AN INVESTIGATION INTO THE USE OF ORM AS A CONCEPTUAL MODELLING TECHNIQUE WITH THE UML DOMAIN MODEL CLASS DIAGRAM AS BENCHMARK

by

MANJU MEREEN JOHN

submitted in part fulfilment of the requirements

for the degree of

MASTER OF SCIENCE

in the subject

COMPUTER SCIENCE

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR: J A van Biljon

JOINT SUPERVISOR: A J Gerber

FEBRUARY 2002

****

ACKNOWLEDGEMENTS

It is only appropriate to acknowledge my gratitude to all persons and institutions that contributed to the completion of this study. With profound thanks I recall the expert guidance given by my supervisors, Mrs Judy van Biljon and Mrs Aurona Gerber, but for whose painstaking efforts at every stage from drafting up to completion this research would not have become a reality.

I also wish to place on record my sincere thanks to the library staff of UNISA for their kind co-operation and assistance in utilising their valuable collections and providing extracts from these.

Last but not least, I wish to express my thanks to Mathew, my husband and Dr M. J. Koshy, my father for their constant encouragement in completing the research process.

DISSERTATION SUMMARY

AN INVESTIGATION INTO THE USE OF ORM AS A CONCEPTUAL MODELLING TECHNIQUE WITH THE UML DOMAIN MODEL CLASS DIAGRAM AS BENCHMARK

MANJU MEREEN JOHN

Supervisor	:	J A van Biljon
Joint Supervisor	:	A J Gerber
Department	:	Computer Science
		UNIVERSITY OF SOUTH AFRICA
Degree	:	Master of Science

This study investigated the use of ORM as a conceptual modelling technique by using the UML domain model class diagram as benchmark. The rationale was that if the ORM-class diagram compared favourably with the benchmark, then ORM could be proposed as an alternate conceptual modelling technique. Proponents of ORM suggest that it has significant advantages over other techniques for conceptual modelling.

The benchmark UML class diagram was developed according to the Unified Process through use-cases and collaboration diagrams. The ORM-class diagram was derived using the Conceptual Schema Design Process and ORM-UML Mapping Process.

The evaluation of the two class diagrams was conducted by means of a questionnaire, based on a set of principles for conceptual models. The study concluded that ORM could not be proposed as a conceptual modelling technique up to the UML domain class diagram level without considering additional techniques for capturing the dynamics of the system.

Key Terms

Unified Modelling Language, Object Role Modelling, Object modelling, Data modelling, Unified Process, Schema Design Process, Mapping Process, Use-case diagrams, Elementary facts, Collaboration diagrams.

TABLE OF CONTENTS

CHAPTER 1: AN INVESTIGATION INTO THE USE OF ORM AS A CONCEPTUAL MODELLING TECHNIQUE WITH THE UML DOMAIN MODEL CLASS DIAGRAM AS BENCHMARK 1
1.1 Introduction and Background 1
1.2 Problem Statement and Objective 2
1.3 Importance of the Topic 3
1.4 Proposed Approach 4
1.5 Limitations 5
1.6 Outline of the Dissertation 5

CHAPTER 2: UML AND ORM FOR CONCEPTUAL MODELLING 7
2.1 Introduction and Definitions 7
2.2 Role of Conceptual Modelling in a Software Life Cycle 7
2.3 Methods, Methodologies, Techniques and Notations 10
2.4 Techniques Used in the Study 10
2.5 Conclusion 12

CHAPTER 3: CONCEPTUAL MODELLING PRINCIPLES 13
3.1 Introduction 13
3.2 Approaches to Software System Modelling 13
3.3 Desirable Characteristics of a Conceptual Model 14
3.3.1 Characteristics Listed by Paige, Ostroff and Brooke 14
3.3.2 Characteristics Listed by Halpin 16
3.3.3 Characteristics Listed by Hay 18
3.3.4 Characteristics Listed by Claxton 18
3.3.5 Characteristics Listed by Simsion 20
3.4 Integrated List of Conceptual Modelling Principles 21
3.5 Conclusion 23

CHAPTER 4: UNIFIED MODELLING LANGUAGE 24
4.1 Introduction 24
4.2 An Overview of Object Oriented Processes 25
4.3 UML Diagrams 27

4.4 Use-case Diagrams 30
4.4.1 Relationships in Use-case Diagrams 31
4.5 Components of UML Class Diagrams 32
4.5.1 Representation of Attributes in UML 35
4.5.2 Associations 39
4.5.3 Multiplicity Constraints on Associations 41
4.5.4 Association Classes 42
4.5.5 Qualified Associations 43
4.5.6 OR Associations 44
4.5.7 Aggregation and Composition 45
4.5.8 Generalisation 47
4.5.9 Dependency Relationships 49
4.5.10 UML Constraints 50
4.5.11 Derivation Rules 52
4.6 Sequence Diagrams 53
4.7 Collaboration Diagrams 56
4.8 Deriving the Domain Model Class Diagram from Use-cases 57
4.9 Conclusion 57

CHAPTER 5: OBJECT ROLE MODELLING 59
5.1 Introduction 59
5.2 Components of ORM Diagram 60
5.2.1 Reference Types 62
5.2.2 Fact Types 63
5.3 Objectified Relationship Types 64
5.4 Representing Constraints in ORM 65
5.4.1 Mandatory Role Constraint 65
5.4.2 Disjunctive Mandatory Role Constraint 66
5.4.3 Uniqueness Constraint 67
5.4.4 Frequency Constraint 69
5.4.5 OR Constraint 70
5.4.6 Subset Constraint 70
5.4.7 Equality Constraint 71

5.4.8 Ring Constraint 71
5.5 Subtyping 72
5.6 Generating Classes from the ORM Model 73
5.7 Conclusion 74

CHAPTER 6: EVALUATION 75
6.1 Introduction 75
6.2 Evaluation of the Questionnaire Results 77
6.2.1 Evaluation for Completeness 78
6.2.2 Evaluation for Accuracy 79
6.2.3 Evaluation for Clarity 79
6.2.4 Evaluation for Simplicity 80
6.2.5 Evaluation for Uniqueness 81
6.2.6 Evaluation for Validation 81
6.3 Evaluation of the Benchmark Class Diagram and the ORM-Class Diagram 82
6.4 Conclusion 84

CHAPTER 7: CONCLUSIONS AND RECOMMENDATIONS 86
7.1 Introduction 86
7.2 Conclusions 86
7.3 Further Research 89

BIBLIOGRAPHY: 90
APPENDIX A: CASE STUDY 93
APPENDIX B: DOMAIN MODEL CLASS DIAGRAM 151
APPENDIX C: ORM SCHEMA 152
APPENDIX D: ORM-CLASS DIAGRAM 190
APPENDIX E: QUESTIONNAIRE 192
GLOSSARY: 197

LIST OF TABLES

Table 1: Conceptual modelling principles 21
Table 2: Use-case 01 105
Table 3: Use-case 02 109
Table 4: Use-case 03 110
Table 5: Use-case 04 112
Table 6: Use-case 05 114
Table 7: Objects identified for use-case 01 115
Table 8: Objects identified for use-case 02 136
Table 9: Objects identified for use-case 03 137
Table 10: Objects identified for use-case 04 141
Table 11: Objects identified for use-case 05 145
Table 12: List of classes 148
Table 13: List of associations 149
Table 14: List of generalisations 150
Table 15: Results of questionnaire survey 196

TABLE OF FIGURES

Figure 1: Use-case diagram 30
Figure 2: Use-case generalisation 31
Figure 3: A customer class 33
Figure 4: Customer class with stereotype 35
Figure 5: Single-valued attribute in UML 37
Figure 6: Multi-valued attribute in UML 37
Figure 7: Instantiation in UML 38
Figure 8: UML association 40
Figure 9: Ternary association in UML 41
Figure 10: Multiplicity constraints in associations 42
Figure 11: Multiplicity constraints in ternary associations 42
Figure 12: Association class 43
Figure 13: Qualified association 44
Figure 14: OR association 45
Figure 15: Shared aggregation 46
Figure 16: Composition in UML 46
Figure 17(a)(b): Composition 47
Figure 18: Generalisation in UML 48
Figure 19: Dependency relationship 49
Figure 20: Subset constraint in UML 50
Figure 21: Textual constraint in UML 51
Figure 22: Enumeration type 52
Figure 23: UML derivation rule 52
Figure 24: Sequence diagram 55
Figure 25: Sequence diagram with object destroyed 55
Figure 26: Collaboration diagram 56
Figure 27: Binary relationship in ORM 61
Figure 28: Ternary association in ORM 62
Figure 29: Reference type in ORM 62
Figure 30: Fact type in ORM 63
Figure 31: Derived fact type 64
Figure 32: Objectified relationship types 64

