } }},
    <!-- write a number of flights to book in a temporary file -->
    update {{ save-number-of-flights-to-book }}
  }
ON
  event {{ book }}
FROM
  {{ var Flight }}
END
```

In response to the evaluated event `book-flight`, both an activity for the actual flight booking and the update of the number of flights, that still have to be completed, are to be made:

```
DO and [
  action {{ booking }},
  <!-- update the number of flights to book -->
  update {{ decrease-number-of-flights-to-book }},

  <!-- all flight are booked -->
  FROM {{ flights-to-book = 0 }}
  action {{ send-invoice }}
ON
  event {{ book-flight }}
END
```

Conclusion The MI With a Priori Runtime Knowledge pattern is not supported in XChange. *Workaround:* A workaround solution is to use a temporary file for saving the status information of the actual process.

MI Without a Priori Runtime Knowledge generates multiple instances of the same activity, whereas the number of instances cannot be determined even at the runtime before the execution of instances is started. Even while some of the instances are being executed or already completed, new ones can be created. For example: *A number of eyewitness reports needed to complete an insurance claim varies and the stopping condition can be determined first after some reports have been already handled.*

For the XChange implementation of this pattern we use the approach proposed for the previous pattern (slightly extended). We use the same idea of simulating a counter with help of a temporary file. In addition, we have to add one more condition in the merging rule. It is important to wait till all the started instances (in our example for handling eyewitness reports) are finished in order to find out if this number of reports is enough to complete the claim. The graphical representation of this scenario is shown in Figure 5.9.

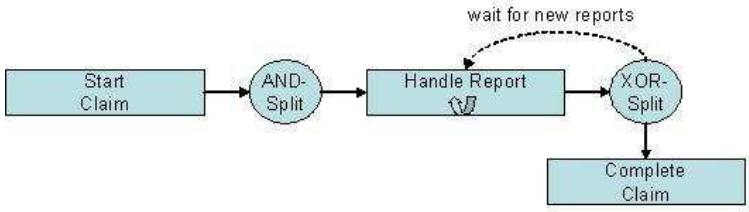

Figure 5.9: Multiple Instances Without a Priori Runtime Knowledge Pattern

First of all, we implement the starting point of the pattern in our example – start of the claim processing:

```
DO and {
    event {{ handle-report { all var Report } }},
    update {{ save-number-of-reports-to-handle }}
  }
ON
  event {{ start-claim }}
<!-- find out all the corresponding reports -->
FROM {{ var Report }}
END
```

In response to the event `handle-report` the actual report handling activity has to be taken, a number of reports to handle has to be decreased, and the condition – if the claim can be closed – checked (note that we use the syntax with condition-action groupings as only the action `complete-claim` is linked with the conditions `reports-to-handle = 0` and `enough-reports`):

```
ON
  event {{ handle-report }}
DO and [
```

```
    action {{ handle-report }},

    <!-- decrease the number of reports to handle -->
    update {{ decrease-number-of-reports-to-handle }},

    IF and {
      <!-- all started reports are finished -->
      {{ reports-to-handle = 0 }},

      <!-- enough information to complete the claim -->
      {{ enough-reports }}
    }
    action {{ complete-claim }}
  ]
END
```

If condition `enough-reports` fails, the claim will be not completed until a new report (or some new reports) for the given claim surfaces and provides enough information. Note that the rule that adds a new report for an active claim has to update the counter of the active instances in the temporary file.

Conclusion The MI Without a Priori Runtime Knowledge pattern is not supported in XChange. *Workaround:* A workaround solution is analogue to the implementation for the MI With a Priori Runtime Knowledge – to use a temporary file for saving the status information of the actual processes – with an additional condition check.

5.1.5 State-Based Patterns

The three patterns of the State-Based group can be also called "event-driven" because the behaviour of a business process described by these patterns is affected by events at the runtime. These three patterns are: Deferred Choice, Interleaved Parallel Routing, and Milestone.

Deferred Choice is a point within a process where one of several branches is chosen. The decision is not data-based as in the case of the Exclusive or Multi Choice patterns; the choice is rather delayed until the processing in one of the alternative branches is actually triggered. Once it happens, the other alternatives are disabled. For example: *After a new item is proposed to a customer of an online book shop, the decision if the item should be added to the purchase is deferred till the customer answers*.

As a reactive language, XChange directly supports such behaviour. We create a rule for each alternative branch following the deferred choice (in our example `add-item` or `complete-order`):

```
DO                              DO
  action {{ add-item }}           action {{ complete-order }}
ON                              ON
  event {{ accept-item }}         event {{ reject-item }}
END                             END
```

In BPEL this pattern is realised through the `pick` construct, whose semantics is awaiting the receipt of one of a number of messages and continuing the execution according to the received message. The sample code below illustrate the use of the construct for the above described situation.

```
<pick>
  <onMessage partnerLink="buyer"
             operation="accept-item">
    <!-- activity to add item to order -->
  </onMessage>
  <onMessage partnerLink="buyer"
             operation="reject-item">
    <!-- activity to perform order completion -->
  </onMessage>
</pick>
```

Conclusion The Deferred Choice pattern is directly supported by XChange. The pattern is conform to the *push* strategy which is XChange subjacent to.

Interleaved Parallel Routing is a mechanism for executing of a set of activities in an arbitrary order. The use of the word "parallel" in the name of this pattern is a bit deceptive, since the activities have to be performed actually sequentially, though in the order that is not known at design time. The pattern is useful in the situations where the activities share or update the same resources (otherwise, Parallel Routing can be used). For example: *At the end of each year, a bank executes two activities for each account:* `add interest` *and* `charge credit card costs` *. These activities can be executed in any order. However, since they both update the account, they cannot be executed at the same time*.

Such updates can be realised in XChange using *unordered conjunction* of updates:

```
DO and {
  update {{ add-interest }},
  update {{ charge-credit-card-costs }}
  }
ON <event>
END
```

As stated in [58] (Section 4.7), the effect of such a complex update specification is the effect of executing of all elementary updates (`add-interest` and `charge-credit-card-costs`) in some arbitrary execution order. "The order of their execution is not given and, thus, the runtime system has the freedom to choose the execution order." [58] (page 121)

The same approach can be used within a complex reaction part invoking one or many `PROCEDURE`s. However, we have to differentiate between a `PROCEDURE` whose action part consists of an (atomic or complex) update and a `PROCEDURE` which reacts with a raising event. In first case, having received *true* from a `PROCEDURE` means the update has been made (i.e., the expected activity is completed); in the second case, *true* as the return value would mean that an event message is sent, but not that the expected activity is completed. Thus, if a set of activities should be executed sequentially in an arbitrary order by *different actors*, XChange realisation would require additional message exchange (see the discussion for the Sequence pattern in Section 5.1.1).

Most BPM vendors lack support for the Interleaved Parallel Routing pattern. BPEL offers a solution which is probably a bit more complex for the end-user as the presented solution in XChange. In the following code sample, adapted from the pattern-based analysis of BPEL [79], the concept of serialisable scopes is used. A `scope` element is used for grouping activities within the same behavior context. When the `variableAccessSerializable` attribute is set to "yes", the scope provides concurrency control in governing access to shared variables. Thus, the activities grouped within the `flow`

element in the example below can potentially be executed in parallel. However, since the serialisable scopes that contain the activities share the same variable, they will be executed one after the other.

```
<flow>
  <scope name="add-interest"
    variableAccessSerializable="yes>
    <sequence>
      <!-- write to variable C -->
      <!-- activity to add interest >
      <!-- write to variable C -->
    </sequence>
  </scope>
  <scope name="charge-credit-card-costs"
    variableAccessSerializable="yes>
    <sequence>
      <!-- write to variable C -->
      <!-- activity to charge credit card costs >
      <!-- write to variable C -->
    </sequence>
  </scope>
</flow>
```

Conclusion The Interleaved Parallel Routing pattern is directly supported by XChange for data updates by means of an unordered conjunction in the event part of an XChange rule. *Problem:* Realisation of interleaved parallel activities that have to be executed by different actors is not supported in XChange (possible is either a parallel execution (more precisely, a parallel *start* of executions) or sequential with explicitly specified order).

Milestone pattern describes the situation in which an activity can be performed only when a certain milestone has been reached (enabling event fires) and cannot be performed after the milestone has been expired (disabling event fires). For example: *A customer can cancel his or her purchase order at any time before the delivery takes place* (see Figure 5.10). In this example, the event that triggers `make-purchase-order` activity is the enabling event, and the event that triggers `deliver` activity is the disabling event. A cancellation is only possible in the time span between these two events.

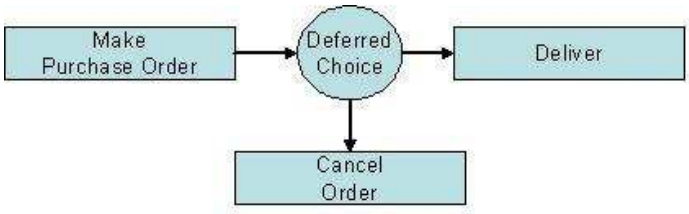

Figure 5.10: Milestone Pattern

In XChange we can implement such a scenario using the construct `without`:

```
DO
  <action-part>
```

```
ON
  without { event {{ delivery }}
    during andthen [
      event {{ make-purchase-order }},
      event {{ cancel }}
    ]
  }
END
```

The exclusion event query is evaluated when `cancel` event occurs after `make-purchase-order` event and only if `delivery` has failed to evaluate successfully on the stream of incoming events during the time interval between them.

Conclusion The Milestone pattern is directly supported by XChange through an exclusion event query introduced by the keyword `without`. *Problem:* For the implementation of this pattern a time restriction is, however, necessary.

5.1.6 Cancellation Patterns

There are two types of cancellation patterns: Cancel Activity and Cancel Case. The intent of this group of patterns is to describe a strategy to cancel a process at any point during its execution. For example, a multi-step insurance claim process has to be stopped as soon as the client revoked the claim.

Cancel Activity pattern describes how to cancel (disable) an enabled activity. For example: *If a customer cancels a request for information, the corresponding activity is disabled.*

Assume that the corresponding activity is a complex update, whereas each simple update takes about an hour. It is desirable to have a mechanism for interrupting such a long running activity if a cancellation signal comes. Unfortunately, XChange lacks such a mechanism. The hard way to solve this problem is to add cancellation checks (check for a specified cancellation input in a temporary file, `not-cancelled` in the example below) before each update. In this case, an already started update cannot be disabled, but the following ones will not be executed any more:

```
ON
    <event>
DO and [
    IF {{ not-cancelled }}
    update {{ updateA }},

    IF {{ not-cancelled }}
    update {{ updateB }},

    IF {{ not-cancelled }}
    update {{ updateC }}

    ...
  ]
END
```

Note that this implementation approach can be applied only using condition-action groupings of the earlier proposed syntax structure. Using the actual syntax of XChange, it is only possible to test the condition `not-cancelled` before the execution of the whole action part rather than before the execution of each atomic update.

Cancel Case is an extension of the Cancel Activity pattern. The intend of the pattern is to describe how an entire process can be stopped in response to a cancellation trigger.

In BPEL the Cancel Case pattern is solved with the `terminate` activity, which is used to abandon all executions within the process instance of which the `terminate` activity is a part. In the following code sample an event handler listening for a `cancel` message is defined; when the message arrives, the handler terminates the process with the `terminate` activity. As a sequence, all currently running activities will be abandoned.

```
<process>
  <eventHandlers>
    <onMessage partnerLink="..."
               operation="cancel"
               ...>
      <terminate/>
  </eventHandlers>
  <sequence>
    <!-- main flow -->
  </sequence>
</process>
```

The modular principle of XChange seems to be a drawback in such a scenario. Multiple XChange rules implementing multiple activities of a business process cannot be defined as being a part of the same process; they are rather independent from each other. Again, the hard way to solve the problem is to add to each rule, linked within a business process, a cancellation condition check.

Conclusion *Problem:* Both cancellation patterns are not supported in XChange since the language lacks the mechanism for integrating multiple XChange rules together within the same process.

The table below summarise the results of the described pattern-based analysis of the language XChange. If the language directly supports the pattern through one of its constructs, it is rated with +. Any workaround solution is rated +-. The same character is used to indicate any incomplete solution (for example, using not legal event queries). Any solution which results in a complex workaround solution, is considered as giving no direct support and is rated -. The differences in the realisation of patterns in terms of data updates versus event messages are indicated respectively with two characters separated with /. A pattern that is not supported by the actual version of XChange but can be realised with a construct proposed in this work is denoted by *. In order to compare XChange with the most widely-used modeling language BPEL, we have added its compliance with the workflow patterns as well.

As we see in the table above, XChange has built-in or easily attainable support for 6 (rated with + or *) of the 20 workflow patterns[15] (13 of 20 are supported by BPEL). Though being not the only

[15] We have presented 21 patterns. Discriminator can be considered, however, as a variant of the N-Out-of-M Join pattern.

Table 5.1: XChange, BPEL and Patterns

PATTERN	XCHANGE	BPEL
Sequence	+/+-	+
Parallel Split	-/+	+
Synchronisation	+-	+
Exclusive Choice	*	+
Simple Merge	+-	+
Multi-Choice	*	+
Sync Merge	+-	+
Multi Merge	+	-
Discriminator	*-	-
N-Out-of-M Join	*-	-
Arbitrary cycles	*	-
Implicit Termination	-	+
MI Without Synchronisation	*	+
MI With Design Time Knowledge	+-	+
MI With Runtime Knowledge	-	-
MI Without Runtime Knowledge	-	-
Deferred Choice	+	+
Interleaved Parallel Routing	+/-	+-
Milestone	+-	-
Cancel Activity	-	+
Cancel Case	-	+

important factor for a process modeling language, the support of the workflow patterns seems to be a good indicator of the language usability to implement workflow functionality. The main restrictions of XChange in this respect are:

- the missing possibility to integrate XChange rules together for the execution of a "process instance";
- the limits in the realisation of synchronous operations;
- the imperative to restrict composite event queries to a finite time interval.

As a result of the first fact, it is impossible to support the Cancellation patterns in XChange. This seems to be a great disadvantage, especially in view of the fact that six of seven modeling languages analysed by the authors of the workflow patterns[16] directly support both these patterns. The Sequence, Simple, and Synchronised Merge patterns can be implemented only with help of additional event messages, simulating "replies" in cases of request/reply communication. The example implementations of the Synchronisation, Simple Merge, Discriminator, N-Out-of-M Join, Synchronised Merge, MI With a Priori Design Time Knowledge, Milestone are realised with help of unbounded composite event queries.

5.2 XChange Implementation of Business Rules

Having discussed the expressiveness of XChange as a business process modeling language, we devote this section to the description of a possible implementation approach of business rules in XChange. As we mentioned in the previous chapter, business rules are a growing area of importance in BPMS as these rules provide governing behaviour to the BPMS. One of the essential points the business rules community pays attention at, is the option for writing business rules. Thus, one of the main goals of each BRMS is to provide a possibility to translate natural language rules into technical rules that can be executed by a business rule engine. This point is out of the scope of this work. We concentrate our attention on the possibility to implement business rules and business rulesets with XChange rules so that the result implementation could support the main functionality of a BPMS and a BRMS - effective execution of a business process correlated with the appropriate business rules.

We remind the main ideas in the realisation of the business rules approach (restricted to implementation issues):

- Business rules meant to be directly executed by an automated system are mostly represented as production (condition-action) rules.
- Logical connected rules are combined together into business rulesets.
- Business rules are separated from business processes.
- Inference engines - the most widely used technical realisation of the business rules approach - are invoked by explicit requests with an explicitly specified set of rules.

If we project these ideas onto the XChange implementation principles, we get the following implementation guidance:

[16] standard Evaluation, `http://is.tm.tue.nl/research/patterns/standards.htm`

- Business rules can be implemented with the `PROCEDURE` construct introduced in Section 5.1.1.
- Logical connected business rules can be specified together in the same XChange rule (to be used remotely) or in the same `PROCEDURE` construct.
- Business processes' activities have to be implemented separately from business rulesets.
- Business processes' activities must invoke business rulesets explicitly when they need a decision or result.

Let's have a look at a small example to illustrate these implementation principles. In the Section 4.2.3 we have analysed the way JRules, a BRMS solution of ILOG, works. We have presented two business rules: `GoldCategory` and `FreeSample`.

Rule: GoldCategory

