EU-Rent as an Artifact-Centric Process Model: Technical Report

Montse Estañol, Anna Queralt, Maria Ribera Sancho, Ernest Teniente

September 2012

Abstract

Business process modeling using an artifact-centric approach has raised a significant interest over the last few years. This approach is usually stated in terms of the BALSA framework which defines the four "dimensions" of an artifact-centric business process model: Business Artifacts, Lifecycles, Services and Associations. One of the research challenges in this area is looking for different diagrams to represent these dimensions. Bearing this in mind, this technical report shows how various UML diagrams can be used to represent all the elements in the BALSA framework by applying them to the EU-Rent case study.

Contents

1 Introduction **2**

2 Artifact-Centric Business Process Models in UML **4**
2.1 Introduction 4
2.2 Business Artifacts 4
2.3 Business Artifact Lifecycle 5
2.4 Services 5
2.5 Associations 6
2.6 Summary 6

3 EU-Rent Car Rental Service as an Artifact-Centric Model in UML **7**
3.1 Introduction 7
3.2 Assumptions 8
3.3 Business Artifacts as a Class Diagram 8
3.4 Lifecycle of *RentalAgreement* as a State Machine Diagram 18
3.5 Associations as Activity Diagrams and Services as Action Contracts 20

Appendices **45**

A Structural Schema in OCL **46**
A.1 Class Diagram 46
A.2 Integrity Constraints 46

Chapter 1

Introduction

Business process design is a key activity in organisations. Business process models have been traditionally based on an activity-centric perspective and thus specified by means of diagrams which define how a business process or workflow is supposed to operate, but giving little importance (or none at all) to the information produced as a consequence of the process execution. Therefore, this approach under-specifies the data underlying the service and the way it is manipulated by the process tasks [4].

Nearly a decade ago, a new information-centric approach to business process modeling emerged [7] and it is still used today. It relies on the assumption that any business needs to record details of what it produces in terms of concrete information. Business artifacts, or simply artifacts, are proposed as a means to record this information. They model key business-relevant entities which are updated by a set of services (specified by pre and postconditions) that implement business process tasks. This approach has been successfully applied in practice and it provides a simple and robust structure for workflow modeling [2, 1].

The artifact-centric approach to business process specification has been shown to have a great intuitive appeal to business managers. However, further research is needed with regards to the "best" artifact-centric model since none of the existing models can adequately handle the broad requirements of business process modeling [6].

This technical report shows the results of a applying our particular proposal for an artifact-centric approach using UML diagrams. We consider that one way of validating it is by applying it to a big case study. EU-Rent, as it is explained in [5], is a case study originally developed by Model Systems, Ltd. EU-Rent is the name of a fictional company which rents cars. It has branches in various countries and it offers the typical car rental services and keeps information about its customers. The technical report [5] includes a detailed description of EU-Rent and its specification using standard notation: UML 2.0 and OCL 2.0.

We considered that EU-Rent would be an appropriate case study for validating our proposal because it presents a service which most people would be familiar with, but at the same time it is complex enough to offer a good testing environment. In order to avoid unnecessary repetition, we take [5] as a starting point for our own report. We refer to it in order to find the detailed description of the EU-Rent company and how it works. Unless otherwise stated, we have followed exactly the same criteria described in it.

This technical report is structured in the following way:

Chapter 1: Introduction presents the purpose of the document and its structure.

Chapter 2: Artifact-Centric Business Process Models in UML summarises our proposal for describing business process from an artifact-centric perspective using UML models.

Chapter 3: EU-Rent Car Rental Service as an Artifact-Centric Model in UML shows how the EU-Rent car rental service would be specified using the proposal summarised in Section 2.

Acknowledgements: The research that resulted in the work presented here has received the financial support of UPC (Universitat Politècnica de Catalunya - Barcelona Tech).

Chapter 2

Artifact-Centric Business Process Models in UML

This chapter describes briefly our proposal for specifying artifact-centric business process models in UML. A very brief summary is presented at the end.

2.1 Introduction

Traditional process-centric business process models are essentially uni-dimensional in the sense that they focus almost entirely on the process model, its constructs and its patterns, and provide little or no support for understanding the structure or the life-cycle of the data that underlies and tracks the history of most workflows [6].

In contrast, the artifact-centric approach provides four explicit inter-related but "separable" dimensions in the specification of the business process [6, 3]. This four-dimensional framework is referred to as "BALSA" - Business Artifacts, Lifecycles, Service and Associations, first described in [6, 3]. By showing the UML diagram which is more appropriate to define each one of these four dimensions we will be able to construct our proposal for the specification of artifact-centric business process models in this language.

However, UML is not enough, as usually UML diagrams make use of some textual notation to precisely specify those aspects that cannot be graphically represented. Currently, the OCL (Object Constraint Language) [10] is probably the most popular one of these notations and we will also use it in our proposal. OCL supplements UML by providing expressions that have neither the ambiguities of natural language nor the inherent difficulty of logic.

The rest of the section gives a brief explanation of the four BALSA dimensions and we explain how we propose representing them using UML diagrams.

2.2 Business Artifacts

The conceptual schema of business artifacts is intended to hold all of the information needed in completing business process execution. A business artifact has an *identity*, which makes it distinguishable from any other artifact, and can

be tracked as it progresses through the workflow of the business process execution. It will usually have also a set of *attributes* to store the data needed for the workflow execution. The relationship of a business artifact with other artifacts must also be shown when this information is relevant for the business being defined. In business terms, an artifact represents the explicit knowledge concerning progress toward a business operational goal at any instant. Therefore, at any time of the execution, the information contained in the set of artifact records all the information about the business operation.

In UML, conceptual schemas are defined by means of class diagrams. We will use a UML class diagram to show the business entities and how they are related to each other, represented as classes and associations respectively. Each class (or business artifact) may have a series of attributes that represent relevant information for the business. Moreover, they can be externally identified by specific attributes or by the relationships they can take part in. A class diagram may also require a list of integrity constraints that, as their name implies, establish a series of restrictions over the class diagram. Constraints can be either specified graphically in the UML class diagram or textually by means of the OCL language.

Furthermore, the UML class diagram allows representing class hierarchies graphically. We will benefit from this by representing the different states in an artifact's lifecycle as subclasses of a superclass, as long as these subclasses hold relevant information or are in relevant relationships. The advantage of having different subclasses for a particular artifact is that it allows having exactly those attributes and relationships that are needed according to its state, preserving at the same time the artifact's original ID and the characteristics that are independent of the artifact's state which are represented in the superclass.

2.3 Business Artifact Lifecycle

The lifecycle of a business artifact states the key, business-relevant, *stages* in the possible evolution of the artifact, from inception to final disposal and archiving. It is natural to represent it by using a variant of state machines, where each state of the machine corresponds to a possible stage in the lifecycle of an artifact from the class [6]. We propose representing the states an artifact may go through in a UML state machine diagram.

2.4 Services

A service (or "task") in a business process encapsulates a unit of work meaningful to the whole business process. The action of services makes business artifacts evolve, e.g. they may cause modifications on the information stored by the artifacts or they may make artifacts to evolve to a new stage, relevant from the business perspective.

Our way of representing services is by means of an OCL operation contract. As we have mentioned before, OCL is a formal language that avoids ambiguities. Moreover, it is declarative, which means that it does not indicate *how* things should be done, but rather *what* should be done.

Operation contracts consist in a set of input parameters and output parameters, a precondition and a postcondition. Both input and output parameters can be classes (i.e. business artifacts) or simple types (e.g. integers, strings, etc.). A precondition states the conditions that must be true before invoking the operation and refers to the values of artifact attributes at the time when the service is called. The postcondition indicates the state of the business artifacts after the execution of the operation. It may refer to the values of artifact attributes at the time when the service is called (appending operator @pre) and to their values after the service has finished execution (no operator or appending operator @post). Those artifacts that do not appear in the postcondition keep their state from before the execution of the operation.

2.5 Associations

The problem, however, is that having the services as detailed above is not enough. We need also a way to establish the conditions under which they can be executed since, in a business process, services make changes to artifacts in a manner that is restricted by a set of constraints.

Since the goal of the *associations* is to define the right sequencing of service execution, we propose using UML activity diagrams for specifying them. In this way, each service is represented as an action (a rounded rectangle) in the activity diagram. Arrows show the order in which actions have to be executed. Swimlanes indicate the main business artifact involved in each action, and the notes stereotypes as *Participant* indicate who is the responsible for carrying out that action.

By modeling associations in this way we achieve our proposal to incorporate also some notions of process awareness, despite its intrinsic artifact-centric nature. Therefore, we may also explicitly capture the control flow of the business process, aspect which is usually lacking in previous artifact-centric proposals.

2.6 Summary

In summary, following the BALSA model described in [6], we will use the following UML diagrams to represent each of its elements:

- UML class diagram to represent the business artifacts.
- State machine diagram to represent the business artifacts' lifecycle.
- Services will be represented as OCL operations with preconditions and postconditions.
- Associations will be shown graphically in a UML activity diagram.

Chapter 3

EU-Rent Car Rental Service as an Artifact-Centric Model in UML

This chapter shows how our proposal is applied to a particular example. As we have already mentioned in the Introduction, we will use the EU-Rent specification described in [5] as a starting point. In the first section of this chapter we give a brief overview of how the EU-Rent company works. Section Assumptions details some considerations and assumptions we have made in order to specify the car rental service provided by EU-Rent. The rest of sections in this chapter show the various diagrams and elements that make up the EU-Rent service specification.

3.1 Introduction

This introduction is meant to give a brief overview of EU-Rent. For a detailed description of how the company works, check pages 1-15 of [5].

EU-Rent is a case study originally developed by Model Systems, Ltd. EU-Rent is a fictional car rental company with branches in multiple countries. It is part of a bigger company, EU-Corporation, which also owns hotels and an airline. A prospective client must be registered with the company in order to rent a car: he/she may make a reservation some days in advance, or rent the car on the spot (what is called a walk-in rental).

Customers are allowed to have many reservations, but they can only have one rental at a time. They are also allowed to return the car to a branch other than the pick-up branch. The company keeps information about the customers, such as a history of their rentals and records any bad experiences (e.g a late return or a damaged car). Therefore, a particular customer may be blacklisted (i.e. he/she will not be allowed to rent a car) if certain conditions are met.

On the other hand, customers may belong to the Loyalty Incentive Scheme. Customers in this program are allowed to pay for their rentals using loyalty points. Moreover, any rental may qualify for a discount, and the customer is always offered the best price for the rental. However, loyalty points can only

pay for the basic price of a rental, i.e. without any discounts applied.

Cars are classified into different groups according to their characteristics, and customers are allowed to choose either a particular car model or a car group. If they do not choose any, they are assigned the cheapest car group. Cars are serviced after a while, and can be bought and sold by EU-Rent. They sometimes have to be transferred from one branch to the other, precisely because customers are allowed to return them to a different branch.

When a car is handed over to the customer, he/she has to fulfill certain conditions: he/she should be able to drive and should not be under the influence of alcohol or drugs, he/she should have a valid driving license and be over 25 years of age. A reservation is held for a customer for 90 minutes after the scheduled pick-up time if the reservation is not guaranteed. If it is guaranteed by a credit card, it is held for the whole day before the car is released and the customer's credit card is charged for not picking it up.

Customers can request rental extensions by phone, and they are granted unless the car is due for maintenance.

3.2 Assumptions

For the following service specification of EU-Rent we follow the same assumptions as in [5]. However, we only want to specify the car rental service provided by the company, i.e. those business processes that are directly involved in the provision of a car rental. This corresponds to a subset of the use cases in the original specification. The rest of use cases are necessary for the provision of the service but transparent to the client and we do not provide their details here.

It is also important to bear in mind that we want to avoid redundancy in the specification. For this reason, we follow the guidelines of [11] both in the specification of the actions (i.e. what in BALSA is referred to as services) and in the activity diagrams. That is, the activity diagrams and action contracts do not check conditions that are already guaranteed somewhere else in the specification (e.g. in integrity constraints). Moreover, consecutive actions will not check for conditions already guaranteed by previous actions. We also consider that parameters can be reused in operations that are part of the same activity diagram.

3.3 Business Artifacts as a Class Diagram

The UML class diagram represents the business artifacts that take part in the provison of the business processes. However, it is important to note that the class diagram presented here is a subset of the one in [5]. Moreover, as the resulting diagram was very big, we have split it into smaller ones. There is one main diagram, that shows the main business artifacts and their relationships, and then we have smaller diagrams showing a business artifacts and its subtypes. At the end there is a diagram showing some data types that we use in our model.

For each class diagram, we include the corresponding integrity constraints and derivation rules. They are defined in natural language. The corresponding OCL definition can be found on Appendix A.

3.3.1 Main Class Diagram

The diagram in Figure 3.1 shows the main artifacts in EU-Rent and the relationships between them.

EU_RentPerson represents someone who has had contact with EU-Rent, either as a driver or as a customer. For this reason, it is linked with exactly one *DrivingLicense*, and a *DrivingLicense* belongs exactly to one *EU_RentPerson*. Notice that this class does not hold any personal information: we have assumed that this information is shared with other EU companies, and the corresponding class is shown in Figure 3.2. An *EU_RentPerson* may take part in any number of *RentalAgreements* as a driver. A *Customer* is a subtype of *EU_RentPerson* and will have, at least, one *RentalAgreement*, but he may not have more than one *RentalAgreement* for a particular *DateTime*[1].

The key class in the diagram is *RentalAgreement*. It can be of the *Reservation* subtype, which means that a reservation was made before the scheduled pick-up date of the car. A *Reservation* is linked to a specific *CarGroup*, and may be linked to a particular *CarModel*. Each *CarModel* belongs to one (and only one) *CarGroup*. *CarGroups* are ordered by their category. A *RentalAgreement* will have certain *RentalDurations*, and therefore may be eligible for some *Discounts*. The *RentalAgreement* will be linked to exactly one pick-up and one drop-off *Branch*, and will also have at least one *Country* where the user will travel to with the car (at least, the *Branches'* countries). It may also have a particular *Car* assigned, which will be of a particular *CarModel*.

A *RentalAgreement* is opened (*OpenRental*) when a customer picks up the car, and is closed (*ClosedRental*) when he/she returns it. It may also be extended (*ExtendedRental*). A *ClosedRental* may be linked to a *BadExperience* (which may be of the *CarDamange* subtype) with an associated *FaultSeriousness*.

The following subsections describe the integrity constraints and derivation rules.

3.3.1.1 Integrity Constraints

- *Branch* is identified by name.
- The pick-up and drop-off branches' *Countries* must be included in the list of countries of the *RentalAgreement*.
- The *initEnding* of a *RentalAgreement* must be later than its *beginning*. The *actualReturn* of a *RentalAgreement* must also be later than its *beginning*.
- *reservationDate* of a *Reservation* must be previous to its *beginning* date.
- Requested car model in a *Reservation* must be in requested car group.
- Rental extension must be done after the *beginning* date of the *RentalAgreement* and the new end date should be later than the initial end date

[1] Note that, for convenience purposes, we have included *DateTime* as a class of the diagram, when it is clearly not a business artifact. However, we considered that a *RentalAgreement* was defined by a *Customer* and a particular *DateTime*; therefore, we decided that *RentalAgreement* should be an association class resulting from the link between these two classes.

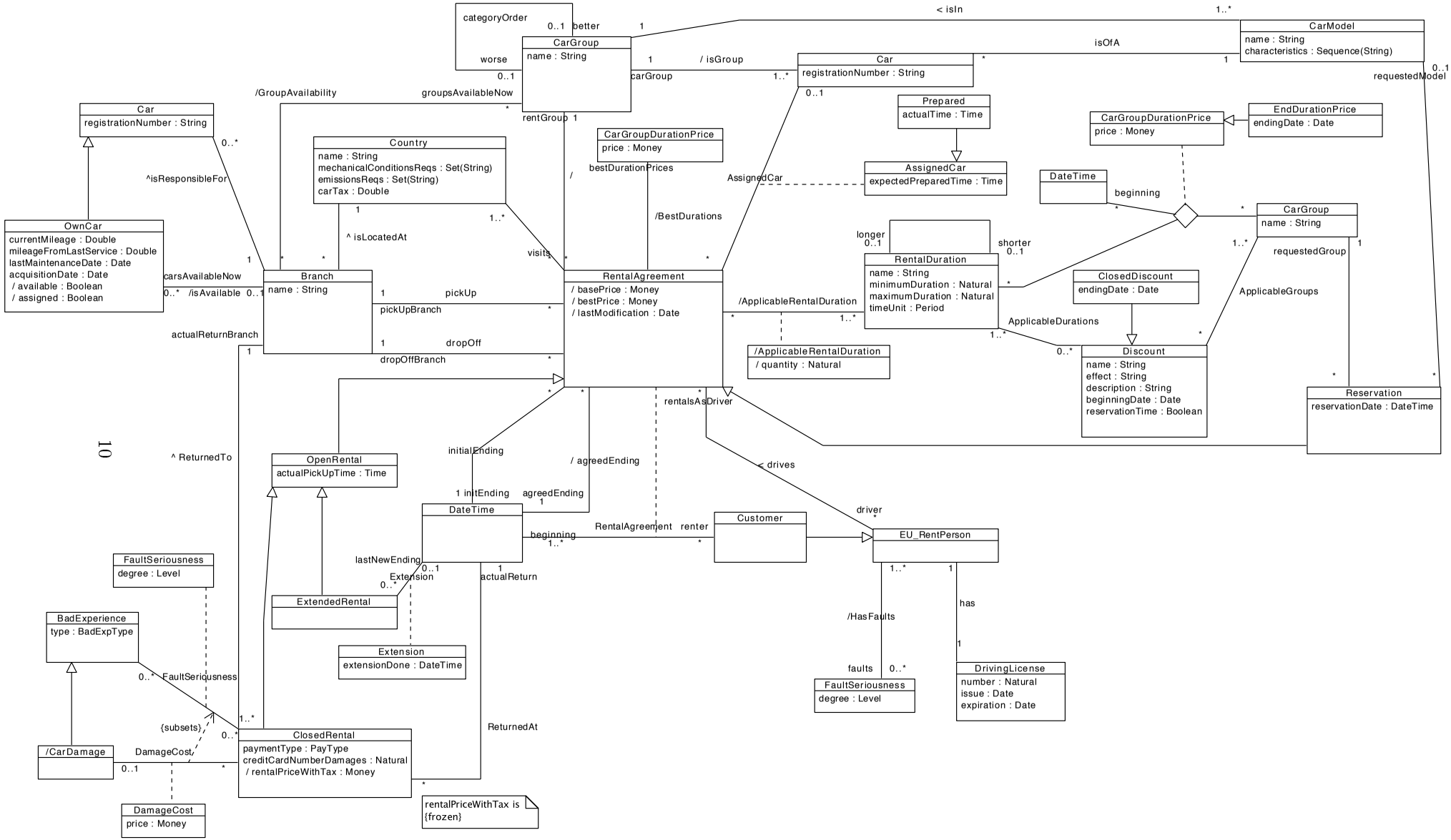

Figure 3.1: Main class diagram with the business artifacts for the provision of car rentals

(*initEnding*). Note that this constraint has been rewritten, as the original code did not tally with the original class diagram.