Figure 33: Mandatory role constraint.................. 66
Figure 34(a), (b): Disjunctive mandatory role constraints.................. 66, 67
Figure 35 (a), (b), (c): Uniqueness constraint.................. 68
Figure 36: External uniqueness constraint.................. 69
Figure 37: ORM frequency constraint.................. 69
Figure 38: Or-constraint in ORM.................. 70
Figure 39: Subset constraint.................. 70
Figure 40: Equality constraint.................. 71
Figure 41: ORM subtype.................. 72
Figure 42: Use-case diagram.................. 104
Figure 43-60: Collaboration for use-case 01.................. 118-135
Figure 61-62: Collaboration for use-case 02.................. 136-137
Figure 63-65: Collaboration for use-case 03.................. 138-140
Figure 66-69: Collaboration for use-case 04.................. 142-145
Figure 70-71: Collaboration for use-case 05.................. 146
Figure 72: UML domain model class diagram.................. 151
Figure 73-85: Step 2 of ORM Schema Design Process.................. 156-165
Figure 86-88: Step 3 of ORM Schema Design Process.................. 166-168
Figure 89-101: Step 4 of ORM Schema Design Process.................. 169-181
Figure 102: Combined schema.................. 182
Figure 103, 104: Step 5 of ORM Design Process.................. 185, 187
Figure 105: Steps 6 and 7 of ORM Schema Design Process.................. 189
Figure 106: ORM-Class diagram.................. 191

Chapter 1

AN INVESTIGATION INTO THE USE OF ORM AS A CONCEPTUAL MODELLING TECHNIQUE WITH THE UML DOMAIN MODEL CLASS DIAGRAM AS BENCHMARK

1.1 *Introduction and Background*

The background of the study is the modelling of software systems at the conceptual level. Conceptual models are constructed for software systems to represent the system requirements. They provide a means of communication between the users, who will make use of the system, and system developers or domain experts such as analysts. The conceptual model forms a basis for the subsequent design and implementation of the software system [Gulla, 2000]. It represents the business objects of interest along with the facts that must be maintained about those objects and the business-based relationships between the objects [Fisher, Schmidt, 1998].

The conceptual model must be capable of representing, verbalising and validating business rules in simple sentences that can be used to communicate with the users involved in the development field [Halpin, 2001]. Halpin defines a business rule as a statement that defines or constrains some aspect of the business and is intended to assert a business structure or to influence the behaviour of the business.

From the above definitions of a conceptual model, it is understood that the construction of an effective conceptual model is critical to the success of the software system. The two techniques used for conceptual modelling in this study are the use-case driven technique of the Unified Modelling Language (UML) and Object Role Modelling (ORM). UML is a standard language used for the object-oriented analysis and design of software systems that includes techniques such as the use-case-driven techniques for creating software

blueprints or conceptual models [Shelly et al., 2001]. ORM is a fact-based conceptual modelling technique that conceptualises the software system in terms of objects that play roles. The reason for selecting UML and ORM for conceptual modelling in this study is explained in 2.4. The two techniques are discussed in chapters 4 and 5 respectively.

1.2 *Problem Statement and Objective*

The Unified Modelling Language (UML) is an accepted standard to visualise, specify, construct and document the artefacts of a software system [Booch et al., 1999a]. An artefact can be any kind of information or deliverable generated by the developers of the software system, such as UML diagrams and their associated text. UML domain model class diagrams are used to model the static aspects of a software system, which encompasses the existence and placement of classes, interfaces, collaborations and their associations [Eriksson, Penker 1998]. When UML domain model class diagrams are used purely for conceptual modelling, they have deficiencies in capturing business rules and verbalizing and validating facts with domain experts [Halpin, 2001, Hay 1999].

Halpin further states that the use of a fact-based modelling technique known by the name Object Role Modelling (ORM) can rectify this deficiency of UML domain model class diagrams in conceptual modelling. The expressiveness of ORM for conceptual modelling is also discussed by Hofstede [Hofstede ter A H M et al., 1997]. According to Zachman, "ORM has incomparable ability to capture semantic intent and unambiguously express the semantics graphically" [Halpin, 2001]. ORM creates a conceptual model of the software system by elaborate verbalisation of facts taken from the problem domain and producing a complete blueprint of the system.

The primary objective of the study is to investigate the use of ORM for conceptual modelling, using the UML domain model class diagram as a benchmark. A secondary objective of the study is to find out how a class diagram can be generated from the ORM conceptual model with a view to

making use of the conceptual model further down the software development process, as discussed in section 2.2. If the study finds that the class diagram generated from the ORM model compares favourably with the benchmark class diagram, then ORM can be used as an alternative conceptual modelling technique to the use-case driven technique of UML. If ORM can be proposed as an alternate conceptual modelling technique, then software developers can exploit the advantages of ORM for conceptual modelling and at the same time use UML for design purposes.

1.3 *Importance of the Topic*

A house is built on a blueprint. The blueprint captures the requirements of the house, depicting the position of the living room, the bedrooms, study rooms, kitchen and other such independent units of the house and making the necessary connections between the units rather than leaving them as separate units. The blueprint gives one a complete overview of the house. A house built on a good blueprint is built on a sound basis. We can substitute the software system for the house and the conceptual model for the blueprint. A house built to a good blueprint will be a good house; similarly, the quality of the conceptual model impacts on the quality of the software system. The principles on which a good conceptual model is designed are discussed in Chapter 3.

ORM helps to capture complex business rules and to verbalise and validate the conceptual model with domain experts using its abstraction mechanism. Halpin [Halpin, 2001] claims that ORM is the only fact-oriented technique with significant support in the industry. It has a linguistic basis, which supports the verbalisation of facts in simple sentences. Furthermore, the ORM model can be mapped into UML classes. Currently, a computer-based tool called Microsoft VisioModeler 3.1 is available that supports conceptual modelling using ORM. As ORM is practically not used in the South African software industry, the study aims to investigate the usefulness of ORM as an alternative technique for conceptual modelling, while using UML further in the software development.

1.4 *Proposed Approach*

A literature survey is conducted on UML and ORM in order to study the techniques they provide for conceptual modelling. These techniques are then applied to the same case study in order to derive the respective conceptual models, of which the UML version is used as a benchmark for the study. Both models are tested against the underlying principles (listed in chapter 3) of a good conceptual model by means of a questionnaire given to end users involved in software development. The reason for using a questionnaire is that conceptual models are used for communication with the user community and therefore testing the model with the user community is an appropriate way to evaluate the model against the principles listed in chapter 3.

The Unified Process, a software development process, uses UML for developing a conceptual model of a software system [Booch et al., 1999b]. This process has a set of activities needed to transform the user requirements into a software system. UML is an integral part of the Unified Process. The Unified Process is discussed in section 2.2.

Following the Unified Process for software development, use cases are written for the case study, followed by dynamic modelling of the use case with collaboration diagrams. Through an iterative process, a conceptual class diagram or domain class diagram is drawn. This class diagram becomes the benchmark of the study.

An ORM model is drawn for the same case using the Conceptual Schema Design Process discussed by Halpin [Halpin, 2001]. The steps involved in this design process are listed in appendix C of the case study. A conceptual model class diagram is derived from the ORM model using the UML-ORM mapping process listed in section 5.6. ORM employs a data perspective, whereas UML employs an object-oriented perspective that encapsulates both data and operations. The data and object-oriented perspectives are

discussed in section 2.4. The two conceptual models constructed using different techniques are evaluated according to the conceptual model principles of chapter 3. If the conceptual class diagram derived using the ORM technique compares favourably with the one derived using UML, then ORM can be used as an alternative conceptual modelling technique.

A case study approach is selected for this study for the following reasons:

- To investigate the use of ORM for conceptual modelling, the technique must be tested in order to determine whether the complex business rules can be captured. This can only be done with the help of a case study that includes complex business rules. The car rental case selected for this study satisfies this requirement by providing a rich collection of business rules, both simple and complex.

- The case is comprehensive enough to test all types of object structures and constraints supported by UML as well as the fact types and constraints of ORM.

- The models derived need to be tested against the principles supporting a good conceptual model. This evaluation can be done on the conceptual models derived from the selected case.