```
IF      the purchase value is greater or equal to $100
THEN    change the customer category to "gold"
```

Rule: FreeSample

```
IF      the item is fish
THEN    add free food sample to shopping cart
```

Thus, each customer of a fish item obtains a free food sample. The `GoldCategory` rule says that each customer who spends more then $100 turns to a "gold" customer.

These rules can be used for example in a business process of an online food supplier. We simplify such a process and assume, that the supplier adds the requested products to the customer shopping cart in reply to his or her request, checks the business rules connected with a purchase process, and finishes the process in the next step.

Notation In the following code sample we use the same XChange-like syntax we used in the previous section to simplify the matter. Further on, we use the rules structure and constructs proposed in the previous section.

The first XChange rule reacts to a customer request by updating his or her shopping cart and invocation of the corresponding ruleset (`purchase-rules`). We illustrate two variants of this XChange rule: one that uses a remotely located ruleset and one that uses a locally located ruleset. In the first case, the ruleset should be implemented as an XChange rule and can be invoked by means of an event message:

```
ON
  event {{ purchase-request }}
DO and [
     update {{ add-products }},
     event {{ purchase-rules }}
  ]

END
```

Having been implemented as a PROCEDURE, the ruleset purchase-rules can be invoked locally without event messaging:

```
ON
  event {{ purchase-request }}
DO and [
     update {{ add-products }},
     purchase-rules,
     action {{ finish-purchase }}
   ]
END
```

Note that using a PROCEDURE we define a synchronous communication. The program can start the action finish-purchase directly after the PROCEDURE purchase-rules is finished, provided it was executed successfully. In case of raising the event purchase-rules, an additional XChange rule waiting for the purchase-rules-verified message has to be specified:

```
ON
  event {{ purchase-rules-verified }}
DO
  action {{ finish-purchase }}
END
```

Before we implement the ruleset purchase-rules, let's specify the related business rules:

```
PROCEDURE gold-category {{ }}
IF {{ purchaseValue >= $100 }}
DO update {{ customerCategory == "gold" }}
END

PROCEDURE free-sample {{ }}
IF {{ purchaseItem = "fish" }}
DO update {{ add-free-sample }}
END
```

As mentioned above, the ruleset can be implemented as an XChange rule:

```
ON
  event {{ purchase-rules }}
DO and [
      and {
         gold-category,
         free-sample
      },
      event {{ purchase-rules-verified }}
  ]
END
```

or as a PROCEDURE:

```
PROCEDURE purchase-rules {{ }}
DO and {
      gold-category,
      free-sample
  }
END
```

In the following chapter we use this approach of business rules realisation for the detailed implementation of business processes and business rules using the correct XChange syntax. As example we have chosen the well-known for BRM community EU-Rent use case which is often used by the BRMS vendors in their presentations.

CHAPTER

SIX

EU-Rent Case Study

6.1 Introduction

EU-Rent is a widely known case study being promoted by the business rules community as a basis for demonstration of business rules product capabilities. EU-Rent is a fictitious car rental company with branches in several countries which provides typical rental services. Though without an in-deep case analysis and specification, this use case provides a good opportunity to demonstrate the XChange solution for business processes and business rules implementation on a basis that is common and well-known for the business rules audience.

This chapter is organised to first introduce the complete specification of the use case and then to present the realisation of particular business processes and business rules of EU-Rent with the Web reactivity language XChange.

6.1.1 Overview

EU-Rent is a most widely known demonstration use case study for business rules products. It was originally developed by Model Systems Ltd[1] and has been used in numerous projects and publications devoted to the business rules domain, including the Business Rules Group's papers and submissions to the OMG for business rules standards. The business requirements are documented on the Web site of the European Business Rules Conference[2]. The more detailed specification from Business Rules Group[3] has been used in this chapter and is referenced as the "original case study".

6.1.2 EU-Rent Business

EU-Rent has 1000 branches in towns in several countries. At each branch cars, classified by car group, are available for rental. All cars in a group are charged at the same rate. Each branch has a manager.

Rentals Most rentals are by advance reservation; the rental period and the car group are specified at the time of reservation. EU-Rent can also accept immediate ("walk-in") rentals (these are, however, not regarded in our work as they are of no interest in terms of Web reactivity).

[1] one of the European Business Rules Conference sponsors, `http://www.eurobizrules.org/org.htm`

[2] EU-Rent Case Study, `http://www.eurobizrules.org/ebrc2005/eurentcs/eurent.htm`

[3] Case Study: EU-Rent Car Rentals, `http://www.businessrulesgroup.org/first_paper/br01ad.htm`

At the end of each day cars are assigned to reservations for the following day. If more cars have been requested than are available at a branch, the branch manager may ask other branches if they have cars they can transfer to him.

Returns Cars rented from one branch of EU-Rent may be returned to a different branch. The renting branch must ensure that the car has been returned to some branch at the end of the rental period. If a car is returned to a branch other than the one that rented it, ownership of the car is assigned to the new branch.

Servicing EU-Rent also has service depots, each serving several branches. Cars may be booked for maintenance at any time provided that the service depot has capacity on the day in question.

Customers A customer can have several reservations but only one car rented at a time. EU-Rent keeps records of customers, their rentals and bad experiences such as late return, problems with payment, and damage to cars. This information is used to decide whether to approve a rental. Customers may join the EU-Rent loyalty club and accumulate points that they can use to pay for rentals.

Pricing and Discounting A rental may cover multiple durations measured by units such as a month, a week, a day, and an hour. From time to time EU-Rent offers discounts and free upgrades. Only one of them (and always the best one) is used to calculate the rental price. At each customer touch point (e.g. reservation, reschedule, pick-up, return of the car) the pricing business rules are applied to determine whether the rental qualifies for a better discount.

6.1.3 EU-Rent Business Rules

In the following section we present EU-Rent business rules. We have restrict the amount of business rules specified in the original case study, having excluded those dealing with handover and walk-in rentals. For example, such a rule as *the car must not be handed over to a driver who appears to be under the influence of alcohol or drugs* can be checked only *in situ* in a EU-Rent branch. The automatisation of such types of rules would imply some visualisation in a textual form on a monitor; this implementation approach is, however, out of the scope of this work. Moreover, in order not to add unnecessary complexity, we do not regard business rules requiring additional information if the implementation of these rules would not demonstrate specific capabilities. For example, a rule *local tax must be collected on the rental charge* is similar to other pricing rules we list below and can be omitted as redundant in terms of business rules implementation. Changes or divergences of particular business rules are indicated individually.

Rental reservation acceptance

- Each driver authorised to drive the car during a rental must have a valid driver's licence.
- If a rental request does not specify a particular car group or model, the default is group A (the lowest-cost group).
- Reservations may be accepted only up to the capacity of the pick-up branch on the pick-up-day.
- If the customer requesting the rental has been blacklisted, the rental must be refused.
- A customer may have multiple future reservations, but may have only one car at a time.

Car allocation for advance reservations At the end of each working day, cars are allocated to rental requests due for pick-up the following working day. The basic rules are applied within a branch:

- Only cars that are physically present in EU-Rent branches may be assigned.
- If a specific model has been requested, a car of that model should be assigned if one is available. Otherwise, a car in the same group as the requested model should be assigned.
- If no specific model has been requested, any car in the requested group may be assigned.
- The end date of the rental must be before any scheduled booking of the assigned car for maintenance or transfer.
- If there are not sufficient cars in a group to meet demand, a one-group free upgrade may be given (i.e. a car of the next higher group may be assigned at the same rental rate) if there is capacity.
- Customers in the loyalty incentive scheme have priority for free upgrades.

If demand cannot be satisfied within a branch under the basic rules, one of the "exception" options may be selected:

- A "bumped upgrade" may be made. *(For example, if a group A car is needed and there is no capacity in group A or B, then a car allocated to a group B reservation may be replaced by a group C car, and the freed-up group B car allocated to the group A reservation).*
- A downgrade may be made (a "downgrade" is a car of a lower group).
- A car from another branch may be allocated, if there is a suitable car available and there is time to transfer it to the pick-up branch.
- A car due for return the next day may be allocated, if there will be time to prepare it before the scheduled pick-up time.
- A car scheduled for service may be used, provided that the rental would not take the mileage more than 10% over the normal mileage for service.

If demand cannot be satisfied within a branch under the "exception" rules, one of the "in extremis" options may be selected:

- Pick-up may have to be delayed until a car is returned and prepared.
- A car may have to be rented from a competitor.

No-shows

- If an assigned car has not been picked up 90 minutes after the scheduled pick-up time, it may be released for walk-in rental[4].

[4] we simplify the original rule as we do not differentiate between guaranteed and not guaranteed rules (see `http://www.businessrulesgroup.org/first_paper/br01ad.htm`)

Late returns

- A customer may request a rental extension[5] – the extension should be granted unless the car is scheduled for maintenance.
- If a car is not returned from rental by the end of the scheduled drop-off day and the customer has not arranged an extension, the customer should be contacted.
- If a car is three days overdue and the customer has not arranged an extension, the police must be informed[6].

Car maintenance and repairs

- Each car must be serviced every three months or 10.000 kilometres, whichever occurs first.
- If there is a shortage of cars for rental, routine maintenance may be delayed by up to 10% of the time or distance interval (whichever was the base for scheduling maintenance) to meet rental demand.
- Cars needing repairs must not be used for rentals[7].

Car purchase and sale

- Only cars on the authorised list can be purchased.
- Cars are to be sold when they reach one year old or 40.000 kilometres, whichever occurs first.

Car ownership

- A branch cannot refuse to accept a drop-off of a EU-Rent car, even if a one-way rental has not been authorised.
- When a car is dropped off at a branch other than the pick-up branch, the car's ownership (and, hence, responsibility for it) switches to the drop-off branch when the car is dropped off.
- When a transfer of a car is arranged between branches, the car's ownership switches to the "receiving" branch when the car is picked up.
- In each car group, if a branch accumulates cars to take it more than 10% over its quota, it must reduce the number back to within 10% of quota by transferring cars to other branches or selling some cars.
- In each car group, if a branch loses cars to take it more than 10% below its quota, it must increase the number back to within 10% of quota by transferring cars from other branches or buying some cars.

[5] we omit the original extension *by phone*
[6] the lapse of insurance cover is not regarded
[7] we do not differentiate the seriousness of needing repair as it is done in the original case study

Loyalty incentive scheme

- To join the loyalty incentive scheme, a customer must have made 4 rentals within a year.
- Each paid rental in the scheme (including the 4 qualifying rentals) earns points that may be used to buy "free rentals".
- Only the basic rental cost of a free rental can be bought with points. Extras, such as insurance, fuel, and taxes must be paid by cash or credit card.
- A free rental must be booked at least fourteen days before the pick-up date.
- Free rentals do not earn points.
- Free rentals attract no discounts.
- Unused points expire three years after the end of the year in which they were earned.

Pricing and discounting

- The lowest price offered for a rental must be honoured.
- EU-Rent's discounting business rules are presented in the following adapted decision table introduced on the EBRC[8]:

NAME	CAR GROUPS	DURATIONS	DISCOUNT	BUSINESS RULES
3-day Advance	All	All	10%	All rentals booked at least 3 days in advance qualify for a 10% discount.
Summer Week	C, D	week	$50.00	Weekly renters of a quality car receive a $50 discount.
New Loyalty Member	A, B	week, month	two car group upgrades	New loyalty club members are eligible for a two level upgrade.

6.1.4 Data Model and Proposed Clarifications

This subsection aims at more detailed specification of the use case and some clarifications. First of all, we describe more clearly the entities of the EU-Rent data model and their attributes while making some hypothesis. Secondly, we clarify the business logic.

In the case study, we can identify the following entities:

- **Person** of interest to EU-Rent can be either a customer or an additional driver. A customer is someone who has rented a car in the past, is currently renting a car, or holds a reservation for a car in the future. A customer is the one who signs the rental agreement and pays for it. An additional driver can (but must not) be an EU-Rent customer. Each additional driver must sign an "additional drivers authorisation" form. The Document Type Definition (DTD) of person's data is presented below:

[8]EU-Rent: Pricing and Discounting Business Rules,
http://www.eurobizrules.org/ebrc2005/eurentcs/eurentprice.htm

```
<!ELEMENT persons (person+)>
<!ELEMENT person (id, first-name, last-name, address, category, uri?,
  driver-licence, credit-card?, loyalty-club-member?, blacklisted?)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT first-name (#PCDATA)>
<!ELEMENT last-name (#PCDATA)>
<!ELEMENT address (street, city, country, postcode)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT postcode (#PCDATA)>
<!ELEMENT category ("customer"|"add-driver")>
<!ELEMENT uri (#PCDATA)>
<!ELEMENT driver-licence (number, date-of-issue, expiration-date)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT date-of-issue (#PCDATA)>
<!ELEMENT expiration-date (#PCDATA)>
<!ELEMENT credit-card (bank, type, number, expiration-date)>
<!ELEMENT bank (#PCDATA)>
<!ELEMENT type ("MASTER"|"VISA")>
<!ELEMENT loyalty-club-member (from, points)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT points (#PCDATA)>
<!ELEMENT blacklisted (from)>
```

- **Branch** represents an organisation unit within EU-Rent, responsible for a location where cars are kept, the cars that are kept there, and rental pick-ups and drop-offs. Additionally to the name and address of a branch, we specify a branch category to be able to determine the appropriate quota of car groups for each branch.

```
<!ELEMENT branches (branch+)>
<!ELEMENT branch (name, address, category, uri?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (street, city, country, postcode)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT postcode (#PCDATA)>
<!ELEMENT category ("large"|"medium"|"small")>
<!ELEMENT uri (#PCDATA)>
```

- **Car** entity represents a rental car that is owned by a EU-Rent branch and can be rented to customers. Each EU-Rent car has a registration number which provides a unique business identifier for a car; is classified by the car manufacturer to a category determined by the element `model`; is owned exactly by one of the EU-Rent branches; has a status that denotes if the car is available, assigned to a future rental, rented, or needs repair. We diverge in our specification

from the one given on the Web site of EBRC[9] where one of the further status values is stated as "needs maintenance". In our data model this status value is denoted as another child element – `maintenance`. The reason is the following business rules, specified by the Business Rules Group and stated above: *a car scheduled for service may be used, provided that the rental would not take the mileage more than 10% over the normal mileage for service* and *a customer may request a rental extension – the extension should be granted unless the car is scheduled for maintenance*. Thus, "needs maintenance" and "rented" are not exclusive statuses. The child elements of `maintenance` – `last-date` (date of the last service) and `last-km` (mileage at the last service) – are used as the indicators for the state of the car (if it is in need of service or not). The optional `new-date` is the date for the next car service.

```
<!ELEMENT cars (car+)>
<!ELEMENT car (registration-number, model, branch, car-status,
  maintenance)>
<!ELEMENT registration-number (#PCDATA)>
<!ELEMENT model (#PCDATA)>
<!ELEMENT branch (#PCDATA)>
<!ELEMENT car-status ("available"|"assigned"|"rented"|
  "scheduled-for-repair")>
<!ELEMENT maintenance (last-date, last-km, new-date?)>
<!ELEMENT last-date (#PCDATA)>
<!ELEMENT last-km (#PCDATA)>
<!ELEMENT new-date (#PCDATA)>
```

- **Rental Duration Category** is a category that specifies a rental duration for which rental prices are defined. A rental price is built up using units in these categories and their maximal and minimal durations. For example, a ten day rental is priced as 1*week+3*day.