- *CarModel* is identified by its name.
- *CarGroup* is identified by its name.
- *CarGroup* order must be coherent (i.e. there are no cycles). Only one *CarGroup* may not have a better *CarGroup*, and only one *CarGroup* may not have a worse *CarGroup* (it may be the same).
- *RentalAgreements* of a *Customer* do not overlap.
- *DrivingLicenses* are identified by their number.
- An *EU_RentPerson* has at least one year of driving experience and the *DrivingLicense* does not expire before the *agreedEnding* of a rental of the driver.
- *RentalDurations* are identified by their name
- Price for a particular *RentalDuration* and *CarGroup* in *CarGroupDurationPrice* must be higher than the price for the same *RentalDuration* but worse *CarGroup*, excluding those *CarGroupDurationPrice* that have ended.
- The order of *RentalDurations* is coherent (i.e. there are no cycles). Only one *RentalDuration* may not have a longer *RentalDuration*, and only one *RentalDuration* may not have a shorter *RentalDuration* (it may be the same).
- *Discounts* are identified by name.
- Ending date of *EndDurationPrice* must be on the same day or later than its beginning date.
- Ending date of *ClosedDiscount* must be on the same day or later than its beginning date.
- *Countries* are identified by name.
- *Car* can only be assigned, at most, to one rental; excluding both closed and canceled rentals.
- *Car* is identified by registration number
- At the time when a car is assigned to a *RentalAgreement* (excluding *ClosedRentals* and *CanceledRentals*) the pick-up branch becomes responible for the car.
- *BadExperience* is identified by type.

3.3.1.2 Derived Classes and Attributes

RentalAgreement

- ⋆ *basicPrice* - Best price for the rental without discounts, considering its duration.
- ⋆ *bestPrice* - Best price for the rental with discounts. When the rental is *PaidWithPointsRental*, it is equal to the *basicPrice* of the rental.
- ⋆ *lastModification* - Last modification of the rental. If it is of the *Reservation* subtype, it corresponds to the reservation date. Otherwise, it corresponds to the beginning date of the rental. In any case, if if it has been extended, it corresponds to the *extensionDone* date.

OwnCar

- ⋆ *available* - An *OwnCar* is available if it is not assigned and is NOT of ANY of the following subtypes: *NeedsMaintenance*, *RepairsScheduled*, *ToBeSoldCar*, *BeingTransferredCar* or *NeedToBeSoldCar*.
- ⋆ *assigned* - An *OwnCar* is assigned if there is a *RentalAgreement* linked to the *OwnCar* that has not been canceled (i.e. it is not of *CanceledReservation* subtype) or closed (i.e. it is not of the *ClosedRental* subtype).

CarDamage

- ⋆ Derived class - Every *BadExperience* that has type *carDamage* will be of the *CarDamage* subtype.

ClosedRental

- ⋆ *rentalPriceWithTax* - Price of the rental plus taxes. It is the result of multiplying the *carTax* in the actual drop-off branch and the *bestPrice* of the rental.

ApplicableRentalDuration

- ⋆ *quantity* - For a particular RentalAgreement and RentalDuration, it holds the number of *RentalDurations* applicable to that *RentalAgreement*. This is calculated by dividing the duration of the rental by the maximumDuration or minimumDuration of *RentalDuration*.

3.3.1.3 Derived Relationships

- ⋆ *BestDuration* - *bestDurationPrices* - Best prices (ordered from best to worst) for the duration of the rental.
- ⋆ *agreedEnding* - Obtains the return date of a rental, considering the extensions a rental may have. If it has no extensions, it corresponds to *initEnding*. If it has been extended, it corresponds to *lastNewEnding*.

- ⋆ *rentGroup* - Returns the *carGroup* that the user will have to pay for, that is, if he has been offered a free promotion, he has to pay for the *carGroup* he asked for, not more. Or, if he has been allocated a worse *carGroup* than what he asked for, then he pays for the worse *carGroup* and not the one he asked for initially.
- ⋆ *ApplicableRentalDuration* - *RentalDurations* into which the RentalAgreement can be split. It is calculated considering the number of days (or hours) of the rental and the minimum and maximum durations of each RentalDuration.
- ⋆ *IsGroup* - *carGroup* - *CarGroup* a particular *Car* belongs to. It is the same car group as the one for the car model of a particular car.
- ⋆ *HasFaults* - *faults* - *FaultSeriousness* associated to the *RentalAgreements* of a particular *EU_RentPerson*, considering his/her faults as both driver and customer.
- ⋆ *IsAvailable* - *carsAvailableNow* - Available *OwnCars* for a particular *Branch*.
- ⋆ *GroupAvailability* - *groupsAvailableNow* - *CarGroups* available for a particular *Branch*, obtained through the available *OwnCars*.

3.3.2 EU_CoPerson and its Subclasses

The diagram in Figure 3.2 shows the classes and subclasses of *EU_CoPerson*. It is important to note that we have decided to show in the diagram both *EU_CoPerson*, representing people who are clients of EU-Corporation, and *EU_RentPerson*, representing people who have used the services of EU-Rent. Most of the information about the customer is kept in *EU_CoPerson* (such as name, address, etc.), unlike in the original EU-Rent specification [5].

As it can be seen in the diagram, *EU_RentPerson* is a subtype of *EU_CoPerson*. An *EU_RentPerson* may be blacklisted, and in that case he/she is not allowed to rent cars. As we have seen in the previous section, a *Customer* is also a subtype of *EU_RentPerson*. Finally, a *Customer* may belong to the Loyalty Incentive Scheme, represented by the subclass *LoyaltyMember*.

3.3.2.1 Integrity Constraints

- An *EU_CoPerson* is identified by its id.
- An *EU_CoPerson* must be 25 or older.
- The reservations or rentals of a *Blacklisted* EU_RentPerson that begin after the *blacklistedDate* must be cancelled.
- *RentalAgreements* of *Customer* do not overlap.
- A *LoyaltyMember* rented at least one car during the last year and does not have any bad experience.

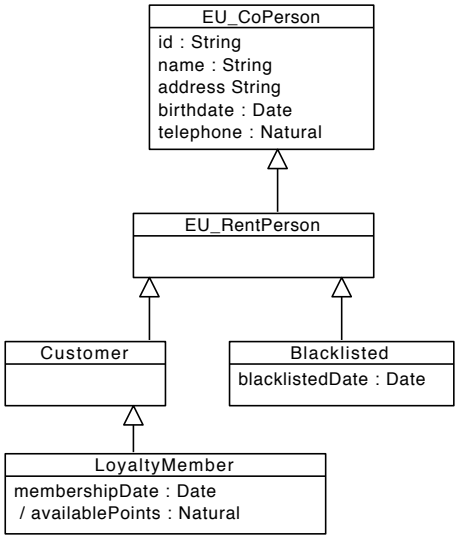

Figure 3.2: Class diagram of EU_CoPerson and its subclasses

3.3.2.2 Derived Attributes and Classes

LoyaltyMember

- ⋆ *availablePoints* - It holds the result of adding the points obtained in the rentals made by the customer which have not been paid with points, and substracting the points spent in the rentals paid with points.

3.3.3 Reservation and its Subclasses

The diagram in Figure 3.3 shows the class *Reservation* and its subclasses.

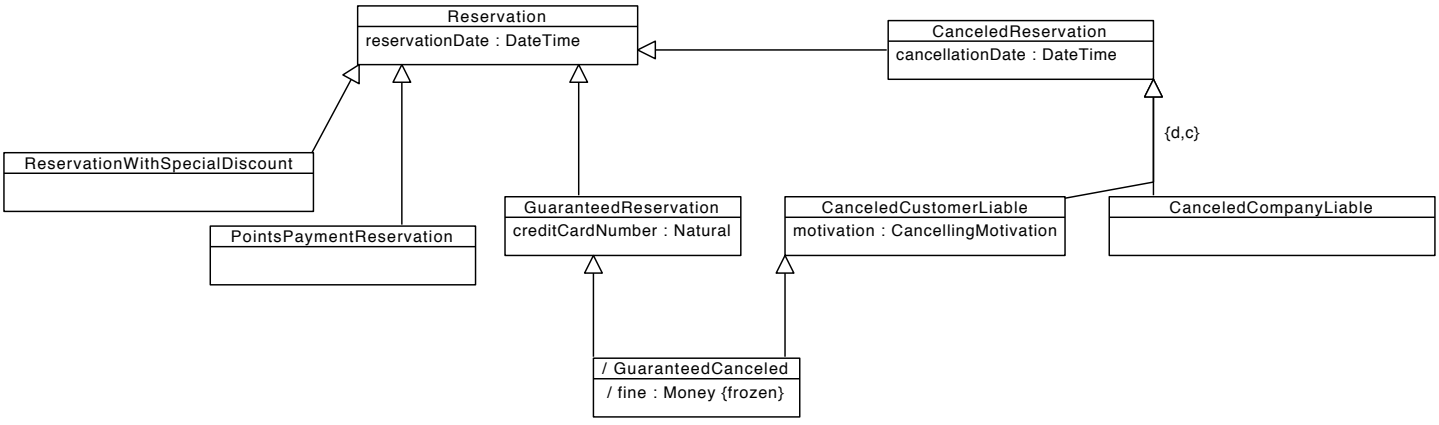

Figure 3.3: Class diagram of Reservation and its subclasses

As it can be seen in the diagram, a *Reservation* may be:

- A *ReservationWithSpecialDiscount*, if it includes a discount on the basic price.
- A *PointsPaymentReservation*, if it can be paid with points and the customer wishes to.
- A *GuaranteedReservation*, if the customer leaves his credit card number.
- A *CanceledReservation*, if the reservation is cancelled. We distinguish two subtypes:
  - *CanceledCompanyLiable*, if EU-Rent is responsible for the cancellation.
  - *CanceledCustomerLiable*, if the customer is the ultimate responsible for the cancellation. A *CanceledCustomerLiable* reservation may also be of *GuaranteedCanceled* subtype if it was also a *GuaranteedReservation*.

It is important to mention that, in the original specification [5], *CanceledReservation* had *CanceledCustomer* and *CanceledCompany* as subclasses, showing whether the reservation had been cancelled at a request from the customer or the company had decided to do so, respectively. In our class diagram, subclasses *CanceledCustomerLiable* and *CanceledCompanyLiable* show who is the ultimate responsible for the cancellation of the reservation: the company may decide to cancel a reservation because a customer is not fit to drive; although it is the company who makes the decision, the customer is liable for it.

3.3.3.1 Integrity Constraints

- *reservationDate* of a *Reservation* must be previous to its *beginning* date.
- Requested car model in a *Reservation* must be in requested car group.
- *PointsPaymentReservation* must be made at least 14 days in advance of its beginning date.
- *cancellationDate* of a *CanceledReservation* must be after or on the same *reservationDate* and before, on the *beginning* date of the *RentalAgreement* or on the day after at the latest. This has been changed from the original report [5].

3.3.3.2 Derived Classes and Attributes

GuaranteedCanceled

⋆ Derived class - All *Reservations* that are both *GuaranteedReservation* and *CanceledCustomerLiable*.

⋆ *fine* - A fine of one day rental must be paid if the rental was guaranteed and the cancelling date is the same day (or later if the customer does not pick up the car) as the expected beginning of the rental. Otherwise, no fine must be paid.

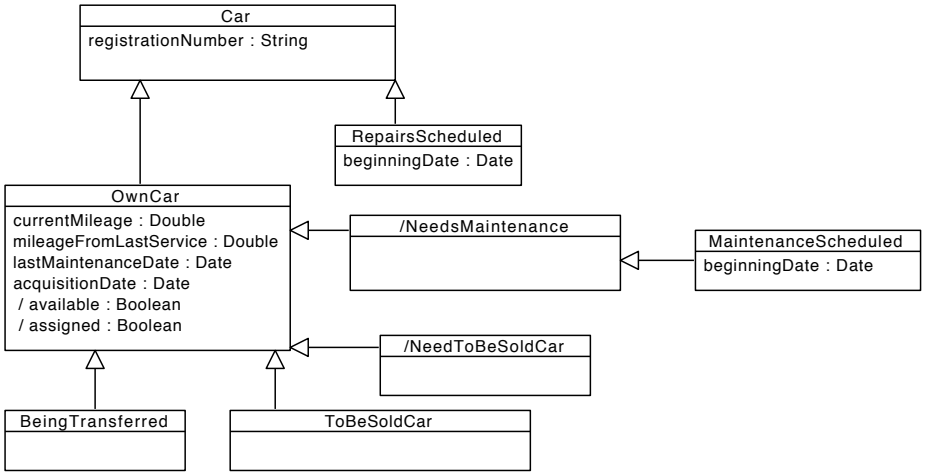

Figure 3.4: Class diagram of Car and its subclasses

3.3.4 Car and its Subclasses

The class diagram of *Car* and its subclasses can be seen in Figure 3.4.

An *OwnCar* represents those cars that are owned by EU-Rent (the company, under special circumstances, can use cars that do not belong to it). An *OwnCar* may be in the process of being transferred from one branch to the other (*BeingTransferred*), may need maintenance (*NeedsMaintenance* subtype), may need to be sold (*NeedToBeSold*) or may be of the *ToBeSoldCar* type, which means that it can no longer be used as it is in the process of being sold. Finally, a *Car* may be scheduled for repairs (*RepairsScheduled*) even if it does not belong to EU-Rent.

3.3.4.1 Integrity Constraints

- *Car* can only be assigned, at most, to one rental; excluding both closed and canceled rentals.
- *Car* is identified by registration number
- A *Car* that needs maintenance cannot have more than 10% of the mileage required for maintenance and not more than 10% of the required time between services may have elapsed.
- A *Car* that is to be sold (*ToBeSoldCar*) cannot be assigned to a rental, excepting those rentals that are closed or canceled.

3.3.4.2 Derived Classes and Attributes

NeedsMaintenance

⋆ Derived class - A car needs maintenance if it was serviced more than 3 months ago or has accumulated more than 10,000 km since the last service.

NeedToBeSoldCar

- ⋆ Derived class - An *OwnCar* is of subtype *NeedToBeSoldCar* if it was bought more than a year ago or has accumulated more than 40,000 km.

3.3.5 ClosedRental and its Subclasses

The diagram in Figure 3.5 shows the class *ClosedRental* and its subclasses.

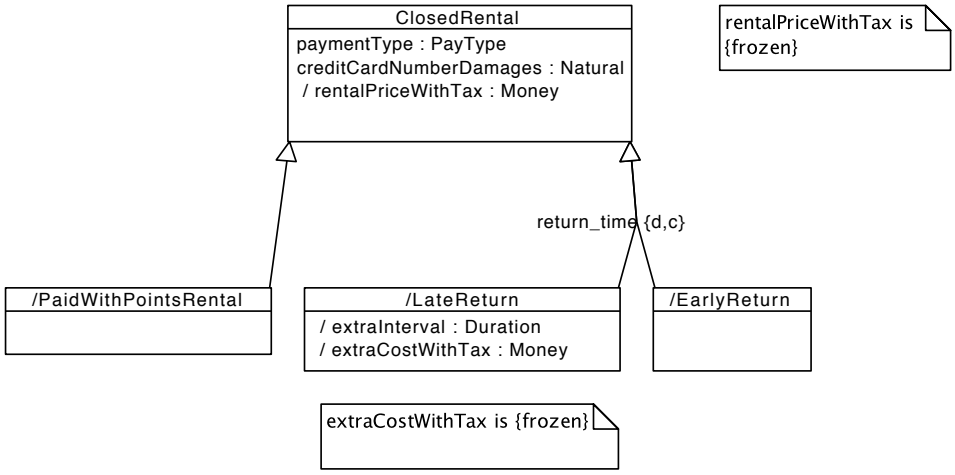

Figure 3.5: Class diagram of ClosedRental and its subclasses

A *ClosedRental* may be *PaidWithPointsRental*, if it has been paid with points; and may also be a *LateReturn* if the car has been returned later than expected or an *EarlyReturn*, if it has been returned more than an hour earlier than expected.

