Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BM-2007-02

Masterarbeit

im Studiengang "Angewandte Informatik"

Evaluation of the Oracle 10g Rules Manager for a Domain Node Architecture in the Semantic Web ECA Framework

Carsten Gottschlich

am Lehrstuhl für

Datenbanken und Informationssysteme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

31. Januar 2007

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel. +49 (5 51) 39-1 44 14
Fax +49 (5 51) 39-1 44 15
Email office@informatik.uni-goettingen.de
WWW www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 31. Januar 2007

Evaluation of the Oracle 10g Rules Manager for a Domain Node Architecture in the Semantic Web ECA Framework

Carsten Gottschlich

January 2007

Abstract

A major challenge of the nascent Semantic Web is the multitude of data formats, languages, schemas and ontologies. "A General Framework for Evolution and Reactivity in the Semantic Web" was designed for bridging the gap torn apart by the heterogeneity. This framework is based on Event-Condition-Action (ECA) rules.

With the release of version 10g Oracle has enhanced its database software by adding a Rules Manager package that allows the user to define and manage ECA rules. In this thesis, the features of the Oracle Rules Manager are examined and it is analysed how and to what extent the Rules Manager can function as a domain node in the ECA Framework for the Semantic Web.

Acknowledgements

I would like to thank Prof. Dr. Wolfgang May and Dr. Erik Behrends for their personal and scientific supervision during my course of studies and the becoming of this thesis.

I appreciate their adjuvant comments and consultations, technical support and the invaluable recommendations I received.

Carsten Gottschlich

Göttingen, January 2007

Contents

Chapter 1

Introduction

1.1 Motivation

The Semantic Web as it has been envisioned by Tim Berners-Lee in the mid-90s [26] is slowly becoming reality [39].[1] The inventor of the World Wide Web expressed his idea of the Semantic Web as "a web of data, in some ways like a global database" [14] [15] [16]. Documents and resources of the World Wide Web shall be extended by metadata that bestow them with a well-defined meaning. In this way, more and more of what previously was only human-readable becomes machine-understandable. Vocabularies of metadata, referred to as ontologies, are developed in order to provide foundations for communication and reasoning.

But the Web of today is still far away from being query-able like a global database. In reality there exists a multitude of data formats, languages, schemas and ontologies, and the Semantic Web is composed of innumerable heterogeneous nodes that provide different resources and different behaviour. For the actualisation of Tim Berners-Lee's vision, the gap torn apart by the heterogeneity needs to be bridged. For this purpose, "A General Framework for Evolution and Reactivity in the Semantic Web" was designed and presented in [1]. This framework is based on *Event-Condition-Action (ECA) rules* which enable nodes to provide behaviour and reactivity, and to propagate knowledge.

With the release of version 10g Oracle has enhanced its database software by adding a Rules Manager package that allows the user to define and manage ECA rules. In this thesis, the features of the Oracle Rules Manager are examined and it is analysed how and to what extent the Rules Manager can function as a domain node in the ECA Framework for the Semantic Web. Hence, the motivation for this thesis is to provide a little mosaic piece on the road towards the Semantic Web.

[1] For instance, the German Government decided in December 2006 to promote a project for the development of a semantic search engine called Theseus [40].

1.2 Structure of the Thesis

The next chapter starts with a general consideration of Event-Condition-Action rules and continues with a thorough investigation of how the components of ECA rules are implemented by the Rules Manager. The features and possible options for specifying simple and complex event structures, rule conditions and rule actions are explored.

In Chapter 3, we undertake an elaborate inspection of the Rules Manager's internal representation within the relational database. For this purpose, two scenarios containing all basic ingredients of ECA rules, that have to be mapped onto SQL tables, objects and triggers, are simulated.

The focus of Chapter 4 is laid on the condition part of ECA rules. An example scenario of a conceived airline is designed and all major cases of rule condition specifications are perused.

Chapter 5 begins with a concise introduction to the origins and basic principles of the Semantic Web, followed by an outline of the ECA Framework for Evolution and Reactivity. Afterwards, we resume with the airline scenario that was developed in Chapter 4 and incorporate it into the ECA Framework. This aim is realised by designing a wrapper software that connects a node based on the Oracle Rules Manager with a domain broker introduced in [12].

Finally, this thesis conludes with an outlook and suggestions for improvement of the Rules Manager.

Chapter 2

Event-Condition-Action Rules

In this chapter, we begin with a general description of ECA rules. Next, we survey how event structures can be specified in Oracle, what properties they can have and how they are managed. Then, we scrutinise the possibilities of defining rule conditions and rule actions.

2.1 Fundamentals of ECA Rules

In general, Event-Condition-Action rules have the following form:

ON	<event>
IF	<condition>
THEN	<action>

An event instance occurs at a certain point in time and conveys some pieces of information. The event properties are accessed in the condition part and the condition evaluates either to true or to false. If the condition is complied with, then a predefined action is executed. Alternatively, if the condition evaluates to false, then no action is taken.

In the ECA Framework (cf. Section 5.2), the rule condition consists of two components: a query part and a testing part. The query part obtains additional static information e.g. from a local database and the subsequent testing part includes information both from the event properties and the queries for checking the condition and evaluating to true or to false.

Emphasising this query part, the general form of Event-Condition-Action rules can be described as follows:

ON	<event>
AND	<query additional information>

IF	<condition>
THEN	<action>

Although the query part is only a component of ECA rules in the ECA Framework [1] and not a component of all ECA rules in general, an implementation of this part suggests itself for a relational database like the Oracle software. The Rules Manager is lacking this feature in its current release.

Now we regard a concrete example of an ECA rule. For instance, the event could be the booking of a flight and have properties like a start airport and a destination airport among others. An airline could have rules that trigger the sending of additional information and promotional offers depending on the flight destination. For example, one rule condition could check whether the destination equals "Oslo" and if so, the action might be to offer a Norwegian language course and to send information about famous sights in Oslo to the customer who booked the flight.

ON	flight booking event
IF	destination airport = 'Oslo'
THEN	offer a Norwegian language course and send information about the famous Vigeland Sculpture Park in Oslo

In the next section, we study the possibilities and limitations of defining event structures for the Rules Manager and subsequently we explore how event instances are managed.

2.2 Events in Oracle

Before being able to deal with events, Oracle has to learn about the structure of a potentially occurring event. This structure of an event corresponds to the concept of a "class" in object-oriented programming while a concrete event corresponds to the concept of an "instance". Before we concern with the details of defining event structures in the next section, we aim to gain an overview of the concepts of the Rules Manager.

ECA rules are managed in groups or sets of resembling rules. Such a set of similar rules is called a *rule class*. There are two essential commonalities of all rules of a rule class. First, they are applied to event instances of the same event structure and secondly, they share the same user-defined action. On the other hand, they distinguish itself from each other in the condition part and each rule can pass different rule-specific values to the action, so that the same action adopts a different behaviour depending on which rule triggers the action. In Section 2.3 we will discuss this in detail.

Event instances, e.g. originating from applications or other internet nodes, are processed by calling `DBMS_RLMGR.PROCESS_RULES`. This function processes one event instance for one rule class. If an event instance shall be processed by multiple rule classes, the function has to be called multiple times. Processing means that the event instance is sequently applied to all rules of the rule class until it is consumed. Details about the consumption and the order in which rules are matched follow in Section 2.4.

2.2.1 Primitive Events

The structure of a simple or *primitive event*[1] is defined by a set of attributes that describe the specific features of this event class.

There are two ways in which a structure for a simple event can be defined. One possibility is given by defining an Oracle object type and the other is by using an existing database table and mapping it to an event structure. The following example illustrates the first way.

Primitive Events Defined by Object Types

```
CREATE TYPE AddAirportEvent AS OBJECT (
 ICAOcode VARCHAR2(4),
 IATAcode VARCHAR2(3),
 name VARCHAR2(50),
 city VARCHAR2(35),
 province VARCHAR2(32),
 country VARCHAR2(4));
);
```

The name of the event structure, `AddAirportEvent`, can later be referenced for defining rule classes and rules. While some properties are self-explanatory, it should be mentioned that the codes of ICAO (International Civil Aviation Organization, an agency of the United Nations) [3] and IATA (International Air Transport Association) [4] uniquely identify an airport. In the following example, we create an instance of this object type by using its constructor function. This object type instance represents a concrete event.

```
AddAirportEvent ('EDDF', 'FRA',
                 'Frankfurt International Airport',
```

[1] We consider "primitive event", "simple event" and "atomic event" as equivalent terms in this thesis.

```
                'Frankfurt am Main', 'Hessen', 'D'
);
```

Assuming that this event instance is processed by a rule class with name `observeAirports`, then the call of `DBMS_RLMGR.PROCESS_RULES` has the following form:

```
BEGIN

 DBMS_RLMGR.PROCESS_RULES (
  rule_class => 'observeAirports',
  event_inst => AddAirportEvent.getVarchar(
                'EDDF', 'FRA',
                'Frankfurt International Airport',
                'Frankfurt am Main', 'Hessen', 'D'),
  event_type => 'AddAirportEvent'
 );

END;
```

Alternatively, `AnyData.convertObject` can be used and the following is equivalent to the previous call.

```
BEGIN

 DBMS_RLMGR.PROCESS_RULES (
  rule_class => 'observeAirports',
  event_inst => AnyData.convertObject(
                 AddAirportEvent.getVarchar(
                  'EDDF', 'FRA',
                  'Frankfurt International Airport',
                  'Frankfurt am Main', 'Hessen', 'D'))
 );

END;
```

Obviously, the structure of a primitive event instance starts with the name of the event structure, followed by the values of the properties which individually are in quotation marks and all values together are in parentheses, and it closes with a semicolon. The sequence of the values correlates with

the sequence of the properties in the definition of the event structures, so that "EDDF" is the ICAO code of the airport, "FRA" is the IATA code and so on.

DML Events

The second possibility of defining an event structure is to use an already existing table. The event structure obtains the same set of attributes from the table. *Data Modification Language (DML) events* are automatically raised whenever a DML operation is executed on the referenced table. Release 10g supports only INSERT operations. UPDATE and DELETE operations will probably[2] be supported in future releases. Rule classes that use DML events have to be taught this behaviour by setting the rule class property <simple dmlevents="I"/> as described in Section 2.3 which discusses rule classes. Implicitly, for a rule class which use DML events, a trigger is created that raises an according event for each `INSERT` operation.

For providing an example, we begin with defining a table. Here, a table for storing flights is used.

```
CREATE TABLE flights (
  customerID NUMBER(10),
  airline VARCHAR2(20),
  departureCity VARCHAR2(30),
  arrivalCity VARCHAR2(30),
  departureDate DATE,
  arrivalDate DATE
);
```

In the next step, we execute a PL/SQL block that creates a primitive event structure with the name `AddFlight` and then maps the attributes of our previously defined table `flights` onto this structure. This mapping is demanded by the line `tab_alias => exf$table_alias('flights')`. The attribute name `f` could later play a role for the specification of the rule condition, if this primitive DML event is part of a composite event. In that case, properties like the customer ID can be referenced via `f.customerID`.

```
BEGIN
```

[2]Comments in the installation scripts indicate that the support of UPDATE and DELETE operations for generating DML events is planned.

```
 DBMS_RLMGR.CREATE_EVENT_STRUCT (
  event_struct => 'AddFlight'
 );
 DBMS_RLMGR.ADD_ELEMENTARY_ATTRIBUTE (
  event_struct => 'AddFlight',
  attr_name => 'f',
  tab_alias => exf$table_alias('flights')
 );

END;
```

Now, we can generate a DML event by inserting a data record into the table `flights`. As we will see in Chapter 3 which investigates the internal representation of the Rules Manager, a database trigger watches out for INSERT operations and raises the according events.

```
INSERT INTO flights (customerID, airline,
                     departureCity, arrivalCity,
                     departureDate, arrivalDate)
VALUES (12345, 'ABC Air',
        'London', 'Madrid',
        '23-May-2007 14:30:00', '23-May-2007 15:20:00');
```

The example above spawns a DML event that has all six attribute values of the INSERT statement. From the rule class' point of view, this is equivalent to the following primitive event instance defined by using an object type[3].

```
CREATE TYPE AddFlightEvent AS OBJECT (
  customerID NUMBER(10),
  airline VARCHAR2(20),
  departureCity VARCHAR2(30),
  arrivalCity VARCHAR2(30),
  departureDate DATE,
  arrivalDate DATE
);

AddFlightEvent (12345, 'ABC Air', 'London', 'Madrid',
                '23-May-2007 14:30:00',
```

[3]The name 'AddFlightEvent' instead of 'flights' was only chosen for reasons of consistency.

```
                '23-May-2007 15:20:00'
);
```

Both event instances described above have the same name, the same attributes, and the attributes have the same values. Being processed by a rule class, they would have exactly the same effect regardless whether the instance was generated using an object type or as a DML event. Having regarded these two ways of defining and producing primitive events, we will now consider a special kind of primitive events, XML events.

XML Events

As XML events we consider primitive events with at least one attribute of Oracle's XMLType. The primitive event can have other non-XMLType attributes or just consist of one single XMLType attribute.

```
CREATE or REPLACE TYPE XMLEvent AS OBJECT (
  doc sys.XMLType
);
```

An XML event seems to be conspicuously different from a normal primitive event since the latter has a fixed nature, because the structure of a primitive event has to be defined before the creation of rule classes, whereas the XML event can possibly have any conceivable shape. But this seemingly vast flexibility of XML events reaches its boundaries when we come to the rule condition part. Because the rules (including the condition part) are specified before the occurrence of XML events, and for being able to formulate reasonable conditions on XML events it is necessary to have information about the XML structure in advance. In this way XML events forfeit a major possible advantage in comparison to standard primitive events.

For the illustration of an XML event instance, we converted the acquainted example of a flight booking.

```
XMLBookingEvent (
 sys.XMLType('<bookingEvent>
               <customerID>12345</customerID>
               <airline>ABC Air</airline>
               <depCity>London</depCity>
               <arrCity>Madrid</arrCity>
               <depDate>23-May-2007 14:30:00</depDate>
               <arrDate>23-May-2007 15:20:00</arrDate>
```

```
            </bookingEvent>')
);
```

In Section 2.3.1, we will resume this discussion when we survey rule conditions for XML events. Next, we address the constitution of composite events.

2.2.2 Composite Events

The aforementioned primitive event structures can be combined in order to define structures for composite events. A *composite event* structure consists of two or more already defined primitive event structures. The Rules Manager fulfils the task of collecting the individual primitive events that will usually occur at different points in time and joins them together so that they can be processed as one composite event. In the following example, we combine a flight booking event and a hotel booking event. A possible join condition for both events could be an equality of the customerID. First, we define two primitive event structures using Oracle's object type and then we compose both to one new structure.

```
CREATE or REPLACE TYPE flightBookingEvent AS OBJECT (
  bookingID NUMBER(10) PRIMARY KEY,
  flightID VARCHAR2(6),
  customerID NUMBER(10),
  price NUMBER(6,2),
  class VARCHAR2(20)
);

CREATE or REPLACE TYPE hotelBookingEvent AS OBJECT (
  customerID NUMBER(10),
  hotelID NUMBER(10),
  checkOut DATE,
  checkIn DATE,
  price NUMBER(6,2),
  options VARCHAR2(100)
);
```

These two primitive event structures are composed in the next step. The definition of a composite event structure is a plain enumeration of two or more primitive event structures. Already defined composite event structures cannot be integrated into the definition of a new composite event structure, so that nesting is not allowed by the Rules Manager.