```
<!ELEMENT durations (duration+)>
<!ELEMENT duration (name, unit, min, max)>
<!ELEMENT name ("month"|"week"|"day"|"hour")>
<!ELEMENT unit (#PCDATA)>
<!ELEMENT min (#PCDATA)>
<!ELEMENT max (#PCDATA)>
```

- **Car Group** is a category in a classification scheme that groups car models, based on common features. Car groups are arranged into a hierarchy for upgrades (in our specification provided through the element `previous-group`). A car group has a quota for each category of EU-Rent branches. This quota corresponds to the desirable number of cars of a concrete type. In addition, a price for each duration category of a car group has to be determined.

```
<!ELEMENT car-groups (car-group+)>
<!ELEMENT car-group (name, previous-group, quotas, prices)>
<!ELEMENT name ("A"|"B"|"C"|"D")>
<!ELEMENT previous-group ("A"|"B"|"C"|"D")>
<!ELEMENT quotas (large|medium|small)>
```

[9] EU-Rent Data Model, `http://www.eurobizrules.org/ebrc2005/eurentcs/eurentldm.htm`

```
<!ELEMENT large (#PCDATA)>
<!ELEMENT medium (#PCDATA)>
<!ELEMENT small (#PCDATA)>
<!ELEMENT prices (duration+)>
<!ELEMENT duration (name, price)>
<!ELEMENT name ("month"|"week"|"day"|"hour")>
<!ELEMENT price (#PCDATA)>
```

- **Car Model** is a name given by a car manufacturer to a category of cars which have the same features, e.g. engine size, fuel type, etc.

```
<!ELEMENT car-models (car-model+)>
<!ELEMENT car-model (name, manufacture, category, group,
  engine-size, fuel-type)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT manufacture (#PCDATA)>
<!ELEMENT category ("sedan"|"coupe"|"hatchback"|"convertible"|"van"|
  "wagon"|"SUV")>
<!ELEMENT group ("A"|"B"|"C"|"D")>
<!ELEMENT engine-size (#PCDATA)>
<!ELEMENT fuel-type ("gasoline"|"diesel")>
```

- **Rental** is a contract between a customer and EU-Rent to rent a car. A rental starts with a reservation request. It must specify a pick-up time, a day of drop-off, and a pick-up branch. Optionally, a reservation request can specify a desired car group and/or model, and a return branch (otherwise, the pick-up branch will be assigned). Thus, one of the possible status of a rental is a "request". When the specified data are checked and the rental is approved, it obtains a "reservation" status. Rentals may have other statuses: a rental is open if the customer has picked up a car; is closed after the rented car has been returned and the rental has been paid; is cancelled at the request of the customer, if the customer has not picked up the car, or if EU-Rent cannot provide a requested car for the specified period. Each rental has also a basic price (price before any discounts have been applied) and a lowest price. The payment type must be determined at the end of the rental. A note about a bad experience (e.g. late return, damage) is an optional element.

```
<!ELEMENT rentals (rental+)>
<!ELEMENT rental (customer, add-drivers?, car, time-period, location,
  rental-status, price, payment?, bad-experience-note?)>
<!ELEMENT customer (#PCDATA)>
<!ELEMENT add-drivers (add-driver+)>
<!ELEMENT add-driver (#PCDATA)>
<!ELEMENT car (group?, model?, assigned-car?)>
<!ELEMENT group ("A"|"B"|"C"|"D")>
<!ELEMENT model (#PCDATA)>
<!ELEMENT assigned-car (#PCDATA)>
<!ELEMENT time-period (from, till, actual-return-date?)>
<!ELEMENT from (#PCDATA)>
```

```
<!ELEMENT till (#PCDATA)>
<!ELEMENT actual-return-date (#PCDATA)>
<!ELEMENT location (pick-up-branch, return-branch?)>
<!ELEMENT pick-up-branch (#PCDATA)>
<!ELEMENT return-branch (#PCDATA)>
<!ELEMENT rental-status ("request"|"reservation"|"open"|"closed"|"cancelled")>
<!ELEMENT price (basic-price, lowest-price)>
<!ELEMENT basic-price (#PCDATA)>
<!ELEMENT lowest-price (#PCDATA)>
<!ELEMENT payment (type, date)>
<!ELEMENT type ("cash"|"credit-card"|"loyalty-club-points")>
<!ELEMENT date (#PCDATA)>
<!ELEMENT bad-experience-note (#PCDATA)>
```

The resulted EU-Rent data model is illustrated in Figure 6.1 as an UML diagram.

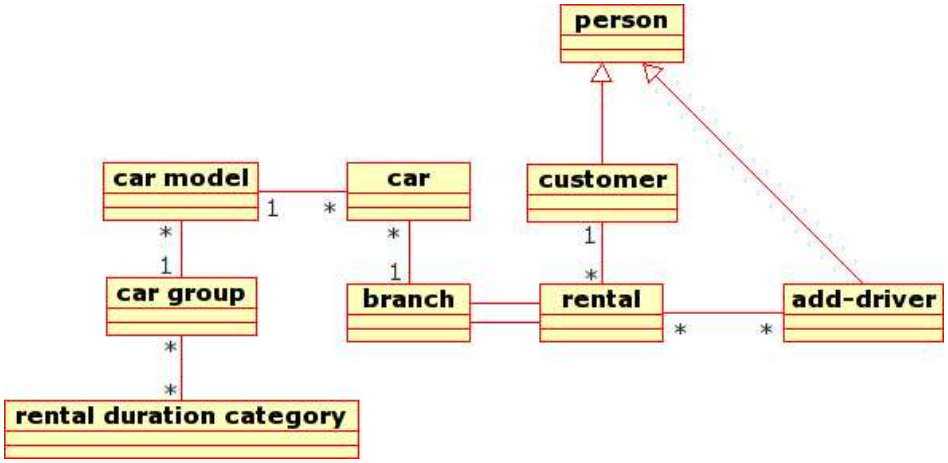

Figure 6.1: EU-Rent data model

Assumed extensions and clarifications In the following we specify some details to the business logic of the business processes, that will be implemented later.

Rental reservation acceptance First of all, some remarks on the following business rules:

- Each driver authorised to drive the car during a rental must have a valid driver's licence.

We interpret this rule as following: the date of expiration of the driver's licence has to be after the expected return date of the requested rental. The side effect of this interpretation is the fact that the rule has also to be checked if a customer requests a rental extension.

- Reservations may be accepted only up to the capacity of the pick-up branch on the pick-up date.

First of all, to get the capacity of the requested branch on the specified date, we add all currently available cars of the branch to those that are already assigned to rentals but are expected to be available on the pick-up date of the current reservation (i.e. the expected return date of the rentals, these cars are assigned to, is earlier as the pick-up date of the current reservation). Secondly, we assume that the change of a car group or model (in case the requested one is unavailable) cannot be taken without the

customer's agreement. Thus, the additional condition to the defined above is that only the cars of the requested group/model (default group A) are to be considered.

We assume that each reservation process comprises a price calculation though each rental is to be paid at the return date (as specified in the original case).

We assume that only registered EU-Rent customers can make a reservation. That means, for a new customer a registration process is necessary. The corresponding business rule is:

- *Only a registered customer can make a rental reservation*.

Car allocation A relevant fact that is not explicitly mentioned in the original case is the order in which car allocations are made, which is especially important when a reservation cannot be eventually accomplished. To follow the order of the corresponding business rules (see Section 6.1.3, *Car allocation for advance reservations*) seems to be logical though a bit deceptive. Thus, four of the business rules for the car allocation process are defined as following:

- *If a specific model has been requested, a car of that model should be assigned if one is available. Otherwise, a car in the same group as the requested model should be assigned.*

- *If there are not sufficient cars in a group to meet demand, a one-group free upgrade may be given (i.e. a car of the next higher group may be assigned at the same rental rate) if there is capacity*.

- *Customers in the loyalty incentive scheme have priority for free upgrades*.

If we follow the order of these rules, the free upgrade should be done after the assignments within a group have been made. This can lead to a situation when cars of the desired (or default) group/model will be assigned to some rentals of customers in the loyalty incentive scheme, whereas not-privileged customers get cars of a higher class group without extra costs. In order not to make the use case more complex, we preserve the same logic order of activities in our implementation. Thus, the schematic order of actions in our implementation is the following: the car allocation within a group; free upgrades; "bumped" upgrades; downgrades; cars transfer; assignment of cars that are expected to be returned the next day (*a car due for return the next day may be allocated, if there will be time to prepare it before the scheduled pick-up time*); assignment of cars that are scheduled for service (*a car scheduled for service may be used, provided that the rental would not take the mileage more than 10% over the normal mileage for service*); rentals delay; rentals from competitors; rentals cancellation.

Thereby, in case of a downgrade, the rental price will be calculated on the basis of the downgrade car group, instead of doing it with the desired car group as usual. This is done to prevent the customer from paying for more than what he is offered. In case of cars transfer (*a car from another branch may be allocated, if there is a suitable car available and there is time to transfer it to the pick-up branch*), we assume a car as "suitable" if it is available and belongs to the same group as the desired one. Then, we face a problem that is not mentioned in the original case: if many branches have such a suitable car, which one is to be chosen? Because of the transportation time constraint, it seems reasonable to use the distances (or expected time of transportation) between branches as a criteria for such a decision. Thus, the corresponding DTD obtains a new element – `nearest-branches`:

```
<!ELEMENT branches (branch+)>
<!ELEMENT branch (name, address, category, uri?, nearest-branches?)>
...
<!ELEMENT nearest-branches (transfer-branch+)>
<!ELEMENT transfer-branch (name, transfer-time)>
<!ELEMENT transfer-time (#PCDATA)>
```

Further on, neither the number of the nearest branches to contact nor the number of positive replies to be regarded is specified. We assume that all the nearest branches listed in the corresponding file are

to be involved in the transfer processes, that only three first transfer agreements are to be regarded, and that the waiting time for the replies is fixed to thirty minutes.

Late returns Some remarks on the following business rules:

- A customer may request a rental extension – the extension should be granted unless the car is scheduled for maintenance.

As we have already mentioned, in our specification the validation of the driver's licence is as well to be made before the rental extension will be granted.

- If a car is not returned from rental by the end of the scheduled drop-off day and the customer has not arranged an extension, the customer should be contacted.

It seems reasonable to start a special business process (or activity) at the end of each working day to guarantee that the rule holds. The same process can be used to execute (if necessary) the action of the following rule:

- If a car is three days overdue and the customer has not arranged and extension, insurance cover lapses and the police must be informed.

Car maintenance and repairs The original case study specifies the following business rule:

- Each car must be serviced every three months or 10.000 kilometres, whichever occurs first.

The necessary checking can be taken for example after each car rental. However, given that a car has not been rented for a period of time, the rule can be hurt. That's why it is reasonable to prove the state of cars in the process of the car allocation, namely before the actual assignment.

The following rule states one more condition in the process of the car allocation:

- Cars needing repairs must not be used for rentals.

Car purchase and sale The appropriate business steps are not described in the original case study. We assume that the car purchase and sale to be the duty of the central management and fulfilled in response to the requests of branch managements to satisfy the following business rules:

- In each car group, if a branch accumulates cars to take it more than 10% over its quota, it must reduce the number back to within 10% of quota by transferring cars to other branches or selling some cars.

- In each car group, if a branch loses cars to take it more than 10% below its quota, it must increase the number back to within 10% of quota by transferring cars to other branches or buying some cars.

Loyalty incentive scheme First of all, it is not clear from the case whether a customer automatically becomes a member of the loyalty incentive scheme when the criteria (4 rentals within a year) is achieved or some actions should be taken by a customer. We suggest, it happens automatically, as the membership does not oblige a customer to anything. It is natural to think that if a bad experience is recorded, points or accumulated rentals are lost as well as the membership in the loyalty incentive club (and all his or her reservations are cancelled). Furthermore, it is not fixed how points are assigned. We assume that points are given upon the cost of rental at the return day. However, the proposal to pay with points (if possible) has to be made during the reservation process.

Data management Different branches share data (information about customers, cars, rentals, etc.). As most of the actions to be managed in EU-Rent are naturally assigned to a branch (reservation, assignments, etc.), we assume that the related operations are to be taken locally.

6.2 Modeling of EU-Rent Business Processes with XChange

In this section we implement particular business processes of EU-Rent along with the related business rules. We use the earlier proposed constructs with the appropriate syntax and semantics and make further on the following assumptions:

- variables can be bound to parameters of event messages. This can be useful, for example, in situations where a variable bound to the element `xchange:sender` in the event query is used as the argument `xchange:recipient` in the action part. Or a variable bound to the event message parameter `xchange:reception-time` is used as a time point within the construct `during FiniteTimeInterval`.
- `where` clause can be applied not only on variables bound on Web data but also on those that are bound on parameters of event messages. This would provide more modularity and flexibility in the support of the business rules approach. As mentioned in Section 5.2 we separate the implementations of business processes and business rules. The eligibility of the business rules execution has to be determined dynamically at runtime, based on the actual context. Since the context depends not only on the present data state but on the actual business process as well, we have to be able to forward the information about the actual business process from one XChange rule to another. Using an extra event data element, that can be queried with the clause `where`, is the most intuitive way. In the following implementation we use the variable `Process` for this purpose.
- as the time related expressions and combinations (for example, `today` value, `var Time + 90 min`) are not supported in the actual version of XChange (though intended to be implemented, see [27], page 113), we use a syntax which is easy to comprehend.

In this section we present XChange code samples for implementing EU-Rent processes, the whole implementation is given in Appendix A.

6.2.1 Reservation Management

We start with the implementation of the reservation process. First of all, the appropriate business rules. These can be realised (adequate to the mostly used representation form of business rules – production rules) using the construct `PROCEDURE`, proposed in the previous section (cf. Section 5.1.1). To remain, a `PROCEDURE` groups a condition and an action together under one name and can be invoked from an action part of any XChange rule or another `PROCEDURE`. In the following we present some code fragments. Other related business rules denoted in the previous section together with the views we use within condition parts of the examples below (cf. `registered-customers`, `blacklisted`, etc. in the samples below) are given in Appendix A. The following XChange code fractions are the implementation of particular business rules dealing with the reservation management. Note that the textual formulations given in the previous section are transformed into `IF-(THEN)` statements to indicate their one-to-one relationship to the given XChange codes.

- *If a rental request does not specify a particular car group or model, the default is group A.*

```
PROCEDURE default-car-group [
      customer {{ id { var Id } }},
      var Period
```

```
   ]
IF reservation-requests {{
      rental {{
         customer { var Id },
         var Period,
         car {{ without group {{ }} }}
      }}
   }}
DO in {resource {"file:rentals.xml"},
      rentals {{
         rental {{
            customer { var Id },
            var Period,
            rental-status {"request"},
            car {{ insert group {"A"} }}
         }}
      }}
   }
END
```

- *If the customer requesting the rental is not registered, the rental must be refused.*

```
PROCEDURE refuse-unregistered [
      var Customer,
      var Uri,
      process { "reservation" }
   ]
IF registered-customers {{
      without var Customer
   }}
DO xchange:event {
      xchange:recipient { var Uri },
      eu-rent-reply {
         status {"Failure"},
         message {"Registration necessary."}
      }
   }
END
```

- *If the customer requesting the rental has been blacklisted, the rental must be refused.*

```
PROCEDURE refuse-blacklisted [
      var Customer,
      var Uri,
      process { "reservation" }
   ]
IF blacklisted {{
```

```
      var Customer
   }}
DO xchange:event {
      xchange:recipient { var Uri },
      eu-rent-reply {
         status {"Failure"},
         message {"You are blacklisted."}
      }
   }
END
```

- *If the customer requesting the rental has invalid driver's licence, the rental must be refused.*

```
PROCEDURE refuse-invalid-licence [
      var Customer,
      time-period {{ till { var Till } }},
      var Uri
   ]
IF invalid-driver-licences {{
      driver-licence {
         var Customer,
         invalid-from { var Date }
      }
   }} where { var Date < var Till }
DO xchange:event {
      xchange:recipient { var Uri },
      eu-rent-reply {
         status {"Failure"},
         message {"Invalid driver licence."}
      }
   }
END
```

The logically connected business rules can be combined together into a ruleset. Hence, the specified rules (together with some related ones from Appendix A) can build two rulesets: `reservation business rules` and `refusal business rules`.

If we want to separate business rules from business processes as it is desired in the business rules approach, we should realise a ruleset as a separate structure that can be explicitly invoked by business processes when they need a decision or result. In a distributed business environment a ruleset can be implemented as a (Web) service, located remotely to the implementations of processes themselves. The appropriate implementation approach in XChange would be to realise a ruleset within a separate XChange rule. For a local realisation of a ruleset we can use the construct `PROCEDURE` again, this time linking all logically connected business rules (each realised as a `PROCEDURE`) together. Given, that the same ruleset can be invoked by different business processes, this solution offers the possibility to avoid redundant implementation of business rules. For example, refusal rules (or at least, a part of them) have to be verified at multiple customer touch points: during the rental reservation, rental reschedule, or rental extension process. Specifying the related business rules separately saves implementation

time and makes programs less error-prone. However, within some business processes rather a part of a ruleset has to be considered. To determine the business process (if it is of importance) within a ruleset, we use the variable `Process`.

The next code example we present is an XChange implementation of the refusal ruleset. It is realised with `PROCEDURE` construct and can be invoked from any business process. Note that in case of a remotely located ruleset, instead of the `PROCEDURE` we would use an XChange rule expecting an event message named `refusal-rules` and specifying an event term, for example, `refusal-rules-verified` to inform the client about the rules checking completeness.