1.5 *Limitations*

Implementation and testing of the system are outside the scope of this investigation. It is therefore not possible to test the compliance of the derived UML and ORM models with all the principles listed in chapter 3.

1.6 *Outline of the dissertation*

Chapter 1 addresses the primary and secondary objectives of the research and provides an outline of the dissertation. The proposed approach of the research is discussed together with the limitations.

Chapter 2 defines the terms used in the study and justifies the use of the conceptual modelling techniques supported by UML and ORM for the study. The role of conceptual modelling in the software life cycle is also discussed in this chapter with the aim of placing the conceptual model into the context of the software development process.

Chapter 3 studies the conceptual modelling principles, definitions and concepts from different perspectives and integrates these into one set of principles that will be used to test against the conceptual models derived for the case study. The aim of the chapter is to identify the relevant principles that can be used as criteria to test a conceptual model.

Chapter 4 describes the Unified Modelling Language conventions and techniques involved. The aim of chapter 4 is to provide background knowledge about UML.

Chapter 5 examines Object Role Modelling conventions and techniques involved in order to provide background knowledge about ORM.

Chapter 6 contains an evaluation of the case study, comprising of an evaluation of the feedback received from the questionnaire and an evaluation of ORM as an alternative conceptual modelling technique.

Chapter 7 concludes the study by examining whether the main aim of the study has been achieved. The techniques are reviewed together with the limitations encountered and benefits identified based on the findings derived from chapter 6. Future areas of research are also highlighted.

The case study is presented in Appendix A, the UML conceptual model in Appendix B, the ORM conceptual model in Appendix C, the class diagram derived from the ORM model in appendix D and the questionnaire used to evaluate the principles supported by a good conceptual model in Appendix E.

Chapter 2

UML AND ORM FOR CONCEPTUAL MODELLING

2.1 *Introduction and Definitions*

Models represent a simplified view of the world that can be used for exploration, communication, testing and making predictions. A model is an abstract representation of something that suppresses unnecessary information, for instance by grouping related things together to form an aggregate or represent a whole class of objects as a single object. Models are often created, used and discarded in the process of software development, where a conceptual model represents the object structures, their characteristics, relationships and the constraints of the software system by capturing the requirements of the system. The definition and purpose of a conceptual model was given in section 1.1.

The following section discusses the role of conceptual modelling in the software development life cycle.

2.2 *Role of Conceptual Modelling in a Software Life Cycle*

UML as such is process-independent. It is not tied to any particular software development process. In order to derive the best benefit from UML, one can consider a process that is use-case driven, architecture centric, iterative and incremental. The Unified Process is such a software development process [Booch et al., 1999b], and hence the Unified Process was selected to develop a conceptual model of the case study.

The Unified Process consists of a series of cycles that covers the entire life of a system, from its birth to its release. Each cycle consists of four phases, namely inception, elaboration, construction and transition, where each phase has a number of iterations [Bennet et al., 2001]. The main workflows

that take place in the four phases concern the requirements, analysis, design, implementation and testing.

In the inception phase, the functionality that is expected from the system is identified. A use-case model can be constructed in this phase, which captures the required functionality of the system. Use cases are discussed in chapter 4. The inception phase is intended to formulate a plan of what needs to be built in the cycle, including a domain analysis [Eriksson, Penker, 1998]. During the elaboration phase, the use cases are specified in detail and the system architecture is designed. The architecture is expressed as views of the use-case model, the analysis model, the design model, the implementation model and the deployment model. In the construction phase, the architecture is built into a functional system, and in the transition phase experienced users try the efficiency of the system and report deficiencies and errors [Booch et al., 1999b].

Among the various models discussed above, the study focuses only on the requirements or conceptual model, which establishes the requirements of the system. In the Unified Process, the domain model captures the requirements of the system by identifying the relevant classes that exist in the system. The domain model is depicted as a UML class diagram, and this class diagram becomes the conceptual model, which is derived through a series of iterations from use-case modelling and dynamic (behavioural) modelling. The domain model UML class diagram developed using the Unified Process forms the benchmark of the study.

ORM is a conceptual modelling technique that is not tied to a particular software development process, and is used in the requirements analysis of software development life cycle. This conceptual model hides the implementation details of the system.

In ORM, the conceptual model yields the conceptual schema, which is used to relate one entity to another. This schema is a visual representation of the conceptual model that constitutes a model of user's knowledge about some subject matter, and it reflects user perceptions of the structure of the

problem domain as completely and unambiguously as possible [Ramesh et al., 1999]. The semantic stability of the conceptual model is partly determined by the expressiveness of the relationship constructs that it can support. Halpin [Halpin, 2001] states that conceptual schemas are constructed for clear communication between the modeller and the domain expert.

The conceptual schema thus -

1. presents a detailed summary of the object structures and constraints involved in the system environment, and

2. is independent of both software and hardware. This means that changes in the hardware or software will have no effect on the conceptual model.

The first step in building a software system is to determine the requirements. This is followed by analysis, design, implementation and testing. The initial step in analysis is to design a conceptual schema, one that accurately and completely defines business rules in a way that users can understand. Conceptual modelling combines three levels of expression:

1. The syntactic level describes the symbols that are used in a particular technique.

2. The positional level explains how entities are laid out.

3. The semantic level describes the different ways of conveying the meaning of business situations [Hay, 1999].

This dissertation focuses on all these levels, as they are all closely tied together to create a conceptual model of the software system.

At this point it is important to distinguish between the terms methods, methodologies, techniques and notations.

2.3 *Methods, Methodologies, Techniques and Notations*

Before elaborating on conceptual modelling techniques, it is important to distinguish between the terms **methods, methodologies, techniques and notations**. A **method** is described as an orderly arrangement of ideas [Rumbaugh et al., 1994]. **Methodology** on the other hand, refers to the study of methods and therefore it is an approach based on a certain way of thinking to carry out some tasks in a systematic order. A methodology comprises techniques that employ notations. Therefore a **technique** includes the steps to be carried out for a development activity; and techniques in turn use **notations** comprising a set of symbols together with a set of rules in order to define the use of the symbols.

From the above explanation of methods, methodologies, techniques and notations, it is evident that conceptual modelling is a technique that provides a platform to construct a model that represents a structural or static view of a software system and is derived by capturing the requirements of the system.

The following section elaborates on the techniques used in this study.

2.4 *Techniques Used in the Study*

The conceptual modelling techniques that form the basis of this study are the use-case-driven technique supported by the Unified Modelling Language (UML) and Object Role Modelling (ORM). The use-case-driven technique uses the UML language to express the requirements of the system. During analysis, the requirements are specified in detail by refining the use-cases as collaborations and developed into a conceptual class diagram to model static structures of the real world. The Object Management Group (OMG) adopted UML as an official standard for object-oriented modelling and design in 1997 [Bennett et al., 2001, Sturm, 1999]. UML supports the development of object-oriented software systems, the current trend in the industry [Deitel, Deitel, 2001], by providing support for static and dynamic modelling of a

system. Hence UML was selected as a benchmark in this study. ORM, on the other hand, is a fact-oriented modelling technique that verbalises the relevant information as elementary facts.

Halpin [Halpin, 2001] claims that ORM is the only fact-oriented technique with significant support in industry. It has a linguistic basis, which supports the verbalisation of facts in simple sentences. The verbalisation can be used to validate the model in terms of expressions that are familiar to the user community. According to Gordon Everest [Halpin, 2001], ORM is capable of capturing and representing semantics in a simplified way and can be interpreted easily by the user community. These reasons justify the use of UML and ORM as conceptual modelling techniques, especially for this study. Both techniques are studied in detail in Chapters 4 and 5 respectively.

UML is considered as a standard notation for object modelling. The domain model class diagram not only models the static structure of a system, but its behaviour as well. Although there are similarities between UML domain model class diagrams and data models, the class diagram cannot be treated as a pure data model; it is an object model that encapsulates the data as well as the behaviour of the system. Therefore UML is considered appropriate for object modelling and not for data modelling. It is used for requirements modelling to support object-oriented analysis and to develop object models [Shelly et al., 2001, Booch, 1999a].

ORM, on the other hand, is a technique that models a system from a data perspective. The conceptual model created using the ORM technique models only the data involved in the system. However, the study aims to derive a conceptual level class diagram from the ORM model by using the ORM-UML Mapping Process.