A composite event structure can be regarded as a *set* of primitive event structures which are not connected in a Boolean way. The formulation of connections via Boolean operators takes place in the definition of rule conditions which we discuss in Section 2.3.1

```
CREATE or REPLACE TYPE compositeBookingEvent AS OBJECT (
  f flightBookingEvent,
  h hotelBookingEvent
);
```

The Oracle Rules Manager collects all occurring primitive events and stores them as long as they are not consumed and their duration is not expired. For example, first a flight booking event might occur. The customer with the ID 12345 books a ticket for the flight with the ID 5582. The seat costs €129.95 and is in the economy class.

```
flightBookingEvent (4733, 5582, 12345, 129.95, 'economy');
```

This event instance will be stored and some time later, a hotel booking event generated by the same customer could occur. The following instance of a `hotelBookingEvent` assumes the booking of a single room with breakfast for €149.95 in the hotel with the ID 553 from May 23rd to May 25th 2007.

```
hotelBookingEvent (12345, 553, '23-May-2007 9:00:00',
                   '25-May-2007 9:00:00', 149.95,
                   'single room, breakfast inclusive'
);
```

If the instance of `hotelBookingEvent` is processed by a rule class that uses the above defined composite event structure `compositeBookingEvent`, and the rule class has previously processed and stored the instance of `flightBookingEvent`, then the Rules Manager detects that both together build an instance of `compositeBookingEvent` and attempts to match the rules of the rule class. The sequence in which the primitive events occur is irrelevant for building the composite event.

In the succeeding section about rules, we will examine in which ways conditions can be specified accessing the event properties. Generally, if a new primitive event arises and is processed by a rule class using composite events, then the new event instance is combined with the already stored primitive

events in order to build a composite event that can be matched against the rules of the rule class until it is consumed. If the newly added primitive event remains unconsumed and no rule was triggered during the attempt to match the rule conditions, then the next combination of building a composite event with the newly added primitive event and the stored primitive events will be tried. This process continues until the event is consumed or all possible combinations of joining the existing primitive events to a composite event have been tried. In the latter case the new primitive event will be stored taking into account the specified duration. The sequence in which primitive events are used for building composite events on the one hand, and the sequence in which the rules are matched on the other hand can be defined for a rule class as described in Section 2.4 which analyses the event management of the Rules Manager.

2.3 Rules in Oracle

Oracle manages rules in groups of similar rules. Such a group of rules is called a *rule class*. As described at the beginning, the Oracle Rules Manager is able to handle rules that can be specified in Event-Condition-Action (ECA) rule semantics and accordingly the definition of a rule in Oracle requires three parts: An event part, a condition part and an action part.

All rules of a rule class automatically have the same event part and the same action part, but each rule is characterised by a different condition part[1].

The event part of a rule class definition refers to an already defined event structure for primitive or composite events. In the following example we define a structure for a simple booking event and afterwards we create a rule class that uses this event structure. This is done by the procedure `DBMS_RLMGR.CREATE_RULE_CLASS`. The parameter `rule_class` sets the name of the rule class and has to be unique because a table with that name is created. `event_struct` references to an already existing event structure, e.g. an appropriate type, and `action_cbk` defines the name of the action procedure. These three parameters are required for all rule classes. Additional optional parameters are:

- `actprf_spec` enables the specification of action preferences. If a rule condition is matched, then the triggered action can access these rule-specific parameters. In the example below, each rule has a different

[1] It is not forbidden to have two rules with the same condition part in one rule class, but in general it presumably is not desired because in case of exclusive event consumption, one of the two rules will never be matched. The decision which rule might be matched by an event and which one will always be ignored is made by the conflict resolution criterion as described in Section 2.4 about event management. In case of shared event consumption, the action will not be triggered or it will be triggered twice.

destinationLanguage. One or more action preferences can be specified and in the created rule class table, each parameter is represented by one column. The values of these parameters are individually specified for each rule at the time the rule is added to the rule class.

- rlst_viewnm creates a result view that shows matching events and rules of a session.
- rlcls_prop can be used for specifying properties of the rule class. Oracle provides a simple XML syntax for this purpose. A detailed examination of the possible options ensues in Section 2.4 about event management. An XML Schema definition can be found in [2]. This parameter has the default value <simple/> and for rule classes that deal with composite events at least rlcls_prop => '<composite/> ' has to be declared.

```
CREATE TYPE bookingEvent AS OBJECT (
 customerID NUMBER(10),
 startCity VARCHAR2(50),
 destinationCity VARCHAR2(50),
 departure DATE,
 arrival DATE
);

BEGIN

 DBMS_RLMGR.CREATE_RULE_CLASS (
  rule_class => 'courseOfferClass',
  event_struct => 'bookingEvent',
  action_cbk => 'courseOfferAction',
  actprf_spec => 'destinationLanguage VARCHAR2(20)'
 );

END;
```

The above-named command DBMS_RLMGR.CREATE_RULE_CLASS creates a relational table that has the same name as the rule class, here courseOfferClass, in which rules can be inserted, and it creates a skeleton procedure with the name courseOfferAction that has the following shape at this time:

```
PROCEDURE courseOfferAction (rlm$event bookingEvent,
                             rlm$rule courseOfferClass%ROWTYPE)
IS

BEGIN
 null;

END;
```

This template can now be replaced with the intended procedure. In the succeeding example we imagine that the airline wants to offer all customers a language course that fly to a country where a language is spoken that is different from the customer's native tongue. We assume that the first language of the customer is stored in the table `customers` and that the actual posting of the offer is done by a function `offerLanguageCourse(cid NUMBER(10), lang VARCHAR2(20))` which takes the customer ID and the destination language as parameters.

The following procedure `courseOfferAction` is called with two arguments. `rlm$event bookingEvent` is an instance of the event structure bookingEvent and all event properties (`customerID`, `startCity`, `destinationCity`, `departure`, `arrival` and `rlm$crttime`) can be accessed by the procedure. `rlm$rule courseOfferClass%ROWTYPE` is one row of rule class table `courseOfferClass` and represents that rule which triggered the action. The row consists of `rlm$ruleid`, `rlm$rulecond`, `rlm$ruledesc` and in our example, the additional parameter for action preferences, `destinationLanguage`.

```
CREATE OR REPLACE PROCEDURE courseOfferAction
                             (rlm$event bookingEvent,
                              rlm$rule courseOfferClass%ROWTYPE)
IS

BEGIN

 DECLARE
  cid NUMBER(10);
  lang1 VARCHAR2(20);
  lang2 VARCHAR2(20);

 BEGIN

  cid := bookingEvent.customerID;
```

```
  lang1 := rlm$rule.destinationLanguage;
  SELECT language
  INTO lang2
  FROM customers
  WHERE customerID=cid;

  IF lang1 != lang2 THEN

   offerLanguageCourse(cid, lang1);

  END IF;

 END;

END;
```

Now we generate a few example rules and add them to the rule class. There are two ways in which rules can be specified. One is to use the procedure `DBMS_RLMGR.ADD_RULE` and the other is to directly insert the new rule into the rule class table which we address afterwards. The procedure `DBMS_RLMGR.ADD_RULE` requires at least the three parameters: `rule_class` indicates to which rule class the rule is added, `rule_id` which has to be unique, and `rule_cond` which describes the rule condition. This condition is specified in XML. The root element is <`condition`> which can be omitted for rule classes using primitive events. A detailed description follows in Section 2.3.1 and additional examples can be found in Chapter 4. The XML Schema definition is given in [2].

The idea of the first rule in the following example is to offer somebody, who books a flight to Oslo, a language course for Norwegian, if he or she does not speak Norwegian as native language, which is checked in the action part.

```
BEGIN

 DBMS_RLMGR.ADD_RULE (
  rule_class => 'courseOfferClass',
  rule_id => 'RULE0001',
  rule_cond => '<condition>
                 destinationCity=''Oslo''
                </condition>',
  actprf_nml => 'destinationLanguage',
  actprf_vall => 'Norwegian'
```

```
 );

END;
```

In the example above, `actprf_nml` is a list of action preference names for which values will be assigned through the `actprf_vall` argument [10].

```
BEGIN

 DBMS_RLMGR.ADD_RULE (
  rule_class => 'courseOfferClass',
  rule_id => 'RULE0002',
  rule_cond => '<condition>
                destinationCity=''Helsinki''
               </condition>',
  actprf_nml => 'destinationLanguage',
  actprf_vall => 'Finnish'
 );

 DBMS_RLMGR.ADD_RULE (
  rule_class => 'courseOfferClass',
  rule_id => 'RULE0003',
  rule_cond => '<condition>
                destinationCity=''Stockholm''
               </condition>',
  actprf_nml => 'destinationLanguage',
  actprf_vall => 'Swedish'
 );

END;
```

Instead of using `DBMS_RLMGR.ADD_RULE` it is possible to directly insert rules into the rule class table. This has exactly the same effect as the above-mentioned statements.

```
INSERT INTO courseOfferClass (rlm$ruleid, rlm$rulecond,
                              destinationLanguage)
VALUES ('RULE0001',
        '<condition>destinationCity=''Oslo''</condition>',
        'Norwegian'
```

```
);

INSERT INTO courseOfferClass (rlm$ruleid, rlm$rulecond,
                              destinationLanguage)
VALUES ('RULE0002',
        '<condition>destinationCity=''Helsinki''</condition>',
        'Finnish'
);

INSERT INTO courseOfferClass (rlm$ruleid, rlm$rulecond,
                              destinationLanguage)
VALUES ('RULE0003',
        '<condition>destinationCity=''Stockholm''</condition>',
        'Swedish'
);
```

With this knowledge of how rule classes are created and how rules are managed by the Rules Manager in background, next we scrutinize Oracle's implementation of the second component of ECA rules: the rule condition.

2.3.1 Rule Conditions

The rule condition part is an expression that accesses attributes of the event instance and then evaluates either to true or to false. This evaluation of the whole rule condition proceeds by evaluating Boolean values for each attribute that is accessed and then typically combining them to a complex Boolean expression which is constructed using the equivalents of the Boolean operators AND, OR and NOT.

Rule Conditions for Primitive Events

Rule conditions for primitive events usually access some attributes of the event and compare them to predefined values. Comparison operators are equal "`=`" and not equal "`!=`", less than "<" and more than "`'>`", less or equal than "`<=`" and more or equal than "`>=`", and "`IS NULL`" and "`IS NOT NULL`". These atomic comparison elements consisting of a predefined value, a comparison operator and an event attribute can be combined with the above-mentioned Boolean operators. The following are two simple condition examples applied to our standard event example of a flight booking. The first condition evaluates to true for all flight bookings from Frankfurt to London, the second for all flight bookings from or to Oslo.

```
CREATE TYPE bookingEvent AS OBJECT (
```

```
 customerID NUMBER(10),
 startCity VARCHAR2(50),
 destinationCity VARCHAR2(50),
 departure DATE,
 arrival DATE
);
```

The following two examples illustrate possible rule conditions (`rlm$rulecond`) that access the properties of a `bookingEvent`.

```
<condition>
 startCity = ''Frankfurt'' AND destinationCity = ''London''
</condition>
```

```
<condition>
 startCity = ''Oslo'' OR destinationCity = ''Oslo''
</condition>
```

Rule Conditions for Composite Events

Rule conditions for composite events can use all aforementioned comparison operations and Boolean operators that combine these comparison elements. Moreover, Oracle provides a simple XML syntax for setting the individual primitive events into relation with each other and formulating conditions on these relations. Primitive events can be joined by an `and` element which corresponds to the Boolean operator AND, they can be connected by an `any` element which corresponds to the Boolean operator OR, and they can be endued with a `not` element which is a logical negation. Furthermore the Rules Manager syntax supplies a `notany` element, which combines the `not` and `any` elements and evaluates to true in case of the non-existence of all primitive events specified within.

The following is a simple example of a rule condition on a composite event structure using the `and` element.

```
CREATE or REPLACE TYPE flightBookingEvent AS OBJECT (
  bookingID NUMBER(10) PRIMARY KEY,
  flightID VARCHAR2(6),
  customerID NUMBER(10),
  price NUMBER(6,2),
  class VARCHAR2(20)
```

```
);

CREATE or REPLACE TYPE hotelBookingEvent AS OBJECT (
  customerID NUMBER(10),
  hotelID NUMBER(10),
  checkOut DATE,
  checkIn DATE,
  price NUMBER(6,2),
  options VARCHAR2(100)
);

CREATE or REPLACE TYPE compositeBookingEvent AS OBJECT (
  f flightBookingEvent,
  h hotelBookingEvent
);

<condition>
 <and join="f.customerID = h.customerID">
  <object name="flightBookingEvent"/>
  <object name="hotelBookingEvent"/>
 </and>
</condition>
```

Detailed examples for all these logical connectives and their possible combinations will follow in Chapter 4 which discusses an example scenario of an airline.

Rule Conditions for XML Events

As described in Section 2.2.1 about the event structure, an XML event has at least one attribute of Oracle's XMLType. In Oracle, there are two functions for accessing the XML document that can be used within the rule condition, `EXISTSNODE` and `EXTRACT`, and both of them use XPath [41] expressions.

The XML Path Language (XPath) is the standard language for addressing and selecting parts of XML documents. XPath is based on the UNIX directory notation and it regards an XML document as a tree that can be navigated. There are basically four kinds of data types that can be returned if an XPath expression is applied to an XML document.

- A node or a set of nodes.
- A string, if a string function is used.
- A number, if the function `sum()` is used.

- A boolean value.

`EXISTSNODE` and `EXTRACT` are the two functions which are available for the specification of rule conditions. `EXISTSNODE` returns 1, if the applied XPath expression has one or more result nodes, and 0, if not. `EXTRACT` returns these result nodes for the XPath expression as an XMLType fragment, or it returns NULL, if the application does not result in any nodes.

Obviously all XPath expressions have to be specified at the time when the rule is added to the rule class, which consequently means that pieces of the structure of a possibly occurring XML event have to be known in advance.

In the following example, we assume that an XML event is generated every time a customer books a flight. First, we define the primitive event structure consisting of one XMLType property. Next, we create the rule class that uses this XML event structure. Then, we replace the template procedure with our own procedure which outputs a line if it is executed. Subsequently, we insert a rule into the rule class table whose condition part checks whether the fragment `/bookingEvent/destinationCity` exists in the XML document and if the string value is equal to "Oslo".