```
PROCEDURE refusal-rules [
      var Customer,
      uri { var Uri },
      var Period,
      var Process -> process {{ }}
   ]
DO or {
      refuse-unregistered { var Customer, var Uri, var Process },
      refuse-blacklisted { var Customer, var Uri, var Process },
      refuse-invalid-credit-card { var Customer, var Uri, var Process },
      refuse-invalid-licence { var Customer, var Period, var Uri },
      refuse-overlap { var Customer, var Period, var Uri },
      refuse-unavailable-group { var Customer, var Period, var Uri,
                                 var Process }
      refuse-maintenance { var Customer, var Period, var Uri, var Process }
   }
END
```

Note that the refusal business rules are exclusive (the reservation is refused if either the customer is not registered, or blacklisted, or has an invalid driver's licence, etc.). This situation corresponds to the Exclusive Choice pattern we have discussed in the previous section. To remain, in case of the Exclusive Choice only one alternative branch (action part) can be chosen. The syntax and informal semantics of our implementation is the same we have proposed for the Exclusive Choice pattern. The set of rules has no sequencing – the runtime system has the freedom to choose the order on which it tries to find and execute successfully the action of one of the `PROCEDURE`s. Thus, we use an unordered disjunction, denoted with the keyword `or` and curly braces {}. The reaction of the XChange rule is the first action part of a `PROCEDURE` which condition part holds true. If no `PROCEDURE` listed within the complex reaction part of the `PROCEDURE rental-rules` can be evaluated successfully, the return value of the `PROCEDURE` would be *false*. To denote at the business process scope that none of the refusal rules have been invoked and the reservation process can be continued, the *procedure negation* can be used, denoted by the keyword `not` (see the last sample code of this section).

Another ruleset we specify is the reservation ruleset. It is a set of two business rules that assign (if necessary) default values for the car group and the return branch. We use an unordered conjunction of the corresponding `PROCEDURE`s to guarantee that the runtime system would try to execute both of them. However, even if both updates specified by these `PROCEDURE`s fail, the reservation process can be continued, as the car group and the return branch might be already assigned. Thus, we have to add a data query that is always successfully evaluated (cf. `true { }` query in Section 5.1.1, Exclusive Choice Pattern) to guarantee the positive return value from the `PROCEDURE`:

```
PROCEDURE reservation-rules [
      var Customer,
      var Period
   ]
DO or [
      and {
         default-car-group { var Customer, var Period },
         default-return-branch { var Customer, var Period }
      },
      true { }
   ]
END
```

The PROCEDURE will always return *true*, provided the XChange program contains the Xcerpt rule:

```
CONSTRUCT
   true { }
END
```

The reservation process is to start in response to the event message reservation-request. The whole process consists of the following steps: save the received information, check reservation rules, check refusal rules, check pricing rules, and answer the customer if the requested reservation has been granted. The subprocesses of checking rules cannot be realised parallel, as it is possible that the car group is settled only in the processing of the reservation rules checking. As we have already discussed, the sequential process requires additional message exchange in XChange (see Section 5.1.1, Sequence). Thus, without the proposed construct PROCEDURE we would need different XChange rules to implement the sequential activities of the reservation processes. Using this construct, however, allows us to combine all of them within the action part of the same XChange rule realising the whole rental reservation process:

```
ON xchange:event {{
      xchange:sender { var Uri },
      reservation-request [
         var Customer -> customer {{ id { var Id } }},
         var Car,
         var Period,
         var Location
      ]
   }}
DO and [
      <!-- add reservation request -->
      in {resource {"file:rentals.xml"},
         rentals {{
            insert rental [
               customer { var Id },
               var Car, var Period, var Location,
               price [ basic-price { }, lowest-price { } ],
               rental-status {"request"}
```

```
            ]
         }}
      },
      or [
         <!-- reservation granted -->
         and [
            reservation-rules { var Customer, var Period },
            not refusal-rules { var Customer, var Uri, var Period,
                                process "reservation" },
            pricing-rules { var Customer, var Period },
            in {resource {"file:rentals.xml"},
               desc rental {{
                  customer { var Id },
                  var Period,
                  rental-status {"request" replaceby "reservation"}
               }}
            },
            xchange:event{
               xchange:recipient { var Uri },
               reply {
                  status {"Finished"},
                  message {"Reservation successful."}
               }
            }
         ],

         <!-- reservation refused -->
         in {resource {"file:rentals.xml"},
            rentals {{
               delete rental {{
                  customer { var Id },
                  var Period,
                  rental-status {"request"}
               }}
            }}
         }
      ]
   ]
END
```

6.2.2 Pricing and Discounting

The pricing and discounting business rules have to be checked again on each customer touch point, such as the rental reservation, extension, or reschedule. In the following we implement the ruleset which combines all pricing and discounting business rules. The realisation approach is the same as in the case of the refusal business rules: each business rule is implemented as a `PROCEDURE`; all pricing and discounting business rules are combined together within (i.e., are invoked from) another

PROCEDURE that represents a pricing and discounting business ruleset. In this section we implement the following business requirements:

- *The basic price of each rental depends on the rental duration and the requested car group*[10].

```
PROCEDURE basic-price [
      var Customer,
      var Period
   ]
IF and {
      in {resource {file:rentals.xml},
         var Customer,
         var Period,
         price {{ basic-price { var Price } }},
         car {{ group { var Group } }}
      },
      in {resource {file:car-group.xml},
         desc car-group {{
            name { var Group },
            prices {{
               duration {
                  name { "day" },
                  price { var DPrice }
               }
            }}
         }}
      }
   }
DO in {resource {"file:rentals.xml"},
      rentals {{
         rental {{
            var Customer,
            var Period,
            price {{
               basic-price { var Price replaceby
                             var DPrice * (var Till - var From) }
            }}
         }}
      }}
   }
END
```

- *For each reservation made in 3-days advance, EU-Rent offers a 10% discount.*

```
PROCEDURE ten-percent-discount {
      customer {{ id { var Id } }},
```

[10] We count the basic price based on the duration day.

```
      var Period -> time-period {{ from { var From } }}
   }
IF in {resource {"file:rentals.xml"},
      desc rental {{
         var Id,
         var Period,
         price [
            basic-price { var BPrice },
            lowest-price { var LPrice }
         ]
      }}
   } where { ( today+3 days <= var From ) &&
             ( LPrice > var BPrice / 10 ) }
DO in {resource {"file:rentals.xml"},
      desc rental {{
         var Id,
         var Period,
         price {{
            lowest-price {
               var LPrice replaceby var BPrice / 10
            }
         }}
      }}
   }
END
```

- *For each weekly reservation of a car from a group C or D, a $50 discount is to be offered.*

```
PROCEDURE fifty-discount [
      customer {{ id { var Id } }},
      var Period -> time-period {{
         from { var From },
         till { var Till }
      }}
   ]
IF in {resource {"file:rentals.xml"},
      desc rental {{
         var Id,
         var Period,
         car {{ group { var Group } }},
         price [
            basic-price { var BPrice },
            lowest-price { var LPrice }
         ]
      }}
   } where { ((var Till - var From) >= 7 days ) &&
             ( LPrice > var BPrice - 50 ) &&
```

```
                ( var Group = "C" || var Group = "D" ) }
    DO in {resource {"file:rentals.xml"},
          desc rental {{
             var Id,
             var Period,
             price {{
                lowest-price {
                   var LPrice replaceby var BPrice - 50
                }
             }}
          }}
       }
    END
```

These business rules are not exclusive as in the case of the rental business rules. They are combined together in an XChange `PROCEDURE` following our implementation proposals for the Sequence (count the basic price at first, then involve discounting rules) and Interleaved Parallel Routing (the order of the discounting rules checking is irrelevant). Since the successful evaluation of all pricing business rules is not essential for processes invoking this ruleset (thus, for some reservations none of the offered discounts can be applied), we need the ordered disjunction of business rules that is always successfully evaluated. We use the data query `true { }` as we have done in the ruleset `reservation-rules`.

```
PROCEDURE pricing-rules [
      var Customer,
      var Period
   ]
DO or [
      and [
         basic-price { var Customer, var Period },
         and {
            ten-percent-discount { var Customer, var Period },
            fifty-discount { var Customer, var Period }
         }
      ],
      true { }
   ]
END

CONSTRUCT
   true { }
END
```

As we see, the basic price of a rental is calculated at the beginning. All the following business rules are checked sequentially, but in the order that is not known at design time.

6.2.3 Car Allocation

As we have appointed in Section 6.1.4 the order of actions in our specification of the car allocation process is the following: the car allocation within a group; free upgrades; "bumped" upgrades; downgrades; cars transfer; assignment of cars that are expected to be returned the next day; assignment of cars that are scheduled for service; rentals delay; rentals from competitors; rentals cancellation. As in the previous cases, we do not demonstrate the whole implementation solution. We concentrate rather on particular parts of the process delivering subject matters for different implementation solutions. Among other car allocation business rules we implement the following:

- *If a specific model has been requested, a car of that model should be assigned if one is available.*

```
PROCEDURE assign-with-model {{ }}
IF and {
      rentals-for-allocation-with-model {{
         desc rental {{
            var Customer -> customer {{ }},
            var Period -> time-period {{ }},
            car {{ model { var Model } }}
         }}
      }},
      in {resource {"file:cars.xml"},
         desc car {{
            registration-number { var Nr },
            model { var Model },
            car-status { "available" }
         }}
      }
   }
DO and {
      in {resource {"file:rentals.xml"},
         desc rental {{
            var Customer,
            var Period,
            car {{ insert assigned-car { var Nr } }}
         }}
      },
      in {resource {"file:cars.xml"},
         desc car {{
            registration-number { var Nr },
            car-status { "available" replaceby "assigned" }
         }}
      }
   }
END
```

- *If there are not sufficient cars in a group to meet demand, a one-group free upgrade may be given if there is capacity.*

```
PROCEDURE free-upgrade {{ }}
IF and {
      rentals-for-allocation {{
         desc rental {{
            var Customer -> customer {{ }},
            var Period -> time-period {{ }},
            car {{ group { var Group } }}
         }}
      }},
      in {resource {"file:car-groups.xml"},
         desc car-group {{
            name { var Group },
            previous-group { var PrGroup }
         }}
      },
      in {resource {"file:car-groups.xml"},
         desc car-group {{
            name { var PrGroup },
            model { var Model }
         }}
      },
      in {resource {"file:cars.xml"},
         desc car {{
            registration-number { var Nr },
            model { var Model },
            car-status { "available" }
         }}
      }
   }
DO and {
      in {resource {"file:rentals.xml"},
         desc rental {{
            var Customer,
            var Period,
            car {{ insert assigned-car { var Nr } }}
         }},
         in {resource {"file:cars.xml"},
            desc car {{
               registration-number { var Nr },
               car-status { "available" replaceby "assigned" }
            }}
         }
      }
   }
END
```

All car allocation business rules are to be checked in the specified order. Thus, to combine all of them in a ruleset, we use the implementation solution proposed for the Sequence Pattern for data updates (as all but the last PROCEDUREs invoked in the XChange rule react with an update):

```
ON
   xchange:event {{
      allocation-rules {{ }}
   }}
DO and [
      assign-with-model {{ }},
      assign-within-group {{ }},
      free-upgrade {{ }},
      bumped-upgrade {{ }},
      downgrade {{ }},
      transfer {{ }}
   ]
END
```

In the specified XChange rules all the PROCEDUREs are invoked in the given order. Their action parts are executed, provided the corresponding conditions hold. The last PROCEDURE implements the business rule *a car from another branch may be allocated, if there is a suitable car available and there is time to transfer it to the pick-up branch*. Thus, after the other possibilities to assign cars to reservations (free upgrade, downgrade, etc.) are exhausted, the cars transfer from other branches will be initiated. The reaction part of this business rule is the event message transferRequest sent to all nearest branches:

```
PROCEDURE transfer {{ }}
IF and {
      rentals-for-allocation {{
         var Rental -> desc rental {{ }}
      }},
      branches-for-transfer {{
         transfer-branch {{ uri { var Uri } }}
      }}
   }
DO all xchange:event {
      xchange:recipient { var Uri },
         transferRequest {
            all var Rental
         }
   }
END
```

The following operations (actual transfer) can be started only after the transferRequests have been answered. Thus, they must be realised in another XChange rule (as we have discussed, XChange does not support synchronous communication directly). Thus, in response to the carTransferAgreements received from requested branches, the proposed agreement is to be written down:

```
ON
   xchange:event {{
      xchange:sender { var Uri },
      carTransferAgreement {
         var Rental -> rental {{ }},
         var Time -> delivery-time {{ }}
      }
   }}
DO in {resource {"file:transfers.xml"},
      agreements {{
         insert agreement {
            from { var Uri },
            var Rental,
            var Time
         }
      }}
   }
END
```

However, after three such agreements have arrived, the transfer process can be started and all the other agreements can be ignored. This is a situation that corresponds to the N-Out-Of-M Join (see Section 5.1.2). For its realisation we deploy the construct introduced by the keyword `first` and presented in the above referenced section. On the other hand, after a time period of thirty minutes, the transfer has to be started even if less than three agreements have occurred (but at least one). Through the ordered conjunction of the event messages `transferRequest` and `carTransferAgreement` we realise the Synchronisation pattern in the second part of the complex event query:

```
ON or {
      andthen [
         xchange:event {{
            transferRequest { var Rental }
         }},
         first 3 of {
            xchange:event {{
               carTransferAgreement {{ var Rental }}
            }}
         }
      ],

      andthen [
         xchange:event {{
            xchange:raising-time { var RTime },
            transferRequest { var Rental }
         }},
         xchange:event {{
            carTransferAgreement {{ var Transfer }}
         }},
```