2.5 *Conclusion*

The terms used in the study are clarified in this chapter. The Unified Process for software development is discussed and the role of conceptual modelling in the Unified Process and the Conceptual Schema Design Process is identified. The reason for selecting the techniques supported by UML and ORM for conceptual modelling is explained. The Unified Process is selected as the software development process for the study primarily because it supports object-oriented techniques that have proven to be of value in constructing a system of any degree of complexity. Secondly, in this study use cases are used as artefacts to represent the behaviour of the system, and the Unified Process is use case driven. This justifies the use of the Unified Process.

ORM is not tied to any software development process. The primary purpose of ORM is to capture the requirements in the form of facts and represent the facts in such a way that they can be read as simple sentences in the model.

Chapter 3

CONCEPTUAL MODELLING PRINCIPLES

3.1 Introduction

The aim of this chapter is to formulate a set of principles for evaluating a conceptual model. Such evaluation is necessary to check that the model fulfils its purpose as explained in section 1.1 of Chapter 1. The desirable characteristics of a conceptual model are studied from various perspectives and integrated into one complete set of evaluation criteria. This chapter first studies the common approaches to software system modelling and then discusses the evaluation and integration of the characteristics of a conceptual model.

3.2 Approaches to Software System Modelling

There are several ways to create a model of a software system, the two most common being the algorithmic and the object-oriented perspectives [Booch et al., 1999a].

The traditional software systems development follows the algorithmic perspective, where the main building block of a software system is the procedure or function. Furthermore, the traditional modelling techniques treat data and processes separately.

From the object-oriented perspective, the main building block of a software system is the object or class. Object orientation is defined as a set of design and development principles based on objects, where an object represents a real-world entity with the ability to interact with itself and with other objects [Rob, Coronel, 2000]. Object-oriented (OO) modelling techniques combine both the data and process aspects into one object. An object has an identity and a state and exhibits behaviour. In an object-oriented environment, the

programmer creates self-contained object classes that contain data and procedures used to operate on the data. Object-oriented techniques are used more frequently than their traditional counterparts because it is more natural for a human being to think of the real world as a set of related objects than to separate the same into processes and data [Rumbaugh et al., 1994]. UML domain model class diagrams are derived from the object-oriented perspective, whereas ORM models are derived from a data perspective. The following section lists the characteristics of a conceptual model from various sources.

3.3 Desirable Characteristics of a Conceptual Model

This section discusses the desirable characteristics of a conceptual model from the perspectives of Paige, Ostroff and Brooke [Paige et al., 2000], Halpin [Halpin, 2001], Hay [Hay, 1999], Claxton [Claxton, Mcdougall, 2000] and Simsion [Simsion, 1994].

3.3.1 Characteristics listed by Paige, Ostroff, Brooke [Paige et al., 2000]

According to Paige, Ostroff, Brooke [Paige et al., 2000], a conceptual model must support the designers and provide assistance to them to perform the following four tasks:

Architectural description – A model must describe a system in terms of abstractions (classes, processes, use cases etc.) and their relationships (inheritance, data flow, sequencing etc.). A good conceptual model will express the abstractions and their relationships at the appropriate level of detail. An architectural description is a document consisting of such abstractions and relationships.

Behavioural description – The architectural description is not sufficient to express the full details of the system. The model will be much more meaningful if it includes the details of interaction between the identified abstractions. The behavioural description will serve this purpose.

System documentation – System documentation is necessary to explain how the system works, the maintenance requirements etc. A model itself is a form of system documentation describing the complete architectural view of the system with all the abstraction details.

Forward and backward generation – The abstractions in the model can easily be transformed into a high-level language. The elements of a model must be mapped easily to computer programs; this is an essential requirement for a good conceptual model.

Paige, Ostroff and Brooke [Paige et al., 2000] list the following characteristics of a conceptual model as desirable.

3.3.1.1 Simplicity

The main principle behind the design of a conceptual model is its simplicity. If a model is simple, then it can be understood easily by its users. This in turn means that the modelling technique must be simple enough to be understood by the users.

3.3.1.2 Uniqueness

A model that satisfies the concept of uniqueness expresses every concept that is of interest to the system and at the same time avoids duplication. Expressing every concept of interest to the system makes a model complete. Even though uniquely representing every valid aspect of the system completes a model, completeness must be treated as a separate principle from uniqueness.

3.3.1.3 Consistency

A consistent conceptual model is one that contains no contradictory information. This means that if two business rule statements that are modelled are contradictory, then the model is inconsistent.

3.3.1.4 Seamlessness

The principle of seamlessness demands that the abstractions used in the model can be used throughout the development process. The model can then support the complete development process and is said to be reliable.

3.3.1.5 Scalability

Scalability means that a system of any size can be modelled. A modelling technique must support the development of physically smaller or bigger systems. If the system grows as more components are added, then the existing model scales efficiently to the new bigger system.

3.3.1.6 Supportability

The conceptual model must support the design and implementation stages of the system development. This principle has the same meaning as seamlessness: both principles indicate that a model must be useful in all phases of the development cycle. In this sense, supportability and seamlessness are synonyms.

3.3.1.7 Space economy

Models should take up as little space on the printed page as possible. Smaller models are easy to analyse and maintenance is simplified. Attribute-based models take up less drawing space than attribute-free models because the latter depicts attributes as relationships, which naturally takes up more space.

3.3.2 Characteristics listed by Halpin [Halpin, 2001]

According to Halpin [Halpin, 2001], a conceptual model ignores the logical and physical level aspects such as the underlying database structures and their implementation. In his view, good conceptual models must reflect all of the following characteristics.

3.3.2.1 Expressibility

A model must express clearly all the details about the software systems that are conceptually relevant. The more features the model can capture, the greater its expressive power. This principle in turn supports the completeness of a model.

3.3.2.2 Clarity

The clarity of a model is a measure of how easy it is to understand and use the model. This means that the diagrams and/or textual expressions used in the model must be meaningful. This principle corresponds to the simplicity principle explained by Paige [Paige et al., 2001].

3.3.2.3 Semantic stability

This refers to the stability of the model in the face of changes due to application changes. The more changes are made to the model with an application change, the less stable the model becomes. If a model becomes unstable, then it becomes unreliable; therefore the principle of semantic stability is directly related to reliability.

3.3.2.4 Semantic relevance

Only conceptually relevant information need be modelled; all irrelevant information must be discarded. This principle supports consistency by modelling only the information that is relevant to the application.

3.3.2.5 Validation mechanisms

The domain expert must be able to check whether the model matches the application. Therefore a conceptual modelling technique must provide support to validate the model against the application.

3.3.3 Characteristics listed by Hay [Hay, 1999]

Hay [Hay, 1998] argues that the evaluation of a conceptual model is based on the technical completeness and readability of the model.

3.3.3.1 Technical completeness

Technical completeness refers to the representation of objects and attributes, relationships between them (if any), unique identifiers, subtypes and supertypes, and also establishes constraints between relationships.

3.3.3.2 Readability

The readability of a model is characterised by its graphic treatment of relationship lines and entity boxes, as well as its adherence to the general principles of good graphic design. Among the most important of the principles of graphic design is that each symbol should have only one meaning that applies wherever that symbol is used, and that each concept should be represented by only one symbol. The fact that each symbol should have only one meaning can be considered as a separate principle called uniqueness. However, a model should not be cluttered with more symbols than are absolutely necessary, and the graphics in the diagram should be intuitively expressive of the concepts involved.

3.3.4 Characteristics listed by Claxton [Claxton, Mcdougall, 2000]

Claxton points out that the quality of a conceptual model is a function of the collective integrity of its components. Detecting loss of information integrity at the more detailed level provides the key for measuring a model's total reliability and subsequent quality. He gives a set of factors for measuring the model's quality.

3.3.4.1 Accuracy

Accuracy is a measure of the correctness of the relationship represented between the model and the business rules. The level of accuracy requires that each assertion in the model reflect the business intent. The accuracy of a model is related to its reliability and can be checked by means of validation mechanisms.

3.3.4.2 *Clarity*

A model is clear if it states its meaning in clear and unambiguous terms. A model lacks clarity when the same statement has two or more interpretations. Clarity is a measure of simplicity and understandability.

3.3.4.3 *Completeness*

Completeness concerns the structural and semantic completeness of the model. Structural completeness refers to the mandatory properties of an object in order to transmit its part of the message. Semantic completeness involves the incorporation of text-based properties such as definitions. This kind of information gives a sense of context to the model.