Regarding the specification of the strings like `rlm$rulecond`, we have to take into consideration that we need two single quotes in order to represent an apostrophe within a string. Two single quotes are different from a double quote which we need later for defining the value of an XML element attribute [9].

```
CREATE or REPLACE TYPE bookingEventXML AS OBJECT (
  doc sys.XMLType
);

BEGIN

DBMS_RLMGR.CREATE_RULE_CLASS (
 rule_class => 'christmasSpecialClass',
 event_struct => 'bookingEventXML',
 action_cbk => 'sendChristmasSpecial'
);

END;


CREATE OR REPLACE PROCEDURE sendChristmasSpecial (
               rlm$event bookingEventXML,
               rlm$rule christmasSpecialClass%ROWTYPE)
```

```
IS

BEGIN

 DBMS_OUTPUT.PUT_LINE('Christmas Action triggered.');

END;


INSERT INTO christmasSpecialClass (rlm$ruleid, rlm$rulecond)
VALUES ('RULE0001',
        '<condition>
          EXISTSNODE(doc,
            ''/bookingEvent/destinationCity'') = 1
          AND EXTRACT(doc,
            ''/bookingEvent/destinationCity/text()=Oslo'')
          IS NOT NULL
         </condition>'
);
```

Now everything is prepared for processing events. For testing purposes, we generate two event instances. The first event instance does not fulfill the rule condition, wheras the second event instance meets the requirements and triggers the action. The function `PROCESS_RULES` takes a name of a rule class and an event instance as arguments, and matches the event against the rules until the event instance is consumed. Details about this process follow in Section 2.4 about event management and the sequence of rule matching.

```
BEGIN

 DBMS_RLMGR.PROCESS_RULES (
  rule_class => 'christmasSpecialClass',
  event_inst => AnyData.convertObject(
                         bookingEventXML(
                          XMLType(
  '<bookingEvent>
    <customerID>12345</customerID>
    <startCity>Madrid</startCity>
    <destinationCity>London</destinationCity>
    <departure>31-Jan-2007 10:15:00</departure>
    <arrival>31-Jan-2007 12:00:00</arrival>
   </bookingEvent>
```

```
  ''''));


 DBMS_RLMGR.PROCESS_RULES (
  rule_class => 'christmasSpecialClass',
  event_inst => AnyData.convertObject(
                          bookingEventXML(
                           XMLType(
  '<bookingEvent>
    <customerID>12345</customerID>
    <startCity>Berlin</startCity>
    <destinationCity>Oslo</destinationCity>
    <departure>23-Dec-2006 14:30:00</departure>
    <arrival>23-Dec-2006 16:15:00</arrival>
   </bookingEvent>
  ''''));

END;
```

In the next section, we survey an example of specifying rule conditions with spatial attributes.

Rule Conditions with Spatial Attributes

If an event has at least one attribute of the MDSYS.SDO_GEOMETRY type, then spatial predicates can be specified within the rule condition. All functions in the MDSYS.SDO_GEOM package are available for this purpose.

For example in case of a customer inquiry event, SDO_NN(geometry1, geometry2, param [, number]) could be used in order to find the three nearest airports for a customer in relation to her or his residence, and the booking engine of the airline could automatically suggest these as favoured departure airports.

Another example application could be a rule class that sends out information about historic sites and other places of interest for sightseeing. One rule in this rule class might post information about the city of Göttingen and its landmark, the Gänseliesel fountain, if a passenger arrives at an airport within a 100 mile radius. The city of Göttingen has the geo coordinates 51 32' North and 9 56' East [7]. The following rule condition accesses a special booking event that has at least two attributes: the customer ID and the location of arrival in the form of Oracle's `SDO_GEOMETRY`. Each rule could have the tourist feature as an action preference. If an event matches a rule condition and triggers an action, then the action procedure could retrieve the email address from the customer table using the customer ID and send

out the appropriate information according to the action preference. Below are the necessary event parts and the rule condition. The value "2001" as SDO_GTYPE indicates a point and the value "8307" as SDO_SRID specifies the so called "Longitude / Latitude (WGS 84)" coordinate system which denominates the World Geodetic System from 1984 [8].

```
CREATE or REPLACE TYPE bookingEvent AS OBJECT (
 CustId NUMBER,
 destinationLocation MDSYS.SDO_GEOMETRY
);
```

```
<condition>
 SDO_WITHIN_DISTANCE (destinationLocation,
  SDO_GEOMETRY(2001, 8307,
   SDO_POINT_TYPE(51.533333, -9.933333, NULL), NULL, NULL),
 'distance=100 units=mile') = 'TRUE'
</condition>
```

For further information on spatial predicates please refer to the "Spatial User's Guide and Reference 10g Release 2 (B14255)" [6].

2.3.2 Rule Action

If an event instance satisfies the condition of a rule, then the procedure associated with the rule class is triggered. This user-defined procedure is coded in PL/SQL and can potentially spawn rule actions of all kinds. Below you will find a list of some possible options for rule actions. Please note that all combinations and quantities of executions of PL/SQL commands are conceivable.

- The rule action could simply be the output of a message using DBMS_OUTPUT.PUT_LINE().
- It is self-evident that an Oracle procedure can execute every thinkable SQL query and consequently all sorts of data retrieval, data manipulation or even data definition activities. Any deeds of inserting data may automatically raise further DML events and in future releases this might also be true for update and delete operations.
- The function DBMS_PIPE can be used in order to pass data to operating-system commands.
- UTL_FILE enables reading and writing in files.

- With `UTL_HTTP` the procedure can send or retrieve data via the http protocol.
- The function `UTL_SMTP` allows the programmer of the rule action to directly send mails.
- The action can raise new events and apply them to other rule classes by using the functions `DBMS_RLMGR.ADD_EVENT` or `DBMS_RLMGR.PROCESS_RULES`. This behaviour can be regarded as an Event-Condition-Event chain instead of a standard ECA rule.

A detailed description of all additional possibilities of rule actions can be found in the Oracle document "PL/SQL User's Guide and Reference" [9].

2.4 Event Management

2.4.1 Consumption of Events

Each rule class has a consumption property which determines whether an event can be used again for other rules of the rule class or it is accounted as consumed and has to be removed once it matches a rule. The default policy is "shared" which means that the event can be reapplied to all other rules of the same rule class. If the consumption property is set to "exclusive", then an event can only be applied to one rule and it is removed afterwards.

Partial Consumption

If a composite event is consumed by a rule of a rule class with an "exclusive" consumption property, then all primitive events that formed the composite event are consumed. Instead of a complete consumption, it is possible to fine-tune a partial consumption. Using the previous example of a composite event, we can define a consumption property that removes only the `hotelBookingEvent` from the system, while the `flightBookingEvent` remains in the storage. As announced in Section 2.3, the rule class adopts this behaviour by setting the parameter `rlcls_prop` at the time of the rule class creation.

```
...
rlcls_prop => '<composite consumption="shared">
                <object type="flightBookingEvent"
                        consumption="shared">
                <object type="hotelBookingEvent"
                        consumption="exclusive">
               </composite>'
```

...

Rule Based Consumption

Rule classes using composite events can have a third option as consumption property: "rule". This option enables the user to postpone the decision which primitive events remain or are removed into the rule action part. It is possible to consume only one of the primitive events that build the composite event, or a few, or all of them. This decision can be made dynamically. The primitive events are removed by using the function `DBMS_RLMGR.CONSUME_PRIM_EVENTS` which returns 1 if all events are successfully consumed and 0, if one or more primitive events could not be removed. Oracle suggests to depend the user-initiated action on the return value of the `CONSUME_PRIM_EVENTS` call.

2.4.2 Ordering and Conflict Resolution

Dealing with rule classes that have a partial or overall "exclusive" consumption policy, it might be very important in which order an event is applied to a rule, and in case of composite events in which order existing primitive events are used in order to merge to a composite event. For example, if there are already two `flightBookingEvent`s in the system and a new `hotelBookingEvent` occurs, we can not predict which `flightBookingEvent` is chosen to build the `compositeBookingEvent` together with the `hotelBookingEvent`. Without the specification of a conflict resolution criterion one of the two available `flightBookingEvent` is chosen at random from the programmer's point of view. The same question emerges if an event, regardless whether primitive or composite, is processed by a rule class that contains more than one rule. If there is no criterion which rule to choose, a decision is made haphazardly. Therefore, if a deterministic behaviour is required, an ordering criterion has to be specified. Potential criteria for assembling composite events are the attributes of the primitive events. Besides the user-defined attributes or the attributes that DML events inherit from a table, every primitive event has a creation time attribute which can be referenced as `rlm$CrtTime`. With the creation time attribute it is feasible to start with the oldest or the newest available primitive event in order to build a composite event. If a rule class has two or more rules, it appears to be reasonable to use the rule ID as ordering criterion which rule is executed first. The following example resumes with the above mentioned composite event consisting of a flight and a hotel booking event.

```
...
rlcls_prop => '<composite ordering="f.rlm$CrtTime,
```

```
                                        h.rlm$CrtTime,
                                        rlm$rule.rlm$ruleid"/>'
...
```

2.4.3 Duration of Events

In a rule class using a primitive event structure, all occurring event instances are consumed immediately after processing regardless whether an event instance triggered a rule action or not. Thus, the specification of a duration concerns only primitive events which are part of a composite event structure, and it relates only to the scope of the rule class that uses this structure. The life cycle of such a primitive event instance begins at the time of its creation and lasts for the specified duration, if it remains unconsumed. There are several ways in which the lifetime of a primitive event can be stipulated.

- Firstly, the shortest life span is call duration. In this case a primitive event or a subset of primitive events in a composite event exists only during the processing of this call. Once the call of `PROCESS_RULES` or `ADD_EVENT` is worked up, the one primitive event or the subset of primitive events with call duration is deleted irrespective if any rule condition was matched.

- Secondly, the event lifetime can take as long as a database transaction. Unconsumed primitive events with transaction lifetime that did not trigger a rule during the transaction are removed from the system so that they can not be part of composite events after the transaction.

- Thirdly, the life span of a primitive event can be as long as a database session. Primitive events with session lifetime can only match rules or form composite events until the `DISCONNECT` takes place. Afterwards unconsumed primitive events are deleted. Session duration and the above mentioned transaction duration can only be specified for composite events, not at the level of primitive events. If one of these two options is used, then it cannot be overridden by specifications at the primitive event level.

- Fourthly, a lifetime can be specified as a number of minutes, hours or days. This period of time is added to the creation date and the result indicates the point in time when the primitive event will be deleted.

- Lastly, if no duration is explicitly specified, then an event has an infinite lifetime. The drawback of this option is that some primitive events may never be part of a composite event that triggers a rule, but they use up resources for storage and attempts of integrating these primitive events into composite events.

Here are syntax examples for all aforementioned cases. Like in Section 2.4.1 about consumption and in Section 2.4.2 about ordering, these are possible values of the `rlcls_prop` parameter.

```
<composite duration="call"/>

<composite duration="session"/>

<composite duration="transaction"/>

<composite duration="5 minutes"/>

<composite duration="10 hours"/>

<composite duration="30 days"/>

<composite/> -- no duration specified, infinite event lifetime
```

Composite events can also have a differentiated duration policy. In this case the duration specified for the composite event is the default duration for all primitive events and it can be overridden by specifications for each primitive event. An exception is a transaction or session duration for the composite event which cannot be overridden.

The composite event in the next example has a default duration of 30 days. This policy is overridden at the primitive event level by setting the lifetime for the `flightBookingEvent` to 8 hours and giving the `hotelBookingEvent` a call duration. Assuming the composite event consists of more than these two primitive events, all other here not explicitly mentioned primitive events have a lifetime of 30 days.

```
<composite duration="30 days">
 <object type="flightBookingEvent" duration="8 hours"/>
 <object type="hotelBookingEvent" duration="call"/>
</composite>
```

In the following section we begin with a general consideration of rules in Oracle and the Rules Manager's method of operation by going through all essential steps of defining and processing rules. After this introduction the chapter closes with two sections that focus on the last two components of ECA rules: the rule condition and the rule action. We examine the possibilities of specifying these parts in the syntax of the Rules Manager.

Chapter 3

Internal Representation of the Rules Manager

In the first section of this chapter, we scrutinize the fundamental structure of the Rules Manager in the relational database. We pursue this target by tracing back the installation process and analysing the installation scripts of the Rules Manager. Knowing the basic structure, we successively simulate two example scenarios, one scenario with primitive events and a second scenario using a composite event structure. After each step, we examine the effects of our actions, what changes are produced in the relational database.

3.1 Fundamental Structure

As the fundamental structure of the Rules Manager we regard those tables that are created during the installation process of the database software. These tables build the foundation for concrete rule classes and event structures. For instance they serve as dictionaries that store names of database objects. In order to examine how the Rules Manager is internally represented in the Oracle database, it is very insightful to analyse the installation scripts.

They can be found in the directory `$ORACLE_HOME/rdbms/admin` and the principal script that controls the installation process of the Rules Manager is called `catrul.sql` which contains statements that execute some other scripts:

```
--- Create object types required for the Rules Manager
@@rultyp.sql

--- Create Rules Manager Dictionary/Static tables
@@rultab.sql
```

```
--- Create Rules Manager Public PL/SQL package specification
@@rulpbs.sql

--- Create Rules Manager Catalog views
@@rulview.sql
```

(i) The first script, `rultyp.sql`, creates a few types and grants execution rights for these types to public. It also creates rlm$table_alias as a synonym for exfsys.exf$table_alias. The table_alias is used for defining DML events, i.e. events that are raised by INSERT statements, or in future releases of the database software, also by UPDATE or DELETE statements.

(ii) The second script, `rultab.sql`, creates eighteen tables that store meta information about rules and rule sets. They serve as the dictionary of the Rules Manager. We will take a closer look at the most important of these tables.

`rlm$eventstruct` is a fundamental table for primitive events. It consists of three columns and two keys. Each event structure of the Rules Manager is a row in this table. An event structure has an owner, a name and a number of properties. The combination of owner and name functions as primary key and as foreign key which connects `rlm$eventstruct` with the table `exf$attrset` of the Expression Filter package. This means that the actual attributes are stored in the table `exf$attrset` with one row per attribute. More details can be found in the script `exftab.sql`. This is a good example of how close the Rules Manager and the Expression Filter are interwoven with each other.

Rule classes that operate with composite events require the table `rlm$rsprimevents`. There is one row in this table for each primitive event that is part of the composite event. Even if a composite event consists of two or more primitive events of the same type, every primitive event has an entry of its own.

The table `rlm$primevttypemap` stores information about properties of primitive event types and in this way it connects the event structures with rule classes. The same event structure can be used as a primitive event structure by different rule classes or for building one or more different composite event structures that can be used by other rule classes. The properties that `rlm$primevttypemap` stores for each rule class include information about the consumption policy and duration of the primitive event.

DML events are automatically generated whenever data of an observed table is modified which means at this point in time that an INSERT statement is executed. Oracle copes with this task by using triggers and these triggers are associated with the corresponding rule classes via the table `rlm$dmlevttrigs`.

The two tables `rlm4j$evtstructs` and `rlm4j$ruleset` manage event structures and rule classes that are created by the Oracle JDeveloper 10g and relates them to the according java packages and java classes.

The principal dictionary table regarding rule classes is called `rlm$ruleset`. For each rule class it keeps track of all the important meta information: `rlm$ruleset` stores among other things the names of dictionaries for event structures and action procedures that are associated with a rule class, it stores the names of the tables in which primitive event objects, expressions and results are collected for a rule class, and it stores rule class properties like consumption policy of events, whether the sequence of primitive events matters or not, and so forth.