3.3.5.1 Integrity Constraints

- In a *PaidWithPointsRental*, the *Reservation* for the corresponding rental was made at least 14 days in advance of the rental's beginning date.
- In a *PaidWithPointsRental*, the *Customer* must be a member of Loyalty Incentive Scheme (i.e. *LoyaltyMember*) in order to pay with points. It is an initial constraint, as the customer must be a Loyalty Incentive Member only at the time of paying; later on he/she may not be a member any longer.

3.3.5.2 Derived Classes and Attributes

ClosedRental

- ⋆ *rentalPriceWithTax* - Price of the rental plus taxes. It is the result of multiplying the *carTax* in the actual drop-off branch and the *bestPrice* of the rental.

PaidWithPointsRental

- ⋆ Derived class - All *ClosedRentals* that have been paid with points (i.e their *paymentType* is *Points*).

LateReturn

- ⋆ Derived class - All *ClosedRentals* such that the *actualReturn* is later than the *agreedEnding*.
- ⋆ *extraInterval* - Duration of the period between the *agreedEnding* and the *actualReturn* of the car.
- ⋆ *extraCostWithTax* - Holds the price of the *extraInterval*, considering the best price for duration without applying any discounts, and the cost of the taxes according to country where the car has been dropped off.

EarlyReturn

- ⋆ Derived class - All *ClosedRentals* such that the *actualReturn* is more than an hour sooner than the *agreedEnding*.

3.3.6 Types

The types that have been defined for EU-Rent can be seen in Figure 3.6. Note that most of them have been defined from scratch or redefined from [5].

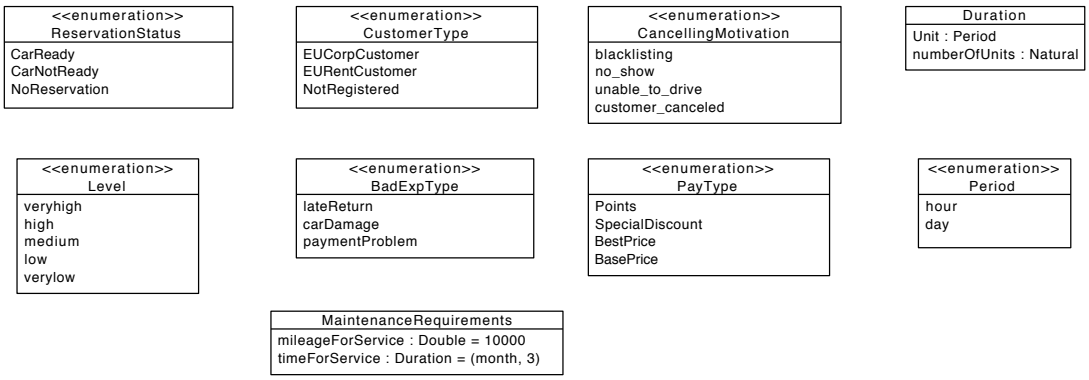

Figure 3.6: Definition of types

3.4 Lifecycle of *RentalAgreement* as a State Machine Diagram

Although many of the business artifacts represented in the class diagram have a lifecycle, in order to keep it simple we will focus only on the lifecycle of what is the main business artifact: *RentalAgreement*.

The state machine diagram for the service can be seen in Figure 3.7. It shows the whole lifecycle of *RentalAgreement*, from the moment a customer

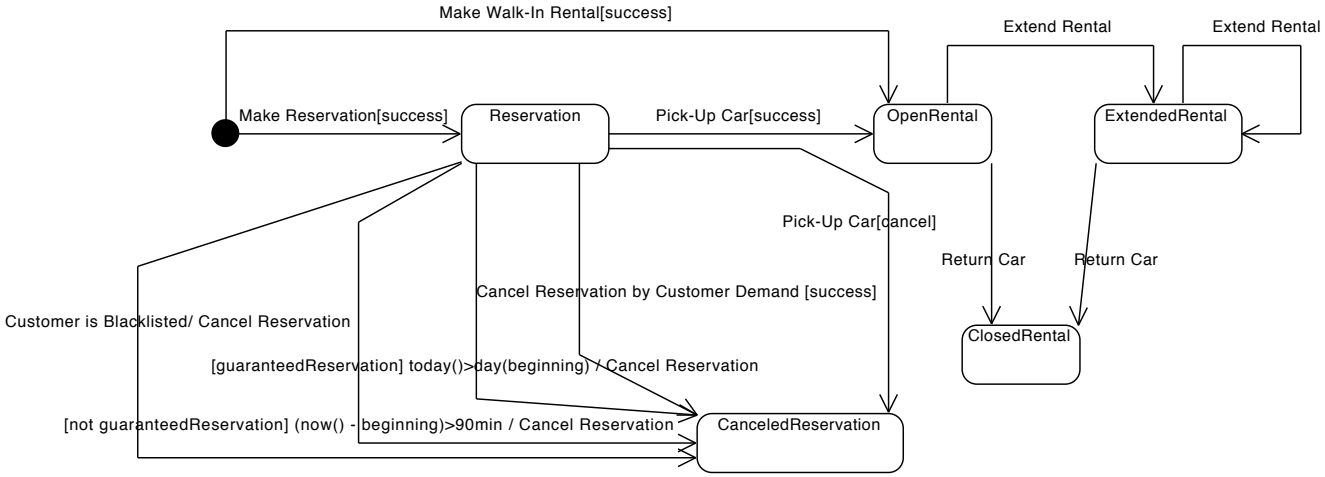

Figure 3.7: State machine diagram for *RentalAgreement*

makes a reservation or rents a car to the moment when the car is returned. It is worth noting that this diagram does not follow exactly the standard described in [9]: we have more than one outgoing transition from the start node. This is necessary because the service can be initialized in different ways (e.g. by making a walk-in rental or a reservation). In any case, the transitions between states are triggered by either domain events, time events or change events [8].

However, our domain events are not always atomic: they can be subprocesses which are further decomposed into actions (or services). These subprocesses may have a condition in square brackets which the subprocess has to meet when it ends in order for the transition to be fired. For example, the transition from *Reservation* to *OpenRental* will only be triggered when: 1- subprocess *Pick Up Car* takes place AND 2- it ends successfully AND 3- *RentalAgreement* is in state *Reservation*. If this same subprocess ends fulfilling the condition *cancel* when the service is in state *Reservation*, then the *RentalAgreement* would be canceled. The postconditions in the state transitions can also be non-atomic. For example, when time event *today() > day(beginning)* takes place and the *Reservation* has been guaranteed, then the reservation must be canceled. This is done through subprocess *Cancel Reservation*.

The state machine diagram in Figure 3.7 shows that there are two possible ways of creating a *RentalAgreement*: either with *Make Reservation* or *Make Walk-In Rental*. In the case of *Make Reservation*, the user has to *Pick-Up Car* before actually using it. It is also important to notice that in state *Reservation*, the reservation may be cancelled either because the customer requests it (*Cancel Reservation by Customer Demand*) or because one of the following conditions is met: 1- the car is not picked up 90 minutes after the scheduled pick-up time and the reservation is not guaranteed, 2- the car is not picked-up in the scheduled day and the reservation was guaranteed, 3- the customer is blacklisted, 4- *Pick-Up Car* is cancelled. In all these cases, the service ends.

While the rental is open, the customer can request an extension (*Extend Rental*. The *RentalAgreement* will become a *ClosedRental* when the customer returns the car (*Return Car*).

3.5 Associations as Activity Diagrams and Services as Action Contracts

The activity diagrams provide the details for each of the subprocesses in the state machine diagrams. Each subprocess is decomposed into actions, which in turn can be atomic (they are *services* as defined in BALSA) or further decomposed in another activity diagram (indicated by a rake-like symbol). Therefore, activity diagrams act as associations between services.

In each activity diagram, the transitions that lead to an end node may be stereotyped with a tag that indicates the outcome of the subprocess. Examples of tags are *succeed* and *fail*, which may be then used in the state machine diagram to determine the following state in the service evolution. Swimlanes indicate the main artifact involved in each of the services or actions, and they are labeled with stereotype *material* if they are dealing with a real, physical object and not its representation. Those actions that deal with information resources are further specified by action contracts using OCL. They correspond to services in BALSA.

Each subsection corresponds to one of the subprocesses in the state machine diagram. However, there are some actions within activity diagrams whose details are defined in another activity diagram: they also have a subsection of their own.

3.5.1 Make Reservation

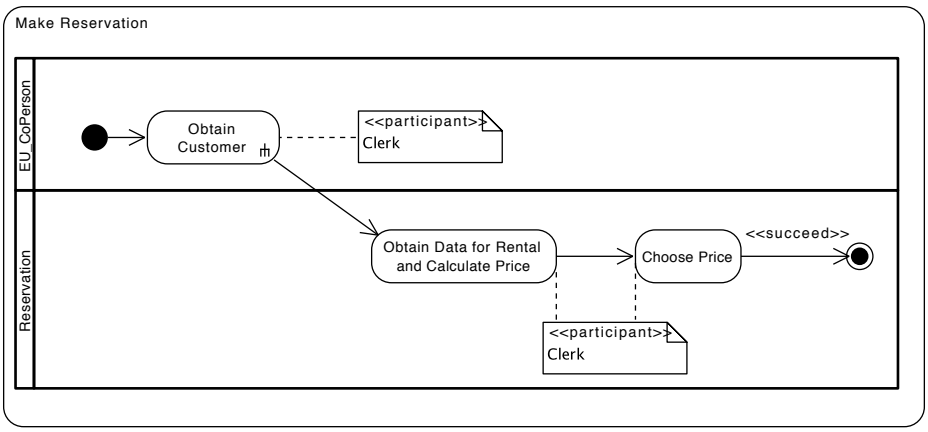

Figure 3.8: Activity Diagram for Make Reservation

3.5.1.1 Obtain Customer

See section 3.5.2 on page 25.

3.5.1.2 Obtain Data for Rental and Calculate Price

Obtains the data for the rental (such as beginning and end date, the countries the user wants to visit, the preferred car model or car group, etc.) and calculates

its price, considering the fact that there may be some applicable offers or the customer may be eligible to pay with points.

There are four different possible prices:

- *Basic Price* - It is calculated according to the rental duration, without considering any discounts.
- *Best Price* - It is calculated considering the existing discounts, excluding those discounts that were must be applied at reservation time. IMPORTANT NOTE: In the original specification [5], apparently *Best Price* and *Price with Special Discount* are calculated in the same way, considering in both cases discounts applicable at reservation time. We have considered that this is a mistake, and for *Best Price* we do not include the reservation-time discounts.
- *Price with Special Discount* - It considers all types of applicable discounts, including those than can only be selected at reservation time.
- *Points* - The payment with points can only be selected if the user is member of the Loyalty Incentive Scheme and has enough points. The cost in points of the rental is calculated from the Basic Price (or Base Price) of the rental.

Additional comments:

- Although there is an integrity constraint that does not allow users to pay with points if the reservation is not made 14 days in advance, this action checks it anyway, to avoid offering the user the option to pay with points if he is not able to.
- *points()* - changes from Money to Points.
- *isBetter()* - checks whether one alternative is better than the other.
- *durationT()* - obtains the corresponding duration given a period and a natural number.
- *applicable()* - used to determine if a particular discount is applicable to a customer.
- *apply()* - applies a discount to a particular price. It is needed because the Discount class contains this information in a String format, as it may be given as a percentage over the final price, certain conditions may have to be met, etc.

```
action obtainDataForRentalAndCalculatePrice (startDate: DateTime,
    endDate: DateTime, pickUpBranch: String, dropOffBranch:
    String, countries: Set(String), carG: String, carM: String,
    person: EU_CoPerson): Set(TupleType(id:PayType, desc:String))

localPre: –

localPost:
```

```
-- Change input EU_CoPerson into EU_RentPerson and again to
    Customer. At this point EU_CoPerson must already be
    EU_RentPerson but may not be a Customer --
person.oclAsType(EU_RentPerson).oclIsTypeOf(Customer) and
let c:Customer=person.oclAsType(Customer) in

-- Create Rental Agreement --
-- 1. Creates the RentalAgreement as a Reservation subtype with
    the input data --
-- 2. Links the EU_RentPerson with this RentalAgreement --
Reservation.allInstances()->exists(r |
        r.oclIsNew() and r.driver=c.oclAsType(EU_RentPerson) and
            r.renter=c and r.beginning=startDate and
            r.initEnding=endDate and r.reservationDate=now() and
        r.pickUpBranch=Branch.allInstances()->select(pub |
            pub.name=pickUpBranch) and
            r.dropOffBranch=Branch.allInstances()->select(dob |
            dob.name=dropOffBranch) and
        (if (carG = '') then
                r.requestedGroup=CarGroup.allInstances()->select(cg
                    | cg.worse->isEmpty())
        else
                r.requestedGroup=CarGroup.allInstances()->select(cg
                    | cg.name=carG)
        endif)
        and
        (if (carM <> '') then
                r.requestedModel=CarModel.allInstances()->select(cm
                    | cm.name=carM)
        else
                true
        endif)
        and
        countries->forAll(co2 |
            r.country->select(co|co.name=co2)->notEmpty()) and
            r.country->includes(Branch.allInstances()->select(b |
            b.name=pickUpBranch).country) and
            r.country->includes(Branch.allInstances()->select(b |
            b.name=dropOffBranch).country))


-- Calculate Price --
-- 1. basePrice and bestPrice are derived attributes in the
    class/business artifact. Therefore, there is no need to
    calculate them.--
        let basePr:Money=r.basicPrice
        let bestPr:Money=r.bestPrice
-- 2. We have to calculate the price considering the discounts
    available at reservation time --
        -- 2.1. We select those discounts applicable to the
            particular rentGroup and the time of the rental. We
            also check if its applicable to the Customer. --
        let applicableDiscounts:Set(Discount) =
            r.rentGroup.discount->select(dis |
            dis.beginningDate<=r.initEnding and
            (dis.oclIsTypeOf(ClosedDiscount) implies
            dis.oclAsType(ClosedDiscount).endingDate>=today() and
            applicable(dis,c)) in
        -- 2.2. We create a function to determine, of all
            applicableDiscounts, the best one for a particular
            duration --
        let bestDiscountPerDuration(rd:RentalDuration, price:
```

```
        Money) : Discount = applicableDiscounts->select(d |
        d.rentalDuration=rd)-> reject(disAct: Discount|
        applicableDiscounts -> select(d2 |
        d2.rentalDuration=rd) -> exists(disOther:Discount|
        apply(disOther, price).isBetter(apply(disAct,
        price)))->any()
    -- 2.3. We calculate the price of the rental including the
        discounts --
            -- 2.3.1. Each RentalAgreement is associated to
                various RentalDurations.
            -- 2.3.2. Each RentalAgreement is linked to
                various CarGroupDurationPrices (through
                bestDurationPrices). This contains the best
                price for each rental duration for the
                CarGroup of the RentalAgreement. That is, for
                every RentalDuration, there is exactly one
                CarGroupDurationPrice. --
            -- 2.3.3. This implies that, if we navigate the
                relationship bestDurationPrices and select the
                CarGroupDurationPrice for a particular
                RentalDuration, there will only be ONE
                CarGroupDurationPrice.
            -- 2.3.4. We calculate the price of the rental by
                iterating through the RentalDurations linked
                to the RentalAgreement and selecting the
                corresponding price in bestDurationPrices. We
                then obtain the best Discount for a particular
                RentalDuration and CarGroup, apply this
                Discount to the price in CarGroupDurationPrice
                and multiply this for the number of a
                particular RentalDuration there is in a
                RentalAgreement. Finally, we add this value to
                the accumulated price and we examine the next
                RentalDuration. --
    let bestSpD: Money =
        r.applicableRentalDurations->iterate(elem;
        tup:Tuple{currentPrice: Money=0, accPrice: Money=0} |
            currentPrice = r.bestDurationPrices -> select
                (cGDP |
                cGDP.rentalDuration=elem.rentalDuration).price
            currentPrice =
                apply(bestDiscountPerDuration(elem.rentalDuration,
                currentPrice), currentPrice)
            accPrice = accPrice + currentPrice*elem.quantity
            ).accPrice
    in
    answerSOptions=Sequence{}->append(Tuple{id=PayType::BasePrice,
        desc=
        basePr.toString})->append(Tuple{id=PayType::BestPrice,
        desc=bestPr.toString}) ->
        append(Tuple{id=PayType::SpecialDiscount,
        desc=bestSpD.toString})

-- Check if able to pay with points --
--- 1. Reservation must be made at least 14 days in advance --
--- 2. Customer must belong to Loyalty Incentive --
--- 3. Customer must have enough points to pay --
if (startDate >= (today()+day(14)) and
    p.oclIsTypeOf(LoyaltyMember) and (points(r.basicPrice) <=
    (c.oclAsType(LoyaltyMember).availablePoints))) then
        answerSOptions->append(Tuple{id=PayType::Points,
            desc=points.toString})
```