3.3.4.4 *Conciseness*

Models must not duplicate information. Repetition of the same facts affects the model's information integrity. This principle is the same as the uniqueness principle discussed by Paige [Paige et al., 2000].

3.3.4.5 *Consistency*

Consistency is necessary for the proper integration of a model. It requires that no statement in the model conflict with any other statement from the same model.

3.3.5 Characteristics listed by Simsion [Simsion, 1994]

Finally, Simsion puts forward a list of characteristics that a data model must have. They are the following:

3.3.5.1 Completeness

A conceptual model must not omit any information that is relevant to the system. If the model omits some information that is required for the current application, it will be difficult to add it at a later stage.

3.3.5.2 Non-redundancy

Recording the same information more than once increases the storage cost, requires extra processing capacity to keep the different copies in step and leads to consistency problem if the copies get out of hand. This principle has the same meaning as the conciseness principle discussed by Claxton [Claxton, Mcdougall, 2000].

3.3.5.3 Enforcement of business rules

Business rules must be accurately represented in the conceptual model. Any misrepresentation of business rules in the model may be difficult to correct later. This principle has the same meaning as the accuracy principle discussed by Claxton [Claxton, Mcdougall, 2000].

3.3.5.4 Stability and flexibility

A conceptual model is said to be stable if we do not need to change it in the event of a change in the requirements. This principle is same as the semantic stability discussed by Halpin [Halpin, 2001]. A model is flexible if it can be readily extended to accommodate new requirements with minimum impact on the existing structure.

3.4 Integrated List of Conceptual Modelling Principles

The above discussion of conceptual modelling principles shows that the purpose of a conceptual model is -

- to describe a software system in terms of abstractions;

- to represent the necessary relationships between the abstractions at various levels of detail;
- to show clearly the meaning and application area of the object structures.

A conceptual model contains the concepts that are relevant to the system, defines them relative to one another, represents different levels of abstraction and provides a formal expression of the meaning of the various concepts.

The characteristics put forward by various sources are integrated into one complete list of conceptual model principles in table 1. These principles are valid and relevant for a conceptual model, since it is possible to test a conceptual model against them and check whether and to what extent the model fulfils its purpose.

Table 1: Conceptual modelling principles

Principle	Meaning
Completeness	A conceptual model is said to be complete if it can model all the relevant aspects of the system. Completeness is one principle that directly affects the effectiveness of a model.
Accuracy	An accurate model correctly represents the relevant aspects of a system. This means that all business rule statements and constraints are represented correctly and no false aspects are represented in the model.
Clarity	A conceptual model must represent the aspects of the software system clearly. Clarity is closely related to understandability. A clear model is a transparent model: one can see the representation of business rules through the model. It is also well readable.
Simplicity	Simplicity is a measure of how easy the model is to

	understand.
Uniqueness	Uniqueness is the ability of a model to represent business rules without duplication. Uniqueness also means conciseness or non-redundancy.
Durability	A durable conceptual model will ensure that a system built on it can accommodate new business constructs without requiring constant reconstruction. A durable model will be flexible enough to accommodate changes affecting the application domain, i.e. it will be adaptable. Durability is also a measure of the stability of the model.
Validation Mechanism	This principle is very important, since the domain expert must be able to test whether the model matches the application.
Usability	A usable model can be used throughout the system development life cycle. This principle is the same as the supportability principle.
Effectiveness	If all of the principles listed in this table are incorporated into a model, then we can say that the model is effective and it fulfils its purpose.

It is evident from the principles listed in table 1 that no principle stands alone; each one is dependent on one or more other principles. The effectiveness principle, for instance, is supported by other principles such as clarity, completeness and usability. A good conceptual model will embody all the principles listed in table 1 above.

Some of the principles listed in table 1 cannot be directly tested on the case study because the scope of the study does not include implementation. The testable principles are completeness, accuracy, clarity, simplicity, uniqueness and validation mechanism. The results of the testing of these principles are captured by a questionnaire. The durability, usability and effectiveness principles can only be tested if one proceeds from conceptual modelling to the implementation of the system and to testing whether the conceptual model supports the implementation.

3.5 Conclusion

The common approaches used in software system modelling are the traditional algorithmic approach and the object-oriented approach. Traditional approaches are based on function and data flow, whereas object-oriented approaches focus on object classes. The principles embodied in a good conceptual model are summarised in the chapter. These criteria can be used to evaluate both the data and object driven conceptual models, as they are general principles that can be applied to any conceptual model. In order to evaluate a conceptual model, it is important to study whether the model exhibits those characteristics that are summarised in this chapter. It is assumed that the techniques need to be implemented correctly before the conceptual modelling principles can be used to evaluate them.

Chapter 4

UNIFIED MODELLING LANGUAGE

4.1 Introduction

This chapter focuses on the Unified Modelling Language (UML) and the techniques supported by UML for conceptual modelling. In 1996, the Object Management Group (OMG) put together a task force to define and approve a notation and meta-model standard for object-oriented analysis and design. The task force was made up of vendors of related tools that grouped themselves into four major camps. One of these camps aggregated around the submission originated by Rational Software and promoted UML, built from the Object Modelling Technique (OMT), Booch and Object Oriented Software Engineering (OOSE) methodologies created by the three methodologists Rumbaugh, Booch and Jacobson [Bennet et al., 2001]. The four proposals were submitted to OMG in January 1997, and in December 1997 the standard was formally adopted [Bezivin, Muller, 1999].

The Unified Modelling Language is emerging as a de facto standard for modelling object-oriented systems [Evans, 1999] and has been adopted by most of the leading software development companies (including IBM, Microsoft, Oracle, Rational Software and Sterling Software) [Post, 1999]. UML can be used for business modelling, software modelling in all phases of the development cycle for all types of systems and for modelling any system that has a static structure and dynamic behaviour [Eriksson, Penker, 1998]. Object Oriented systems can be modelled visually with the help of a number of diagram models. UML diagrams are broadly divided into static and behavioural diagrams. The static diagram describes the relationships between object instances, and the behavioural diagram describes the interactions between objects to perform some function [Evans, 1999].

UML is becoming widely used for software modelling and is being evaluated by the Object Management Group as a standard language for object-oriented analysis and design [Halpin, 2001]. For object modelling purposes, UML makes use of domain model class diagrams that may be annotated with expressions in a textual constraint language. UML has constructs designed to assist developers of object-oriented code.

UML includes diagrams for static structures (class and object diagrams), behaviour (use-case, state-chart, activity, sequence and collaboration diagrams) and implementation (component and deployment diagrams). These diagrams are explained in section 4.3. The main focus area of this study is the support provided by UML for requirements analysis. This chapter explains the use-case diagrams, class diagrams, collaboration diagrams and sequence diagrams which are used for the requirements analysis of the case study presented in Appendix A. The other diagrams are not used in the case study and therefore not dealt with in detail.

The abstract syntax of the different language constructs in UML is specified with the graphical notation of class diagrams, while the rules of UML are given in an object-oriented constraint language. This makes the structure of the language rigorous, whereas the semantics of the language is quite informal [France R, et al, 1999]. At this stage, it is important to discuss the object-oriented processes.

4.2 An Overview of Object-Oriented (OO) Processes

Humans think in terms of objects and they learn about objects by studying their attributes and observing their behaviours [Deitel, Deitel, 2001]. Different objects can have similar attributes and exhibit similar behaviours. Object-oriented programming (OOP) models real-world objects with software counterparts taking advantage of the class relationship, where objects of a certain class have the same characteristics.

OOP includes inheritance and multiple inheritance relationships where newly created classes of objects are derived by inheriting characteristics of existing classes and adding unique characteristics of their own. Object-oriented programming gives humans a more natural and intuitive way to view the programming process by modelling real-world objects, their attributes and behaviours.

Object-Oriented (OO) techniques are best suited to projects that will implement systems using object technologies. The object-oriented approach is centred on a technique referred to as object modelling which is a technique for identifying objects within the systems environment and the relationships between those objects [Whitten et al., 2000]. These techniques are completely different from the techniques used for data and process modelling. The typical object-oriented approach encapsulates data (attributes) and functions (behaviour) into packages called objects; the data and functions of an object are intimately tied together. Attributes are the data that represent characteristics of interest about an object. Behaviour refers to those things that the object can do and that correspond to functions that act on the object's data (or attributes) [Whitten et al., 2000]. Objects have the property of information hiding. This means that although objects have the ability to communicate with one another across well-defined interfaces, objects are not aware of how other objects are implemented, i.e., implementation details are hidden within the objects themselves. In an object-oriented programming language such as C++, the unit of programming is the class, a user-defined data type from which objects are created. Classes can have relationships with other classes. A class can be reused many times to make many objects of the same class. On the whole, object-orientation promotes software reusability.