(iii) The third sub-script, `rulpbs.sql`, defines and declares all procedures that build the Rule Manager package and the Rule Manager for Java (rlm4j) package. It also states the XML schema for rule class properties and the syntax of rule conditions.

(iv) The fourth sub-script, `rulview.sql`, creates catalog views that allow the user to behold the event structures and rule classes stored in the database, as well as some information on status and privileges. The script accomplishes this by creating views that show information from protected tables and then granting `SELECT` rights on these views to public. For example, the first view is called `USER_RLMGR_EVENT_STRUCTS` and shows data from the table rlm$eventstruct. The view lists the event structure names and the two properties "has timestamp (YES/NO)" and "is primitive (YES/NO)" for all those event structures that are owned by the user who is executing the `SELECT` statement. All views beginning with "USER" display those pieces of information that belongs to the username that executed the "`SELECT * FROM view_name`" command. Here is a complete list of all views that are generated by the script.

USER_RLMGR_EVENT_STRUCTS
ALL_RLMGR_EVENT_STRUCTS
ADM_RLMGR_EVENT_STRUCTS
USER_RLMGR_RULE_CLASSES
ALL_RLMGR_RULE_CLASSES
ADM_RLMGR_RULE_CLASSES
USER_RLMGR_RULE_CLASS_STATUS
ALL_RLMGR_RULE_CLASS_STATUS
ADM_RLMGR_RULE_CLASS_STATUS
USER_RLMGR_PRIVILEGES
ADM_RLMGR_PRIVILEGES
USER_RLMGR_COMPRCLS_PROPERTIES
ALL_RLMGR_COMPRCLS_PROPERTIES

ADM_RLMGR_COMPRCLS_PROPERTIES
USER_RLMGR_ACTION_ERRORS
ALL_RLMGR_ACTION_ERRORS
ADM_RLMGR_ACTION_ERRORS
USER_RLM4J_EVENT_STRUCTS
USER_RLM4J_RULE_CLASSES
USER_RLM4J_ATTRIBUTE_ALIASES

Now we will do a step by step run through of two complete example scenarios. One example features the definition and application of a primitive event structure and the other example avails a composite event structure. After each single step we will examine all SQL tables of the Rules Manager and seek for changes made by INSERTs, UPDATEs or DELETEs, and we will search for new tables, procedures, triggers or objects that might have been created as a result of our recent step.

3.2 Structure of a Scenario with Primitive Events

Step 1: Creating a Primitive Event Structure

First of all, we have to define the structure of a potentially occurring event. Let us assume that in our scenario news can emerge from different sources and they shall be distributed to various target groups of the airline like e.g. customers, administrative personnel, flight personnel, pilots or all personnel. The delivery of these messages shall be prioritised from 1 = high to 3 = low. This information will be represented by events of a specific type:

```
CREATE OR REPLACE TYPE newsEvent AS OBJECT (
 msg VARCHAR2(1000),
 priority NUMBER,
 target VARCHAR2(24)
);
```

This first step only creates a new object type with three attributes. There are no changes in the tables belonging to the Rules Manager.

Step 2: Creating a Rule Class

Next, we create a rule class that uses the above specified event structure. As an action preference we define the variable `smtpServer`, so that in our example, each rule of the rule class decides which SMTP server is used in

Table 3.1: Entry of newsRuleClass in the table rlm$ruleset

RSET_OWNER	VARCHAR2(32)	AIRLINE
RSET_NAME	VARCHAR2(32)	NEWSRULECLASS
RSET_PACK	VARCHAR2(75)	RLM$RULECLS_PACK_91539
RSET_EVENTST	VARCHAR2(75)	NEWSEVENT
ACTION_CBK	VARCHAR2(75)	NEWSACTION
RSET_RSLTVW	VARCHAR2(32)	RLM$SESSRSLTVIEW_91539
RSET_RSLTTAB	VARCHAR2(32)	RLM$SESSRSLTTTAB_91539
RSET_PRMEXPT	VARCHAR2(32)	
RSET_PRMOBJT	VARCHAR2(32)	
RSET_PRMRSLT	VARCHAR2(32)	
RSET_STATUS	NUMBER	0
RSET_PROP	NUMBER	1
RSET_DURMIN	NUMBER	
RSET_DURTCL	VARCHAR2(100)	N/A
RSET_ORDRCL	VARCHAR2(4000)	
RSET_REWOCL	VARCHAR2(4000)	
RSET_STGCLS	VARCHAR2(4000)	
RSET_EQCLS	VARCHAR2(1000)	
RSET_OBJNM	NUMBER	91539

order to distribute the message, depending on the combination of the priority and the target group.

```
BEGIN

DBMS_RLMGR.CREATE_RULE_CLASS (
 rule_class => 'newsRuleClass',
 event_struct => 'newsEvent',
 action_cbk => 'newsAction',
 actprf_spec => 'smtpServer VARCHAR2(50)'
);

END;
```

This second step has a number of consequences which can be conjectured alone by reason of the time consumed for the execution of this command in comparison to other commands. First, we examine the major table for rule

classes, `rlm$ruleset`. The query

```
SELECT * FROM EXFSYS.RLM$RULESET
```

leads to our new entry which is shown in the Table 3.1. The first column states the column name in the `rlm$ruleset` table, the second column shows the data type and the third one displays the value.
The sequence in which all further objects connected with this rule class are created can be anticipated by the content of the table `rlm$rulesetstcode`. It describes the meaning of the status code during the creation process.

RSET_STCODE	RSET_STDESC	RSET_STNEXT
0	VALID	
1	DICTIONARY SETUP	Creating Event Structure Objects
2	EVENT STRUCTURE CREATED	Creating and Configuring Rule Class Repository
3	RULE CLASS TABLE(S) CREATED	Configuring Incremental Results tables
4	INCREMENTAL RESULTS TABLES CREATED	Creating Action Callback Procedure
5	ACTION CALLBACK PROCEDURE CREATED	Creating Rule Set Access Package(s)
6	RULE SET ACCESS PACKAGES CREATED	Creating Expression Filter Indexes for Rule Conditions
7	EXPRESSION FILTER INDEXE(S) CREATED	

When the creation process is completed, RSET_STATUS in the `rlm$ruleset` table is set to 0 (= VALID).

Now, we examine the creation of the event structure. The object type newsEvent which we defined in step one with three attributes was altered by our second step. It received an additional timestamp attribute called RLM$CRTTIME and six new methods. Now it looks like this in detail:

Attributes		
Name	Null?	Type
MSG		VARCHAR2(1000)
PRIORITY		NUMBER
TARGET		VARCHAR2(24)
RLM$CRTTIME		TIMESTAMP(6)

The new methods are the following:

METHOD STATIC FUNCTION GETTIMESTAMPCOMP RETURNS TIMESTAMP			
Argument Name	Type	In/Out	Default?
EXF$TSFORM	VARCHAR2	IN	
MSG	VARCHAR2	IN	
PRIORITY	NUMBER	IN	
TARGET	VARCHAR2	IN	
RLM$CRTTIME	TIMESTAMP	IN	DEFAULT

METHOD FINAL CONSTRUCTOR FUNCTION NEWSEVENT RETURNS SELF AS RESULT			
Argument Name	Type	In/Out	Default?
MSG	VARCHAR2	IN	
PRIORITY	NUMBER	IN	
TARGET	VARCHAR2	IN	
RLM$CRTTIME	TIMESTAMP	IN	DEFAULT

METHOD STATIC FUNCTION GETVARCHAR RETURNS VARCHAR2			
Argument Name	Type	In/Out	Default?
MSG	VARCHAR2	IN	
PRIORITY	NUMBER	IN	
TARGET	VARCHAR2	IN	
RLM$CRTTIME	TIMESTAMP	IN	DEFAULT

METHOD MEMBER FUNCTION GETVARCHAR RETURNS VARCHAR2 METHOD STATIC FUNCTION GETEQUIVQUERY RETURNS VARCHAR2			
Argument Name	Type	In/Out	Default?
EXF$EXPR	VARCHAR2	IN	

METHOD STATIC PROCEDURE DYNAMICEVAL			
Argument Name	Type	In/Out	Default?
EXF$SPEVAL	NUMBER	IN/OUT	
EXF$RID	UNDEFINED	IN	
EXF$SPRED	VARCHAR2	IN	
MSG	VARCHAR2	IN	
PRIORITY	NUMBER	IN	
TARGET	VARCHAR2	IN	
RLM$CRTTIME	TIMESTAMP	IN	DEFAULT

METHOD STATIC PROCEDURE DYNAMICEVAL			
Argument Name	Type	In/Out	Default?
EXF$SPEVAL	NUMBER	IN/OUT	
EXF$RID	UNDEFINED	IN	
EXF$SPRED	VARCHAR2	IN	
EXF$ANYD	ANYDATA	IN	

The implementation of these methods is only in binary format available which is not human readable, so that unfortunately we cannot retrace further details of the implementation.

The function `GETTIMESTAMPCOMP` is used by the constructor function which creates and returns an event instance with a timestamp of the creation time.

The methods `GETVARCHAR` and `DYNAMICEVAL`, both with two different signatures, represent the three ways in which a rule class can process an event instance: firstly, by calling `DBMS_RLMGR.PROCESS_RULES` and `EventName.GETVARCHAR` which converts the event instance attributes to a string of name-value pairs, secondly, by calling `DBMS_RLMGR.ADD_EVENT` which uses the `DYNAMICEVAL` procedure with all event attributes in its signature, and thirdly, by calling `DBMS_RLMGR.PROCESS_RULES` in combination with `AnyData.convertObject` which is handled by the other `DYNAMICEVAL` procedure.

As stated in the general description above, another consequence of promoting the object type `newsEvent` to an event structure is the insertion of a row in table `rlm$eventstruct`, which stores all event structures of the Rules Manager.

EVST_OWNER	VARCHAR2(32)	AIRLINE
EVST_NAME	VARCHAR2(32)	NEWSEVENT
EVST_PROP	NUMBER	3

The event structure property encodes the information that it has a creation timestamp attribute (1) and (+) it can only be primitive (2), so that the value is 1 + 2 = 3. This bitmap encodes only these two attributes and the explanation is stated in the script `rultab.sql`. The combination of owner and name acts both as primary key of the table and as foreign key referencing to the table `exf$attrset` which stores a complete list of attribute sets for Expression Filter, with one row per set.

ATSOWNER	VARCHAR2(32)	AIRLINE
ATSNAME	VARCHAR2(32)	NEWSEVENT
ATSTABTYP	VARCHAR2(75)	EXF$NTT_91507
ATSFLAGS	NUMBER	1

The name `exf$ntt_91507` indicates that we are dealing with a nested table type which handles an unordered set of elements. `exf$ntt_91507` is a multicolumn table and has one column for each attribute.

These changes have been made regarding the event structure. For the storage of rules, a table called `newsRuleClass` was created with the following colums:

Name	Null?	Type
RLM$RULEID	NOT NULL	VARCHAR2(100)
SMTPSERVER		VARCHAR2(50)
RLM$RULECOND		VARCHAR2(4000)
RLM$RULEDESC		VARCHAR2(1000)

With the execution of `DBMS_RLMGR.CREATE_RULE_CLASS`, Oracle generated a dummy procedure called newsAction that does nothing at the moment (action body = "null;") and it shall be replaced by a user-defined action which will happen in the next step.

The third row of Table 3.2 refers to the rule class package `rlm$rulecls_pack_91539` which is a collection of eight methods for managing rules and events: the two procedures `ADD_RULE` and `DELETE_RULE` can be used instead of directly inserting rows into or deleting rows from the rule class table. `RESET_SESSION` makes a clean sweep and resets the current session. A consequence of calling `RESET_SESSION` would be the removal of all primitive events with a session lifetime in a composite event scenario. The

two procedures ADD_EVENT and EXEC_RULES both take an event instance as argument and both are available with two different signatures: VARCHAR2, if the event instance is converted to a long string with name-value pairs of all attributes, or ANYDATA. The last function, CONSUME_EVENT returns a number that indicates whether the consumption was successful or not.

Step 3: Defining the Action

In step three we replace the dummy procedure with a user-defined action. A simple output suffices our testing purposes, so that we can see if the action was duly triggered.

```
CREATE OR REPLACE PROCEDURE newsAction (rlm$event newsEvent,
      rlm$rule newsRuleClass%ROWTYPE) is
BEGIN

 DBMS_OUTPUT.PUT_LINE('News Action triggered.');
 -- Action implementation or forward the task
 -- by calling an appropriate function like:
 -- sendNews(rlm$event.msg, rlm$event.target,
 --          rlm$rule.smtpServer);

END;
```

Step 4: Adding Rules

We merely add a rule and the only alteration in the database is one new row in the newsRuleClass table. This direct method with INSERT INTO is equivalent to the usage of the procedure DBMS_RLMGR.ADD_RULE.

```
INSERT INTO newsRuleClass (rlm$ruleid, rlm$rulecond,
                           smtpServer)
VALUES (
 'RULE01',
 '<condition>priority=1 AND target=''pilots''</condition>',
 'mail42.abcair.com'
);
```

Step 5: Processing Events

Now we process an adequate event instance that matches the condition of our test rule.

```
BEGIN

 DBMS_RLMGR.PROCESS_RULES (
  rule_class => 'newsRuleClass',
  event_inst => newsEvent.getVarchar('Christmas bonus doubled.',
                                     1, 'pilots')
 );
END;
```

The output shows that the action is triggered as expected.
`rlm$sessrsltttab_91539` and `rlm$sessrsltview_91539` are empty. There are no stored events because a rule class using a primitive event structure consumes all event instances immediately, irrespective of whether the event instance fulfilled a condition and triggered an action or not.

Now we move on to a more elaborate scenario with composite events in which the rule class has to store primitive events that potentially might be part of future composite events matching a rule.

3.3 Structure of a Scenario with Composite Events

In our second example, we will discuss only those details of the internal structure that have not been already examined in the first scenario. This especially concerns the storage of unconsumed primitive events for a future processing.

Let us image a simple scenario of flight scheduling. A flight is unambiguously characterised by a flight ID. An `aeroplaneEvent` assigns an aircraft to the flight. This event also contains two pieces of information: Which licence is at least required by the pilot and the copilot and how many stewardesses and stewards are needed for operating the flight. This number shall be two, three or four in our example. Besides, we claim that each flight requires one pilot and one copilot. Each `pilotEvent` or `serviceStaffEvent`s corresponds to attempted assignment of one person to the flight. The scheduling of a flight shall be complete if all requirements are met (rule number 3, 4 and 5 below), and the according action shall be carried out. In reality that would be the actual assignment, in our example it is the insertion of a log entry into a table.