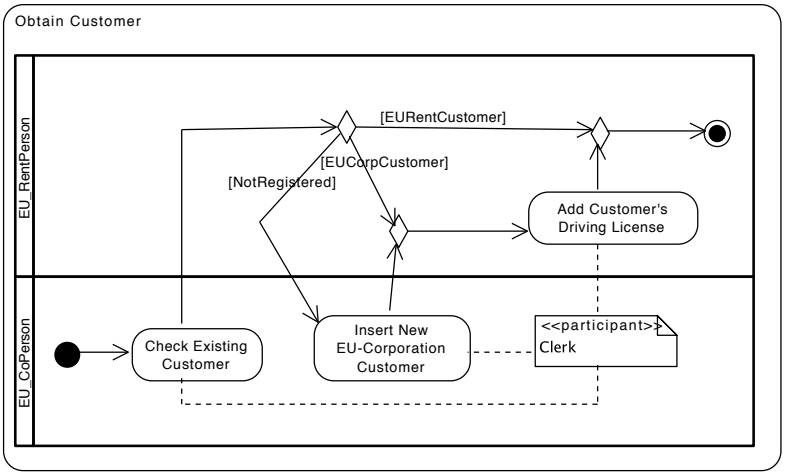

Figure 3.9: Activity Diagram for Obtain Customer

```
else
        true
endif

-- Return all prices and discounts available --
result = answerSOptions
```

3.5.1.3 Choose Price

The user chooses the price for his/her rental and decides whether to guarantee it or not (if he provides the credit card number, he wants to guarantee the rental).

```
action ChoosePrice (r: Reservation, pm: PayType, cc: Natural)

localPre: -

localPost:
if (pm = PayType::Points) then
        r.oclIsTypeOf(PointsPaymentReservation)
else
        if (pm = PayType::SpecialDiscount) then
                r.oclIsTypeOf(ReservationWithSpecialDiscount)
        else
                true
        endif
endif
if (cc <> null) then
        r.oclIsTypeOf(GuaranteedReservation) and
            r.oclAsType(GuaranteedReservation).creditCardNumber =
            cc
else
        true
endif
```

3.5.2 Obtain Customer

3.5.2.1 Check Existing Customer

Checks whether the user is already a customer and of what type (*EU_RentPerson*, *EU_CoPerson* or not registered).

```
action CheckExistingCustomer (cid: String):
    TupleType(cType:CustomerType, EU_CoP: EU_CoPerson)

localPre: -

localPost:
if (EU_RentPerson.allInstances() -> select(id=cid)->notEmpty())
    then
        result = Tuple{cType=CustomerType::EURentCustomer,
            EU_CoP=EU_CoPerson.allInstances()->select(id=cid)}
else
        if (EU_CoPerson.allInstances() ->
            select(id=cid)->notEmpty()) then
                result = Tuple{cType=CustomerType::EUCorpCustomer,
                    EU_CoP=EU_CoPerson.allInstances()->select(id=cid)}
        else
                result = Tuple{cType=CustomerType::NotRegistered,
                    EU_CoP=null}
        endif
endif
```

3.5.2.2 Insert New EU_CorporationCustomer

Inserts a new *EU_CoCustomer* after acquiring the customer's personal information.

Additional comments:

- The customer id (*cid*) in this operation must be the same as in the previous one.

```
action InsertNewEU-CorpCustomer (cid: String, cname: String,
    cbirthday: Date, cAddress: String, cTelephone: Natural):
    EU_CoPerson

localPre: -

localPost:
EU_CoPerson.allInstances()->exists (p | p.oclIsNew() and p.id=cid
    and p.name=cname and p.birthday=cbirthday and
    p.address=cAddress and p.telephone=cTelephone) and
result=p
```

3.5.2.3 Add Customer's Driving License

Adds a driving license to an *EU_CoPerson*, so that it becomes an *EU_RentPerson* and therefore eligible for renting a car with the company.

Additional comments:

- After having executed this operation, the *EU_CoPerson* will have been converted into an *EU_RentPerson*. Therefore, it is not necessary to return

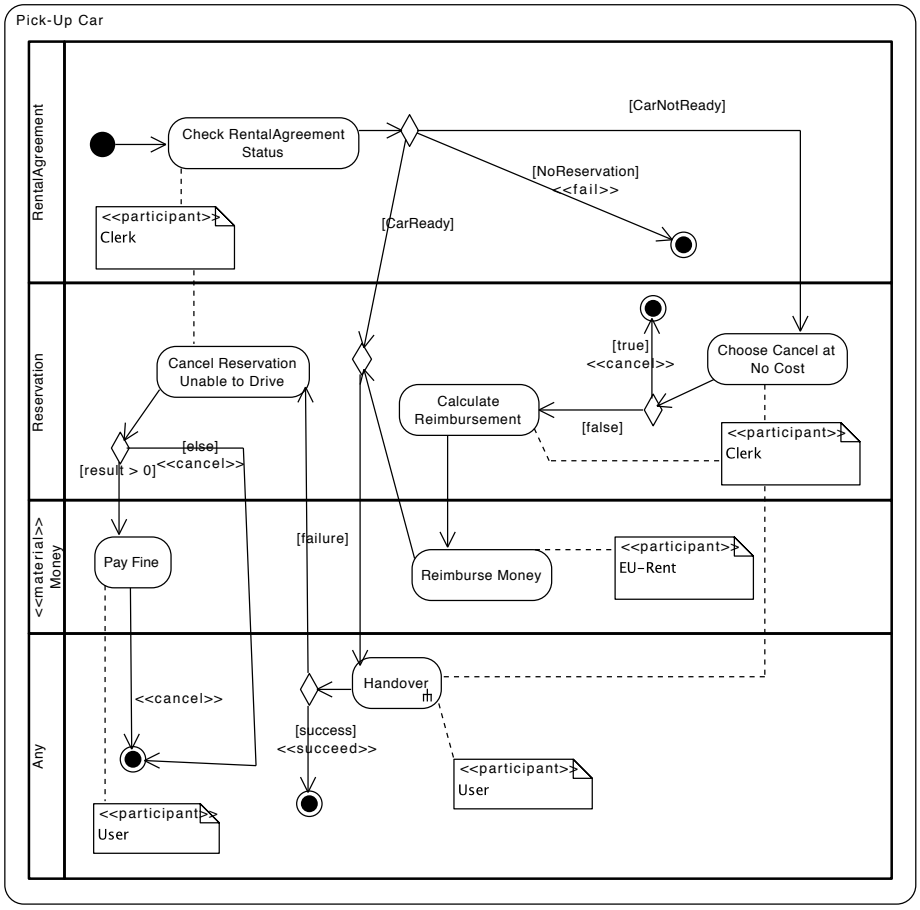

Figure 3.10: Activity Diagram for Pick-Up Car

the *EU_RentPerson* for the following operations, as it will be guaranteed that the *EU_CoPerson* is an *EU_RentPerson* as well.

```
action AddCustomersDrivingLicense (EU_CoP: EU_CoPerson, dExpiry:
    Date, dIssue: Date, lnumber: Natural): EU_RentPerson

localPre: –

localPost:
EU_CoP.oclIsTypeOf(EU_RentPerson) and
DrivingLicense.allInstances()->exists(l | l.oclIsNew() and
    l.number=lnumber and l.issue=dIssue and l.expiration=dExpiry
    and l.EU_RentPerson=EU_CoP.oclAsType(EU_RentPerson)) and
result = EU_CoP.oclAsType(EU_RentPerson)
```

3.5.3 Pick-up Car

3.5.3.1 Check RentalAgreement Status

Checks whether the customer identified by a certain id has a reservation for a car at the moment the action is called.

Additional comments:

- It returns a *RentalAgreement* because the *Handover* actions expect it. It cannot work with *Reservations* because *Handover* is also called from *Make Walk-In Rental*, where there is no *Reservation* for the *RentalAgreement*.

```
action CheckRentalAgreementStatus (cid: String): TupleType(status:
    ReservationStatus, time: DateTime, ra: RentalAgreement)

localPre: -

localPost:
-- Existing Reservation for Now --
let reserv: Reservation = Reservation.allInstances()-> select(r |
    r.renter.id=cid and r.beginning<=now() and
    not(r.oclIsKindOf(CanceledReservation)) and not
    r.oclIsKindOf(OpenRental)) in
if reserv->isEmpty() then
        result = Tuple{status = ReservationStatus::NoReservation,
            time=null, res=null}
else
        if (reserv.assignedCar->notEmpty()) then
                if (reserv.assignedCar.oclIsTypeOf(Prepared)) then
                        result = Tuple{status =
                            ReservationStatus::CarReady,
                            time=reserv.assignedCar.oclAsType(Prepared).actualTime,
                            res=reserv.oclAsType(RentalAgreement)}
                else
                        result = Tuple{status =
                            ReservationStatus::CarNotReady,
                            time=reserv.assignedCar.expectedPreparedTime,
                            res=reserv.oclAsType(RentalAgreement)}
                endif
        else
                result = Tuple{status =
                    ReservationStatus::CarNotReady, time=null,
                    res=oclAsType(RentalAgreement)}
        endif
 endif
```

3.5.3.2 Choose Cancel at No Cost

As the car is not ready, the customer is given the opportunity to cancel the reservation at no cost. In case the customer chooses to cancel it, the action cancels the reservation stating that the company (i.e. EU-Rent) is liable for the cancellation.

Additional comments:

- The *RentalAgreement* is also of the *Reservation* subtype, as we have chosen an existing *Reservation* in the previous operation.

```
action ChooseCancelAtNoCost (ra: RentalAgreement, cancel:
    Boolean): Boolean

localPre: -

localPost:
if (cancel) then
        ra.oclIsTypeOf(CanceledCompanyLiable) and
            ra.oclAsType(CanceledCompanyLiable).cancellationDate =
            now() and result = true
else
        result = false
endif
```

3.5.3.3 Calculate Reimbursement

The action calculates the reimbursement the company has to give to the customer in case the car was not ready at the scheduled pick-up time.

Additional comments:

- Car must be ready in order to calculate the appropriate refund.

```
action CalculateReimbursement (ra: RentalAgreement): Money

localPre carReady: ra.assignedCar->notEmpty() and
    ra.assignedCar.oclIsTypeOf(Prepared)

localPost:
let hourlyPaid: Money=
    ra.bestDurationPrices->select(b|b.rentalDuration.minimumDuration=1
    and b.rentalDuration.timeUnit=hour).price
let hours: Integer= (ra.assignedCar.oclAsType(Prepared).actualTime
    - self.reservation.beginning.Time()).floor() in
result=hours*hourlyPaid
```

3.5.3.4 Reimburse Money

EU-Rent reimburses money to the customer for not having the car ready.

```
Deals with material resources.
```

3.5.3.5 Handover

Check section 3.5.4 on page 29.

3.5.3.6 Cancel Reservation Unable to Drive

Cancels the reservation because the customer is not fit to drive the car, e.g. he may be under the influece of illegal drugs or alcohol.

```
action cancelReservationUnableToDrive(ra: RentalAgreement): Natural

localPre: -

localPost:
```

```
ra.oclIsTypeOf(CanceledCustomerLiable) and
    ra.oclAsType(CanceledCustomerLiable).cancellationDate = now()
    and
    ra.oclAsType(CanceledCustomerLiable).motivation=CancellingMotivation::unable_to_drive
    and
if (ra.oclIsTypeOf(GuaranteedCancel)) then
        result=ra.oclAsType(GuaranteedCancel).fine
else
        result=0
endif
```

3.5.3.7 Pay Fine

The user has to pay a fine for not being in an appropriate condition to drive the car.

```
Deals with material resources.
```

3.5.4 Handover

3.5.4.1 Verify State of Customer

Checks that the customer is in a right state (e.g physically capable, not under the influence of illegal drugs or drunk, etc.) to drive the car.

```
Deals with material resources.
```

3.5.4.2 Verify State of Driver

Checks that the driver is in a right state (e.g physically capable, not under the influence of illegal drugs or drunk, etc.) to drive the car.

```
Deals with material resources.
```

3.5.4.3 Check Requirements Fulfilment

Check section 3.5.5 on page 31.

3.5.4.4 Sign Additional Driver's Authorization

The additional driver has to sign an authorization in order to be allowed to drive the car.

```
Deals with material resources.
```

3.5.4.5 Add Driver to Rental

Given an *EU_CoPerson* and a *RentalAgreement*, the action adds the driver to the given rental.

Additional comments:

- The previous operation makes sure that the *EU_CoPerson* is also a *EU_RentPerson*.

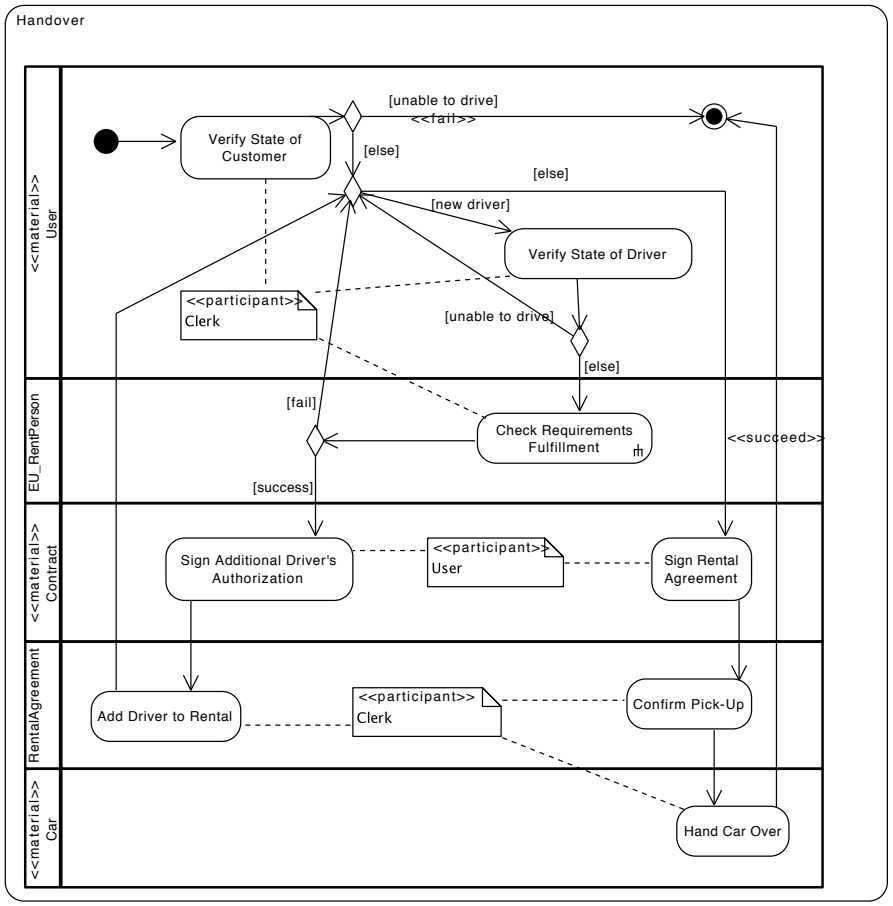

Figure 3.11: Activity Diagram for Handover

```
action addDriverToRental (p: EU_CoPerson, ra: RentalAgreement)

localPre:
-- We need to check these conditions here as we do not want the
    whole activity diagram to abort execution if they are not met
    --
ra.beginning<=now() and ra.assignedCar.oclIsTypeOf(Prepared)

localPost:
ra.driver->includes(p.oclAsType(EU_RentPerson))
```

3.5.4.6 Sign Rental Agreement

The customer signs the rental agreement in order to accept the rental conditions and be able to rent the car.

```
Deals with material resources
```

3.5.4.7 Confirm Pick-Up

Confirms that the car has been picked up and the rental is open.

```
action confirmPickUp (ra: RentalAgreement)

localPre: -

localPost:
ra.oclIsTypeOf(OpenRental) and
    ra.oclAsType(OpenRental).actualPickUpTime=now()
```

3.5.4.8 Hand Car Over

The car is given to the customer.

```
Deals with material resources
```

3.5.5 Check Requirements Fulfilment

3.5.5.1 Check Existing Person

Has the same OCL code as action *Check Existing Customer* in section 3.5.2.1, page 25.

3.5.5.2 Insert New EU-Corporation Driver

Inserts a new EU-Corporation customer using his/her personal data.

Additional comments:

- The postcondition checks whether the person is over 25 years of age, a condition which is guaranteed by the integrity constraints. However, in this particular case, we do not want to cancel the whole process if the person does not fulfill the requirements, as it is simply an additional driver and not the customer.