```
      ] during [ var RTime, var RTime + 30 minute ]
   }
IF and {
      in {resource {"file:transfers.xml"},
         desc agreement {{
            var Rental,
            from { var Uri },
            delivery-time { var BestTime }
         }}
      },
      in {resource {"file:transfers.xml"},
         desc agreement {{
            var Rental,
            from { var AnotherUri },
            delivery-time { var AnotherTime }
         }}
      }
   } where { var BestTime < var AnotherTime }
DO and {
      xchange:event {
         xchange:recipient { var Uri },
         start-transfer { var Rental }
      },
      all xchange:event {
         xchange:recipient { var AnotherUri },
         cancel-transfer { var Rental }
      }
   }
END
```

Thirty minutes after sending of `transferRequest`s without any reply, the next allocation activity has to be taken – rentals' delay:

```
ON andthen [
      xchange:event {{
         xchange:raising-time { var RTime },
         transferRequest { var Rental }
      }},
      without {
         xchange:event {{
            carTransferAgreement {{ var Rental }}
         }}
      }
   ] during [ var RTime, var RTime + 30 minute ]
DO rentals-delay {{ }}
END
```

In the above implemented XChange rule a variant of the Synchronisation pattern is used. This case – a synchronisation of a message event with an event that should *not* occur – is not described among the workflow patterns. The same *Synchronisation with negation* can be used in the implementation of the reservation monitor process. As stated in Section 6.1.3: *If an assigned car has not been picked up 90 minutes after the scheduled pick-up time, it may be released for walk-in rental*. This monitoring process can be implemented with the following XChange rule:

```
ON andthen [
      xchange:event {{
         xchange:reception-time { var Reception },
         start-reservation-monitor {
            var Rental -> rental {{
               customer {{ var Id }},
               var Period -> time-period {{ }},
            }}
         }
      }},
      without {
         xchange:event {{
            stop-reservation-monitor { var Rental }
         }}
      } during [ var Reception, var Reception+90 minutes ]
   ]
DO and {
      <!-- cancel reservation -->
      in {resource {"file:rentals.xml"},
         desc rental {{
            customer { var Id },
            var Period,
            car {{ assigned-car { var Nr } }},
            rental-status {"reservation" replaceby "cancelled"}
         }}
      },

      <!-- update car status -->
      in {resource {"file:cars.xml"},
         desc car {{
            registration-number { var Nr },
            car-status {"assigned" replaceby "available"}
         }}
      },

      <!-- message to the customer -->
      IF in {resource {"file:persons.xml"},
            desc person {{
               id { var Id },
               uri { var Uri }
```

```
            }}
         }
      DO xchange:event {
            xchange:recipient { var Uri },
            rental-info {
               message "Your reservation has been cancelled."
            }
         }
   }
END
```

Note that the event message `start-reservation-monitor` can be sent after a reservation is made, the event `stop-reservation-monitor` after a rental cancellation or a pick-up of the reserved car.

After the `transferRequest` is sent, there are two possible processing variants: a car transfer or a rental delay. As the decision is not data-based but depends upon the event message that trigger one or another alternative branches, it is the case of the Deferred Choice pattern.

The rental delay can be implemented alike the rental transfer. First of all, all customers which reservations couldn't be successfully handled, receive a delay request:

```
PROCEDURE rentals-delay {{ }}
IF rentals-for-assignment {{
      var Rental -> desc rental {{
         customer { var Id },
      }},
      in {resource {"file:persons.xml"},
         desc customer {
            id { var Id },
            uri { var Uri }
         }
      }
   }}
DO all xchange:event {
      xchange:recipient { var Uri },
      rentalDelayRequest {
         var Rental
      }
   }
END
```

The actual delay can be accomplished only after the customer's affirmation:

```
ON andthen [
      xchange:event {
         xchange:recipient { var Customer },
         rentalDelayRequest {{ }}
      },
```

```
      xchange:event {
         xchange:sender { var Customer },
         rentalDelayAgreement {{
            rental {{
               var Customer -> customer {{ }},
               var From -> from {{ }},
               till {{ var Till }}
            }},
            date { var Date }
         }}
      }
   ]
DO in {resource {"file:rentals.xml"},
      desc rental {{
         var Customer,
         time-period {{
            var From,
            till { var Till replaceby var Date }
         }}
      }}
   }
END
```

The next operation of the allocation process would be to rental some cars from the competitors. We do not represent the XChange implementation of this part, as it requires the same implementation approach as the car transfer subprocess.

Conclusion In this section we have implemented the EU-Rent business processes and business rules by using the Web reactive language XChange. We have restricted the business domain, as the implementation approach of many business logic parts is either simular or is the same. We have seen that the implementation of business rules and business rulesets with the `PROCEDURE` construct is a modular, flexible solution preventing redundancy and error-proneness, since each business rule and ruleset are implemented separately. We have used the implementation solution proposed for the following design patterns: Sequence, Synchronisation, N-Out-Of-M Join, Parallel Split, Exclusive Choice, Interleaved Parallel Routing, Deferred Choice. We have found the application example for a Synchronisation pattern, where the temporal occurrences order plays an important role, and for a Synchronisation with negation. We have not found the application examples for some patterns like Cancel Activity, Cancel Case, MI Without Runtime Knowledge, or Arbitrary Cycles. This should not mean, however, that the mentioned patterns do not have their range of use.

Part IV

Conclusion

CHAPTER
SEVEN

Conclusion

7.1 Summary

The research topic investigated by this thesis is the *business process modeling using ECA rules* by means of the reactive language XChange. First of all, we have analysed currently used Business Process Management Systems and Business Rules Management Systems, including their fundamental principles, supporting commercial products, and used standards (Chapters 3 and 4). The objective of this part of the research work was to specify the requirements, challenges, and essentials of the business process modeling so as to have a basis for analysing the language XChange. In the following we outline the most significant determinations of this investigation:

Service-Oriented Architecture and Web Services In order to work effectively, a BPMS often requires that the underlying software is constructed according to the principles of the Service-Oriented Architecture. The loosely-coupled components of this architecture – services – can be implemented on different platforms using different programming languages. All the clients need to know in order to use a service is its interface. XChange follows the similar approach: different Web sites (assume, a client and a service provider) communicate with event messages, that can be seen as an interface defining a name of the service and its arguments. The preferred way to realise a Service-Oriented Architecture is to use Web services. Using Web services has an advantage over using XChange programs: the former can be public registered and, thus, easily found on the Web. In XChange, the service provider (the message recipient) has to be known in advance.

Business Process Execution Language Amongst other standards, we have analysed the Business Process Execution Language – a language for the definition and execution of business processes using Web services (Section 3.4.2). The gain information was used in the (workflow-)pattern-based analysis of XChange as the foundation for the appraisal of XChange in terms of business processes modeling. As a language developed specific to the requirements of business processes, BPEL provides more constructs for support of typical processes behaviour patterns (e.g., sequence, exclusive choice, etc.).

Business Rules Approach We have investigated the business rules approach – an area of a growing importance in Business Process Management. Business rules are atomic, highly specific, and structured statements that constrain some aspect of a business, control or influence its behaviour (i.e., the behaviour of business processes). They encompass the business logic and tend to change quite

frequently. Business applications can gain on adaptability and flexibility if business analysts themselves create and maintain business rules. These, written in a natural or semi-natural language, are to be automatically translated in a programming language. For the efficient evaluation the implementation language is of a great importance. As we have recognised, the typical realisation form of a business rule – a production rule – is easy to implement with XChange, provided, however, that the language comprises a new structural element we have called `PROCEDURE`. The fact, that a `PROCEDURE` is a condition-action rule provides the one-to-one relationship between the rule specification and the implementation definitely simplifies the process of translating of the rules' specification into an executive code. As a sequence, such systems are more adaptive and flexible. Normally, business rules are unified into manageable-sized rulesets that can be invoked by different business applications. The implementation of rulesets in XChange is also straightforward: either within a rule or within a `PROCEDURE`.

In contrast, this type of applications (rules and rulesets) is difficult to develop and maintain using Java or any other procedural programming language. Java objects consist of methods for executing procedures and data members to hold the inputs and outputs of these procedures. They lack a mechanism for capturing rules that determine when and how procedures should be executed.

Provided with an extensive overview over the basic principles, standards, and technology in modern process and rules management systems, we have investigated the strengths and limits of the rule-based method of business process modeling. First of all, we have made a (workflow-)pattern-based analysis of XChange. Then, we have specified the use case EU-Rent (business processes and business rules of a fictitious car rental company) and implemented it.

Pattern-Based Analysis Workflow-patterns represent typical process behaviour. Our pattern-based analysis of XChange has revealed some limits of the language in support of business process modeling. The most serious ones are:

- Lack of a "process instance" principle in XChange, that means that multiple XChange rules cannot be combined together at the process scope. As a sequence, a cancellation of a started process, for example, cannot be fulfilled.
- Missing possibility to implement synchronous operations. As a result, even a basic pattern *Sequence* has to be implemented mostly with additional event messages.
- The imperative to restrict composite event queries to a finite time interval. Such restrictions cannot be combined with business process modeling, as business processes are repeatable and can take place at any point in time.

EU-Rent Use Case The implementation of the EU-Rent use case has motivated the existing language constructs, has given to refinements, and has served to identify constructs that are yet missing in XChange. The contributions of the work presented in this thesis are explicitly given in the following section.

7.2 Contributions

Our work on rule-based business process modeling contributes on the further development of XChange with the following:

7.2.1 Syntax Refinements

Structural Construct We have formally defined a new structural construct `PROCEDURE` realising condition-action (or production) rules. In this thesis condition-action rules have been used for implementing business rules. Having not discovered any application exigency for other structural constructs we have not made any other proposals in this direction. However, it can be an issue for future researches on XChange.

Rule Structure We have proposed to reorganise the structure of an XChange rule. The rationale is that in real-life situations it is often possible to react upon *the same* world's state change (event or event combination) *differently*, often depending on different conditions. Hence, a relative binding between different action- and condition parts is necessary. With such a rule structure (introduced in Section 5.1.1) a commonplace behaviour pattern corresponding to the construct `IF-THEN-ELSE` can be also implemented.

Sequence Order of Rule's Parts In order to conform with the typical Event-Condition-Action rule's structure, we have proposed to change the sequential order in the specification of XChange rules.

Keywords Introducing Rule's Parts Using of the keyword `FROM` introducing the condition part of an XChange rule is historical determined: the same keyword is used in Xcerpt rules. However, to emphasise the conditional meaning of data queries in XChange rules, the keyword `IF` seems to be more appropriate.

7.2.2 Language Constructs

New Language Constructs A new construct whose necessity has been proved in the implementation of patterns Discriminator and N-Out-of-M Join is a construct introduced with the keyword `first` (Section 5.1.2).

Motivation for Existing Language Constructs Through pattern-based analysis of XChange and the implementation of the use case EU-Rent the practicability of almost all (except for mentioned in the next paragraph) XChange language constructs has been confirmed on examples.

Not Used Language Constructs As no application situations for the following language constructs has been found, the further use case developing can be concentrated on involving them to bear evidence of their usefulness: repetitions and ranks of event queries.

Used Assumptions In our implementation we have made some assumptions. These are in fact intended to be formally defined and implemented in future. Nevertheless, we would like once more to emphasise their practicability demonstrated on the application examples (Chapter 6). These assumptions are:

- variables can be bound on the event message arguments;
- `where` clause can be applied on variables bound on event message arguments;
- XChange supports advanced time specifications.

7.2.3 Future Research Directions

While analysing the literature we have encountered different application areas (Web Service applications) that seem to be effective and efficient realisable with ECA rules [11, 1, 57, 21, 10]. These applications can be used for the future research on XChange:

- **Personalisation** For example, suitable site views should be assigned for clients who are classified respectively as kids or as frequent purchasers.
- **Alerts** For example, if a manager of an insurance company wishes to receive an alert if a customer request was reassigned by different agents at least three times.
- **Classification** The simplest example is the e-mail classification. Classification of users in the example for **Personalisation** is another one.
- **Metadata Management** application is, for example, a rule for incrementing the number of visitors of the pages of given books in an online library.
- **Publish/Subscribe Systems** where events are filtered according to their attribute values, using filtering criteria defined by the subscribers, and then sent to the interested subscribers.

7.3 Conclusion

Through the investigation of business process modeling and business rules approach requirements, analysis of XChange to satisfy them, and through the implementation of (simplified versions of) real-life applications, we arrive at a conclusion that the declarative, reactive, rule-based character of XChange is advantageous for the implementation of business rules, normally specified as production rules. XChange in its actual version is less advantageous for the implementation of business process behaviour, can be, however, further developed based on the outcomes of this research. Our work has demonstrated the practicability of almost all language constructs and revealed some structural and syntactic shortcomings. Elimination of these shortcomings in future would surely strengthen the expressive power of the language irrespective of the application areas it will be used for.

Part V

Appendix

APPENDIX

A

XChange Implementation of EU-Rent Business Processes

The following provides XChange implementations of EU-rent business processes and business rules. The given Document Type Definitions do not belong to XChange programs and are given to simplify the code understanding.

A.1 Rental Management

A.1.1 Constructs

Registered customers

```
<!DOCTYPE registered-customers [
   <!ELEMENT registered-customers (customer+)>
   <!ELEMENT customer (id, first-name, last-name)>
   <!ELEMENT id (#PCDATA)>
   <!ELEMENT first-name (#PCDATA)>
   <!ELEMENT last-name (#PCDATA)>
]>

CONSTRUCT
   registered-customers {
      all customer { var Id, var FName, var LName }
   }
FROM
   in { resource { "file:persons.xml"},
      desc person {{
         category { "customer" },
         var Id -> id {{ }},
         var FName -> first-name {{ }},
         var LName -> last-name {{ }},
      }}
   }
END
```

Blacklisted customers

```
<!DOCTYPE blacklisted [
   <!ELEMENT blacklisted (customer+)>
```

```
   <!ELEMENT customer (id, first-name, last-name)>
   <!ELEMENT id (#PCDATA)>
   <!ELEMENT first-name (#PCDATA)>
   <!ELEMENT last-name (#PCDATA)>
]>

CONSTRUCT
   blacklisted {
      all customer { var Id, var FName, var LName }
   }
FROM
   in { resource { "file:persons.xml"},
      desc person {{
         category { "customer" },
         blacklisted {{ }},
         var Id -> id {{ }},
         var FName -> first-name {{ }},
         var LName -> last-name {{ }},
      }}
   }
END
```

Invalid credit cards

```
<!DOCTYPE invalid-credit-cards [
   <!ELEMENT invalid-credit-cards (credit-card+)>
   <!ELEMENT credit-card (customer, bank, number)>
   <!ELEMENT customer (id, first-name, last-name)>
   <!ELEMENT id (#PCDATA)>
   <!ELEMENT first-name (#PCDATA)>
   <!ELEMENT last-name (#PCDATA)>
   <!ELEMENT bank (#PCDATA)>
   <!ELEMENT number (#PCDATA)>
]>

CONSTRUCT
   invalid-credit-cards {
      all credit-card {
         customer {
            var Id,
            var FName,
            var LName
         },
         var Bank,
         number { var KNr }
      }
   }
FROM
   and [
      in {resource {"file:persons.xml"},
         desc person {{
            var Id -> id {{ }},
            var FName -> first-name {{ }},
```

```
            var LName -> last-name {{ }},
            credit-card {{
               var Bank -> bank {{ }},
               number {{ var KNr }}
            }}
         }}
      },
      in {resource {"file:banks.xml"},
         desc bank {
            name { var Bank },
            uri { var Uri }
         }
      },
      in {resource {var Uri},
         without {
            desc credit-card {{
               var FName,
               var LName,
               kontoNr { var KNr }
            }}
         }
      }
   ]
END
```

Invalid driver's licences