Step 1: Creating a Composite Event Structure

First, we define three different structures of primitive events using Oracle's object types, and based on these primitive event structures we then define our composite event structure which represents a flight scheduling.

```
CREATE OR REPLACE TYPE aeroplaneEvent AS OBJECT (
 flightID NUMBER,
 planeType VARCHAR2(50),
 requiredPilotLicenceClass NUMBER,
 requiredServicePersonnel NUMBER
);

CREATE OR REPLACE TYPE pilotEvent AS OBJECT (
 flightID NUMBER,
 staffID NUMBER,
 pilotLicenceClass NUMBER,
 name VARCHAR2(50),
 position VARCHAR2(7) -- 'pilot' or 'copilot'
);

CREATE OR REPLACE TYPE serviceStaffEvent AS OBJECT (
 flightID NUMBER,
 staffID NUMBER,
 name VARCHAR2(50)
);

CREATE OR REPLACE TYPE FSEvent AS OBJECT (
 plane aeroplaneEvent,
 pilot pilotEvent,
 copilot pilotEvent,
 service1 serviceStaffEvent,
 service2 serviceStaffEvent,
 service3 serviceStaffEvent,
 service4 serviceStaffEvent
);
```

Step 2: Creating a Rule Class

In the following definition, we endow our rule class with an exclusive consumption policy as default value, whereas the `aeroplaneEvent` is specified with a shared consumption, so that even if `pilotEvent`s with an insufficient

licence class occur, the `aeroplaneEvent` is stored in the system, whereas the improper `pilotEvent` is consumed. Initially the following rule class definition contained the name "`FlightSchedulingEvent`" instead of "`FSEvent`". This lead to the error message:

ORA-38417: attribute set AIRLINE.FLIGHTSCHEDULINGEVENT does not exist

The attribute set existed, but it turned out that the name was too long. After renaming the composite event structure to `FSEvent` everything worked fine, and the rule class can be created:

```
BEGIN

DBMS_RLMGR.CREATE_RULE_CLASS (
 rule_class  => 'flightSchedulingRuleClass',
 event_struct  => 'FSEvent',
 action_cbk  => 'flightSchedulingAction',
 rslt_viewnm  => 'flightSchedulingResultView',
 actprf_spec  => 'msg VARCHAR2(100),
                  code NUMBER', -- 0=ok, 1=error
 rlcls_prop  =>
  '<composite consumption="exclusive"
              ordering="plane.rlm$CrtTime,
                        rlm$rule.rlm$ruleid DESC"
              duration="7 days">
    <object type="pilotEvent" duration="5 days"/>
    <object type="serviceStaffEvent" duration="3 days"
            consumption="shared"/>
    <object type="aeroplaneEvent" consumption="shared"/>
   </composite>'
);

END;
```

In comparison to the first example, the new row in the table `rlm$ruleset` has some extra attributes: the order clause for the sequence of rule matching, the duration as a string and in minutes, as well as the names of three tables for managing primitive events. The value of `RSET_PROP`, 61, is added up by different properties of the rule class. It is indexed (1), it uses composite

events (4), sequence is enabled (8), autocommit after each add rule, process rule or delete rule operation is activated (16) and the default consumption is exclusive (32). All these properties (1 + 4 + 8 + 16 + 32) sum up to 61. This bitmap is explained in the comments of the script `rultab.sql`.

RSET_OWNER	VARCHAR2(32)	AIRLINE
RSET_NAME	VARCHAR2(32)	FLIGHTSCHEDULINGRULECLASS
RSET_PACK	VARCHAR2(75)	RLM$RULECLS_PACK_93045
RSET_EVENTST	VARCHAR2(75)	FSEVENT
ACTION_CBK	VARCHAR2(75)	FLIGHTSCHEDULINGACTION
RSET_RSLTVW	VARCHAR2(32)	RLM$SESSRSLTVIEW_93045
RSET_RSLTTAB	VARCHAR2(32)	RLM$SESSRSLTTTAB_93045
RSET_PRMEXPT	VARCHAR2(32)	RLM$PRMEXPRT_93045
RSET_PRMOBJT	VARCHAR2(32)	RLM$PRMEVENTS_93045
RSET_PRMRSLT	VARCHAR2(32)	RLM$PRMINCRSLT_93045
RSET_STATUS	NUMBER	0
RSET_PROP	NUMBER	61
RSET_DURMIN	NUMBER	10080
RSET_DURTCL	VARCHAR2(100)	7 DAYS
RSET_ORDRCL	VARCHAR2(4000)	plane.rlm$CrtTime, rlm$rule.rlm$ruleid DESC
RSET_REWOCL	VARCHAR2(4000)	
RSET_STGCLS	VARCHAR2(4000)	
RSET_EQCLS	VARCHAR2(1000)	
RSET_OBJNM	NUMBER	93045

Another consequence of creating a rule class with composite events is the addition of all primitive events to the table `rlm$rsprimevents`. The two columns RSET_OWNER and RSET_NAME which have the values "AIRLINE" and "FLIGHTSCHEDULINGRULECLASS" are left out for lack of space.

PRIM_ATTR	PRIM_ATTRPOS	PRIM_ATTRALS	PRIM_ASETNM
PLANE	1	RLM$PRMEVT_1	AEROPLANEEVENT
PILOT	2	RLM$PRMEVT_2	PILOTEVENT
COPILOT	3	RLM$PRMEVT_3	PILOTEVENT
SERVICE1	4	RLM$PRMEVT_4	SERVICESTAFFEVENT
SERVICE2	5	RLM$PRMEVT_5	SERVICESTAFFEVENT
SERVICE3	6	RLM$PRMEVT_6	SERVICESTAFFEVENT
SERVICE4	7	RLM$PRMEVT_7	SERVICESTAFFEVENT

Step 3: Defining the Action

Our action is the simple adding of a log entry into the table `FSActionlog` with timestamp, so that we can keep track of which rule triggers the action and when this does take place.

```
CREATE TABLE FSActionLog(
 msg VARCHAR2(100),
 code NUMBER,
 ts TIMESTAMP
);

CREATE OR REPLACE PROCEDURE flightSchedulingAction (
 plane aeroplaneEvent,
 pilot pilotEvent,
 copilot pilotEvent,
 service1 serviceStaffEvent,
 service2 serviceStaffEvent,
 service3 serviceStaffEvent,
 service4 serviceStaffEvent,
 rlm$rule flightSchedulingRuleClass%ROWTYPE
) is

BEGIN

 INSERT INTO FSActionLog (msg,code,ts)
 VALUES  (rlm$rule.msg, rlm$rule.code, SYSDATE);

END;
```

Step 4: Adding Rules

The task of the first two rules is to check whether the pilot and the copilot have the required licence class for operating the aircraft. If the licence is insufficient, the pilot or copilot event is consumed and an entry is added to the log table, whereas the `aeroplaneEvent` stays in the system. Rule number three, four and five audit whether the required number of service personnel is assigned to the flight. It would be desirable being able to specify this condition in just one rule with a syntax like <`any count="3 + plane.requiredServicePersonnel" ...`> , but this is not possible, because the attribute "count" accepts only a number as value and not a string

like "`3 + plane.requiredServicePersonnel`" that has to be computed to a number. Therefore we have to phrase three rules in our example in order to cover all possible cases and it is easy to imagine other scenarios in which hundreds or thousands of rules have to be specified because of this limitation. The fifth rule uses the `and` element instead of the `any` element due to the fact that the value of count can be a number that is smaller than the cardinality of the whole set which is seven in our example.

RULE01

```
INSERT INTO flightSchedulingRuleClass (rlm$ruleid,
                                       rlm$rulecond,
                                       msg, code)
VALUES (
 'RULE01',
 '<condition>
   <and join="plane.flightID = pilot.flightID and
              plane.requiredPilotLicenceClass
              > pilot.pilotLicenceClass"
        sequence="yes">
    <object name="plane"/>
    <object name="pilot">position = ''pilot''</object>
   </and>
  </condition>',
 'Error: Pilot license insufficient.',
 1
);
```

Firstly, the condition demands that the `flightID`s of the both primitive events shall be equal, secondly that the licence class of the `pilot` shall be smaller than the one required by the aircraft and thirdly, the sequence attribute postulates that the `aeroplaneEvent` occurs before the `pilotEvent`. The occurrence of an `aeroplaneEvent` and a `pilotEvent` is required by the rule, the other primitive events of the composite event structure can be `NULL`.

RULE02

```
INSERT INTO flightSchedulingRuleClass (rlm$ruleid,
                                       rlm$rulecond,
                                       msg, code)
```

```
VALUES (
 'RULE02',
 '<condition>
   <and join="plane.flightID = copilot.flightID and
              plane.requiredPilotLicenceClass
              > copilot.pilotLicenceClass"
        sequence="yes">
    <object name="plane"/>
    <object name="copilot">position = ''copilot''</object>
   </and>
  </condition>',
 'Error: Copilot license insufficient.',
 1
);
```

The second rule is very similar to the first rule. The condition checks whether the licence class is lower than the one required for operating the plane. If so, it evaluates to true.

RULE03

```
INSERT INTO flightSchedulingRuleClass (rlm$ruleid,
                                       rlm$rulecond,
                                       msg, code)
VALUES (
 'RULE03',
 '<condition>
   <any count="5"
     equal="plane.flightID, pilot.flightID, copilot.flightID,
            service1.flightID, service2.flightID,
            service3.flightID, service4.flightID"
     join="plane IS NOT NULL and pilot IS NOT NULL
           and copilot IS NOT NULL
           and plane.requiredServicePersonnel=2">
    <object name="plane"/>
    <object name="pilot"/>
    <object name="copilot"/>
    <object name="service1"/>
    <object name="service2"/>
    <object name="service3"/>
    <object name="service4"/>
```

```
   </any>
  </condition>',
 'Flight scheduling ok. (Rule 3)',
 0
);
```

The condition of the third rule uses an `any` element in its specification. `count="5"` demands that at least five out of the seven primitive events that build the composite event have to occur. Ergo, two primitive events can be `NULL`. The join attribute ensures that `plane`, `pilot` and `copilot` are not `NULL`.

RULE04

```
INSERT INTO flightSchedulingRuleClass (rlm$ruleid,
                                       rlm$rulecond,
                                       msg, code)
VALUES (
 'RULE04',
 '<condition>
   <any count="6"
     equal="plane.flightID, pilot.flightID, copilot.flightID,
            service1.flightID, service2.flightID,
            service3.flightID, service4.flightID"
     join="plane IS NOT NULL and pilot IS NOT NULL
           and copilot IS NOT NULL
           and plane.requiredServicePersonnel=3">
    <object name="plane"/>
    <object name="pilot"/>
    <object name="copilot"/>
    <object name="service1"/>
    <object name="service2"/>
    <object name="service3"/>
    <object name="service4"/>
   </any>
  </condition>',
 'Flight scheduling ok. (Rule 4)',
 0
);
```

The condition of rule four is similar to the previous one. Here, the occurrence of at least six out of seven primitive events is required.

RULE05

```
INSERT INTO flightSchedulingRuleClass (rlm$ruleid,
                                       rlm$rulecond,
                                       msg, code)
VALUES (
 'RULE05',
 '<condition>
   <and equal="plane.flightID, pilot.flightID, copilot.flightID,
               service1.flightID, service2.flightID,
               service3.flightID, service4.flightID"
     join="plane.requiredServicePersonnel=4">
    <object name="plane"/>
    <object name="pilot"/>
    <object name="copilot"/>
    <object name="service1"/>
    <object name="service2"/>
    <object name="service3"/>
    <object name="service4"/>
   </and>
  </condition>',
 'Flight scheduling ok. (Rule 5)',
 0
);
```

The `and` element postulates that all seven primitive events have to occur. The `equal` attribute ensures that all `flightID`s are equal and the `join` attribute garantees that this condition evaluates to true only for a `plane` that requires four crew members for the service.

Step 5: Processing Events

Now we can process primitive events with our rule class and observe how they are stored until they are consumed and how the compositing process is prepared within the database.

```
BEGIN
```

```
 DBMS_RLMGR.PROCESS_RULES (
  rule_class => 'flightSchedulingRuleClass',
  event_inst => AnyData.ConvertObject(
                aeroplaneEvent(4344, 'Airbus A320', 3, 2))
 );

END;
```

The table `rlm$prmevents_93045` stores all unconsumed primitive events for the `flightSchedulingRuleClass`. It has four columns. The first is RLM$DELTIME and the value indicates the point in time when the primitive event will be removed from the system. This date is simply calculated by adding the lifetime of the primitive event to the date of creation. For our rule class we defined a default lifetime of seven days, so that the aeroplaneEvent will be kept in this table for exactly one week.

RLM$ DELTIME	RLM$ PRMEVST_1_EVT(FLIGHTID, PLANETYPE, REQUIRED-PILOTLICENCECLASS, REQUIRED-SERVICEPERSONNEL, RLM$CRTTIME)	RLM$ PRMEVST_2_EVT(FLIGHTID, STAFFID, PILOTLICENCECLASS, NAME, POSITION, RLM$CRTTIME)	RLM$ PRMEVST_3_EVT(FLIGHTID, STAFFID, NAME, RLM$CRTTIME)
23-NOV-06 10.54.50 PM	AEROPLANEEVENT(4344, 'Airbus A320', 3, 2, '16-NOV-06 10.54.50.465383 PM')		

After the execution of `DBMS_RLMGR.PROCESS_RULES`, there are five new rows in the table `rlm$prmexprt_93045` that stores primitive event expressions, one row for each rule the primitive has been applied to.

RULE01
The first row has the ID "AAAWt1AAEAAAAaOAAF" as a unique identifier (RLM$RULEROWID) and the join predicate (RLM$JOINPRED):

```
(plane.flightID = pilot.flightID and
 plane.requiredPilotLicenceClass > pilot.pilotLicenceClass)
and
```

```
exfsys.rlm$seqchk(
 exfsys.rlm$dateval(PLANE.rlm$crttime , PILOT.rlm$crttime ),0)
= 1
```

This is the translation of the first rule in our `flightSchedulingRuleClass`. We demanded that the flightIDs of both primitive events shall be equal, that the licence class of the pilot shall be smaller than the one required by the aircraft and thirdly, the sequence attribute postulates that the `aeroplaneEvent` occurs before the `pilotEvent`. Entries in the columns RLM$PRMEVT_1_EXP and RLM$PRMEVT_2_EXP indicate that the first rule requires an `aeroplaneEvent` and `pilotEvent`, whereas the other primitive events are irrelevant for this rule. The value of RLM$PRMEVT_1_EXP is "1 = 1" because we only demand the existence of this primitive event without further conditions. RLM$PRMEVT_2_EXP has the value "position = 'pilot'" for the first rule. The attribute "position" distinguishes pilots from copilots who share the same event structure.

RULE02
The second row looks very similar. The only difference is the usage of copilot instead of a pilot.

RULE03
The condition of the third rule uses an `any` element in its specification. This is translated into a sum using the function decode as an existence operator. decode(PLANE,null,0,1) compares PLANE with null. If PLANE is null, then 0 is returned, otherwise decode returns 1. In rule three we demanded that the count of all objects within the `any` element shall be five. This is realised by the first part "(decode(...)+...+decode(...) >= 5)".

```
((decode(PLANE,null,0,1)
 +decode(PILOT,null,0,1)
 +decode(COPILOT,null,0,1)
 +decode(SERVICE1,null,0,1)
 +decode(SERVICE2,null,0,1)
 +decode(SERVICE3,null,0,1)
 +decode(SERVICE4,null,0,1) >= 5 ) and
exfsys.rlm$eqlchk(
        exfsys.rlm$keyval(PLANE.FLIGHTID,PILOT.FLIGHTID,
                          COPILOT.FLIGHTID,SERVICE1.FLIGHTID,
                          SERVICE2.FLIGHTID,SERVICE3.FLIGHTID,
                          SERVICE4.FLIGHTID))=1 ) and
```

```
(plane IS NOT NULL and
 pilot IS NOT NULL and
 copilot IS NOT NULL
 and plane.requiredServicePersonnel=2)
```

RULE04
The fourth row is likewise. Both conditions possibly involve all of the seven primitive events, but for rule three the occurence of five primitive events would be sufficient, for rule four there have to be six primitive events.