Object-oriented analysis and design involves analysing a problem and developing an approach to solve it. As problems get more complicated, efficient use of object modelling techniques play a vital role in creating a good conceptual model for the system to be built. UML is a notation that has techniques designed for object modelling. UML modellers are free to develop

systems using various processes, but all developers can express those systems with one standard of notations. The trend towards using object-oriented techniques to design and developing software is steadily increasing [Becker, 2000]. The following section explains the various UML diagrams and their purposes.

4.3 UML Diagrams

Software analysis diagrams are visual representations that show the structure of a system. A system includes both static and dynamic structures. Static structures are the static parts of a model, representing the elements that are conceptual, while dynamic structures emphasise the behavioural parts of the system [Booch et al., 1999a].

The concepts that are used in the diagrams are called model elements [Eriksson, Penker, 1998]. The model element conveys the meaning of what is stated in business rules. Examples of model elements can be class, object, state etc. UML offers different diagrams to model a system. Each UML diagram provides the development team with a different perspective of the information system. The various UML diagrams are the following: [Whitten et al., 2000].

1. ***Use-case Diagrams*** – These diagrams graphically represent the interactions between the system, external systems and users. They indicate graphically who will use the system and in what ways the user expects to interact with the system.

   The use-case diagram depicts a number of external actors and their connection to the use cases supplied by the system. In general, a use case is a description of the functionality of a system [Eriksson, Penker, 1998]. Use-case diagrams are described in more detail in section 4.4.

2. ***Class Diagrams*** – Class diagrams represent the object structure of the system. They identify the object classes the system is composed of as well as the relationships between the object classes. Class diagrams represent the static structure of classes in the underlying system as well the relationships between the classes. A detailed study of class diagrams can be found in section 4.5.

3. ***Object Diagrams*** – Object diagrams model object instances, showing the current values of the instance's attributes. The object diagram provides the system developer with a "snapshot" of the system's objects at one point in time.

4. ***Sequence Diagrams*** – These diagrams represent how objects interact with one another via messages in the execution of a use case or operation. They illustrate how messages are sent and received between objects and in what sequence. A sequence diagram represents a dynamic collaboration between objects [Eriksson, Penker, 1998]. Sequence diagrams are explained in detail in section 4.6.

5. ***Collaboration Diagrams*** – These diagrams are similar to sequence diagrams, except that collaboration diagrams do not focus on the sequence of messages; instead, they depict the interaction between objects in a network format. Collaboration diagrams depict objects and their links to one another as well as how messages are sent between the linked objects [Eriksson, Penker, 1998]. Collaboration diagrams are discussed in section 4.7.

6. ***State Diagrams*** – State diagrams are used to model the dynamic behaviour of an object. They show the various states that an object can assume and the events that cause the object to change from one state to another.

7. ***Activity Diagrams*** – Activity diagrams represent the sequential flow of activities of a business process or a use case. They are also used to model actions that will be performed when an operation is executing as well as the result of those actions. These diagrams are similar to flowcharts because they graphically depict the sequential flow of activities of a business process. They are different from flowcharts in that they provide a mechanism to depict activities that occur in parallel. Thus activity diagrams capture actions and their results in terms of the change of state of an object.

8. ***Component Diagrams*** – These diagrams represent the physical architecture of the system. They can also be used to show how programming code is divided into components and depict the dependencies between the components. Component diagrams are implementation-type diagrams.

9. ***Deployment Diagrams*** – Deployment diagrams describe the physical architecture of the hardware and software in the system. These diagrams are also implementation-type diagrams depicting the software components, processors and devices that make up the system's architecture.

Class diagrams represent a static model of a system, whereas use-case, state, sequence, collaboration and activity diagrams help to represent the behaviour of a system. This study focuses on the Unified Process, which makes use of the use-case-driven conceptual modelling technique supported by UML, and hence the discussion includes the UML diagrams that support static and dynamic modelling.

Having listed the different diagrams provided by UML for static and dynamic modelling, it is now important to discuss the diagrams. The discussion begins with use-case diagrams.

4.4 Use-case Diagrams

A use case represents a set of sequences where each sequence is an interaction of things outside the system called its actors. A use-case diagram represents the functional requirement of a system. Use cases involve the interaction of actors and the system, where an actor represents a set of roles played by the users of the use cases while interacting with those use cases [Booch et al., 1999a]. Actors can be human beings or other systems. Actors can be primary actors or secondary actors; a primary actor is the main actor that will perform the use case, while secondary actors are additional actors that will perform the use case [Sturm, 1999]. For example, in a car rental system, processing a reservation involves the interaction between a customer and a booking clerk. Use-case diagrams depict use cases and actors and their relationships. Figure 1 shows a use-case diagram for a car rental system.

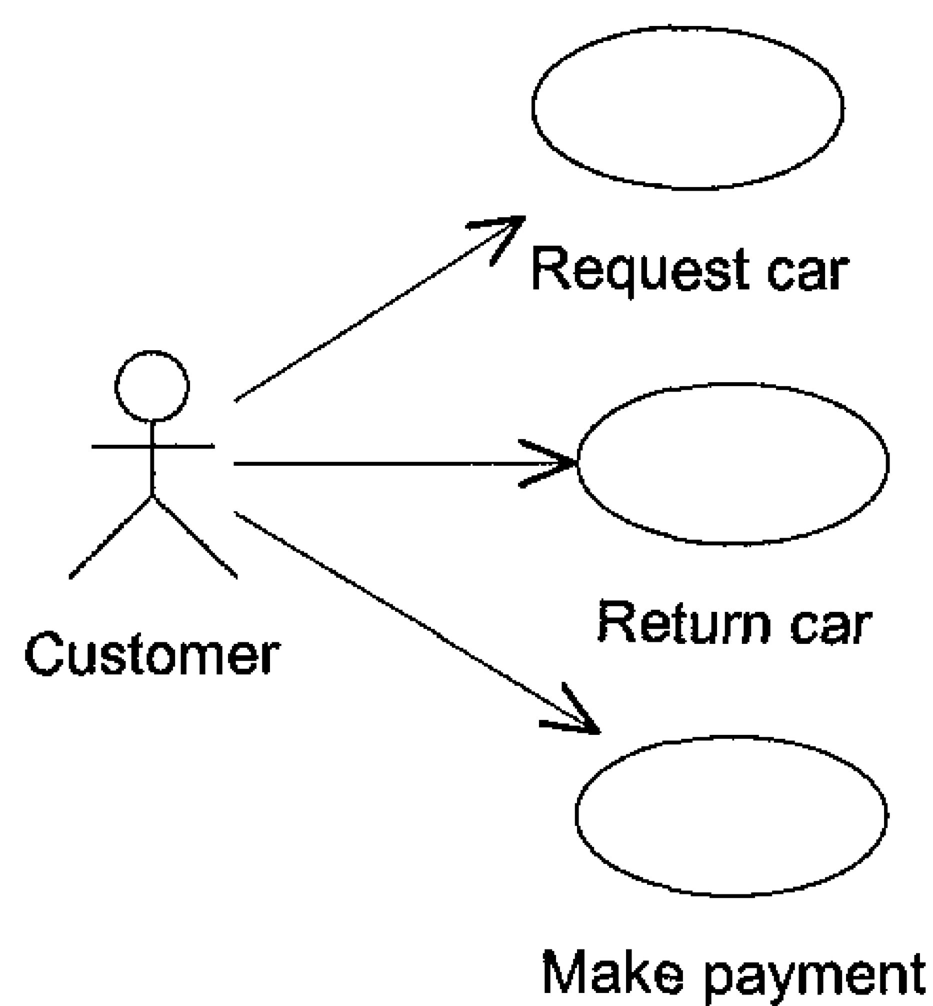


Figure 1: Use-case diagram for a car rental system

Use cases are represented graphically in a use-case diagram as an ellipse with the name of the use case written inside the ellipse or beneath it [Bennett et al., 2001]. Actors are depicted in a use-case diagram as a person with the name written beneath. Actors are connected to use cases by means of relationships, shown as an arrow between the actor and the use case. In figure 1, the actor "Customer" and the use case "Request car" is connected by a relationship line.

An actor can have relationships with more than one use case or with other actors; likewise, a use case can have relationships with more than one actor.