```
<!DOCTYPE invalid-driver-licences [
   <!ELEMENT invalid-driver-licences (driver-licence+)>
   <!ELEMENT driver-licence (customer, invalid-from)>
   <!ELEMENT customer (id, first-name, last-name)>
   <!ELEMENT id (#PCDATA)>
   <!ELEMENT first-name (#PCDATA)>
   <!ELEMENT last-name (#PCDATA)>
   <!ELEMENT invalid-from (#PCDATA)>
]>

CONSTRUCT
   invalid-driver-licences {
      all driver-licence {
         customer {
            var Id,
            var FName,
            var LName,
         },
         invalid-from { var Date }
      }
   }
FROM
   in {resource {"file:persons.xml"},
      desc person {{
         var Id -> id {{ }},
         var FName -> first-name {{ }},
```

```
         var LName -> last-name {{ }},
         driver-licence {{
            expiration-date { var Date }
         }}
      }}
   }
END
```

Open rentals of each customer

```
<!DOCTYPE open-rentals [
   <!ELEMENT open-rentals (customer+)>
   <!ELEMENT customer (id, rental+)>
   <!ELEMENT rental (from, till)>
   <!ELEMENT from (#PCDATA)>
   <!ELEMENT till (#PCDATA)>
]>

CONSTRUCT
   open-rentals {
      all customer {
         id { var Id },
         all rental { var From, var Till }
       }
   }
FROM
   in {resource {"file:rentals.xml"},
      desc rental {{
         customer { var Id },
         rental-status { var RStatus },
         var From -> from { },
         var Till -> till { }
      }}
   } where { var RStatus = "reservation" || var RStatus = "open" }
END
```

Available cars within each car group

```
<!DOCTYPE available-cars [
   <!ELEMENT available-cars (group+)>
   <!ELEMENT group (name, car+)>
   <!ELEMENT car (number, avail-from)>
   <!ELEMENT number (#PCDATA)>
   <!ELEMENT avail-from (#PCDATA)>
]>

CONSTRUCT
   available-cars {
      all group {
         name { var Group },
         all car {
            number { var Nr },
            avail-from {
```

```
               optional var Till with default today
            }
         }
      }
   }
FROM
   and {
      or {
         in {resource {"file:cars.xml"},
            desc car {{
               model { var Model },
               car-status { "available" }
            }}
         },
         and {
            in {resource {"file:cars.xml"},
               desc car {{
                  registration-number { var Nr },
                  model { var Model },
                  car-status { "assigned" }
               }}
            },
            in {resource {"file:rentals.xml"},
               desc rental {{
                  assigned-car { var Nr },
                  till { var Till }
               }}
            }
         }
      },
      in {resource {"file:models.xml"},
         desc model {{
            name { var Model },
            group { var Group }
         }}
      }
   }
END
```

Reservation requests

```
<!DOCTYPE reservation-requests [
   <!ELEMENT reservation-requests (rental+)>
   <!ELEMENT rental (customer, car, time-period, location,
     rental-status, price)>
   <!ELEMENT customer (#PCDATA)>
   <!ELEMENT car (group?, model?)>
   <!ELEMENT group ("A"|"B"|"C"|"D")>
   <!ELEMENT model (#PCDATA)>
   <!ELEMENT time-period (from, till)>
   <!ELEMENT from (#PCDATA)>
   <!ELEMENT till (#PCDATA)>
   <!ELEMENT location (pick-up-branch, return-branch?)>
```

```
   <!ELEMENT pick-up-branch (#PCDATA)>
   <!ELEMENT return-branch (#PCDATA)>
   <!ELEMENT rental-status ("request")>
   <!ELEMENT price (basic-price, lowest-price)>
   <!ELEMENT basic-price (#PCDATA)>
   <!ELEMENT lowest-price (#PCDATA)>
]>

CONSTRUCT
   reservation-requests {
      all var Rental
   }
FROM
   in {resource {"file:rentals.xml"},
      var Rental -> desc rental {{
         rental-status {"request"}
      }}
   }
END
```

A.1.2 Refusal Business Rules

Refuse unregistered customer

```
PROCEDURE refuse-unregistered [
      var Customer,
      var Uri,
      process { "reservation" }
   ]
IF registered-customers {{
      without var Customer
   }}
DO xchange:event {
      xchange:recipient { var Uri },
      eu-rent-reply {
         status {"Failure"},
         message {"Registration necessary."}
      }
   }
END
```

Refuse blacklisted customer

```
PROCEDURE refuse-blacklisted [
      var Customer,
      var Uri,
      process { "reservation" }
   ]
IF blacklisted {{
      var Customer
   }}
DO xchange:event {
      xchange:recipient { var Uri },
      eu-rent-reply {
```

```
         status {"Failure"},
         message {"You are blacklisted."}
      }
   }
END
```

Refuse if driver's licence is invalid

```
PROCEDURE refuse-invalid-licence [
      var Customer,
      time-period {{ till { var Till } }},
      var Uri
   ]
IF invalid-driver-licences {{
      driver-licence {
         var Customer,
         invalid-from { var Date }
      }
   }} where { var Date < var Till }
DO xchange:event {
      xchange:recipient { var Uri },
      eu-rent-reply {
         status {"Failure"},
         message {"Invalid driver licence."}
      }
   }
END
```

Refuse if credit card is invalid

```
PROCEDURE refuse-invalid-credit-card [
      var Customer,
      var Uri,
      process {"reservation"}
   ]
IF invalid-credit-cards {{
      credit-card {{ var Customer }}
   }}
DO xchange:event {
      xchange:recipient { var Uri },
      eu-rent-reply {
         status {"Failure"},
         message {"Invalid credit card."}
      }
   }
END
```

Refuse if rentals overlap

```
PROCEDURE refuse-overlap [
      customer {{ id { var Id } }},
      time-period {{ from { var From }, till { var Till } }},
      var Uri
   ]
```

```
IF open-rentals {{
      customer {{
         id { var Id },
         rental {
            from { var AnotherFrom },
            till { var AnotherTill }
         }
      }}
   }} where {
         (var AnotherFrom <= var From && var AnotherTill >= var From) ||
         (var AnotherFrom >= var From && var AnotherTill <= var Till)
      }
DO xchange:event {
      xchange:recipient { var Uri },
      eu-rent-reply {
         status {"Failure"},
         message {"Overlap with another rental. Change rental period."}
      }
   }
END
```

Refuse if the specified car group is unavailable

```
PROCEDURE refuse-unavailable-group [
      customer {{ id { var Id } }},
      var Period -> time-period {{ from { var From } }},
      var Uri,
      process {"reservation"}
   ]
IF and {
      in reservation-requests {{
          rental {{
            customer { var Id },
            var Period,
            car {{ group { var Group } }}
         }}
      }},
      not available-cars {{
         group {{
            name { var Group },
            avail-from { var Date }
         }}
      }} where { var From <= var Date }
   }
DO xchange:event {
      xchange:recipient { var Uri },
      eu-rent-reply {
         status {"Failure"},
         message {"Car of this group is unavailable.
                   Please choose another one."}
      }
   }
END
```

Refuse rental's extension if the car is scheduled for maintenance

```
PROCEDURE refuse-maintenance [
      customer {{ id { var Id } }},
      time-period {{ from { var From } }},
      var Uri,
      process {"extension"}
   ]
IF and {
      in {resource{file:rentals.xml},
         rentals {{
            rental {{
               customer { var Id },
               time-period {{ from { var From } }},
               assigned-car { var Nr }
            }}
         }}
      },
      in {resource{file:cars.xml},
         cars {{
            car {{
               registration-number { var Nr },
               maintenance {{
                  optional new-date { var MaintDate }
               }}
            }}
         }}
      } where { var MaintDate <= var Till }
   }
DO xchange:event {
      xchange:recipient { var Uri },
      eu-rent-reply {
         status {"Failure"},
         message {"The car is scheduled for maintenance."}
      }
   }
END
```

Refusal Ruleset

```
PROCEDURE refusal-rules [
      var Customer,
      uri { var Uri },
      var Period,
      var Process -> process {{ }}
   ]
DO or {
      refuse-unregistered { var Customer, var Uri, var Process },
      refuse-blacklisted { var Customer, var Uri, var Process },
      refuse-invalid-credit-card { var Customer, var Uri, var Process },
      refuse-invalid-licence { var Customer, var Period, var Uri },
      refuse-overlap { var Customer, var Period, var Uri },
      refuse-unavailable-group { var Customer, var Period, var Uri,
```

```
                                    var Process }
      refuse-maintenance { var Customer, var Period, var Uri, var Process }
   }
END
```

A.1.3 Reservation Business Rules

Default car group

```
PROCEDURE default-car-group [
      customer {{ id { var Id } }},
      var Period
   ]
IF reservation-requests {{
      rental {{
         customer { var Id },
         var Period,
         car {{ without group {{ }} }}
      }}
   }}
DO in {resource {"file:rentals.xml"},
      rentals {{
         rental {{
            customer { var Id },
            var Period,
            rental-status {"request"},
            car {{ insert group {"A"} }}
         }}
      }}
   }
END
```

Default return branch

```
PROCEDURE default-return-branch [
      customer {{ id { var Id } }},
      var Period
   ]
IF reservation-requests {{
      rental {{
         customer { var Id },
         var Period,
         location {{
            pickup-branch { var Branch },
            without return-branch {{ }}
         }}
      }}
   }}
DO in {resource {"file:rentals.xml"},
      rentals {{
         rental {
            customer { var Id },
            var Period,
            rental-status {"request"},
```

```
            location {{ insert return-branch { var Branch } }}
         }
      }}
   }
END
```

Reservation Ruleset

```
PROCEDURE reservation-rules [
      var Customer,
      var Period
   ]
DO or [
      and {
         default-car-group { var Customer, var Period },
         default-return-branch { var Customer, var Period }
      },
      true { }
   ]
END

CONSTRUCT
   true { }
END
```

A.1.4 Rental Reservation Process

```
<!DOCTYPE reservation-request [
   <!ELEMENT reservation-request {customer, car, time-period, location}>
   <!ELEMENT customer (id, first-name, last-name)>
   <!ELEMENT id (#PCDATA)>
   <!ELEMENT first-name (#PCDATA)>
   <!ELEMENT last-name (#PCDATA)>
   <!ELEMENT car (group?, model?)>
   <!ELEMENT group ("A"|"B"|"C"|"D")>
   <!ELEMENT model (#PCDATA)>
   <!ELEMENT time-period (from, till)>
   <!ELEMENT from (#PCDATA)>
   <!ELEMENT till (#PCDATA)>
   <!ELEMENT location (pick-up, return-branch?)>
   <!ELEMENT pick-up (#PCDATA)>
   <!ELEMENT return-branch (#PCDATA)>
]>

ON xchange:event {{
      xchange:sender { var Uri },
      reservation-request [
         var Customer -> customer {{ id { var Id } }},
         var Car,
         var Period,
         var Location
      ]
```

```
   }}
DO and [
      <!-- add reservation request -->
      in {resource {"file:rentals.xml"},
         rentals {{
            insert rental [
               customer { var Id },
               var Car, var Period, var Location,
               price [ basic-price { }, lowest-price { } ],
               rental-status {"request"}
            ]
         }}
      },
      or [
         <!-- reservation granted -->
         and [
            reservation-rules { var Customer, var Period },
            not refusal-rules { var Customer, var Uri, var Period,
                                process "reservation" },
            pricing-rules { var Customer, var Period },
            in {resource {"file:rentals.xml"},
               desc rental {{
                  customer { var Id },
                  var Period,
                  rental-status {"request" replaceby "reservation"}
               }}
            },
            xchange:event{
               xchange:recipient { var Uri },
               reply {
                  status {"Finished"},
                  message {"Reservation successful."}
               }
            }
         ],

         <!-- reservation refused -->
         in {resource {"file:rentals.xml"},
            rentals {{
               delete rental {{
                  customer { var Id },
                  var Period,
                  rental-status {"request"}
               }}
            }}
         }
      ]
   ]
END
```

A.1.5 Rental Extension Process

```
<!DOCTYPE extension-request [
   <!ELEMENT extension-request {customer, time-period}>
   <!ELEMENT customer (id, first-name, last-name)>
   <!ELEMENT id (#PCDATA)>
   <!ELEMENT first-name (#PCDATA)>
   <!ELEMENT last-name (#PCDATA)>
   <!ELEMENT time-period (from, till)>
   <!ELEMENT from (#PCDATA)>
   <!ELEMENT till (#PCDATA)>
]>

ON xchange:event {{
      xchange:sender { var Uri },
      extension-request [
         var Customer -> customer {{ id { var Id } }},
         var Period -> time-period {
            from { var From },
            till { var Till }
         }
      ]
   }}
DO and [
      <!-- check refusal rules -->
      not refusal-rules { var Customer, var Uri, var Period,
                          process "extension" },
      and {
         <!-- check pricing rules -->
         pricing-rules { var Customer, var Period },

         <!-- update rental data -->
         in {resource {"file:rentals.xml"},
            desc rental {{
               customer { var Id },
               time-period {{
                  from { var From },
                  till { var Dummy replaceby var Till }
               }}
            }}
         },

         <!-- send message to the customer -->
         xchange:event{
            xchange:recipient { var Uri },
            reply {
               status {"Finished"},
               message {"Rental extension successful."}
            }
         }
      }
   ]
END
```

A.2 Pricing and Discounting Business Rules

Count basic price

```
PROCEDURE basic-price [
      var Customer,
      var Period
   ]
IF and {
      in {resource {file:rentals.xml},
         var Customer,
         var Period,
         price {{ basic-price { var Price } }},
         car {{ group { var Group } }}
      },
      in {resource {file:car-group.xml},
         desc car-group {{
            name { var Group },
            prices {{
               duration {
                  name { "day" },
                  price { var DPrice }
               }
            }}
         }}
      }
   }
DO in {resource {"file:rentals.xml"},
      rentals {{
         rental {{
            var Customer,
            var Period,
            price {{
               basic-price { var Price replaceby
                             var DPrice * (var Till - var From) }
            }}
         }}
      }}
   }
END
```

10% discount for a rental booked at least 3 days in advance

```
PROCEDURE ten-percent-discount {
      customer {{ id { var Id } }},
      var Period -> time-period {{ from { var From } }}
   }
IF in {resource {"file:rentals.xml"},
      desc rental {{
         var Id,
         var Period,
         price [
            basic-price { var BPrice },
            lowest-price { var LPrice }
```

```
         ]
      }}
   } where { ( today+3 days <= var From ) &&
             ( LPrice > var BPrice / 10 ) }
DO in {resource {"file:rentals.xml"},
      desc rental {{
         var Id,
         var Period,
         price {{
            lowest-price {
               var LPrice replaceby var BPrice / 10
            }
         }}
      }}
   }
END
```

$50 discount for a week rental (for a car group C or D)

```
PROCEDURE fifty-discount [
      customer {{ id { var Id } }},
      var Period -> time-period {{
         from { var From },
         till { var Till }
      }}
   ]
IF in {resource {"file:rentals.xml"},
      desc rental {{
         var Id,
         var Period,
         car {{ group { var Group } }},
         price [
            basic-price { var BPrice },
            lowest-price { var LPrice }
         ]
      }}
   } where { ((var Till - var From) >= 7 days ) &&
             ( LPrice > var BPrice - 50 ) &&
             ( var Group = "C" || var Group = "D" ) }
DO in {resource {"file:rentals.xml"},
      desc rental {{
         var Id,
         var Period,
         price {{
            lowest-price {
               var LPrice replaceby var BPrice - 50
            }
         }}
      }}
   }
END
```

Pricing Ruleset