RULE05
The join predicate of the fifth row translates an `and` element, not an `any` element like the last two rows, so that our last condition postulates that the plane requires four stewardesses and stewards, and that all seven primitive events do actually occur and that all have the same flightID.

```
(exfsys.rlm$eqlchk(
  exfsys.rlm$keyval(PLANE.FLIGHTID, PILOT.FLIGHTID,
                    COPILOT.FLIGHTID, SERVICE1.FLIGHTID,
                    SERVICE2.FLIGHTID, SERVICE3.FLIGHTID,
                    SERVICE4.FLIGHTID))=1 )
and (plane.requiredServicePersonnel=4)
```

Now we process the second primitive event by trying to assign a pilot with an insufficient licence class to our flight.

```
BEGIN

 DBMS_RLMGR.PROCESS_RULES (
  rule_class => 'flightSchedulingRuleClass',
  event_inst => AnyData.ConvertObject(
                 pilotEvent(4344, 55, 2, 'John', 'pilot'))
 );

END;
```

The primitive event is accounted for in the `rlm$prmevents_93045` table as deleted immediately after processing. It was consumed by our first rule as expected and a new log entry in `FSActionlog` says "Error: Pilot license

insufficient.". According to the timestamp in `FSActionlog`, the action was executed within the same second the primitive event was created.

The `aeroplaneEvent` was given a shared consumption policy and it is still stored in the database for building future composite events. Next we add pilot and copilot assignment events with appropriate values and then service staff events, so that eventually the condition of rule three will be satisfied and the according action will be triggered.

```
BEGIN

 DBMS_RLMGR.PROCESS_RULES (
  rule_class => 'flightSchedulingRuleClass',
  event_inst => AnyData.ConvertObject(
                pilotEvent(4344, 56, 3, 'Milton', 'pilot'))
 );

END;
```

The `pilotEvent` is stored in `rlm$prmevents_93045`. The previous primitive event with a deletion time in the past has been removed. It appears that the table is cleaned up at regular intervals. No changes are made in `rlm$prmexprt_93045`.

```
BEGIN

 DBMS_RLMGR.PROCESS_RULES (
  rule_class => 'flightSchedulingRuleClass',
  event_inst => AnyData.ConvertObject(
                pilotEvent(4344, 57, 3, 'Richard', 'copilot'))
 );

END;
```

Like before, the primitive event is stored in `rlm$prmevents_93045` and nothing else happened.

```
BEGIN

 DBMS_RLMGR.PROCESS_RULES (
```

```
  rule_class => 'flightSchedulingRuleClass',
  event_inst => AnyData.ConvertObject(
                 serviceStaffEvent(4344, 324, 'Christina'))
 );

END;
```

Again the primitive event was preserved for later composition, but no rule matched.

```
BEGIN

 DBMS_RLMGR.PROCESS_RULES (
  rule_class => 'flightSchedulingRuleClass',
  event_inst => AnyData.ConvertObject(
                 serviceStaffEvent(4344, 325, 'Robert'))
 );

END;
```

This last primitive event and all stored primitive events together build a composite event that satisfies the condition of rule three and triggers the action. The log shows "Flight scheduling ok. (Rule 3)". The `pilotEvent`s and serviceStaffEvents are consumed.

Chapter 4

Example Scenario: An Airline

In this chapter, we will look at a practical application of the Oracle Rules Manager. We will design an example scenario of a conceived airline and peruse different cases in order to detect which rule conditions can be handled by the Rules Manager, and what are the limitations of specifying rule conditions. A complete listing of all tables, events, rules and actions in SQL/Oracle syntax can be found at the end of this document in Appendix A.

In the following examples, we will especially make use of the table `bookings`. If a customer of ABC Air books a flight, this will result in inserting a row in the table `bookings`. Additionally, we map the table `bookings` to an event structure, so that each `INSERT` will raise a DML event of a booking.

```
CREATE TABLE bookings (
  bookingID NUMBER(10) PRIMARY KEY,
  flightID VARCHAR2(6),
  customerID NUMBER(10),
  departureTime DATE,
  departureAirport VARCHAR2(4),
  arrivalTime DATE,
  arrivalAirport VARCHAR2(4),
  price NUMBER(6,2)  CHECK (price > 0),
  class VARCHAR2(20) CHECK (class IN ('Economy','Business')),
  FOREIGN KEY (flightID) REFERENCES flights (flightID),
  FOREIGN KEY (customerID) REFERENCES customers (customerID),
  FOREIGN KEY (departureAirport) REFERENCES airports (ICAOcode),
```

```
  FOREIGN KEY (arrivalAirport) REFERENCES airports (ICAOcode)
);


BEGIN

 DBMS_RLMGR.CREATE_EVENT_STRUCT(
      event_struct => 'bookingEvent'
 );
 DBMS_RLMGR.ADD_ELEMENTARY_ATTRIBUTE (
      event_struct => 'bookingEvent',
      attr_name => 'bEvt',
      tab_alias => exf$table_alias('bookings')
 );

END;
```

Using these booking events we simulate our example scenario. At the same time, we focus on the rule condition part which is formulated in XML syntax. In Section 2.2, we examined the possibilities of specifying event structures and the options what rule actions can perform. Now in this chapter, we scrutinize the rule condition part.

4.1 Simple Rule Conditions

The following condition evaluates true for all flight bookings from Hanover/Langenhagen International Airport (EDDV) to Munich International Airport (EDDM). The airports are referenced to by their ICAO code [3]. Details like the international name of the airport and the location can be found in the table `airports`[1]. A primitive event structure suffices for this example.

```
<condition>
 departureAirport = 'EDDV' and arrivalAirport = 'EDDM'
</condition>
```

[1]The location of an airport is determined by the combination of the triple city VARCHAR2(35), province VARCHAR2(32) and country VARCHAR2(4) which reference to the MONDIAL database [5].

4.2 Rule Conditions Using and-join

Whereas the previous example worked with a simple event structure, for more sophisticated examples composite event structures are needed which consist of two or more simple events.

In the next case, two booking events are joined on the condition that they have the same customerID. That means the same customer has booked both flights. The first flight travels from London Heathrow (EGLL) to Hanover (EDDV) from where the second flight heads to Munich (EDDM).

```
<condition>
 <and join="bookingEvent1.customerID
            = bookingEvent2.customerID">
  <object name="bookingEvent1">
   departureAirport = 'EGLL' and arrivalAirport = 'EDDV'
  </object>
  <object name="bookingEvent2">
   departureAirport = 'EDDV' and arrivalAirport = 'EDDM'
  </object>
 </and>
</condition>
```

4.3 Rule Conditions Using Sequence

In the following condition, the Rules Manager checks if the events occur in the specified order. This is demanded by the attribute `sequence="Yes"` in the `and` element. Assuming that our airline generates an opening event when a flight is ready for check-in and a closing event when no more check-ins are accepted and the plane will be prepared for take-off, then this condition ensures that a check-in event of a passenger occurs after the opening event and before the closing event. The `join="..."` condition assures that all events relate to the same flight by requiring equality of the flightIDs.

```
<condition>
 <and join="openFlightForCheckInEvent.flightID
            = CheckInEvent.flightID
            and
            CheckInEvent.flightID
            = closeForCheckInFlightEvent.flightID"
            sequence="Yes">
```

```
  <object name="openFlightForCheckInEvent"/>
  <object name="CheckInEvent"/>
  <object name="closeFlightForCheckInEvent"/>
 </and>
</condition>
```

4.4 Rule Conditions Using Negation

Negations in rule conditions are represented by the `not` and the `notany` elements. There is only one negation permitted per rule condition. Negation signifies the non-occurrence of the event that is surrounded by the `not` element within a certain time slot. This time slot is usually defined as a delay in comparison to another event that is part of the composite event. Within the `not` element can only be one event and the `not` element itself has to be part of an `and` element. The primitive event whose non-occurrence is postulated, has to be part of the composite event structure.

For example, let us assume that ABC Air usually sends a confirmation immediately after a flight booking. The following condition checks whether a confirmation was sent within one day after the booking. If it did not happen on time, a possible rule action could be a retry or alerting the service personnel. The code segment `by="bookingEvent.rlm$CrtTime+1"` specifies the time slot which begins at the creation time of the `bookingEvent` and ends one day later.

```
<condition>
 <and join="bookingEvent.customerID
            = sendConfirmationEvent.customerID and
            bookingEvent.flightID
            = sendConfirmationEvent.flightID">
  <object name="bookingEvent"/>
  <not by="bookingEvent.rlm$CrtTime+1">
   <object name="sendConfirmationEvent"/>
  </not>
 </and>
</condition>
```

The `notany` element can be used for constructing a condition that checks the non-occurrence of two or more elements. For example, if a flight booking has neither been confirmed by mail nor by phone within one day, then this condition will evaluate to true:

```
<condition>
 <and equal="bookingEvent.customerID,
             confirmByMailEvent.customerID,
             confirmByPhoneEvent.customerID">
  <object name="bookingEvent"/>
  <notany count="1" by="bookingEvent.rlm$CrtTime+1"
          join="bookingEvent.flightID
                = confirmByMailEvent.flightID and
                bookingEvent.flightID
                = confirmByPhoneEvent.flightID">
   <object name="confirmByMailEvent"/>
   <object name="confirmByPhoneEvent"/>
  </notany>
 </and>
</condition>
```

This example takes into account that Oracle suggests in the documentation of the Rules Manager that "the primitive events appearing within the `not` or `notany` elements should not be referenced in the join attribute specification of the `and` element. However, they (primitive events) can be used within the EQUAL property specifications. If there is a need to specify a join condition (other than those already captured by the EQUAL property specifications), the join attribute for the `not` element can be used. The conditional expression specified for this join attribute can reference all the primitive events that appear in the rule condition, including those appearing within the `not` element" [2].

4.5 Rule Conditions Using Set Semantics

A set is a collection of objects. These objects are simple events in our case. The following condition checks whether a customer has booked three or more flights with seats in the business class. A possible action could be the invitation to a bonus programme.

```
<condition>
 <and equal="bookingEvent1.customerID, bookingEvent2.customerID
             bookingEvent3.customerID">
  <object name="bookingEvent1">class = Business</object>
  <object name="bookingEvent2">class = Business</object>
  <object name="bookingEvent3">class = Business</object>
```

```
 </and>
</condition>
```

4.6 Rule Conditions Using any n Semantics

The `any` element allows us to construct rule conditions that require only a certain subset of simple events in order to evaluate to true. For instance if we have a composite event that consists of the three simple events flightBookingEvent, rentCarEvent and hotelBookingEvent, then this condition will be true, if at least one of the three simple events occur. The other two can be NULL. The objects within the `any` element are logically connected with non-exclusive OR.

```
<condition>
 <any>
  <object name="flightBookingEvent"/>
  <object name="rentCarEvent"/>
  <object name="hotelBookingEvent"/>
 </any>
</condition>
```

In the case that at least two simple events have to occur, then the attribute `count` can be used to specify the minimum number of simple event objects that are needed to fulfil the condition. If `count` is not specified, it has the default value 1.

```
<condition>
 <any count="2">
  <object name="flightBookingEvent"/>
  <object name="rentCarEvent"/>
  <object name="hotelBookingEvent"/>
 </any>
</condition>
```

The attributes `join`, `equal` and `sequence` can be utilised in the same way as in the `and` element. For example, if additionally to the previous condition the occurrence of the object `flightBookingEvent` is required, this can be expressed as follows. This is equivalent to the logical `"flightBookingEvent AND (rentCarEvent OR hotelBookingEvent)"`.

```
<condition>
 <any count="2" join="flightBookingEvent IS NOT NULL">
  <object name="flightBookingEvent"/>
  <object name="rentCarEvent"/>
  <object name="hotelBookingEvent"/>
 </any>
</condition>
```

Chapter 5

Embedding the Oracle Rules Manager Example Scenario into the ECA Framework

In this chapter, we will start with an outline of the Semantic Web as it is today. The description of the concepts and the important components of the Semantic Web aims at giving the reader an impression of the current heterogeneity.

In the concluding section, "A General Framework for Evolution and Reactivity in the Semantic Web" [1] is described that was designed for bridging this heterogeneity. This framework is based on ECA rules which provide the aspect of reactivity.

In the final section of this chapter, we will incorporate our airline scenario into this ECA Framework for the Semantic Web. For this purpose we sketch out a wrapper software that mediates between the Oracle Rules Manager and a domain broker developed in [12]. The Rules Manager builds together with the wrapper software a fully functional domain node, and the domain broker acts as a gateway for this bundle, and connects the node to the Semantic Web.

5.1 The Semantic Web

The *Semantic Web* augments the current World Wide Web (WWW) with meta information that endow the existing data with a well-defined meaning and makes them computer-processable.

For example, if somebody queries a search engine of the WWW for the keyword "Oracle", then the search engine does not know, whether the person is looking for (1) a source of prophetic foreseeing like the Delphic Sibyl, (2) a novel by Edwin O'Connor, (3) a shopping mall in Reading, England, (4)

a comic book superhero, (5) a database software or maybe something else [25]. In the Semantic Web, the word might be accompanied with meta information clarifying that a database software is meant. A search engine for the Semantic Web could in a certain sense "understand" what is searched for and accordingly select only those hits that fit in that context.

The WWW was designed for linking human-readable documents and it is essentially based on three standards that were introduced by Tim Berners-Lee: the HyperText Markup Language (HTML) [20], the Hypertext Transfer Protocol (HTTP) [19] and Uniform Resource Locators (URLs)[1] [22].

The substantial components and standards of the Semantic Web are XML, XML Schema, RDF, RDF Schema and OWL. Below ensues a short description of their function and interaction among themselves and other languages that all together draw the picture of heterogeneity.

For a start, the Semantic Web needs a model that enables the formulation of semantic assertions. In order to allow a mapping of any conceivable application onto this model, it has to be as general as possible. The Resource Description Framework (RDF) was introduced for this purpose. RDF uses the Extensible Markup Language (XML) [28] for the formulation of these statements about resources. An RDF statement consists of three parts called RDF triple: subject, predicate and object. The subject is a resource which has to be uniquely identified. It could be a web page identified by a URL or a book identified by a Uniform Resource Name (URN) [29]. The predicate describes an aspect or a trait of resource in the subject part and the object associates a value to this predicate.

In the context of the Oracle database it appears worthy to mention that there is a direct mapping between RDF and relational databases. A relational database consists of tables and each table consists of columns and rows. An RDF node corresponds to a row entry, the columns correspond to the properties and the content of a table cell corresponds to the value of the property [27].