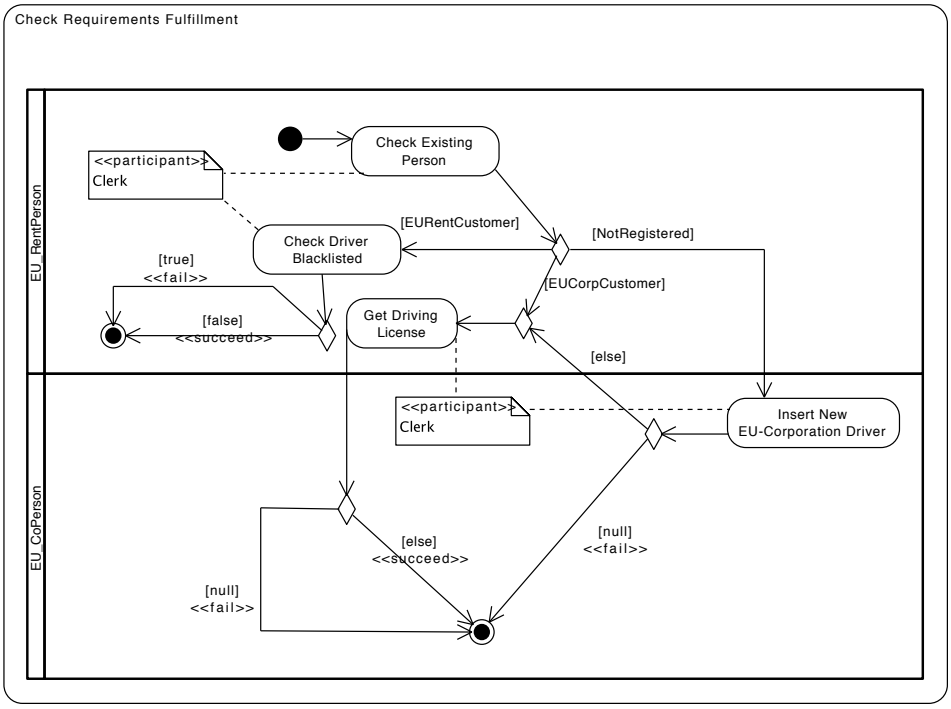

Figure 3.12: Activity Diagram for Check Requirements Fulfilment

```
action insertNewEU-CorporationDriver (cid: String, cname: String,
    cbirthday: Date, cAddress: String, cTelephone: Integer):
    EU_CoPerson

localPre: -

localPost:
if ( (today()-cbirthday)<year(25) ) then
        result=null
else
        EU_CoPerson.allInstances()->exists(p | p.oclIsNew() and
            p.id=cid and p.name=cname and p.birthday=cbirthday and
            p.address=cAddress and p.telephone=cTelephone) and
        result=p
endif
```

3.5.5.3 Get Driving License

Obtains the driver's license information and creates a *EU_RentPerson*.
Additional comments:

- The postcondition checks whether the driving license is valid, a condition which is guaranteed by the integrity constraints. However, in this particular case, we do not want to cancel the whole process if the license is not valid, as it is simply the driving license of an additional driver and not the customer.

```
action getDrivingLicense (p: EU_CoPerson, dExpiry: Date, dIssue:
    Date, lnumber: Integer)

localPre: -

localPost:
if ((dExpiry < dIssue) or (dExpiry < today()) or ((today() -
    dIssue) > year(1))) then
        result=null
else
        p.oclIsTypeOf(EU_RentPerson) and
        DrivingLicence.allInstances()->exists(l | l.oclIsNew() and
            l.number=lnumber and l.issue=dIssue and
            l.expiry=dExpiry and r.drivingLicense=l) and
        result=r
```

3.5.5.4 Check Driver Blacklisted

Checks whether the *EU_RentPerson* has been blacklisted.

Additional comments:

- We need to check whether the additional driver has been blacklisted. If he has, the operation fails but it does not imply the failure of the whole subprocess, just the insertion of the new driver (as it is shown in the activity diagram).

```
action checkDriverBlacklisted (EU_CoP: EU_CoPerson): Boolean

localPre: -

localPost:
result=EU_CoP.oclAsType(EU_RentPerson).oclIsTypeOf(Blacklisted)
```

3.5.6 Make Walk-In Rental

3.5.6.1 Obtain Customer

See section 3.5.2 on page 25.

3.5.6.2 Obtain Rental Data

The action obtains the data for the rental (such as the beginning and end dates, the countries the customer wants to travel to with the car, the preferred car group or car model, etc.) and creates the RentalAgreement.

Additional comments:

- The operation creates the *RentalAgreement*. There is no need to create a Reservation because the *Customer* will take the car with him immediately.
- As we have previously called the action *Obtain Customer*, we can guarantee that the *EU_CoPerson* is already a *EU_RentPerson*.
- We need a way to identify the branch from which the system is being run. So far, we have a function, *currentBranch()*, that returns the *Branch* from which the *Reservation* is being made.

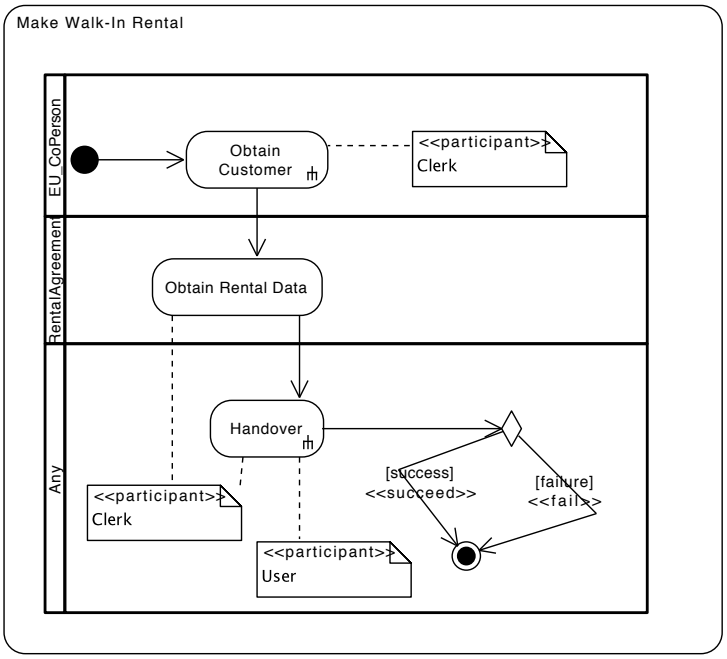

Figure 3.13: Activity Diagram for Make Walk-In Rental

- We assign the car directly to the *RentalAgreement*. To do so, we first make sure that only currently available *CarModels* and *CarGroups* are selected. If the user chooses a *CarModel*, then we assign the car with least mileage belonging to that *CarModel*. If the user selects a *CarGroup* (or if he has not selected any), then we assign the car with the least mileage from that group (or with the least mileage if he has not specified a group).
- The description of the case study states that, when assigning cars, the absolute mileage should be considered instead of the car's mileage since its last service. However, in the original operation's specification, the mileage since the last service is used. We have considered that this is a mistake, and therefore we have used the car's absolute mileage.
- The specification in the original technical report does not calculate nor show the cost of the rental to the customer, as he/she will not be able to select any special offers. Apparently, then, there is no need to calculate the cost of the rental.

```
action obtainRentalData(endDate: Date, pickUpBranch: String,
    dropOffBranch: String, countries: Set(String), carG: String,
    carM: String, p: EU_CoPerson): RentalAgreement

localPre availableCarModel: carM<>'' implies
    currentBranch().carsAvailableNow.carModel.name->includes(carM)
localPre availableCarGroup: carG<>'' implies
    currentBranch().groupsAvailableNow.name->includes(carG)
localPre availableCars: (carM = '' and carG = '' ) implies
    currentBranch().carsAvailableNow->notEmpty()
```

localPost:

```
let c:EU_RentPerson=p.oclAsType(EU_RentPerson) in
-- Create Rental Agreement --
RentalAgreement.allInstances() -> exists(ra.oclIsNew() and
    ra.driver=c and c.isTypeOf(Customer) and
    ra.renter=c.oclAsType(Customer) and ra.beginning=now() and
    ra.initEnding=endDate and ra.pickUpBranch=currentBranch() and
    ra.dropOffBranch=Branch.allInstances()->select(dob |
    dob.name=dropOffBranch) and
-- We assign the car model with the least mileage --
(if (carM <> '') then
        ra.car = currentBranch().carsAvailableNow -> select(c |
            c.carModel.name=carM)->sortedBy(currentMileage) ->
            first()
 else
        (if (carG = '') then
                ra.car = currentBranch().carsAvailableNow ->
                    sortedBy(currentMileage) -> first()
         else
                ra.car = currentBranch().carsAvailableNow ->
                    select(c | c.carGroup.name=carG) ->
                    sortedBy(currentMileage) -> first()
        endif)
 endif)
and
-- We add the countries to the list, including the branches'
    countries--
countries->forAll(co2 |
    ra.country->select(count|count.name=co2)->notEmpty()) and
ra.country->includes(currentBranch().country) and
ra.country->includes(Branch.allInstances()->select(b |
    b.name=dropOffBranch).country) and
-- We return the Rental Agreement --
result = ra)
```

3.5.6.3 Handover

Check section 3.5.4 on page 29.

3.5.7 Extend Rental Agreement

3.5.7.1 Call Branch

The customer calls an EU-Rent branch to ask for a rental extension.

```
Deals with material resources.
```

3.5.7.2 Obtain ID, Data for Extension and Verify

This action extends a Rental Agreement as long as the customer has an open rental that has not been closed, the new end date is later that the previous end date and the car is not in need of maintenance.

Additional comments:

- It does not check for overlapping rentals as this is guaranteed by the integrity constraints.

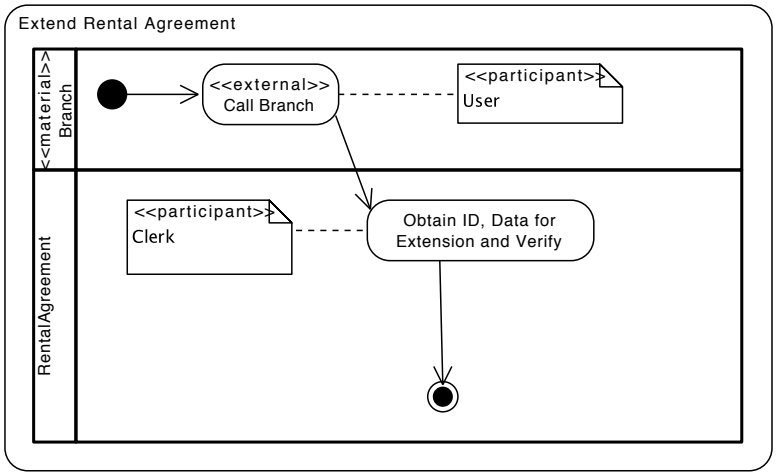

Figure 3.14: Activity Diagram for Extend Rental Agreement

- Extension must be applied to the currently *OpenRental*.
- It is necessary to check that Rental is not *ClosedRental* because *ClosedRental* is a subclass of *OpenRental*.
- New end date must be later than *agreedEnding*.

```
action obtainIDDataExtensionVerify (cid: String, newEndDate:
    DateTime): Boolean

localPre customerHasOpenRental:
        Customer.allInstances()->
            select(c|c.id=cid).rentalAgreement -> select (ra |
            ra.oclIsTypeOf(OpenRental) and not
            ra.oclIsTypeOf(ClosedRental))->notEmpty()

localPre laterReturnDate:
        let rental: OpenRental = (Customer.allInstances()->
            select(c|c.id=cid).rentalAgreement -> select (ra |
            ra.oclIsTypeOf(OpenRental) and not
            ra.oclIsTypeOf(ClosedRental))).oclAsType(OpenRental) in
        rental.agreedEnding < newEndDate

localPost:
let currentRental: OpenRental=
Customer.allInstances()-> select(c|c.id=cid).rentalAgreement ->
    select (ra | ra.oclIsTypeOf(OpenRental) and not
    ra.oclIsTypeOf(ClosedRental)).oclAsType(OpenRental)
in
if (currentRental.car.oclIsTypeOf(NeedsMaintenance)) then
        result=false
else
        currentRental.oclIsTypeOf(ExtendedRental) and
            currentRental.oclAsType(ExtendedRental).lastNewEnding
            = newEndDate and
            currentRental.oclAsType(ExtendedRental).extension.extensionDone
            = now() and result=true
endif
```

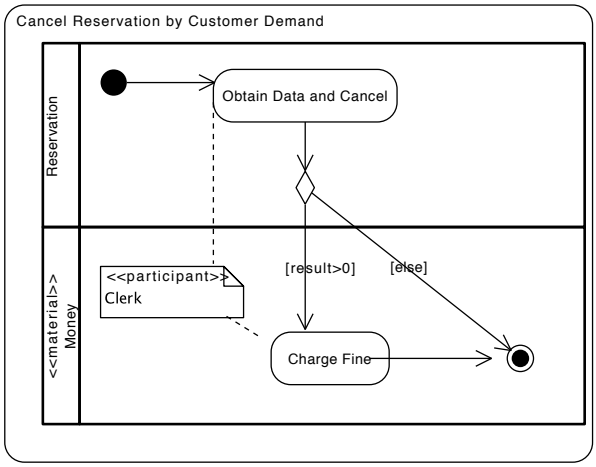

Figure 3.15: Activity Diagram for Cancel Reservation by Customer Demand

3.5.8 Cancel Reservation by Customer Demand

3.5.8.1 Obtain Data and Cancel

This action obtains the startDate of a rental and a user's id and cancels the corresponding reservation. It returns the money that the customer has to be charged for the cancellation (may be 0).

Additional comments:

- The description of the use case states that a car should be freed if it had been previously assigned to a no-show reservation. However, in the original operation's specification this is not taken care of. It is not taken care of here either.
- A fine should be charged if reservation is cancelled on pick-up day. This action returns money that has to be charged.

```
action obtainDataAndCancel (cid: String, startDate: DateTime):
    Money

localPre: -

localPost:
let ra:RentalAgreement = Customer.allInstances()->
    select(c|c.id=cid and
    c.beginning=startDate).rentalAgreement->select(r |
    r.oclIsTypeOf(Reservation) and not
    r.oclIsKindOf(CanceledReservation) and not
    r.oclIsKindOf(OpenRental)) in
ra.oclIsTypeOf(CanceledReservation) and
    ra.oclIsTypeOf(CanceledCustomerLiable) and
    ra.oclAsType(CanceledCustomerLiable).motivation=CancelingMotivation::customer_canceled
    and
    ra.oclAsType(CanceledCustomerLiable).cancellationDate=now() and
if (ra.oclIsTypeOf(GuaranteedCancel)) then
        result=ra.oclAsType(GuaranteedCancel).fine
```

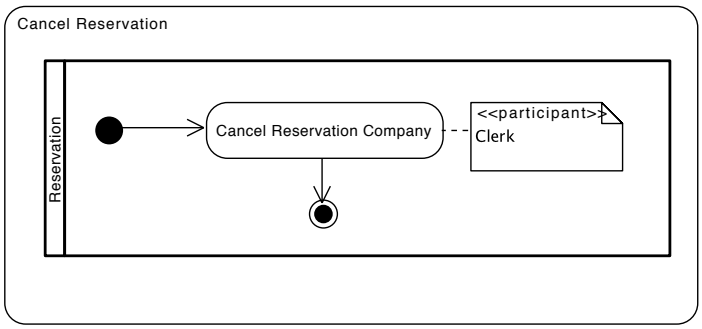

Figure 3.16: Activity Diagram for Cancel Reservation

```
else
        result=0
endif
```

3.5.8.2 Charge Fine

The customer is charged a fine for the cancellation of the reservation.

```
Deals with material resources.
```

3.5.9 Cancel Reservation

3.5.9.1 Cancel Reservation Company

This action cancels a user's reservation at the request of EU-Rent. However, the customer may also have to pay a fine if the company is forced to cancel it due to a customer's fault (e.g. becoming blackslisted).

Additional comments:

- The original specification for this operation did not charge the user for cancelling the reservation. However, we consider that if the company is forced to cancel a reservation because of the user's fault, the user should be charged as if it had been a no-show reservation. The original description, in fact, states that this is so.
- We have included the *charge* operation in this action, instead of having a separate action for it, because this is done automatically and there is no interaction with the user.
- *charge()* - charges the cancellation cost to the user.

```
action CancelReservationCompany (res: Reservation, reason:
    CancellingMotivation)

localPre: -

localPost:
```

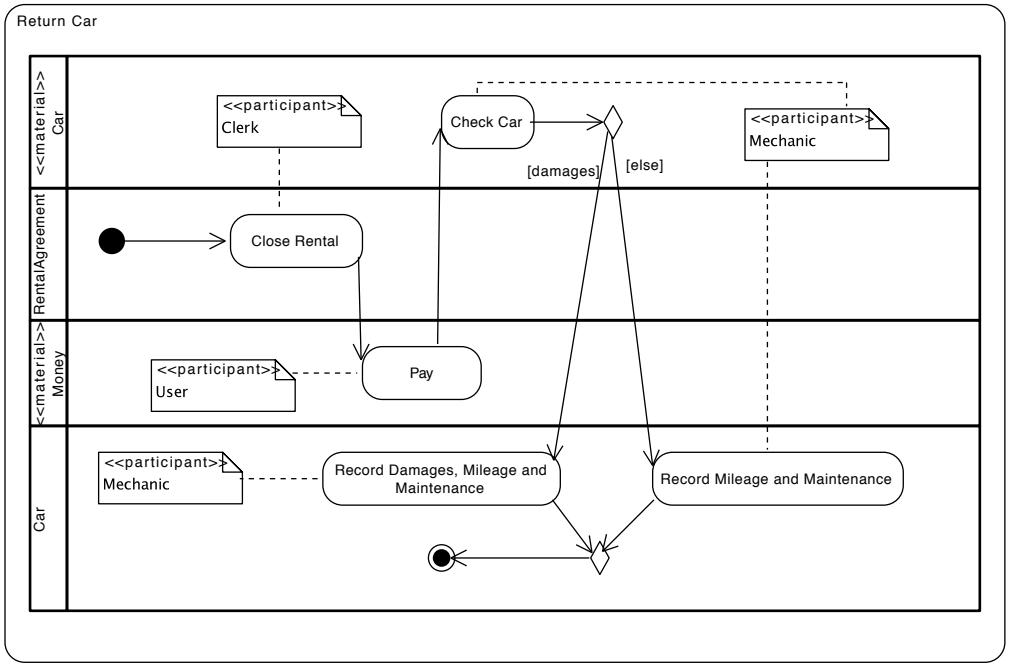

Figure 3.17: Activity Diagram for Return Car

```
res.oclIsTypeOf(CanceledCustomerLiable) and
    res.oclAsType(CanceledCustomerLiable).motivation=reason and
    res.oclAsType(CanceledCustomerLiable).cancellationDate=now()
if (res.oclIsTypeOf(GuaranteedCanceled)) then
        charge(res.oclAsType(GuaranteedCanceled).fine)
else
        true
endif
```

3.5.10 Return Car

3.5.10.1 Close Rental

This action closes the corresponding rental after a customer returns the car. It calculates the final price of the rental and, if the customer has returned the car later than expected, a bad experience is recorded.