```
PROCEDURE pricing-rules [
      var Customer,
      var Period
   ]
DO or [
      and [
         basic-price { var Customer, var Period },
         and {
            ten-percent-discount { var Customer, var Period },
            fifty-discount { var Customer, var Period }
         }
      ],
      true { }
   ]
END

CONSTRUCT
   true { }
END
```

A.3 Car Allocation

A.3.1 Constructs

Rentals for allocation with a specified model

```
<!DOCTYPE rentals-for-allocation-with-model [
   <!ELEMENT rentals-for-allocation-with-model (rentals+)>
   <!ELEMENT rental (customer, add-drivers?, car, time-period,
     location, rental-status, price)>
   <!ELEMENT customer (#PCDATA)>
   <!ELEMENT add-drivers (add-driver+)>
   <!ELEMENT add-driver (#PCDATA)>
   <!ELEMENT car (group, model)>
   <!ELEMENT group ("A"|"B"|"C"|"D")>
   <!ELEMENT model (#PCDATA)>
   <!ELEMENT time-period (from, till)>
   <!ELEMENT from (#PCDATA)>
   <!ELEMENT till (#PCDATA)>
   <!ELEMENT location (pick-up-branch, return-branch)>
   <!ELEMENT pick-up-branch (#PCDATA)>
   <!ELEMENT return-branch (#PCDATA)>
   <!ELEMENT rental-status ("reservation")>
   <!ELEMENT price (basic-price, lowest-price)>
   <!ELEMENT basic-price (#PCDATA)>
   <!ELEMENT lowest-price (#PCDATA)>
]>

CONSTRUCT
   rentals-for-allocation-with-model {
       all rental
   }
FROM
   in {resource {"file:rentals.xml"},
      var Rental -> desc rental {{
         rental-status { "reservation" },
         time-period {{ from { tomorrow } }},
         car {{
            model {{ }},
            without assigned-car {{ }}
         }}
      }}
   }
END
```

Rentals for allocation without model specification

```
<!DOCTYPE rentals-for-allocation-without-model [
   <!ELEMENT rentals-for-allocation-without-model (rentals+)>
   <!ELEMENT rental (customer, add-drivers?, car, time-period,
     location, rental-status, price)>
   <!ELEMENT customer (#PCDATA)>
   <!ELEMENT add-drivers (add-driver+)>
   <!ELEMENT add-driver (#PCDATA)>
```

```
   <!ELEMENT car (group)>
   <!ELEMENT group ("A"|"B"|"C"|"D")>
   <!ELEMENT time-period (from, till)>
   <!ELEMENT from (#PCDATA)>
   <!ELEMENT till (#PCDATA)>
   <!ELEMENT location (pick-up-branch, return-branch)>
   <!ELEMENT pick-up-branch (#PCDATA)>
   <!ELEMENT return-branch (#PCDATA)>
   <!ELEMENT rental-status ("reservation")>
   <!ELEMENT price (basic-price, lowest-price)>
   <!ELEMENT basic-price (#PCDATA)>
   <!ELEMENT lowest-price (#PCDATA)>
]>

CONSTRUCT
   rentals-for-allocation-without-model {
       all rental
   }
FROM
   in {resource {"file:rentals.xml"},
      var Rental -> desc rental {{
         rental-status { "reservation" },
         time-period {{ from { tomorrow } }},
         car {{
            without model {{ }},
            without assigned-car {{ }}
         }}
      }}
   }
END
```

All rentals for allocation

```
<!DOCTYPE rentals-for-allocation [
   <!ELEMENT rentals-for-allocation (rentals+)>
   <!ELEMENT rental (customer, add-drivers?, car, time-period,
     location, rental-status, price)>
   <!ELEMENT customer (#PCDATA)>
   <!ELEMENT add-drivers (add-driver+)>
   <!ELEMENT add-driver (#PCDATA)>
   <!ELEMENT car (group, model?)>
   <!ELEMENT group ("A"|"B"|"C"|"D")>
   <!ELEMENT model (#PCDATA)>
   <!ELEMENT time-period (from, till)>
   <!ELEMENT from (#PCDATA)>
   <!ELEMENT till (#PCDATA)>
   <!ELEMENT location (pick-up-branch, return-branch)>
   <!ELEMENT pick-up-branch (#PCDATA)>
   <!ELEMENT return-branch (#PCDATA)>
   <!ELEMENT rental-status ("reservation")>
   <!ELEMENT price (basic-price, lowest-price)>
   <!ELEMENT basic-price (#PCDATA)>
   <!ELEMENT lowest-price (#PCDATA)>
```

```
]>

CONSTRUCT
   rentals-for-allocation {
       all rental
   }
FROM
   in {resource {"file:rentals.xml"},
      var Rental -> desc rental {{
         rental-status { "reservation" },
         time-period {{ from { tomorrow } }},
         car {{ without assigned-car {{ }} }}
      }}
   }
END
```

Nearest branches

```
<!DOCTYPE branches-for-transfer [
   <!ELEMENT branches-for-transfer (branch+)>
   <!ELEMENT branch (name, transfer-time, uri)>
   <!ELEMENT name (#PCDATA)>
   <!ELEMENT transfer-time (#PCDATA)>
   <!ELEMENT uri (#PCDATA)>
]>

CONSTRUCT
   branches-for-transfer {
      all branch {
         var Name,
         var TTime,
         uri { var Uri }
      }
   }
FROM and {
      in {resource {"file:branches.xml"},
         desc branch {{
            name { "EuRentBranch" },
            nearest-branches {{
               transfer-branch {{
                  var Name -> name {{ }},
                  var TTime -> transfer-time {{ }}
               }}
            }}
         }}
      },
      in {resource {"file:branches.xml"},
         desc rental {{
            var Name,
            uri { var Uri }
         }}
      }
   }
```

```
END
```

A.3.2 Car Allocation Business Rules

Assign cars to rentals with a specified model

```
PROCEDURE assign-with-model {{ }}
IF and {
      rentals-for-allocation-with-model {{
         desc rental {{
            var Customer -> customer {{ }},
            var Period -> time-period {{ }},
            car {{ model { var Model } }}
         }}
      }},
      in {resource {"file:cars.xml"},
         desc car {{
            registration-number { var Nr },
            model { var Model },
            car-status { "available" }
         }}
      }
   }
DO and {
      in {resource {"file:rentals.xml"},
         desc rental {{
            var Customer,
            var Period,
            car {{ insert assigned-car { var Nr } }}
         }}
      },
      in {resource {"file:cars.xml"},
         desc car {{
            registration-number { var Nr },
            car-status { "available" replaceby "assigned" }
         }}
      }
   }
END
```

Assign within the group

```
PROCEDURE assign-within-group {{ }}
IF and {
      rentals-for-allocation-without-model {{
         desc rental {{
            var Customer -> customer {{ }},
            var Period -> time-period {{ }},
            car {{ group { var Group } }}
         }}
      }},
      in {resource {"file:car-models.xml"},
         desc car-model {{
            name { var Model },
```

```
            group { var Group }
         }}
      },
      in {resource {"file:cars.xml"},
         desc car {{
            registration-number { var Nr },
            model { var Model },
            car-status { "available" }
         }}
      }
   }
DO and {
      in {resource {"file:rentals.xml"},
         desc rental {{
            var Customer,
            var Period,
            insert assigned-car { var Nr }
         }},
      },
      in {resource {"file:cars.xml"},
         desc car {{
            registration-number { var Nr },
            car-status { "available" replaceby "assigned" }
         }}
      }
   }
```

Free upgrade

```
PROCEDURE free-upgrade {{ }}
IF and {
      rentals-for-allocation {{
         desc rental {{
            var Customer -> customer {{ }},
            var Period -> time-period {{ }},
            car {{ group { var Group } }}
         }}
      }},
      in {resource {"file:car-groups.xml"},
         desc car-group {{
            name { var Group },
            previous-group { var PrGroup }
         }}
      },
      in {resource {"file:car-groups.xml"},
         desc car-group {{
            name { var PrGroup },
            model { var Model }
         }}
      },
      in {resource {"file:cars.xml"},
         desc car {{
            registration-number { var Nr },
```

```
            model { var Model },
            car-status { "available" }
         }}
      }
   }
DO and {
      in {resource {"file:rentals.xml"},
         desc rental {{
            var Customer,
            var Period,
            car {{ insert assigned-car { var Nr } }}
         }},
         in {resource {"file:cars.xml"},
            desc car {{
               registration-number { var Nr },
               car-status { "available" replaceby "assigned" }
            }}
         }
      }
   }
END
```

Downgrade

```
PROCEDURE downgrade {{ }}
IF and {
      rentals-for-allocation {{
         desc rental {{
            var Customer -> customer {{ }},
            var Period -> time-period {{ }},
            car {{ group { var PrGroup } }}
         }}
      }},
      in {resource {"file:car-groups.xml"},
         desc car-group {{
            name { var Group },
            previous-group { var PrGroup }
         }}
      },
      in {resource {"file:car-groups.xml"},
         desc car-group {{
            name { var Group },
            model { var Model }
         }}
      },
      in {resource {"file:cars.xml"},
         desc car {{
            registration-number { var Nr },
            model { var Model },
            car-status { "available" }
         }}
      }
   }
```

```
DO and {
      in {resource {"file:rentals.xml"},
         desc rental {{
            var Customer,
            var Period,
            car {{ insert assigned-car { var Nr } }}
         }},
         in {resource {"file:cars.xml"},
            desc car {{
               registration-number { var Nr },
               car-status { "available" replaceby "assigned" }
            }}
         }
      }
   }
END
```

Bumped upgrade

```
PROCEDURE bumped-upgrade {{ }}
IF and {
      rentals-for-allocation {{
         desc rental {{
            var Customer -> customer {{ }},
            var Period -> time-period {{ }},
            car {{ group { var Group } }}
         }}
      }},
      in {resource {"file:car-groups.xml"},
         desc car-group {{
            name { var Group },
            previous-group { var PrGroup }
         }}
      },
      in {resource {"file:car-groups.xml"},
         desc car-group {{
            name { var PrGroup },
            previous-group { var PrPrGroup }
         }}
      },
      in {resource {"file:car-models.xml"},
         desc car-model {{
            name { var PrModel },
            group { var PrGroup }
         }}
      },
      in {resource {"file:car-models.xml"},
         desc car-model {{
            name { var PrPrModel },
            group { var PrPrGroup }
         }}
      },
      in {resource {"file:cars.xml"},
```

```
         desc car {{
            registration-number { var Nr },
            model { var PrPrModel },
            car-status { "available" }
         }}
      },
      in {resource {"file:cars.xml"},
         desc car {{
            registration-number { var AnotherNr },
            model { var PrModel },
         }}
      },
      in {resource {"file:rentals.xml"},
         desc rental {{
            rental-status {"reservation"},
            car {{ assigned-car { var AnotherNr } }},
            var AnotherCustomer -> customer {{ }},
            var AnotherPeriod -> time-period {{ }}
         }}
      }
   }
DO and {
       in {resource {"file:rentals.xml"},
         desc rental {{
            var Customer,
            var Period,
            car {{ insert assigned-car { var AnotherNr } }}
         }}
       },
       in {resource {"file:rentals.xml"},
         desc rental {{
            var AnotherCustomer,
            var AnotherPeriod,
            car {{ assigned-car { var AnotherNr replaceby var Nr } }}
         }}
       },
       in {resource {"file:cars.xml"},
          desc car {{
            registration-number { var Nr },
            car-status { "available" replaceby "assigned" }
          }}
      }
   }
END
```

Allocation Ruleset

```
ON
   xchange:event {{
      allocation-rules {{ }}
   }}
DO and [
      assign-with-model {{ }},
```

```
      assign-within-group {{ }},
      free-upgrade {{ }},
      bumped-upgrade {{ }},
      downgrade {{ }},
      transfer {{ }}
   ]
END
```

Initiate car transfer

```
PROCEDURE transfer {{ }}
IF and {
      rentals-for-allocation {{
         var Rental -> desc rental {{ }}
      }},
      branches-for-transfer {{
         transfer-branch {{ uri { var Uri } }}
      }}
   }
DO all xchange:event {
      xchange:recipient { var Uri },
         transferRequest {
            all var Rental
         }
   }
END
```

Update data on each transfer agreement

```
ON
   xchange:event {{
      xchange:sender { var Uri },
      carTransferAgreement {
         var Rental -> rental {{ }},
         var Time -> delivery-time {{ }}
      }
   }}
DO in {resource {"file:transfers.xml"},
      agreements {{
         insert agreement {
            from { var Uri },
            var Rental,
            var Time
         }
      }}
   }
END
```

Confirm transfer after 3 agreements or after 1 agreement within 30 minutes

```
ON or {
      andthen [
         xchange:event {{
```

```
            transferRequest { var Rental }
         }},
         first 3 of {
            xchange:event {{
               carTransferAgreement {{ var Rental }}
            }}
         }
      ],

      andthen [
         xchange:event {{
            xchange:raising-time { var RTime },
            transferRequest { var Rental }
         }},
         xchange:event {{
            carTransferAgreement {{ var Transfer }}
         }},
      ] during [ var RTime, var RTime + 30 minute ]
   }
IF and {
      in {resource {"file:transfers.xml"},
         desc agreement {{
            var Rental,
            from { var Uri },
            delivery-time { var BestTime }
         }}
      },
      in {resource {"file:transfers.xml"},
         desc agreement {{
            var Rental,
            from { var AnotherUri },
            delivery-time { var AnotherTime }
         }}
      }
   } where { var BestTime < var AnotherTime }
DO and {
      xchange:event {
         xchange:recipient { var Uri },
         start-transfer { var Rental }
      },
      all xchange:event {
         xchange:recipient { var AnotherUri },
         cancel-transfer { var Rental }
      }
   }
END
```

Start rentals' delay after timeout without agreements

```
ON andthen [
      xchange:event {{
         xchange:raising-time { var RTime },
         transferRequest { var Rental }
```

```
      }},
      without {
         xchange:event {{
            carTransferAgreement {{ var Rental }}
         }}
      }
   ] during [ var RTime, var RTime + 30 minute ]
DO rentals-delay {{ }}
END
```

Rentals' delay

```
PROCEDURE rentals-delay {{ }}
IF rentals-for-assignment {{
      var Rental -> desc rental {{
         customer { var Id },
      }},
      in {resource {"file:persons.xml"},
         desc customer {
            id { var Id },
            uri { var Uri }
         }
      }
   }}
DO all xchange:event {
      xchange:recipient { var Uri },
      rentalDelayRequest {
         var Rental
      }
   }
END

ON andthen [
      xchange:event {
         xchange:recipient { var Customer },
         rentalDelayRequest {{ }}
      },
      xchange:event {
         xchange:sender { var Customer },
         rentalDelayAgreement {{
            rental {{
               var Customer -> customer {{ }},
               var From -> from {{ }},
               till {{ var Till }}
            }},
            date { var Date }
         }}
      }
   ]
DO in {resource {"file:rentals.xml"},
      desc rental {{
         var Customer,
         time-period {{
```

```
            var From,
            till { var Till replaceby var Date }
         }}
      }}
   }
END
```

Monitor reservations

```
ON andthen [
      xchange:event {{
         xchange:reception-time { var Reception },
         start-reservation-monitor {
            var Rental -> rental {{
               customer {{ var Id }},
               var Period -> time-period {{ }},
            }}
         }
      }},
      without {
         xchange:event {{
            stop-reservation-monitor { var Rental }
         }}
      } during [ var Reception, var Reception+90 minutes ]
   ]
DO and {
      <!-- cancel reservation -->
      in {resource {"file:rentals.xml"},
         desc rental {{
            customer { var Id },
            var Period,
            car {{ assigned-car { var Nr } }},
            rental-status {"reservation" replaceby "cancelled"}
         }}
      },