A compact and better human-readable alternative to RDF's XML syntax is Notation 3 (N3) which was also designed by Tim Berners-Lee [30]. The triples in N3 are presented in a way that resembles the natural language.

For accessing data in RDF a number of query languages are being developed [31]. Among these SPARQL seems to be the most promising. In an interview in May 2006 Tim Berners-Lee stated "SPARQL will make a huge difference" [32].
Since ontologies in computer science originate from the field of artificial intel-

[1] URLs are a subset of Uniform Resource Identifiers (URIs) [21]. Originally the U stood for "Universal" instead of "Uniform" in order to "indicate the importance to the Web architecture of the single universal information space" [24], but Tim Berners-Lee complied with a group at IETF that preferred the term "Uniform".

ligence, several ontology languages have been developed before the existence of the Semantic Web. KIF, F-logic, LOOM, CycL to name a few.

RDF Schema (RDFS) is RDF's vocabulary description language and provides together with the aforementioned RDF the basic elements for the defining an ontology. Please refer to [38] for details.

DAML+OIL (DARPA Agent Markup Language + Ontology Inference Layer) is another ontology language. It is based on RDF and XML.

The Web Ontology Language (OWL) can be regarded as a successor of DAML+OIL. It was developed under the guidance of the World Wide Web Consortium (W3C) and is also based on XML, RDF and RDF Schema[1]. OWL currently consists of three sublanguages: OWL Lite, OWL DL and OWL Full.

OWL Lite supports the phrasing of a classification hierarchy and simple constraint features.

OWL DL[2] provides more expressiveness than OWL Lite, but has some restrictions that guarantee decidability which means that all computations will finish in finite time [37].

OWL Full provides maximum expressiveness and the syntactic freedom of RDF but no computational guarantees [37].

Having delivered specifications, use cases and many related works concerning the Semantic Web standards RDF and OWL, the Web-Ontology Working Group [36] was closed in May 2004.

5.2 ECA Framework

The Semantic Web in Tim Berners-Lee's vision is a 'web of data, in some ways like a global database'. In reality there exists a multitude of data formats, languages, schemas and ontologies, and the Semantic Web is composed of innumerable heterogeneous nodes that provide different resources and different behaviour. For the actualisation of Tim Berners-Lee's vision and for being able to query the whole Semantic Web like a global database, the gap torn apart by the heterogeneity needs to be bridged.

'A General Framework for Evolution and Reactivity in the Semantic Web' was designed for this purpose and has been presented in [1]. In this context, reactivity of nodes in the Semantic Web is understood as their ability to react to events. This behaviour is provided by software that implements Event-Condition-Action rules like e.g. the Oracle Rules Manager. Events and ECA rules can facilitate the communication between individual

[1] OWL extends RDF Schema and for example allows the specification of complex relations between different RDFS classes [35].

[2] The name OWL DL alludes to Description Logics, a field of research which deals with a decidable fragment of first order logic [37].

nodes and knowledge can be propagated via events. From a greater perspective, these local changes in nodes appear as an evolution of the whole Semantic Web.

5.3 Domain Broker

In this section, we will describe the embedding of our airline example scenario into the general ECA Framework for the Semantic Web. Therefore we design a wrapper software along general lines that connects our Oracle Rules Manager to the domain broker that was introduced in [12].

The domain broker stages several services of communication from the perspective of a node. The domain broker simultaneously performs as an event broker, an action broker and a query broker. Our wrapper will call upon the first two offers of service. The event broker takes events from event producing nodes of its domain like e.g. our airline node in the travel domain, and passes them on to event consuming nodes that previously registered at the event broker. The action broker takes actions and forwards them to the appropiate nodes for execution.

Firstly, our wrapper registers on behalf of the airline as an event producer at an event broker for the travel domain. The registration contains a URI of the domain, because the event broker possibly handles more than one domain, and information about the structure of a potentially occurring event. A simple event could for instance be the status change of a flight. A flight might be delayed, rescheduled or canceled and the airline wishes to communicate this efficiently to all who are interested in this information. So this primitive event structure would be registered for the travel domain by the wrapper. On the other side, there are approximately 13000 travel agencies in Germany that could register at the same event broker as event consumers. If a flight of our airline changes its schedule, an event instance will be raised and turned over to the event broker who delivers it to all registered consumers. The nodes of the travel agencies could check whether they have customers who booked that particular flight and if so, send them an update about the new schedule.

As described in [12], another group of event consuming nodes are atomic[2] event matchers (AEM). They also register at the event broker and deliver the atomic events to composite event detection services (CED) that in turn have previously registered at the AEM. CEDs like the SNOOP engine collect atomic events for building composite events. Please refer to [12] for further details.

Regarding action brokering, there are two possibilities available at the

[2]The term 'atomic event' is equivalent to terms 'primitive event' and 'simple event' we used so far.

domain broker level: Action forwarding via broadcast which means that an action is forwarded by the broker to all known nodes that support the action, and secondly, the decision to which nodes the action will be forwarded can be made depending on data that comes along with the action.

Recapitulatory, the tasks of the wrapper are:

- To register, and if necessary to deregister structures of events to be propagated.
- To forward event instances generated by the Rules Manager to the domain broker.
- To register as an event consumer if desired.
- To pass on incoming event instances from domain brokers to the Rules Manager.
- To forward actions from the Rules Manager to the domain broker.
- To accept incoming actions if desired.

Chapter 6

Conclusions

In this chapter, we first discuss some drawbacks and limitations of the current release of the Rules Manager that emerged during the examination, and we make some suggestions for improvement that might be incorporated into future releases. Finally, we conclude this thesis with some final comments.

6.1 Drawbacks

6.1.1 Inflexibility of the Event Structure

A main drawback of the way the Oracle Rules Manager handles events is the fixed nature of composite events. As described in Section 2.3 about rule classes, the structure of events has to be precisely defined before the creation of a rule class that uses this structure. In case of a composite event structure, this definition consists of a finite number of primitive event structures, and this number is fixed at the time of definition.

In the reality of everyday life there are examples of use in which the number of possibly occurring primitive events is uncertain at the time the rule is specified. In Snoop [42], these kinds of events are called cumulative events. For example a composite event structure that reproduces a registration process could consist of an opening event, an unknown number of registration events and a closing event. This scenario cannot be dealt with as one composite event unless some maximum number of registration events is assumed. But that might result in an inflated, inelegant composite event structure that allows many of the registration events to be NULL, or it will lead to inconsistencies and errors, if the actual number of registration events was bigger than the assumed maximum number because in case of a shared consumption of the registration events, they will be processed multiple times, and in case of exclusive consumption, some registration events will remain unprocessed.

An appropriate solution to this problem would be to allow a range of

occurrences of a primitive event instead of a fixed number. The syntax for describing the range could be based on regular expression, like for example:

"1-4" for at least one and maximum four occurrences,
"5+" for five or more occurrences,
"2,4,6" for exactly two, four or six occurrences and
"*" for any number from zero to many occurrences.

6.1.2 Missing DML Events

In the current release of the Oracle Rules Mangager, DML events are only raised by INSERT operations. It would be preferable if UPDATE and DELETE operations could also generate DML events.

Comments in the installation scripts indicate that this feature shall be implemented in the future.

6.1.3 Lacking Query Part

From the viewpoint of the ECA Framework [1], an ECA rule consists of two components: a query part and a testing part. The query part obtains additional information from a local database. Although an implementation of the query part suggests itself for a relational database like the Oracle software, the Rules Manager is lacking this part.

Since this feature can be easily implemented and it would improve the possibilities of specifying rule conditions exceedingly, it is very desirable to install this feature into the next release of the Rules Manager.

6.2 Final Comments

Despite the aforementioned drawbacks, we come to the conclusion that the Oracle Rules Manager provides full functionality for defining and managing ECA rules.

Together with a wrapper software as outlined in the prior chapter, the Oracle Rules Manager can function as a domain node in the Semantic Web ECA Framework.

Since enormous amounts of information are stored in relational databases nowadays, Oracle provides with the Rules Manager package an important possibility for the changeover towards the Semantic Web.