4.4.1 Relationships in use-case diagrams

The following types of relationships or associations can be represented in use-case diagrams [Eriksson, Penker, 1998].

- Generalisation
- Extends relationship
- Uses relationship
- Grouping

Generalisation is a special form of relationship where several actors as part of their roles play more generalised roles. The specialised actors inherit the behaviour of the super class actor. Generalisation in use-case diagrams is depicted as a line with a hollow triangle at the end of the more general superclass actor. Figure 2 indicates a generalisation in a use-case diagram.

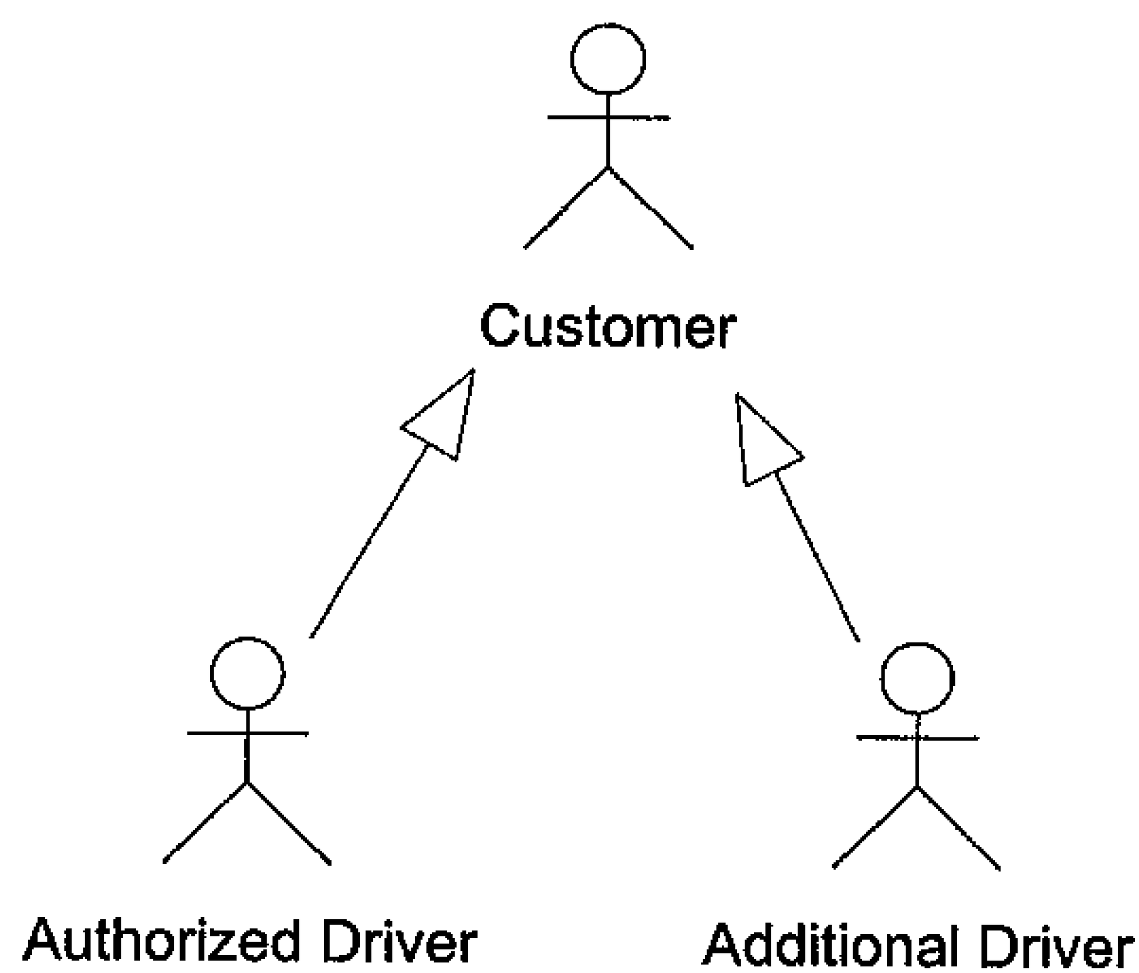


Figure 2: Use-case generalisation

Extends relationship is a generalisation relationship where one use case extends an existing use case by adding new actions to the general use case. The "extends" relationship is depicted in a use-case diagram by adding the stereotype <<extends>> to the generalisation.

Uses relationship is a generalisation relationship where one use case uses another use case and is depicted in a use-case diagram by adding the stereotype <<uses>> together with the generalisation.

Grouping occurs when many use cases can be grouped together because of some similar functionality. Packages in UML do the grouping process, where a package is shown as a large rectangle with a small rectangle connected to the upper left-hand corner of the large rectangle.

The following section explains UML class diagrams.

4.5 Components of UML Class Diagrams

Class diagrams represent a visual model of the classes and associations in a software system. The basic features that must be included in a class diagram are the class names (entities) in boxes and the associations (relationships) connecting them [Post, 1999]. Classes, objects and associations are the primary modelling elements in UML. Class diagrams are drawn by identifying the classes in the problem domain.

UML class diagrams include the following diagrammatic objects:

- ***Classes***, which describe the different kinds of objects existing in the system. Objects are class instances. The symbol for a class is a rectangle with up to three compartments. The top compartment contains the class name and its properties, if any. The class name can be derived from the problem domain and should be a noun [Eriksson, Penker, 1998]. Every class has a unique class name.

The middle compartment contains the list of attributes (member data), which is a list of data representing the properties of an object of interest. An attribute has a type indicating whether the attribute is integer, Boolean etc. The attribute can also have visibility, informing whether it can be referenced from other classes. **Public** visibility indicates that the attribute can be referenced outside the class and can be indicated in the model with a plus sign (+) preceding the attribute; **private** visibility indicates that the class cannot be referenced from other classes and can be denoted by a minus sign (-) preceding the attribute, and **protected** visibility is indicated by the (#) sign, which is used with generalisation and specialisation [Eriksson, Penker, 1998].

The bottom compartment contains the list of operations (member functions) and their properties. Operations are used to carry out any action using the attributes. In an object-oriented programming language, an operation may be treated as a function and therefore specified using a name, return type and parameters. Compartments 2 and 3 are optional. To illustrate the meaning of the three compartments, consider a class with the name "Customer". CustomerID, Licence and Payment are considered as attributes of the Customer class. In an object-oriented language like C++, these attributes are called member data. MakeReservation can be considered as an operation that the object "Customer" can do, and in an object-oriented language such as C++, this can be called a member function denoted as MakeReservation(). A class can be implemented directly in an object-oriented language. Thus the class "Customer" can be represented as shown in Figure 3.

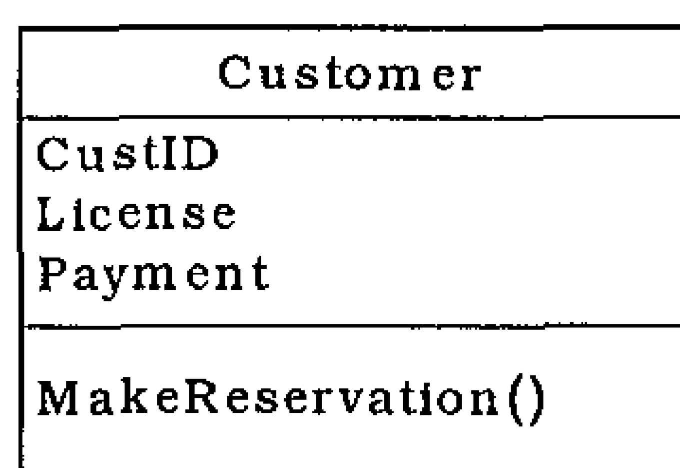


Figure 3: A customer class

- ***Associations***, which describe the natural relationships between the object classes. Classes interact with one another through relationships known as association in UML. Associations are described in detail in section 4.5.2.

- ***Multiplicity constraints***, which explain how many instances of one object or class can be associated with one instance of another object or class. All associations being bi-directional, multiplicities can be defined in both directions for every relationship [Whitten et al., 2000]. Multiplicity constraints are explained in section 4.5.3.

- ***Aggregation*** is a special type of association that represents a whole/part relationship where objects or classes are made up of other objects or classes. Aggregation is depicted in UML by a small diamond on the association line next to the class. Aggregation is explained in detail in section 4.5.7.

- ***Composition*** is a stronger aggregate association where the individual items become a new object [Post, 1999]. Composition is shown by means of a solid filled diamond at the association end. Composition is explained in depth in section 4.5.7.