      <!-- update car status -->
      in {resource {"file:cars.xml"},
         desc car {{
            registration-number { var Nr },
            car-status {"assigned" replaceby "available"}
         }}
      },

      <!-- message to the customer -->
      IF in {resource {"file:persons.xml"},
            desc person {{
               id { var Id },
               uri { var Uri }
            }}
         }
      DO xchange:event {
            xchange:recipient { var Uri },
```

```
            rental-info {
               message "Your reservation has been cancelled."
            }
         }
   }
END
```

BIBLIOGRAPHY

[1] Asaf Adi, David Botzer, Opher Etzion, and Tali Ytzkar-Haham. Push Technology Personalization Through Event Correlation. In *Proc. 26th Int. Conference on Very Large Databases*, pages 643–645, 2000.

[2] Dimitrios Tombros Andreas Geppert, Markus Kradolfer. Workflow Specification and Event-Driven Workflow Execution. *SI-INFORMATIK*, April 1998.

[3] Joonsoo Bae, Hyerim Bae, Suk-Ho Kang, and Yeongho Kim. Automatic Control of Workflow Processes Using ECA Rules. *IEEE Transactions on Knowledge and Data Engineering*, 16(8):1010–1023, 2004.

[4] James Bailey, François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. Flavours of XChange, a rule-based reactive language for the (Semantic) Web. In *Proc. Int. Conference on Rules and Rule Markup Languages for the Semantic Web*, number 3791 in LNCS, pages 187–192. Springer, 2005.

[5] James Bailey, François Bry, and Paula-Lavinia Pătrânjan. Composite Event Queries for Reactivity on the Web. Technical Report PMS-FB-2004-26, Institute for Informatics, University of Munich, 2004.

[6] Daniel Barbara, Sharad Mehrotra, and Marek Rusinkiewicz. INCAS: A Computation Model for Dynamic Workflows in Autonomous Distribued Environments. Technical report, Matsushita Information Technology Laboratory, 1994.

[7] Tim Berners-Lee, Roy Fielding, and Henrik Frystyk. Hypertext Transfer Protocol (HTTP) 1.0. Internet Informational RFC 1945, 1996.

[8] Tim Berners-Lee, Roy Fielding, Henrik Frystyk, Jim Gettys, and Jeffrey C. Mogul. Hypertext Transfer Protocol (HTTP) 1.1. Internet Informational RFC 2068, 1997.

[9] Elisa Bertino, Barbara Catania, Vincenzo Gervasi, and Alessandra Raffaeta. *B. Freitag et al.(Eds.): Transactions and Change in Logic DBs*. Springer-Verlag Berlin Heidelberg, 1998.

[10] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Active rules for XML: A new paradigm for E-services. *The VLDB Journal*, 10(1):39–47, 2001.

[11] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Pushing reactive services to XML repositories using active rules. In *World Wide Web*, pages 633–641, 2001.

[12] François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. Querying Composite Events for Reactivity on the Web. In *Proc. of Int. Workshop on XML Research and Applications (XRA 2006)*, January 2006.

[13] François Bry, Tim Furche, Paula-Lavinia Pătrânjan, and Sebastian Schaffert. Data retrieval and evolution on the (Semantic) Web: A deductive approach. In *Proc. of Workshop on Principles and Practice of Semantic Web Reasoning 2004 (PPSWR 2004)*, volume 3208 of *Lecture Notes in Computer Science*. REWERSE, Springer-Verlag, September 2004.

[14] François Bry and Massimo Marchiori. Ten Theses on Logic Languages for the Semantic Web. In *Proc. of W3C Workshop on Rule Languages for Interoperability*, Washington D.C., USA, April 2005. W3C.

[15] François Bry and Paula-Lavinia Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *20th Annual ACM Symposium on Applied Computing (SAC'2005)*, volume 2, pages 1645–1649, Santa Fe, New Mexico,USA, March 2005. ACM Press.

[16] François Bry, Paula-Lavinia Pătrânjan, and Sebastian Schaffert. Xcerpt and XChange: Deductive Languages for Data Retrieval and Evolution on the Web. In *Proc. of Workshop on Semantic Web Services and Dynamic Networks*, volume 51 of *LNI*, pages 562–568, Ulm, Germany, September 2004. GI.

[17] Bruce G. Buchanan. *Rule-based expert systems: the MYCIN experiments of the Standord Heuristic Programming Project*. Addison-Wesley, Juni 1985.

[18] Business Rules Group. *Defining Business Rules - What Are They Really?*, Juli 2000. `http://www.businessrulesgroup.org/first_paper/br01c0.htm`.

[19] Business Rules Group. *The Business Rules Manifesto.*, November 2003. `http://www.businessrulesgroup.org/brmanifesto.htm`.

[20] Alison Cawsey. *Databases and Artificial Intelligence 3. Artificial Intelligence Segment*. Heriot Watt University (online), 1994. `http://www.cee.hw.ac.uk/~alison/ai3notes/section2_5_5.html`.

[21] Stefano Ceri, Piero Fraternali, and Stefano Paraboschi. Data-Driven, One-To-One Web Site Generation for Data-Intensive Applications. In *The VLDB Journal*, pages 615–626, 1999.

[22] C.J.Date. *What Not How. The Business Rules Approach to Application Development*. Addison-Wesley, 2000.

[23] Thomas H. Davenport. *Process Innovation: Reengineering Work through Information Technology*. Harvard Business School Press, 1993.

[24] Umeshwar Dayal, Meichung Hsu, and Rivka Ladin. Organizing Long-Running Activities with Triggers and Transactions. In *Proc. of Int. Conference on Management of Data*, 1990.

[25] Christian de Sainte Marie. *Why we need an XML standard for representing Business Rules*. W3C (online). `http://www.w3.org/2004/08/ws-cc/cdsm-20040903`.

[26] Klaus R. Dittrich and Stella Gatziu. *Aktive Datenbanksysteme, Konzepte und Mechanismen*. Internat. Thompson Publ., 1996.

[27] Michael Eckert. Reactivity on the Web: Event Queries and Composite Event Detection in XChange. Master's thesis, Institute for Informatics, University of Munich, Germany, 2005.

[28] Fair Isaac Corporation. *Fair Isaac Blaze Advisor: How it Works*, August 2005. Whitepaper, `http://www.fairisaac.com/Fairisaac/Solutions/Enterprise+Decision+Management/`.

[29] Fair Isaac Corporation. *Fair Isaac Blaze Advisor. Product Description*, Juli 2005. Whitepaper, `http://www.fairisaac.com/Fairisaac/Solutions/Enterprise+Decision+Management/`.

[30] FITECH Laboratories. *xTier Workflow Service*, 2003. Technical Whitepaper.

[31] Kevin Geminiuc. *A Services-Oriented Approach to Business Rules Development*. ORACLE(online), 2005. `http://www.oracle.com/technology/pub/articles/bpel_cookbook/geminiuc.html`.

[32] Andreas Geppert and Dimitros Tombros. Event-based Distributed Workflow Execution with EVE. Technical Report 96.05, University of Zurich, 1998.

[33] Ian Graham. *Service Oriented Business Rules Management Systems*. TriReme International Ltd. (online), Juni 2005. `http://www.trireme.com/`.

[34] John Hall. Business Rules Boot Camp. Tutorial at the European Business Rules Conference (EBRC) 2005.

[35] Michael Havey. *Essential Business Process Modeling*. O'Reilly Media, August 2005.

[36] Michael Havey. *What is Business Process Modeling?* O'Reilly Media, Inc.'s Online Publishing Group, July 2005. `http://www.onjava.com/pub/aonjava/2005/07/20/businessprocessmodeling.html`.

[37] Thomas Heimrich and Günter Specht. Enhancing ECA Rules for Distributed Active Database Systems. In *Web, Web-Services and Database*, pages 199–205. Springer, 2003.

[38] ILOG. *ILOG Rules*, Oktober 2002. Whitepaper, `http://www.ilog.com/products/businessrules/whitepapers/`.

[39] ILOG. *ILOG JRules 4.6 Technical White Paper*, April 2004. Whitepaper, `http://www.ilog.com/products/businessrules/whitepapers/`.

[40] ILOG. *ILOG JRules Business White Paper*, March 2005. Whitepaper, `http://www.ilog.com/products/businessrules/whitepapers/`.

[41] ILOG. *ILOG Rules for .NET and Microsoft Biz Talk Server*, September 2005. Whitepaper, `http://www.ilog.com/products/businessrules/whitepapers/`.

[42] J.A.Bubenko, D.Brash, and J.Stirna. EKG user guide. Technical report, Dept. of Computer and Systems Science, Royal Institute of Technology (KTH) and Stockholm University, 1998.

[43] Matjaz Juric. *A Hands-on Introduction to BPEL*. Oracle Technology Network (online), 2005. http://www.oracle.com/technology/pub/articles/matjaz_bpel1.html.

[44] Gerti Kappel, Peter Lang, S. Rausch-Schott, and Werner Retschitzegger. Workflow Management Based on Objects, Rules, and Roles. *Data Engineering Bulletin*, 18(1):11–18, 1995.

[45] Gerhard Knolmayer, Rainer Endl, and Marcel Pfahrer. Modeling Processes and Workflows by Business Rules. In *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[46] Qusay H. Mahmoud. *Getting Started With the Java Rule Engine API(JSR 94): Toward Rule-Based Applications*. Java Sun Developer Network, 2005. http://java.sun.com/developer/technicalArticles/J2SE/JavaRule.html.

[47] Colleen McClintock and Carole Ann Berlioz. *Implementing Business Rules in Java*. SYS-CON Media (online), May 2000. http://jdj.sys-con.com/read/36608.htm.

[48] Duane Nickull. *Service Oriented Architecture*. Adobe Systems Inc., 2005. Whitepaper.

[49] OASIS. *UDDI Version 3.0.2 Specification*, Juli 2004. http://uddi.org/pubs/uddi_v3.htm.

[50] Object Management Group. *CORBA BASICS*, November 2005. http://www.omg.org/gettingstarted/corbafaq.htm.

[51] Object Management Group (OMG). *Semantics of Business Vocabulary and Business Rules (SBVR)*, August 2005. Revised Submission, http://www.omg.org/docs/bei/05-08-01.pdf.

[52] OMG. *Business Semantics of Business Rules.*, January 2004. Request For Proposal.

[53] Ed Ort. *Service-Oriented Architecture and Web Services: Concepts, Technologies, and Tools*. Sun Developer Network, April 2005. http://java.sun.com/developer/technicalArticles/WebServices/soa2/.

[54] James Owen. *Blaze Advisor 5.1 shines for developers and business analysts*. InfoWorld (online), January 2004. http://www.infoworld.com/article/04/01/16/03TCblaze_1.html.

[55] G. Papamarkos, A. Poulovassilis, and P. Wood. Event-Condition-Action Rule Languages for the Semantic Web. In *Proc. of Workshop on Semantic Web and Databases, at VLDB'03*, September 2003.

[56] Norman W. Paton. *Active Rules in Database Systems*. Springer, 1999.

[57] Joao Pereira, Francoise Fabret, Francois Llirbat, and Dennis Shasha. Efficient Matching for Web-Based Publish/Subscribe Systems. In *Conference on Cooperative Information Systems*, pages 162–173, 2000.

[58] Paula-Lavinia Pătrânjan. *The Language XChange: A Declarative Approach to Reactivity on the Web*. PhD thesis, University of Munich, Germany, September 2005.

[59] Juha Puustjärvi, Henry Tirri, and Jari Veijalainen. Managing Overlapping Transactional Workflows. In *Proc. of the 8th Int. Conference on Advanced Information Systems Engineering (CAISE'96)*, pages 345–361, Crete,Greece, May 1996. Springer.

[60] J. Reinert and N. Ritter. Applying ECA Rules in DB-based Design Environments. In *Proc. CAD 98*, March.

[61] Dirk Riehle and Heinz Zuellighoven. Understanding and Using Patterns in Software Development. *Theor. Pract. Object Syst.*, 2(1):3–13, 1996.

[62] Colette Rolland. A Comprehensive View of Process Engineering. In *10th Int. Conference on Advanced Information System Engineering*, LNCS, London, UK, 1998. Springer.

[63] Florian Rosenberg and Schahram Dustdar. Towards a Distributed Service-Oriented Business Rules System. In *Proc. of the European Conference on Web Services 2005 (ECOWS 2005*, pages 14–24. IEEE Computer Society Press, November 2005.

[64] Anthony Rowe, Susie Stephens, and Yike Guo. *The Use of Business Rules with Workflow Systems*. W3C (online), 20044.
`www.w3.org/2004/12/rules-ws/paper/105/.`

[65] Sebastian Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, University of Munich, Germany, December 2004.

[66] Bruce Silver. *BPMS Watch: Is Visio Your Next BPMS Design Tool?* BrainStorm Group, June 2005. http://www.bpminstitute.org/articles/article/article/bpms-watch-is-visio-your-next-bpms-design-tool.html.

[67] Jennifer Widom Stefano Ceri, Roberta J. Cochrane. Practical Applications of Triggers and Constraints: Successes and Lingering Issues. In *Proc. 29th Int. Conference on Very Large Databases (VLDB)*, 2000.

[68] W. M. P. van der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[69] Gerd Wagner. How to Design a General Rule Markup Language? In *XSW*, pages 19–37, 2002.

[70] Stephen A. White. Introduction to BPMN. Technical report, Object Management Group/ Business Process Management Initiative, May 2004. `http://www.bpmn.org/.`

[71] Jennifer Widom and Stefano Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.

[72] Wikipedia, The Free Encyclopedia. *Business Process Management*, 2005.
`http://en.wikipedia.org/wiki/Business_process_management.`

[73] Wikipedia, The Free Encyclopedia. *Business Process Modeling*, 2005.
`http://en.wikipedia.org/wiki/Business_Process_Modeling.`

[74] Wikipedia, The Free Encyclopedia. *Geschäftsprozesse*, 2005.
`http://de.wikipedia.org/wiki/Gesch%C3%A4ftsprozess.`

[75] Wikipedia, The Free Encyclopedia. *MYCIN*, 2005. http://en.wikipedia.org/wiki/MYCIN.

[76] Wikipedia, The Free Encyclopedia. *Rete algorithm*, 2005. http://en.wikipedia.org/wiki/Rete_algorithm.

[77] Wikipedia, The Free Encyclopedia. *Service-Oriented Architecture*, 2005. http://en.wikipedia.org/wiki/Service-oriented_architecture.

[78] Wikipedia, The Free Encyclopedia. *SOAP*, 2005. http://en.wikipedia.org/wiki/SOAP.

[79] Petia Wohed, Wil M.P. van der Aalst, Marlon Dumas, and Arthur H.M. ter Hofstede. Pattern Based Analysis of BPEL4WS. Technical report, Department of Computer and Systems Sciences, Stockholm University, 2002. xml.comverpages.org/AalstBPEL4WS.pdf.

[80] Workflow Management Coalition. *Terminology and Glossary*, Februar 1999. Technical Report WFMSTC-1011.

[81] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.1*, February 2004. W3C Recommendation, http://www.w3.org/TR/2004/REC-xml11-20040204/.

[82] World Wide Web Consortium (W3C). *SOAP Version 1.2*, June 2004. W3C Recommendation, http://www.w3.org/TR/soap/.

[83] World Wide Web Consortium (W3C). *Web Services Choreography Description Language*, April 2004. W3C Working Draft, http://www.w3.org/TR/2006/WD-ws-cdl-10-20040427/.

[84] World Wide Web Consortium (W3C). *Web Services Description Language (WSDL)*, January 2006. W3C Recommendation, http://www.w3.org/TR/2006/CR-wsdl20-20060106/.