Appendix A

SQL/Oracle Syntax of the Airline Scenario

```
-- This listing consists of the following parts
--
-- 1. Defining the tables
-- 2. Defining Event Structures
-- 3. Running through All Examples of the Airline Scenario.
--    For each example we do the following steps:
--     3.1. Defining the rule class
--     3.2. Replacing the action
--     3.3. Adding rules
-- 4. Processing Event Instances


-- Part 1. Defining the Tables

CREATE TABLE airports (
 ICAOcode VARCHAR2(4) PRIMARY KEY,
 IATAcode VARCHAR2(3),
 internationalName VARCHAR2(50),
 localName VARCHAR2(50),
 city VARCHAR2(35),
 province VARCHAR2(32),
 country VARCHAR2(4));

CREATE TABLE airportCity (
 ICAOcode VARCHAR2(4),
 city VARCHAR2(35),
```

```
 province VARCHAR2(32),
 country VARCHAR2(4),
 PRIMARY KEY(ICAOcode, city, province, country));

CREATE TABLE aircrafts (
 aircraftID NUMBER(4) PRIMARY KEY,
 name VARCHAR2(50),
 range NUMBER(6),
 economySeats NUMBER(4),
 businessSeats NUMBER(4),
 cargo NUMBER(10),
 maxPayload NUMBER(10),
 fuel NUMBER(10));

CREATE TABLE staff (
  staffID NUMBER(10) PRIMARY KEY,
  position VARCHAR2(20) CHECK (
   position IN ('pilot','co-pilot',
               'steward','stewardess')),
  salary NUMBER(6),
  lastname VARCHAR2(50),
  firstname VARCHAR2(50),
  street VARCHAR2(50),
  city VARCHAR2(50),
  country VARCHAR2(50),
  mail VARCHAR2(50),
  phone VARCHAR2(50));

CREATE TABLE flights (
  flightID VARCHAR2(6) PRIMARY KEY,
  aircraftType NUMBER(4),
  departureTime DATE,
  departureAirport VARCHAR2(4),
  arrivalTime DATE,
  arrivalAirport VARCHAR2(4),
  economyPrice NUMBER(6,2),
  businessPrice NUMBER (6,2),
  FOREIGN KEY (departureAirport) REFERENCES airports (ICAOcode),
  FOREIGN KEY (arrivalAirport) REFERENCES airports (ICAOcode),
  FOREIGN KEY (aircraftType) REFERENCES aircrafts (id));

CREATE TABLE customers (
```

```
  customerID NUMBER(10) PRIMARY KEY,
  lastname VARCHAR2(50),
  firstname VARCHAR2(50),
  street VARCHAR2(50),
  city VARCHAR2(50),
  country VARCHAR2(50),
  mail VARCHAR2(50),
  phone VARCHAR2(50));

CREATE TABLE bookings (
  bookingID NUMBER(10) PRIMARY KEY,
  flightID VARCHAR2(6),
  customerID NUMBER(10),
  price NUMBER(6,2)  CHECK (price > 0),
  class VARCHAR2(20) CHECK (class IN ('Economy','Business')),
  FOREIGN KEY (flightID) REFERENCES flights (flightID),
  FOREIGN KEY (customerID) REFERENCES customers (customerID),
  FOREIGN KEY (departureAirport) REFERENCES airports (ICAOcode),
  FOREIGN KEY (arrivalAirport) REFERENCES airports (ICAOcode)
);

-- The tables above are connected with each other
-- in order to provide referential integrity.
-- A booking has a unique bookingID,
-- relates to the customers table via the customerID
-- and to the flights table via the flightID.
-- The flights table relates to the aircrafts table
-- using aircraftType and to the table airports
-- via departureAirport and arrivalAirport.
-- The table airports should have the foreign keys
-- city, province and country to the MONDIAL database
-- in a complete implementation.
-- For the purpose of giving a practical example
-- that can easily be played through,
-- the structure has been simplified to the following table
-- which is the basis of our DML events.
-- Airports are here equivalent to the cities,
-- That wrongly assumes each city has only one airport,
-- but it is better to comprehend than the cryptic ICAO code.

CREATE TABLE simpleBookings (
  customerID NUMBER(5),
```

```
  departureTime DATE,
  departureAirport VARCHAR2(20),
  arrivalTime DATE,
  arrivalAirport VARCHAR2(20)
);

-- All rule actions will insert a row into the following log table,
-- so that it can be easily retraced which rule of which rule class
-- was triggered at what time, and which was not triggered.

CREATE TABLE log (
  ts TIMESTAMP,
  msg VARCHAR2(255)
);


-- Part 2. Defining Event Structures

BEGIN

 DBMS_RLMGR.CREATE_EVENT_STRUCT(event_struct => 'bkEvent');
 DBMS_RLMGR.ADD_ELEMENTARY_ATTRIBUTE (
event_struct => 'bkEvent',
attr_name => 'bk',
tab_alias => exf$table_alias('simpleBookings')
 );

END;

BEGIN

 DBMS_RLMGR.CREATE_EVENT_STRUCT(event_struct => 'bk2Event');
 DBMS_RLMGR.ADD_ELEMENTARY_ATTRIBUTE (
event_struct => 'bk2Event',
attr_name => 'booking1',
tab_alias => exf$table_alias('simpleBookings')
 );
 DBMS_RLMGR.ADD_ELEMENTARY_ATTRIBUTE (
event_struct => 'bk2Event',
attr_name => 'booking2',
tab_alias => exf$table_alias('simpleBookings')
 );
```

```
END;

BEGIN

 DBMS_RLMGR.CREATE_EVENT_STRUCT(event_struct => 'bk3Event');
 DBMS_RLMGR.ADD_ELEMENTARY_ATTRIBUTE (
event_struct => 'bk3Event',
attr_name => 'booking1',
tab_alias => exf$table_alias('simpleBookings')
 );
 DBMS_RLMGR.ADD_ELEMENTARY_ATTRIBUTE (
event_struct => 'bk3Event',
attr_name => 'booking2',
tab_alias => exf$table_alias('simpleBookings')
 );
 DBMS_RLMGR.ADD_ELEMENTARY_ATTRIBUTE (
event_struct => 'bk3Event',
attr_name => 'booking3',
tab_alias => exf$table_alias('simpleBookings')
 );

END;



CREATE or REPLACE TYPE openFlightForCheckInEvent AS OBJECT (
  flightID VARCHAR2(6)
);

CREATE or REPLACE TYPE checkInEvent AS OBJECT (
  customerID NUMBER(10),
  flightID VARCHAR2(6)
);

CREATE or REPLACE TYPE closeFlightForCheckInEvent AS OBJECT (
  flightID VARCHAR2(6)
);

CREATE or REPLACE TYPE compCheckEvt AS OBJECT (
  open openFlightForCheckInEvent,
```

```
  checkin checkInEvent,
  close closeFlightForCheckInEvent
);




-- Part 3. Running through all examples of the airline scenario.
--
-- For each example, the following steps are performed:
-- 3.1. Defining the rule class
-- 3.2. Replacing the action
-- 3.3. Adding rules
-- 3.4. Processing event instances

-- Example 1: 'Simple Rule Condition'

-- Step 1: Defining the rule class

BEGIN

 DBMS_RLMGR.CREATE_RULE_CLASS (
  rule_class => 'example1Class',
  event_struct => 'bkEvent',
  action_cbk => 'example1Action',
  rlcls_prop => '<simple dmlevents="I"/>'
 );

END;

-- Step 2: Replacing the action

CREATE OR REPLACE PROCEDURE example1Action
                   (booking bkEvent,
                    rlm$rule example1Class%ROWTYPE)
IS

BEGIN

 INSERT INTO log (ts, msg)
 VALUES (SYSDATE, 'Example 1: Action triggered.');
```

```
END;

-- Step 3: Adding rules

INSERT INTO example1Class (rlm$ruleid, rlm$rulecond)
VALUES ('RULE01',
        '<condition>
          departureAirport = ''Frankfurt''
          and arrivalAirport = ''London''
         </condition>'
);


-- Example 2: 'Rule Conditions Using and-join'

-- Step 1: Defining the rule class

BEGIN

 DBMS_RLMGR.CREATE_RULE_CLASS (
  rule_class => 'example2Class',
  event_struct => 'bk2Event',
  action_cbk => 'example2Action',
  rlcls_prop => '<composite dmlevents="I"/>'
 );

END;

-- Step 2: Replacing the action

CREATE OR REPLACE PROCEDURE example2Action
                    (booking1 ROWID,
                     booking2 ROWID,
                     rlm$rule example2Class%ROWTYPE)
IS

BEGIN

 INSERT INTO log (ts, msg)
 VALUES (SYSDATE, 'Example 2: Action triggered.');

END;
```

```
-- Step 3: Adding rules

INSERT INTO example2Class (rlm$ruleid, rlm$rulecond)
VALUES ('RULE01',
        '<condition>
          <and join="booking1.customerID
                     = booking2.customerID">
           <object name="booking1">
            departureAirport = ''Hanover''
            and arrivalAirport = ''Frankfurt''
           </object>
           <object name="booking2">
            departureAirport = ''Frankfurt''
            and arrivalAirport = ''New York''
           </object>
          </and>
         </condition>'
);


-- Example 3: 'Rule Conditions Using Sequence'

-- Step 1: Defining the rule class

BEGIN

 DBMS_RLMGR.CREATE_RULE_CLASS (
  rule_class => 'example3Class',
  event_struct => 'compCheckEvt',
  action_cbk => 'example3Action',
  rlcls_prop => '<composite/>'
 );

END;

-- Step 2: Replacing the action

CREATE OR REPLACE PROCEDURE example3Action
                   (open openFlightForCheckInEvent,
                    checkin checkInEvent,
                    close closeFlightForCheckInEvent,
```

```
                    rlm$rule example3Class%ROWTYPE)
IS

BEGIN

 INSERT INTO log (ts, msg)
 VALUES (SYSDATE, 'Example 3: Action triggered.');

END;

-- Step 3: Adding rules

INSERT INTO example3Class (rlm$ruleid, rlm$rulecond)
VALUES ('RULE01',
        '<condition>
          <and join="open.flightID
                     = checkin.flightID
                     and close.flightID
                     = checkin.flightID"
               sequence="Yes">
           <object name="open"/>
           <object name="checkin"/>
           <object name="close"/>
          </and>
         </condition>'
);


-- Example 4: 'Rule Conditions Using Negation'
--            'The not Element'

-- Step 1: Defining the rule class

BEGIN

 DBMS_RLMGR.CREATE_RULE_CLASS (
  rule_class => 'example4Class',
  event_struct => 'bk2Event',
  action_cbk => 'example4Action',
  rlcls_prop => '<composite dmlevents="I"/>'
 );
```

```
END;

-- Step 2: Replacing the action

CREATE OR REPLACE PROCEDURE example4Action
                   (booking1 ROWID,
                    booking2 ROWID
                    rlm$rule example4Class%ROWTYPE)
IS

BEGIN

 INSERT INTO log (ts, msg)
 VALUES (SYSDATE, 'Example 4: Action triggered.');

END;

-- Step 3: Adding rules

INSERT INTO example4Class (rlm$ruleid, rlm$rulecond)
VALUES ('RULE01',
        '<condition>
          <and join="booking1.customerID
            = booking2.customerID">
           <object name="booking1"/>
           <not by="booking1.rlm$CrtTime+1">
            <object name="booking2"/>
           </not>
          </and>
         </condition>'
);


-- Example 5: 'Rule Conditions Using Negation'
--            'The notany Element'

-- Step 1: Defining the rule class

BEGIN

 DBMS_RLMGR.CREATE_RULE_CLASS (
  rule_class => 'example5Class',
```

```
  event_struct => 'bk3Event',
  action_cbk => 'example5Action',
  rlcls_prop => '<composite dmlevents="I"/>'
 );

END;

-- Step 2: Replacing the action

CREATE OR REPLACE PROCEDURE example5Action
                   (booking1 ROWID,
                    booking2 ROWID,
                    booking3 ROWID,
                    rlm$rule example5Class%ROWTYPE)
IS

BEGIN

 INSERT INTO log (ts, msg)
 VALUES (SYSDATE, 'Example 5: Action triggered.');

END;

-- Step 3: Adding rules

INSERT INTO example5Class (rlm$ruleid, rlm$rulecond)
VALUES ('RULE01',
        '<condition>
          <and equal="booking1.customerID,
                      booking2.customerID,
                      booking3.customerID">
           <object name="booking1">
            arrivalAirport = ''Madrid''
           </object>
           <notany count="1" by="booking1.rlm$CrtTime+1">
            <object name="booking2">
             departureAirport = ''Madrid''
             and arrivalAirport = ''London''
            </object>
            <object name="booking3">
             departureAirport = ''Madrid''
             and arrivalAirport = ''Paris''
```

```
            </object>
           </notany>
          </and>
         </condition>'
);


-- Example 6: 'Rule Conditions Using Set Semantics'


-- Step 1: Defining the rule class

BEGIN

 DBMS_RLMGR.CREATE_RULE_CLASS (
  rule_class => 'example6Class',
  event_struct => 'bk3Event',
  action_cbk => 'example6Action',
  rlcls_prop => '<composite dmlevents="I"
                            consumption="exclusive"/>'
 );

END;

-- Step 2: Replacing the action

CREATE OR REPLACE PROCEDURE example6Action
                   (booking1 ROWID,
                    booking2 ROWID,
                    booking3 ROWID,
                    rlm$rule example6Class%ROWTYPE)
IS

BEGIN

 INSERT INTO log (ts, msg)
 VALUES (SYSDATE, 'Example 6: Action triggered.');

END;

-- Step 3: Adding rules
```

```
INSERT INTO example6Class (rlm$ruleid, rlm$rulecond)
VALUES ('RULE01',
        '<condition>
          <and equal="booking1.customerID,
                      booking2.customerID,
                      booking3.customerID">
           <object name="booking1">
            departureAirport = ''Rome''
           </object>
           <object name="booking2">
            departureAirport = ''Rome''
           </object>
           <object name="booking3">
            departureAirport = ''Rome''
           </object>
          </and>
         </condition>'
);


-- Example 7: 'Rule Conditions Using any n Semantics'


-- Step 1: Defining the rule class

BEGIN

 DBMS_RLMGR.CREATE_RULE_CLASS (
  rule_class => 'example7Class',
  event_struct => 'bk3Event',
  action_cbk => 'example7Action',
  rlcls_prop => '<composite dmlevents="I"/>'
 );

END;

-- Step 2: Replacing the action

CREATE OR REPLACE PROCEDURE example7Action
                   (booking1 ROWID,
                    booking2 ROWID,
                    booking3 ROWID,
```

```
                    rlm$rule example7Class%ROWTYPE)
IS

BEGIN

 INSERT INTO log (ts, msg)
 VALUES (SYSDATE, 'Example 7: Action triggered.');

END;

-- Step 3: Adding rules

INSERT INTO example7Class (rlm$ruleid, rlm$rulecond)
VALUES ('RULE01',
        '<condition>
          <any count="2">
           <object name="booking1"/>
           <object name="booking2"/>
           <object name="booking3"/>
          </any>
         </condition>'
);

-- Part 4. Processing event instances

SELECT * FROM log;

-- For example 3: 'Rule Conditions Using Sequence'

BEGIN

 DBMS_RLMGR.ADD_EVENT (
  rule_class => 'example3Class',
  event_inst => openFlightForCheckInEvent.getVarchar('AB4223'),
  event_type => 'openFlightForCheckInEvent'
 );

END;

SELECT * FROM log;

BEGIN
```

```
 DBMS_RLMGR.PROCESS_RULES (
  rule_class => 'example3Class',
  event_inst => checkInEvent.getVarchar(1234, 'AB4223'),
  event_type => 'checkInEvent'
 );

END;

SELECT * FROM log;

BEGIN

 DBMS_RLMGR.PROCESS_RULES (
  rule_class => 'example3Class',
  event_inst => closeFlightForCheckInEvent.getVarchar('AB4223'),
  event_type => 'closeFlightForCheckInEvent'
 );

END;

SELECT * FROM log;

-- For example 1: 'Simple Rule Condition'

INSERT INTO simpleBookings (
  customerID, departureTime, departureAirport,
  arrivalTime, arrivalAirport)
VALUES (
  4567,
  TO_DATE('31-Jan-2007 10:05','DD-MON-YYYY HH24:MI'),
  'Frankfurt',
  TO_DATE('31-Jan-2007 11:55','DD-MON-YYYY HH24:MI'),
  'London'
);

SELECT * FROM log;

-- For example 2: 'Rule Conditions Using and-join'

INSERT INTO simpleBookings (
  customerID, departureTime, departureAirport,
```

```
  arrivalTime, arrivalAirport)
VALUES (
  4567,
  TO_DATE('31-Jan-2007 12:00','DD-MON-YYYY HH24:MI'),
  'Hanover',
  TO_DATE('31-Jan-2007 12:30','DD-MON-YYYY HH24:MI'),
  'Frankfurt'
);

SELECT * FROM log;

INSERT INTO simpleBookings (
  customerID, departureTime, departureAirport,
  arrivalTime, arrivalAirport)
VALUES (
  4567,
  TO_DATE('31-Jan-2007 15:00','DD-MON-YYYY HH24:MI'),
  'Frankfurt',
  TO_DATE('31-Jan-2007 23:00','DD-MON-YYYY HH24:MI'),
  'New York'
);

SELECT * FROM log;

-- For example 6: 'Rule Conditions Using Set Semantics'

INSERT INTO simpleBookings (
  customerID, departureTime, departureAirport,
  arrivalTime, arrivalAirport)
VALUES (
  4567,
  TO_DATE('01-Jan-2007 12:00','DD-MON-YYYY HH24:MI'),
  'Rome',
  TO_DATE('01-Jan-2007 14:00','DD-MON-YYYY HH24:MI'),
  'Paris'
);

SELECT * FROM log;

INSERT INTO simpleBookings (
  customerID, departureTime, departureAirport,
  arrivalTime, arrivalAirport)
```

```
VALUES (
  4567,
  TO_DATE('01-Jan-2007 12:00','DD-MON-YYYY HH24:MI'),
  'Rome',
  TO_DATE('01-Jan-2007 14:00','DD-MON-YYYY HH24:MI'),
  'Paris'
);

SELECT * FROM log;

INSERT INTO simpleBookings (
  customerID, departureTime, departureAirport,
  arrivalTime, arrivalAirport)
VALUES (
  4567,
  TO_DATE('15-Jan-2007 18:15','DD-MON-YYYY HH24:MI'),
  'Rome',
  TO_DATE('15-Jan-2007 21:00','DD-MON-YYYY HH24:MI'),
  'Oslo'
);

SELECT * FROM log;
```

Bibliography

[1] A General Framework for Evolution and Reactivity in the Semantic Web. December 2006

[2] Oracle Database: Application Developer's Guide - Rules Manager and Expression Filter 10g Release 2 (10.2) B14288-01

[3] http://en.wikipedia.org/wiki/List_of_airports_by_ICAO_code

[4] http://en.wikipedia.org/wiki/IATA

[5] http://www.dbis.informatik.uni-goettingen.de/Mondial/

[6] Oracle Database: Spatial User's Guide and Reference 10g Release 2 (10.2) Part Number B14255-01

[7] http://en.wikipedia.org/wiki/G%C3%B6ttingen

[8] http://en.wikipedia.org/wiki/World_Geodetic_System

[9] Oracle Database: PL/SQL User's Guide and Reference 10g Release 2 (10.2) Part Number B14261-01

[10] Oracle Database: PL/SQL Packages and Types Reference 10g Release 2 (10.2) Part Number B14258-01

[11] Oracle Database: SQL Reference 10g Release 2 (10.2) Part Number B14200-02

[12] Tobias Knabke: Development and Implementation of a Domain Broker for the Semantic Web

[13] Oracle's SQL Scripts: catrul.sql, rultyp.sql, rultab.sql, rulpbs.sql and rulview.sql

[14] Tim Berners-Lee: Semantic Web Road map. September 1998

[15] Tim Berners-Lee, Mark Fischetti: Weaving the Web. HarperCollins Publishers. 1999

[16] Tim Berners-Lee, James Hendler and Ora Lassila: The Semantic Web A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. May 2001

[17] http://www.w3.org/Proposal.html

[18] http://www.w3.org/People/Berners-Lee/WorldWideWeb

[19] ftp://ftp.rfc-editor.org/in-notes/rfc1945.txt

[20] ftp://ftp.rfc-editor.org/in-notes/rfc1866.txt

[21] ftp://ftp.rfc-editor.org/in-notes/rfc1630.txt

[22] ftp://ftp.rfc-editor.org/in-notes/rfc1738.txt

[23] ftp://ftp.rfc-editor.org/in-notes/rfc3986.txt

[24] http://www.w3.org/DesignIssues/Architecture.html

[25] http://en.wikipedia.org/wiki/Oracle_%28disambiguation%29

[26] Tim Berners-Lee, Wendy Hall and Nigel Shadbolt: The Semantic Web Revisited. 2006

[27] Tim Berners-Lee: What the Semantic Web can represent. http://www.w3.org/DesignIssues/RDFnot.html

[28] http://www.w3.org/XML/

[29] ftp://ftp.rfc-editor.org/in-notes/rfc2141.txt

[30] http://www.w3.org/DesignIssues/Notation3.html

[31] http://www.w3.org/2001/11/13-RDF-Query-Rules/

[32] http://en.wikipedia.org/wiki/SPARQL

[33] http://en.wikipedia.org/wiki/Ontology_%28computer_science%29

[34] John Davies, Rudi Studer, Paul Warren: Semantic Web Technologies. 2006

[35] http://www.w3.org/2003/08/owlfaq.html.en

[36] http://www.w3.org/2001/sw/WebOnt/

[37] http://www.w3.org/TR/owl-guide/

[38] http://www.w3.org/TR/2004/REC-rdf-schema-20040210/

[39] http://www.heise.de/newsticker/meldung/81747

[40] http://www.heise.de/newsticker/meldung/82708

[41] http://www.w3.org/TR/xpath20/

[42] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In Proceedings of the 20th VLDB, pages 606 to 617, 1994.