Additional comments:

- Checks if the user fulfils any blacklisting criterion. If *ClosedRental (closedR)* is a *LateReturn*, then we add a *BadExperience* of this type to the *ClosedRental*, we calculate the degree of the *BadExperience* and the *Customer* loses his membership to the Loyal Incentive Scheme.
- As the car has been returned, we must indicate the new type of the rental, *ClosedRental*.
- If the car is returned to a branch other than the *pickUpBranch*, then car ownership is transfered to the *dropOffBranch*.

- Obtains the data to charge the user for the rental (payment type and credit card number).
- To calculate the final price, *LateReturn* and a drop-off charge are considered.
- It is not clear whether we should record a bad experience and membership loss for the customer and the additional drivers. We have considered that it only affects the renter (i.e. the customer).
- *currentBranch()* - returns the branch from which the system is being executed.
- *degree()* - calculates the degree of the customer's fault and returns a Level.
- *dropOffPenalty()* - calculates cost of dropping off the car at a different branch than expected.

```
action CloseRental (pid: String, paymentT: PayType, cc: Integer):
    TupleType(retCar: Car, money: Money)

localPre: -

localPost:
OpenRental.allInstances()->select(r | r.renter.id=pid and not
    r@pre.oclIsTypeOf(ClosedRental)).oclIsTypeOf(ClosedRental)
let closedR: closedRental = OpenRental.allInstances()->select(r |
    r.renter.id=pid and not
    r@pre.oclIsTypeOf(ClosedRental)).oclAsType(ClosedRental) in
let dropPenalty: Boolean = closedR.dropOffBranch <>
    currentBranch() in
closedR.actualReturn=now() and
    closedR.actualReturnBranch=currentBranch() and
(closedR.actualReturnBranch <> closedR.pickUpBranch implies
        closedR.actualReturnBranch.car->includes(closedR.car) and
            closedR.pickUpBranch.car->excludes(closedR.car)) and
if (closedR.oclIsKindOf(LateReturn)) then
        FaultSeriousness.allInstances()->exists(fs | fs.oclIsNew()
            and fs.badExperience.type=BadExpType::lateReturn and
            fs.closedRental=closedR and
            fs.degree=degree(closedR.oclAsType(LateReturn).extraInterval))
            and
        not closedR.customer.oclIsTypeOf(LoyaltyMember)
        if dropPenalty then
        result = Tuple{retCar=closedR.car,
            money=closedR.rentalPriceWithTax +
            closedR.oclAsType(LateReturn).extraCostWithTax +
            dropOffPenalty()}
        else
        result=Tuple{retCar=closedR.car,
            money=closedR.rentalPriceWithTax +
            closedR.oclAsType(LateReturn).extraCostWithTax}
        endif
else
        if dropPenalty then
                result=Tuple{retCar=closedR.car,
                    money=closedR.rentalPriceWithTax+dropOffPenalty()}
        else
                result=Tuple{retCar=closedR.car,
                    money=closedR.rentalPriceWithTax}
```

```
        endif
endif
and
closedR.paymentType=paymentT and closedR.creditCarNumberDamages=cc
```

3.5.10.2 Pay

The customer pays for the rental.

```
Deals with material resources.
```

3.5.10.3 Check Car

The mechanic checks the car for any damages.

```
Deals with material resources
```

3.5.10.4 Record Damages, Mileage and Maintenance

This action records the new mileage of a car and, as the car has been damaged, it records a bad experience for the customer and schedules the car reparations. It also checks if the car needs maintenance.

Additional comments:

- It is not clear if all drivers lose Loyalty Incentive Membership or only the renter. However, drivers don't have to be customers, and the ones that can belong to the Loyalty Incentive are customers. Therefore, we have considered that the renter is the only one who loses the Loyalty Incentive Membership.
- If the customer is blacklisted, when this operation ends the integrity constraints are not satisfied, as the customer's reservations are not cancelled. This is, apparently, a contradiction with the assumption that, if at the end of an activity diagram the system's constraints are not fulfilled, then the whole activity diagram is reverted. However, in the state machine diagram, it is shown how a customer being blacklisted implies a cancellation of the reservation.
- *charge()* - Automatically charges customer's credit card. We have not included this as a separate action because we have considered that it is done automatically without the user's nor the clerk's involvement.
- *blacklistingCriteriaAchieved()* - Checks if the user fulfills the blacklisting criteria.
- *getMaintenanceDate()* - Obtains a date for which car maintenance can be performed.

```
action RecordDamagesMileageMaintenance (retCar: Car, deg: Level,
    dcost: Money, mileage: Double)

localPre CorrectMileage:
        if (retCar.oclIsTypeOf(OwnCar)) then
```

```
                retCar.oclAsType(OwnCar).currentMileage < mileage
        else
                true
        endif

localPost:

-- Obtain the last rental associated with the car --
let closedR: closedRental = retCar.rentalAgreement->select(ra |
    ra.oclIsKindOf(ClosedRental))->forAll(ra |
    ra.oclAsType(ClosedRental))->sortedBy(actualReturn)->last()
in
-- Record a bad experience--
FaultSeriousness.allInstances()->exists(fs | fs.oclIsNew() and
    fs.closedRental=closedR and
    fs.badExperience.type=BadExpType::carDamage and fs.degree=deg)
    and
-- Record Car Damage --
DamageCost.allInstances()->exists(dc.oclIsNew() and
    dc.closedRental=closedR and dc.price=dcost and
    dc.carDamage.type=BadExpType::carDamage) and
-- Charge Cost of Damages --
charge(closedR.creditCardNumberDamages, dcost) and
-- Schedule car reparations --
closedR.car.oclIsTypeOf(RepairsScheduled) and
    closedR.car.oclAsType(RepairsScheduled).beginningDate=today()
    and
-- Customer loses Loyalty Incentive membership --
not closedR.renter.oclIsTypeOf(LoyaltyMember) and
-- Check if customer should be blacklisted. If he had been
    blacklisted before he would not have been able to rent the car
    in the first place --
if blacklistingCriteriaAchieved (closedR.renter) then
        closedR.renter.oclAsType(EU_RentPerson).oclIsTypeOf(Blacklisted)
            and
            closedR.renter.oclAsType(Blacklisted).blacklistedDate=today()
else
        true
endif and
-- Update mileage if car belongs to EU_Rent and check if it needs
    maintenance
if retCar.oclIsTypeOf(OwnCar) then
        retCar.oclAsType(OwnCar).currentMileage=mileage and
        if (retCar.oclIsKindOf(NeedsMaintenance)) then
                retCar.oclIsTypeOf(MaintenanceScheduled) and
                    retCar.oclAsType(MaintenanceScheduled).beginningDate
                    = getMaintenanceDate()
        else
                true
        endif
else
        true
endif
```

3.5.10.5 Record Mileage, Maintenance

The actions updates the car mileage and checks if it needs maintenance or has to be sold.

```
action RecordMileageMaintenance (retCar: Car, mileage: Double)
```

```
localPre CorrectMileage:
        if (retCar.oclIsTypeOf(OwnCar)) then
                retCar.oclAsType(OwnCar).currentMileage < mileage
        else
                true
        endif

localPost:
-- Update mileage if car belongs to EU_Rent and check if it needs
    maintenance
if retCar.oclIsTypeOf(OwnCar) then
        retCar.ocAsType(OwnCar).currentMileage=mileage and
        if (retCar.oclIsKindOf(NeedsMaintenance)) then
                retCar.oclIsTypeOf(MaintenanceScheduled) and
                    retCar.oclAsType(MaintenanceScheduled).beginningDate
                    = getMaintenanceDate()
        else
                -- Car doesn't need maintenance and therefore we
                    check if it needs to be sold. We don't check
                    if it has been assigned because it has just
                    been returned --
                if (retCar.oclIsKindOf(NeedToBeSoldCar)) then
                        retCar.oclIsTypeOf(ToBeSoldCar)
                else
                        true
                endif
        endif
else
        true
endif
```

Bibliography

[1] K. Bhattacharya, N. S. Caswell, S. Kumaran, A. Nigam, and F. Y. Wu. Artifact-centered operational modeling: lessons from customer engagements. *IBM Syst. J.*, 46(4):703–721, October 2007.

[2] K. Bhattacharya, R. Guthman, K. Lyman, F. F. Heath III, S. Kumaran, P. Nandi, F. Wu, P. Athma, C. Freiberg, L. Johannsen, and A. Staudt. A model-driven approach to industrializing discovery processes in pharmaceutical research. *IBM Syst. J.*, 44(1):145–162, January 2005.

[3] Kamal Bhattacharya, Richard Hull, and Jianwen Su. A Data-Centric Design Methodology for Business Processes. In *Handbook of Research on Business Process Management*, pages 1–28. 2009.

[4] Elio Damaggio, Alin Deutsch, Richard Hull, and Victor Vianu. Automatic verification of data-centric business processes. In Stefanie Rinderle-Ma, Farouk Toumani, and Karsten Wolf, editors, *BPM 2011*, volume 6896, pages 3–16. Springer, 2011.

[5] Leonor Frías, Anna Queralt, and Antoni Olivé. EU-Rent Car Rentals Specification. Technical Report Technical report LSI-03-59-R, Universitat Politècnica de Catalunya, 2003.

[6] Richard Hull. Artifact-centric business process models: Brief survey of research results and challenges. In Robert Meersman and Zahir Tari, editors, *OTM 2008*, volume 5332 of *LNCS*, pages 1152–1163. Springer Berlin / Heidelberg, 2008.

[7] A Nigam and N S Caswell. Business artifacts: an approach to operational specification. *IBM Syst. J.*, 42(3):428–445, 2003.

[8] Antoni Olivé. *Conceptual Modeling of Information Systems*. Springer, 2007.

[9] OMG. Unified Modeling Language superstructure 2.4.1, 2011. Available at: `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF`.

[10] OMG. OMG Object Constraint Language version 2.3.1, 2012. Available at: `http://www.omg.org/spec/OCL/2.3.1/PDF/`.

[11] Anna Queralt and Ernest Teniente. Specifying the semantics of operation contracts in conceptual modeling. In *Journal on Data Semantics VII*, volume 4244 of *LNCS*, pages 33–56. Springer Berlin / Heidelberg, 2006.

Appendices

Appendix A

Structural Schema in OCL

The following appendix includes the definition of the integrity constraints and the derivation rules in OCL corresponding to class diagrams in Chapter 3.

A.1 Class Diagram

Following the method described in [8], in this section we present the class diagram with the corresponding operations that define the derivation rules for attributes and their relationships, together with the integrity constraints, also represented as operations.

A.2 Integrity Constraints

The following section defines, for each class in Chapter 3, its integrity constraints and derivation rules.

A.2.1 Branch

Id id key:

```
context Branch::nameIsKey() : Boolean
body: result=Branch.allInstances()-> isUnique(name)
```

Derived relationship *carsAvailableNow*

```
context Branch::carsAvailableNow(): Set(OwnCar)
body: result = self.car->select(c | c.oclIsKindOf(OwnCar) and
    c.oclAsType(OwnCar).available).oclAsType(OwnCar)
```

Derived relationship *groupsAvailableNow*

```
context Branch::carsAvailableNow(): Set(CarGroup)
body: result = self.carsAvailableNow.carModel.carGroup->asSet()
```

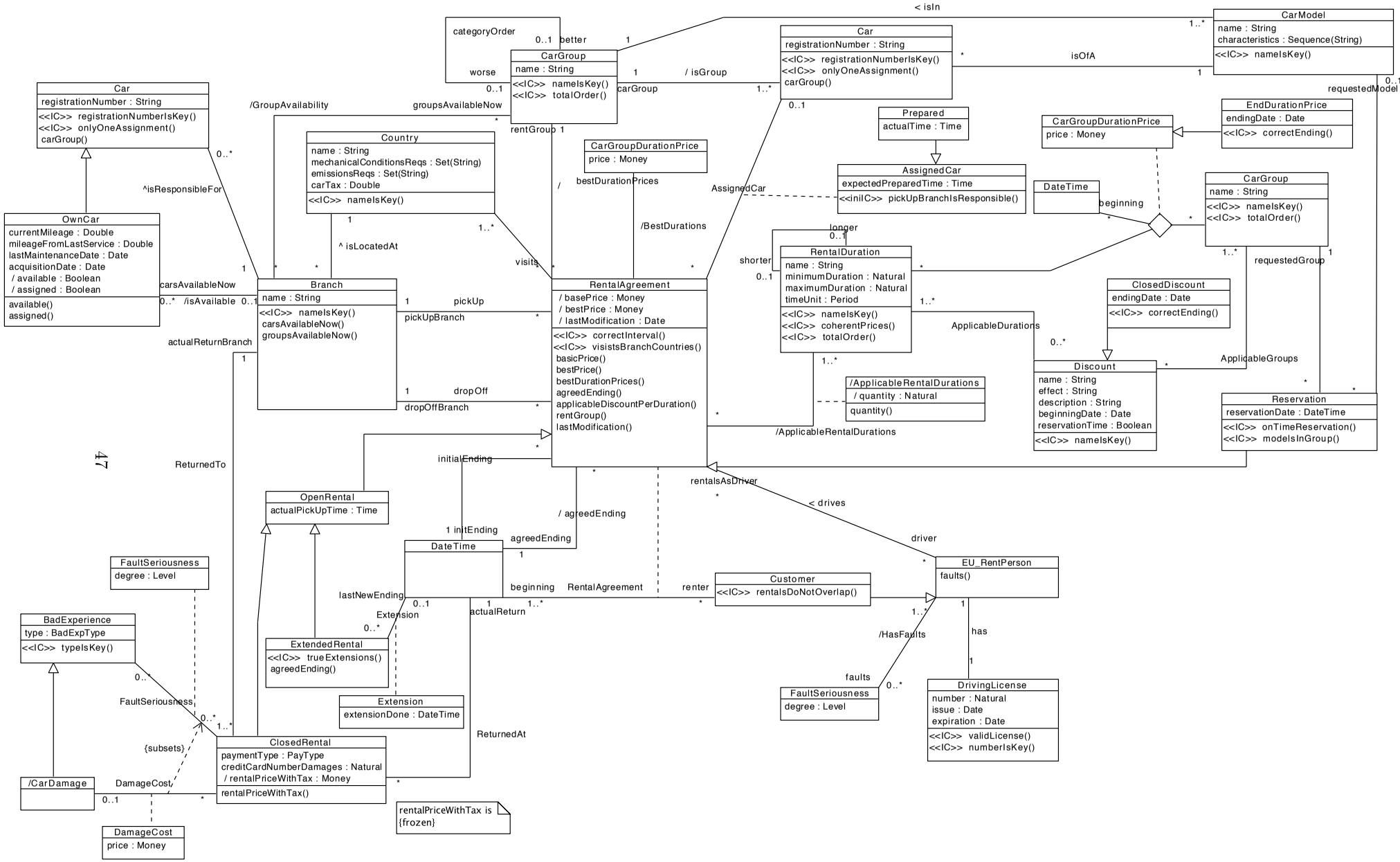

Figure A.1: Main class diagram for EU-Rent Car Rental Service.

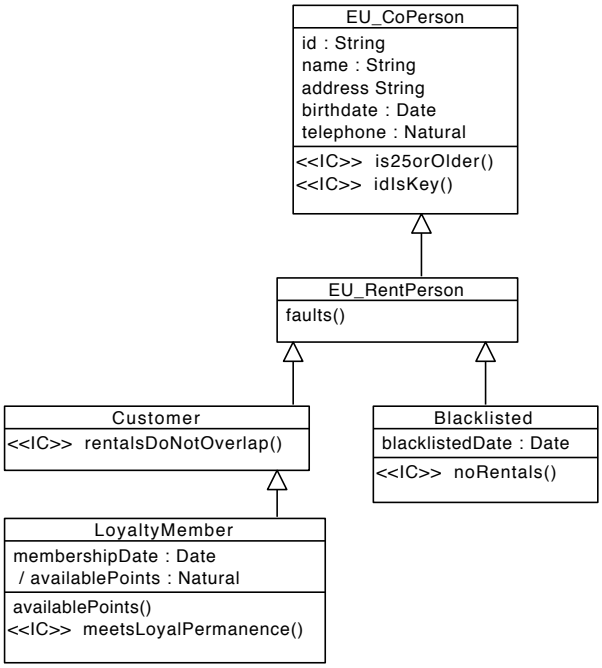

Figure A.2: Class diagram of EU_CoPerson and its subclasses

A.2.2 EU_CoPerson

Id is key:

```
context EU_CoPerson::idIsKey() : Boolean
body: result = EU_CoPerson.allInstances()->isUnique(id)
```

Must be 25 or older:

```
context EU_CoPerson::is25OrOlder() : Boolean
body: result = today()-self.birthdate() >= year(25))
```

A.2.3 EU_RentPerson

Derived relationship *faults*:

```
context EU_RentPerson::faults() : Boolean
body:
let faultsAsDriver: FaultSeriousness = self.rentalsAsDriver ->
    select(rA |
    rA.oclIsTypeOf(ClosedRental)).oclAsType(ClosedRental).faultSeriousness
let faultsAsRenter: FaultSeriousness = Customer.allInstances() ->
    select (c | c.id = self.id).rentalAgreement -> select(rA |
    rA.oclIsTypeOf(ClosedRental)).oclAsType(ClosedRental).faultSeriousness
in
result = faultsAsDriver->asSet()-> union(faultsAsRenter)->asSet()
```

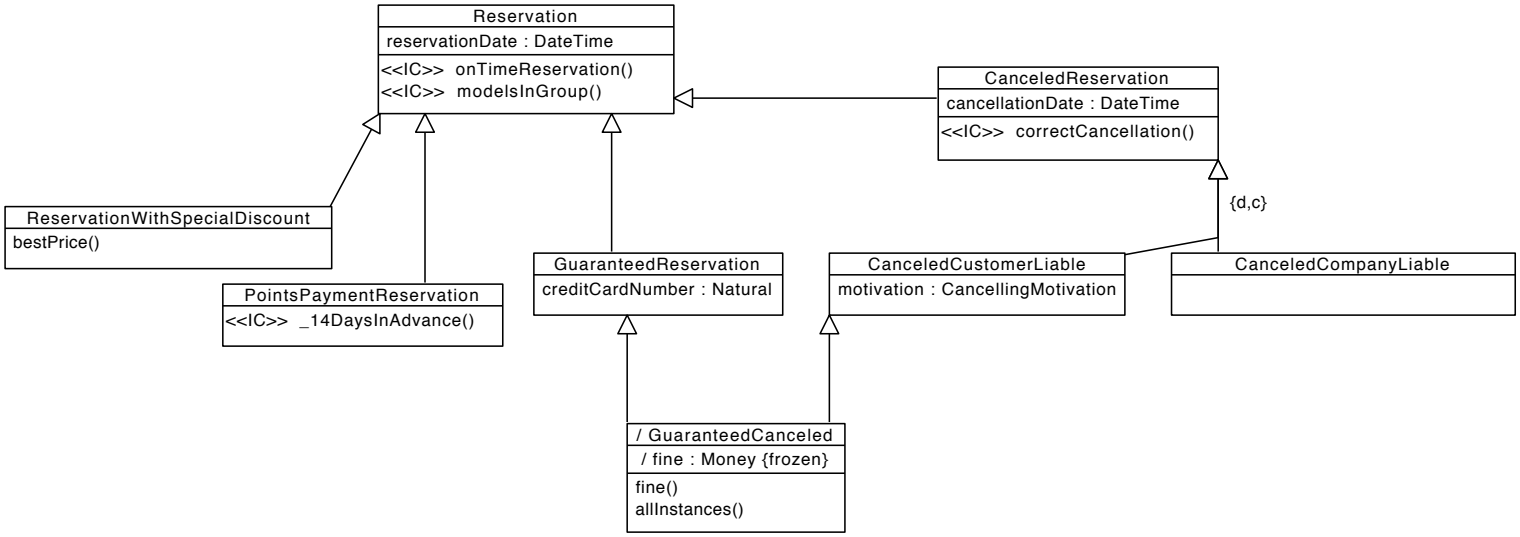

Figure A.3: Class diagram of Reservation and its subclasses

A.2.4 RentalAgreement

The pick-up and drop-off branches' countries must be included in the list of countries of the *RentalAgreement*:

```
context RentalAgreement:: visitsBranchCountries() : Boolean
body: result = self.Countries->includes(self.PickUpBranch.Country)
    and self.Countries->includes(self.DropOffBranch.Country)
```

Correct interval for rental agreement:

```
context RentalAgreement:: correctInterval() : Boolean
body: result=self.beginning< self.initEnding and
    self.actualReturn> self.beginning
```

Derived attribute *basicPrice*[1]:

```
context RentalAgreement:: basicPrice(): Money
body:
-- We have to calculate the price considering the best applicable
    prices, but without any discounts. --
-- 1. Each RentalAgreement is associated to various
    RentalDurations.
-- 2. Each RentalAgreement is linked to various
    CarGroupDurationPrices (through bestDurationPrices). This
    contains the best price for each rental duration for the
    CarGroup of the RentalAgreement. That is, for every
    RentalDuration, there is exactly one CarGroupDurationPrice. --
-- 3. This implies that, if we navigate the relationship
    bestDurationPrices and select the CarGroupDurationPrice for a
    particular RentalDuration, there will only be ONE
    CarGroupDurationPrice.
-- 2.3.4. We calculate the price of the rental by iterating
    through the RentalDurations linked to the RentalAgreement and
    selecting the corresponding price in bestDurationPrices. We
```

[1]This code has been changed from the original specification in [5].

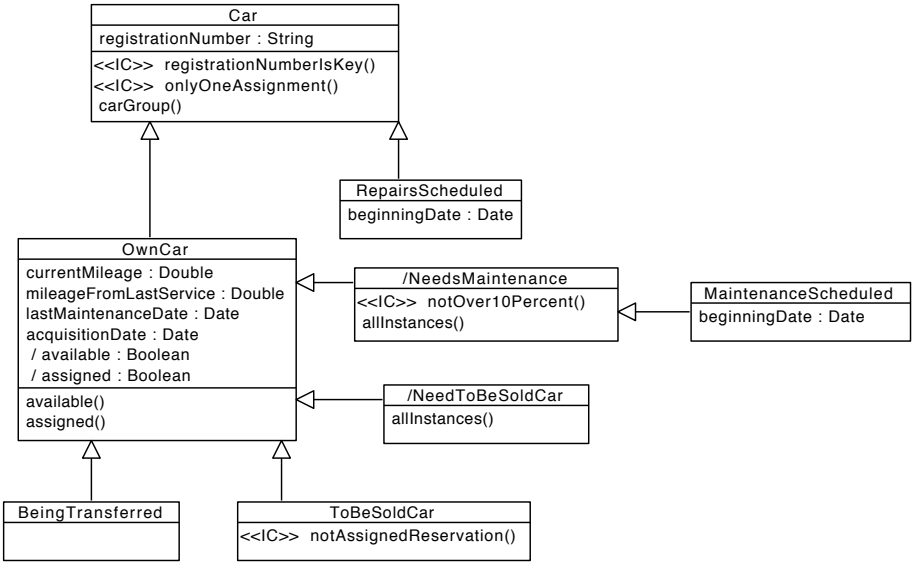

Figure A.4: Class diagram of Car and its subclasses

```
    then multiply this for the number of a particular
    RentalDuration there is in a RentalAgreement. Finally, we add
    this value to the accumulated price and we examine the next
    RentalDuration. --
                result =
                    self.applicableRentalDuration->iterate(elem;
                    tup:Tuple{currentPrice: Money=0, accPrice:
                    Money=0} |
                        currentPrice = self.bestDurationPrices ->
                            select (cGDP |
                            cGDP.rentalDuration=elem.rentalDuration).price
                        accPrice = accPrice +
                            currentPrice*elem.quantity
                        ).accPrice
```

Derived attribute *bestPrice*[2] :

```
context RentalAgreement:: bestPrice(): Money
body:

-- We have to calculate the price considering the discounts
    available. However, we must exclude those discounts that are
    only applicable at reservation time, as the function is in
    RentalAgreement and may not be of the Reservation subtype. --
-- 1. We select those discounts applicable to the particular
    rentGroup and the last modification of the rental, excluding
    those that must be selected at reservation time. We also check
    if its applicable to the Customer. --
        let
            applicableDiscounts:Set(Discount)=self.rentGroup.discount->select(dis
            | dis.beginningDate<=self.initEnding and
            (dis.oclIsTypeOf(ClosedDiscount) implies
            dis.oclAsType(ClosedDiscount).endingDate>=self.lastModification)
            and dis.reservationTime=false and applicable(dis,c)) in
```

[2]This code has been modified from the original specification in [5].

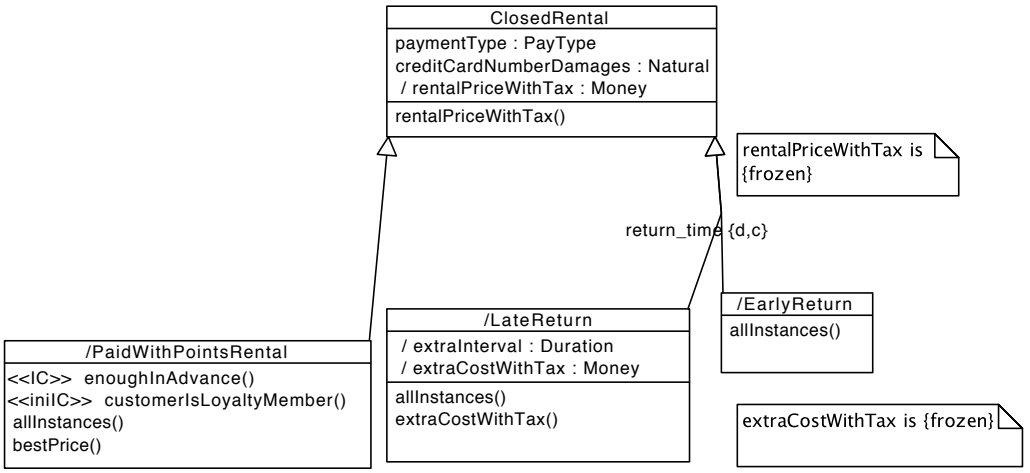

Figure A.5: Class diagram of ClosedRental and its subclasses

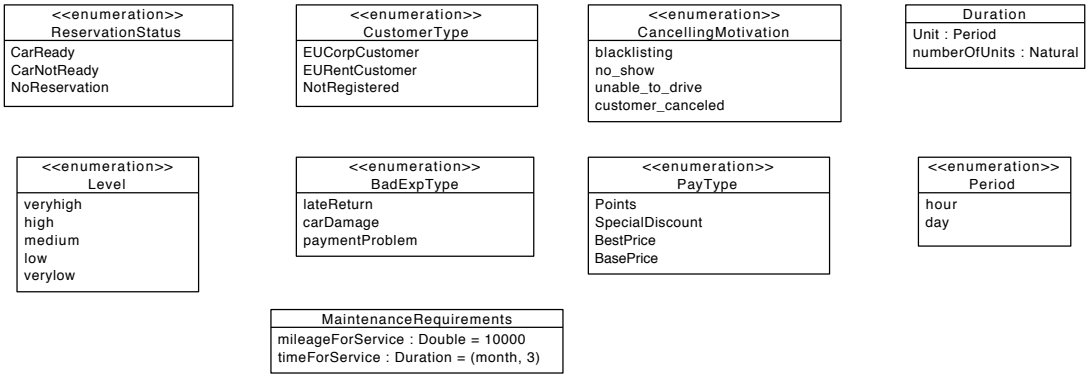

Figure A.6: Definition of types

```
-- 2. We create a function to determine, of all
    applicableDiscounts, the best one for a particular duration --
        let bestDiscountPerDuration(rd:RentalDuration, price:
            Money) : Discount = applicableDiscounts->select(d |
            d.rentalDuration=rd)-> reject(disAct: Discount|
            applicableDiscounts -> select(d2 |
            d2.rentalDuration=rd) -> exists(disOther:Discount|
            apply(disOther, price).isBetter(apply(disAct,
            price)))->any()
-- 3. We calculate the price of the rental including the discounts
    --
        -- 3.1. Each RentalAgreement is associated to various
            RentalDurations. --
        -- 3.2. Each RentalAgreement is linked to various
            CarGroupDurationPrices (through bestDurationPrices).
            This contains the best price for each rental duration
            for the CarGroup of the RentalAgreement. That is, for
            every RentalDuration, there is exactly one
            CarGroupDurationPrice. --
```

```
    -- 3.3. This implies that, if we navigate the relationship
       bestDurationPrices and select the
       CarGroupDurationPrice for a particular RentalDuration,
       there will only be ONE CarGroupDurationPrice. --
    -- 3.4. We calculate the price of the rental by iterating
       through the RentalDurations linked to the
       RentalAgreement and selecting the corresponding price
       in bestDurationPrices. We then obtain the best
       Discount for a particular RentalDuration and CarGroup,
       apply this Discount to the price in
       CarGroupDurationPrice and multiply this for the number
       of a particular RentalDuration there is in a
       RentalAgreement. Finally, we add this value to the
       accumulated price and we examine the next
       RentalDuration. --
        result =
            self.applicableRentalDuration->iterate(elem;
            tup:Tuple{currentPrice: Money=0, accPrice:
            Money=0} |
                currentPrice = self.bestDurationPrices ->
                    select (cGDP |
                    cGDP.rentalDuration=elem.rentalDuration).price
                currentPrice =
                    apply(bestDiscountPerDuration(elem.rentalDuration,
                    currentPrice), currentPrice)
                accPrice = accPrice +
                    currentPrice*elem.quantity
                ).accPrice
```

Derived attribute *lastModification*:

```
context RentalAgreement:: lastModification(): DateTime
body:
if self.oclIsTypeOf(Reservation) then
        result = self.reservationDate
else
        result = self.beginning
endif
```

Derived relationship *bestDurationPrices*

```
context RentalAgreement:: bestDurationPrices():
    Set(CarGroupDurationPrice)
body:
let applicableDuration: Set(CarGroupDurationPrice)=
    self.rentGroup.carGroupDurationPrice -> select(cg:
    CarGroupDurationPrice | cg.beginning<= self.ending and
    (cg.oclIsTypeOf(EndDurationPrice) implies
    cg.oclAsType(EndDurationPrice).endingDate >=
    self.lastModification)
let bestCurrentDuration: Set(CarGroupDurationPrice)=
    applicableDuration->reject(cgCur: CarGroupDurationPrice |
    applicableDuration-> exists(cgOther:CarGroupDurationPrice |
    cgOther.rentalDuration=cgCur.rentalDuration and
    cgOther.carGroup= cgCur.carGroup and
    cgOther.price<cgCur.price))
in
result = bestCurrentDuration -> sortedBy(rentalDuration.shorter)
```

Derived relationship *rentalDuration*[3]

[3]This relationship and its correspoding associative class do not appear in the original specification in [5]. We suppose that duration of a rental is measured either in days or hours.

```
context RentalAgreement:: rentalDuration(): Set(RentalDuration)
body:
let rentalDur:Duration = durationT(self.agreedEnding -
    self.initEnding)
let rentalDays:Natural = rentalDur.numberOfUnits in
        let possibleRentalDur:Set(RentalDuration) =
        if (rentalDur.unit = Period::day) then
                RentalDuration.allInstances()->select(rd |
                    rd.timeUnit=Period::day)->sortedBy(maximumDuration)->reverse()
        else
                -- The rental will only be for a few hours --
                RentalDuration.allInstances()->select(rd |
                    rd.timeUnit=Period::hour)->sortedBy(maximumDuration)->reverse()
        endif
        in
                possibleRentalDur->iterate(elem;
                    selRentalDur:OrderedSet(RentalDuration)->isEmpty()
                    |
                        if (rentalDays >= elem.maximumDuration)
                            then
                                selRentalDur=selRentalDur->including(elem)
                                rentalDays=rentalDays%maximumDuration
                        else
                                true
                        endif
                        if (rentalDays >= elem.minimumDuration)
                            then
                                selRentalDur=selRentalDur->including(elem)
                                rentalDays=rentalDays%minimumDuration
                        else
                                true
                        endif
                        )
        result = selRentalDur
```

Derived relationship *agreedEnding*.

```
context RentalAgreement:: agreedEnding(): DateTime
body: result= initEnding
```

Derived relationship *rentGroup*:

```
context RentalAgreement:: rentGroup(): CarGroup
body:
if self.oclIsKindOf(Reservation) then
        if self.car->isEmpty() or
            self.car.carGroup<>self.carGroup.worse then
                result=self.carGroup
        else
                result=self.carGroup.worse
        endif
else
        result=self.car.carGroup
endif
```

A.2.5 Reservation

Reservation date of a rental must be previous to its beginning date.

```
context Reservation:: onTimeReservation() :      Boolean
body: result=self.reservationDate < self.beginning
```

Requested car model must be in requested car group.

```
context Reservation:: modelIsInGroup() : Boolean
body: result=self.requestedModel->notEmpty() implies
    self.requestedModel.carGroup=self.requestedGroup
```

A.2.6 ReservationWithSpecialDiscount

Derived attribute *bestPrice*:

- This code has been modified from the original specification in [5].

```
context ReservationWithSpecialDiscount:: bestPrice() : Money
body:
-- We have to calculate the price considering the discounts
    available at reservation time --
-- 1. We select those discounts applicable to the particular
    rentGroup and last modification of the rental. We also check
    if its applicable to the Customer. --
        let
            applicableDiscounts:Set(Discount)=self.rentGroup.discount->select(dis
            | dis.beginningDate<=self.initEnding and
            (dis.oclIsTypeOf(ClosedDiscount) implies
            dis.oclAsType(ClosedDiscount).endingDate>=self.lastModification
            and applicable(dis,c)) in
-- 2. We create a function to determine, of all
    applicableDiscounts, the best one for a particular duration --
        let bestDiscountPerDuration(rd:RentalDuration, price:
            Money) : Discount = applicableDiscounts->select(d |
            d.rentalDuration=rd)-> reject(disAct: Discount|
            applicableDiscounts -> select(d2 |
            d2.rentalDuration=rd) -> exists(disOther:Discount|
            apply(disOther, price).isBetter(apply(disAct,
            price)))->any()
-- 3. We calculate the price of the rental including the discounts
    --
        -- 3.1. Each RentalAgreement is associated to various
            RentalDurations. --
        -- 3.2. Each RentalAgreement is linked to various
            CarGroupDurationPrices (through bestDurationPrices).
            This contains the best price for each rental duration
            for the CarGroup of the RentalAgreement. That is, for
            every RentalDuration, there is exactly one
            CarGroupDurationPrice. --
        -- 3.3. This implies that, if we navigate the relationship
            bestDurationPrices and select the
            CarGroupDurationPrice for a particular RentalDuration,
            there will only be ONE CarGroupDurationPrice. --
        -- 3.4. We calculate the price of the rental by iterating
            through the RentalDurations linked to the
            RentalAgreement and selecting the corresponding price
            in bestDurationPrices. We then obtain the best
            Discount for a particular RentalDuration and CarGroup,
            apply this Discount to the price in
            CarGroupDurationPrice and multiply this for the number
            of a particular RentalDuration there is in a
            RentalAgreement. Finally, we add this value to the
            accumulated price and we examine the next
            RentalDuration. --
```

```
result =
    self.applicableRentalDuration->iterate(elem;
    tup:Tuple{currentPrice: Money=0, accPrice:
    Money=0} |
        currentPrice = self.bestDurationPrices ->
            select (cGDP |
            cGDP.rentalDuration=elem.rentalDuration).price
        currentPrice =
            apply(bestDiscountPerDuration(elem.rentalDuration,
            currentPrice), currentPrice)
        accPrice = accPrice +
            currentPrice*elem.quantity
        ).accPrice
```

A.2.7 PointsPaymentReservation

PointsPaymentReservation must be made at least 14 days in advance of its beginning date.

```
context PointsPaymentReservation:: _14DaysInAdvance() : Boolean
body: result=(self.beginning-self.reservationDate)>=day(14)
```

A.2.8 CanceledReservation

Cancellation date of a reservation must be after or on the same reservation date and before the beginning date, on the same date or the day after. This has been changed from the original report.

```
context CanceledReservation:: correctCancellation() : Boolean
body: result=(self.cancellationDate>=self.reservationDate and
    self.cancellationDate<=(self.beginning+day(1)))
```

A.2.9 GuaranteedCanceled

Derived class:

```
context GuaranteedCanceled:: allInstances():
    Set(GuaranteedCanceled)
body: result=CanceledCustomerLiable.allInstances()->
    intersection(GuaranteedReservation.allInstances())
```

Derived attribute *fine*:

```
context GuaranteedCanceled:: fine() : Money
body:
if self.beginning=self.cancellationDate then
        result = self.bestDurationPrices->select(cGDP |
            not(cGDP.oclIsTypeOf(EndDurationPrice)) and
            cGDP.rentalDuration.timeUnit= Period::day and
            cGDP.rentalDuration.minimumDuration=1)->first().price
else
        result = 0
endif
```

A.2.10 ExtendedRental

Rental extension must be done after the beginning date of the rental agreement and the new end date should be later than initial end date. Note that this constraint has been rewritten, as the original code did not tally with the original class diagram.

```
context ExtendedRental::trueExtension() : Boolean
body: result= self.extension.extensionDone > self.beginning and
    self.lastNewEnding > self.initialEnding
```

Derived attribute *lastModification*. Note that this constraint has been rewritten:

```
context ExtendedRental::lastModification(): DateTime
body: result=self.extension.extensionDone
```

Derived attribute *agreedEnding*:

```
context ExtendedRental::agreedEnding(): DateTime
body: result=self.lastNewEnding
```

A.2.11 ClosedRental

Derived attribute *rentalPriceWithTax*:

```
context ClosedRental::rentalPriceWithTax(): Money
body: result= self.bestPrice *
    self.actualReturnBranch.country.carTax
```

A.2.12 PaidWithPointsRental

The *Reservation* for the corresponding rental was made at least 14 days in advance of the rental's beginning date.

```
context PaidWithPointsRental::enoughInAdvance() : Boolean
body: result= (self.oclIsTypeOf(Reservation) and
    (self.beginning.day()-
    self.oclAsType(Reservation).reservationDate.day())>=day(14))
```

Customer must be member of Loyalty Incentive Scheme in order to pay with points. It is a initial constraint as the customer must be a Loyalty Incentive Member only at the time of paying; later on he/she may not be a member any longer.

```
context PaidWithPointsRental::customerIsLoyaltyMember() : Boolean
body: result = self.renter.oclIsTypeOf(LoyaltyMember)
```

Derived class:

```
context PaidWithPointsRental::allInstances():
    Set(PaidWithPointsRental)
body: result= ClosedRental.allInstances->select(cR|cR.paymentType=
    payType::points)
```

Derived attribute *bestPrice*:

```
context PaidWithPointsRental::bestPrice(): Money
body: result=basicPrice
```

A.2.13 LateReturn

Derived class:

```
context LateReturn:: allInstances(): Set(LateReturn)
body: result= ClosedRental.allInstances()->select(cR|
    cR.actualReturn > cR.agreedEnding)
```

Derived attribute *extraInterval*

```
context LateReturn:: extraInterval(): Duration
body: result = self.actualReturn-self.agreedEnding
```

Derived attribute *extraCostWithTax*:

```
let timeUnit: Period=
        if self.extraInterval.unit=Period::hour and
            self.extraInterval.numberOfUnits <= 6 then
                Period::hour
        else
                Period::day
        endif
in
let durationPrice: Money= self.bestDurationPrices->select(cGDP |
    not(cGDP.oclIsTypeOf(EndDurationPrice)) and cGDP.timeUnit=
    timeUnit and minimumDuration=1)->first().price
let extraPrice: Money=
    durationPrice*(extraInterval/durationT(timeUnit,1))+
    durationPrice*(extraInterval%durationT(timeUnit,1)) in
result= extraPrice * self.actualReturnBranch.country.carTax
```

A.2.14 EarlyReturn

Derived class:

```
context EarlyReturn:: allInstances(): Set(EarlyReturn)
body: ClosedRental.allInstances()->select(initEnding-
    actualReturn> hour(1))
```

A.2.15 Car

Car can only be assigned, at most, to one rental; excluding both closed and canceled rentals.

```
context Car:: onlyOneCarAssignment() : Boolean
body: result = self.rentalAgreement->select(rA |
    not(rA.oclIsTypeOf(CanceledReservation)) and
    not(rA.oclIsTypeOf(ClosedRental)))->size()<=1
```

Car is identified by registration number

```
context Car:: registrationNumberIsKey() : Boolean
body: result = Car.allInstances()->isUnique(registrationNumber)
```

Derived relationship *carGroup*:

```
context Car:: carGroup(): CarGroup
result=self.carModel.carGroup
```

A.2.16 OwnCar

Derived attribute *available*

```
context OwnCar::available() : Boolean
body: result= not(self.oclIsTypeOf(NeedsMaintenance)) and
    not(self.oclIsTypeOf(RepairsScheduled)) and
    not(self.oclIsKindOf(ToBeSoldCar)) and not(self.assigned) and
    not(self.oclIsTypeOf(BeingTransferredCar)) and
    not(self.oclIsTypeOf(NeedToBeSoldCar))
```

Derived attribute *assigned*

```
context OwnCar::assigned() : Boolean
body: result= car.rentalAgreement->exists(rA |
    not(rA.oclIsTypeOf(CanceledReservation) and
    not(rA.oclIsTypeOf(ClosedRental)))
```

A.2.17 AssignedCar

At the time when a car is assigned to a *RentalAgrement* (exluduing closed rentals and canceled rentals) the pick-up branch becomes responible for the car.

```
context AssignedCar::pickUpBranchisResponsible(): Boolean
body: result= self.car.branch = self.rentalAgreement.pickUpBranch
```

A.2.18 NeedsMaintenance

A Car that needs maintenance cannot have more than 10% of the mileage required for maintenance and not more than 10% of the required time between services may have elapsed.

```
context NeedsMaintenance::notOver10Percent(): Boolean
body: result = ((currentMileage - mileageFromLastService) <=
    (1,1*MaintenanceRequirements.mileageForService)) or ((now() -
    lastMaintenanceDate) <=
    (1,1*MaintenanceRequirements.timeForService))
```

Derived class:

```
context NeedsMaintenance::allInstances(): Set(NeedsMaintenance)
result= OwnCar.allInstances()->select(currentMileage -
    mileageFromLastService >=
    MaintenanceRequirements.mileageForService or now()
    -lastMaintenanceDate > MaintenanceRequirements.timeForService)
```

A.2.19 NeedToBeSoldCar

Derived class:

```
context NeedToBeSoldCar::allInstances(): Set(NeedToBeSoldCar)
body: OwnCar.allInstances()->select(c|today()-c.acquisitionDate >=
    year(1) or self.currentMileage>=40,000)
```

A.2.20 ToBeSoldCar

A *Car* that is to be sold cannot be assigned to a rental, excepting those rentals that are closed or canceled.

```
context ToBeSoldCar:: notAssignedReservation(): Boolean
body: result = self.rentalAgreement->forAll(r|
    r.oclIsKindOf(ClosedRental) or
    r.oclIsKindOf(CanceledReservation))
```

A.2.21 CarModel

CarModel is identified by its name.

```
context CarModel:: nameIsKey() : Boolean
body: result = CarModel.allInstances()->isUnique(name)
```

A.2.22 CarGroup

CarGroup is identified by its name.

```
context CarGroup:: nameIsKey() : Boolean
body: result = CarGroup.allInstances()->isUnique(name)
```

Makes sure that the order of *CarGroups* is coherent (i.e there are no cycles).

```
context CarGroup:: totalOrder(): Boolean
let isWorse(w,b:CarGroup):Boolean= b.worse=w or isWorse(w,b.worse)
let isBetter(b,w:CarGroup):Boolean= w.better=b or
    isBetter(b,w.better)
in result = CarGroup.allInstances()->one(cg|cg.worse->isEmpty())
    and CarGroup.allInstances()->one(cg|cg.better->isEmpty()) and
    CarGroup.allInstances()->forall(cg1,cg2| isWorse(cg1,cg2)
    implies not isBetter(cg1,cg2) and isBetter(cg1,cg2) implies
    not isWorse(cg1,cg2))
```

A.2.23 Customer

RentalAgreements of a *Customer* do not overlap.

```
context Customer:: rentalsDoNotOverlap() : Boolean
body: result=self.rentalAgreement-> reject(rA|
    rA.oclIsKindOf(CanceledReservation)->notExists(rA |
    self.rentalAgreement->select(rAOther |
    rAOther.beginning.day()> rA.beginning.day())->exists(rAOther|
    rAOther.beginning.day() <= rA.agreedEnding.day())))
```

A.2.24 LoyaltyMember

A member of the loyalty incentive scheme rented at least one car during the last year and does not have any bad experience.

```
context LoyaltyMember:: meetsLoyalPermanence() : Boolean
body: result = (self.rentalAgreement.beginning->exists(dT|
    dT>(now()-year(1))) and self.faults->isEmpty())
```

Derived attribute *availablePoints*:

```
let candidateRentals: Set(ClosedRental)= self.RentalAgreement->
    select(rA| rA.oclIsTypeOf(ClosedRental) and (now()-
    rA.agreedEnding)< year(3) and rA.agreedEnding >
    (membershipDate - year(1)).oclAsType(ClosedRental)->asSet()
let earnRentals: Set(ClosedRental)= candidateRentals->
    reject(cR|cR.oclIsTypeOf(PaidWithPointsRental)
let accumulatedPoints: Integer= earnRentals->forAll(r |
    result->including(pointsEarned(r.bestPrice)))->sum()
let spendRentals: Set(ClosedRental)=
    candidateRental->select(oclIsTypeOf(PaidWithPointsRental))
let spentPoints: Integer= spendRentals->forAll(r
    |result->including (pointsSpent(r.bestPrice)))->sum() in
result= accumulatedPoints-spentPoints
```

A.2.25 Blacklisted

The reservations or rentals of a blacklisted driver that begin after the *blacklistedDate* must be cancelled.

```
context Blacklisted:: noRentals() : Boolean
body: result= self.rentalsAsDriver->select(rA| rA.beginning >
    self.blacklistedDate)->
    forAll(rA|ra.oclIsTypeOf(CanceledReservation))
```

A.2.26 DrivingLicense

DrivingLicenses are identified by their number.

```
context DrivingLicense:: numberIsKey() : Boolean
body: result = DrivingLicense.allInstances()->isUnique(number)
```

Driver has at least one year of experience and the license does not expire before the agreed end of a rental of the driver.

```
context DrivingLicense:: validLicence() : Boolean
body: result = today()-self.issue>year(1) and
    self.eU_RentPerson.rentalsAsDriver.agreedEnding->
    forAll(d|d<self.expiration)
```

A.2.27 RentalDuration

RentalDurations are identified by their name.

```
context RentalDuration:: nameIsKey() : Boolean
body: result = RentalDuration.allInstances()->isUnique(name)
```

Price for a particular rental duration and car group must be higher than the price for the same rental duration but worse car group, excluding those that have ended.

```
context RentalDuration:: coherentPrices() : Boolean
body: let curCGDPrices: Set(CarGroupDurationPrice) =
    self.carGroupDurationPrice->reject(cgdp|cgdp.oclIsTypeOf(EndDurationPrice))
    in
result = curCGDPrices->forAll(cgdp|cgdp.price >=
    (curCGDPrices.carGroup.worse.carGroupDurationPrice->
    select(cg|cg.rentalDuration=self)).price)
```

Makes sure that the order of *RentalDurations* is coherent (i.e. there are no cycles).

```
context RentalDuration:: totalOrder(): Boolean
let isShorter(s,l:RentalDuration):Boolean= l.shorter=s or
    isShorter(s,l.shorter)
let isLonger(l,s:RentalDuration):Boolean= s.longer=l or
    isLonger(l,s.longer) in
result = (RentalDuration.allInstances()->one(rd|
    rd.shorter->isEmpty()) and RentalDuration.allInstances()->
    one(rd|rd.longer->isEmpty()) and RentalDuration.allInstances()
    ->forAll(rd1,rd2| isShorter(rd1,rd2) implies not
    isLonger(rd1,rd2) and isLonger(rd1,rd2) implies not
    isShorter(rd1,rd2)))
```

A.2.28 ApplicableRentalDuration

Derived class[4] :

- Will be the result of the derived relationship between *RentalAgreement* and *RentalDuration*.

Derived attribute *quantity*:

```
context ApplicableRentalDuration::quantity():Natural
body:
let rentalDur:Duration = durationT(self.agreedEnding -
    self.initEnding)
let rentalDays:Natural = rentalDur.numberOfUnits
-- All RentalDurations related to a particular RentalAgreement can
    be obtained by accessing the RentalAgreement and, from there,
    navigating to RentalDuration. Accessing the RentalDuration
    from this class will only return one RentalDuration. --
let allRentalDur:Set(RentalDuration) =
    self.rentalAgreement.rentalDuration in
        allRentalDur->iterate(elem; qty:Natural = 0 |
                if (rentalDays >= elem.maximumDuration) then
                        if (elem = self.rentalDuration) then
                                qty=qty+rentalDays/maximumDuration
                        else
                                true
                        endif
                        rentalDays=rentalDays%maximumDuration
                else
                        true
                endif
                if (rentalDays >= elem.minimumDuration) then
                        if (elem = self.rentalDuration) then
                                qty=qty+rentalDays/minimumDuration
                        else
                                true
                        endif
                        rentalDays=rentalDays%minimumDuration
                else
                        true
                endif
                )
result = qty
```

[4]This class and its corresponding relationship do not appear in the original specification in [5]. We suppose that duration of a rental is measured either in days or hours.

A.2.29 Discount

Discounts are identified by name.

```
context Discount:: nameIsKey() : Boolean
body: result = Discount.allInstances()->isUnique(name)
```

A.2.30 EndDurationPrice

Ending date of *EndDurationPrice* must be on the same day or later than its beginning date.

```
context EndDurationPrice:: correctEnding(): Boolean
body: result = self.endingDate >= self.beginning
```

A.2.31 ClosedDiscount

Ending date of *ClosedDiscount* must be on the same day or later than beginning date.

```
context ClosedDiscount:: correctEnding(): Boolean
body: result = self.beginningDate <= self.endingDate
```

A.2.32 BadExperience

BadExperience is identified by type.

```
context BadExperience:: typeIsKey() : Boolean
body: result = BadExperience.allInstances()->isUnique(type)
```

A.2.33 CarDamage

Derived class:

```
context CarDamage::allInstances():Set(CarDamage)
result = BadExperience.allInstances()->
    select(b|b.type=BadExpType::carDamage)
```

A.2.34 Country

Countries are identified by name.

```
context Country:: nameIsKey() : Boolean
body: result = Country.allInstances()->isUnique(name)
```