- ***Constraints*** are restrictions that are placed on relationships between classes and are represented by dashed lines between pairs of associations. A detailed treatment of UML constraints can be found in section 4.5.10.

- ***Generalisations / Specialisation*** group attributes and behaviours that are common to several types of object classes into a class called supertype. Subtype is an object class whose instances inherit some common attributes from a supertype class and then add other attributes that are unique to an instance of the subtype. The subtype object class inherits the attributes and

methods of the supertype object class. Generalisation is described in detail in section 4.5.8.

In general, a class diagram consists of a number of classes that are related by associations and generalisations [Juric R., 1998]. The purpose of a class diagram is to represent the classes in the underlying conceptual model. The following section explains how attributes are depicted in UML.

4.5.1 Representation of attributes in UML

This section discusses how attributes can be depicted in UML. UML represents single-valued and multi-valued attributes. An attribute can possess one or more of the following elements [Hay, 1999]:

- Stereotype – An additional annotation that can be used to enhance the standard UML notation by adding additional semantics to a class. It is depicted in angled brackets called guillemets (<< >>) and can be used to extend entity, attribute and association definitions, generalisations and dependencies. Alternatively, stereotypes can be used to designate attributes and relationships that constitute unique identifiers. When a stereotype is used to extend a class, the class is read as an element type of the specified stereotype [Eriksson, Penker, 1998]. Figure 4 indicates a Customer class with the stereotype <<Driver>> that adds extra semantics to the class.

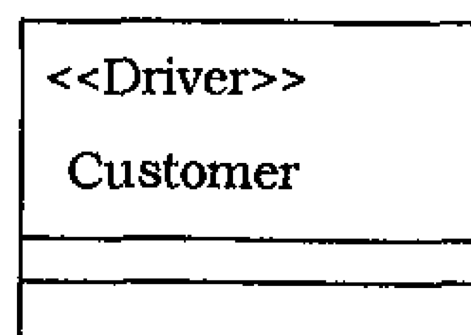


Figure 4: Customer class with stereotype

- Visibility - A measure of the visibility of a class to all the classes, or only to the subtypes of this class, or to this class only.

- Name - The only mandatory element.

- Multiplicity – The multiplicity of an attribute is an indication of the number of separate values the attribute can hold.

- Type - Defines the type of the attribute whose values depend on the model's environment.

- =initial value – An initial value can be specified for an attribute.

{other} – Any additional named properties that could be added to the attribute.

The following section describes how single-valued attributes are represented in UML.

4.5.1.1 Single-valued attributes

In UML, relationships can be modelled as attributes. In object models, entities are called object classes [Hay, 1999]. For example, figure 5 shows a Branch class with five attributes. The class attributes, namely "BranchNr", "Location", "Manager", "Capacity" and "Group_quota", capture the information that identifies an instance of the Customer class. The attributes "BranchNr", "Location" and "Manager" are unique to a Branch and therefore they are single-valued. UML does not have a standard graphic notation for these attribute uniqueness constraints. Hence notations are chosen by the modeller, and textual constraints are appended in braces after the attribute names: (P = primary identifier, U = unique, with numbers appended if needed to disambiguate cases where the same U constraint might apply to a combination of attributes) [Halpin, Bloesh, 1999].

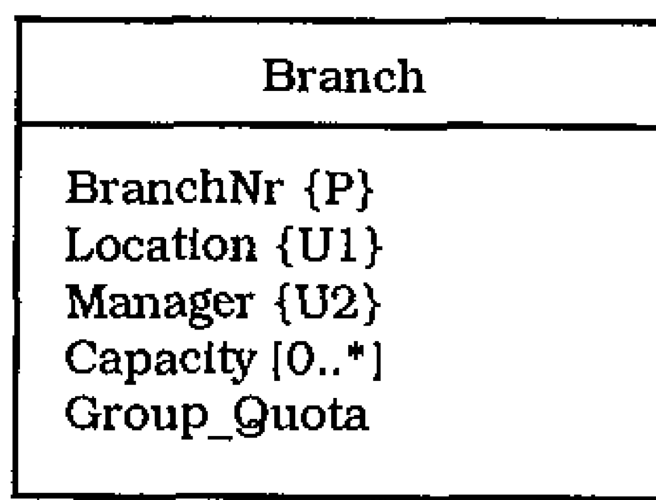


Figure 5: Single-valued attribute in UML

The concept of single-valued attributes is extended to the situation where attributes have multiple values.

The following section describes how multiple valued attributes can be represented in UML.

4.5.1.2 Multi-valued attributes

Consider the situation where we are interested in recording the different models of a car. A car can have different models. In UML, this situation may be modelled as shown below in figure 6:

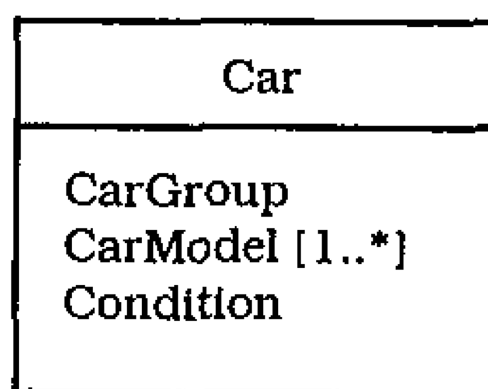


Figure 6: Multi-valued attribute in UML

The information that a car can have one or many models is depicted in UML by appending "[1..*]" to the CarModel attribute. Thus CarModel becomes a multi-valued attribute. The "*" in "[1..*]" indicates that there is no upper limit to the number of models of a car - a car may have many models. If "*" is used without a lower limit, this is taken as an abbreviation for "0..*" [Halpin, 2001].

An attribute with no explicit multiplicity constraint is assumed to be mandatory and single-valued (exactly one). This can be depicted explicitly by appending "[1]" to the relevant attribute. For example, to indicate explicitly that each car has exactly one model, we would use "CarModel[1]".

UML provides object-diagrams for the purpose of instantiation. These are essentially class diagrams in which each object is shown as a separate class instance, with data values supplied for its attributes. Figure 7 is a population, which is depicted in a UML object diagram.

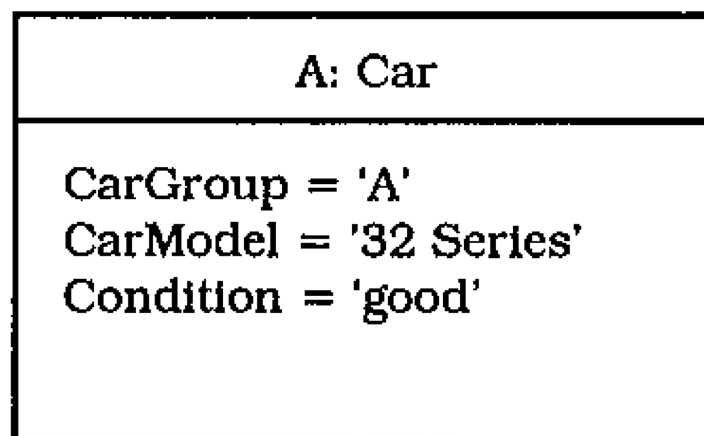


Figure 7: Instantiation in UML

Figure 7 is a population of figure 6. The object 'A' is an instance of the car class with attribute set as shown in figure 7. For simple cases like this, object diagrams are useful. However, if multiple instances are to be displayed for more complex cases, the diagram gets more complicated [UML semantics, ver1.1, 1997].

Classes have attributes that describe the characteristics of the objects. UML gives us the choice of modelling a feature as an attribute or an association. This choice helps us to specify, visualise, construct and document the artefacts of a software system [Alhir, 1998]. It also enables us to express various constraints involving the "role played by the attribute" in a standard notation.

The following section explains how associations can be represented in UML class diagrams.

4.5.2 Associations

An association represents a semantic link between the objects of the classes participating in the association [Eriksson, Penker, 1998]. An association can indicate the roles connected to each class taking part in the association, depicted by role names. This facility is optional and they are part of associations. The number of roles in a relationship is called its "arity". Associations can be unary, binary, ternary and higher orders. The following section discusses the unary, binary and ternary associations and how each type can be represented in UML.

4.5.2.1 Unary associations

Unary associations are modelled in UML as Boolean attributes but allow relationships of all other arities [Halpin, 2001]. An example of a unary association is "Customer drives". This situation can be modelled in UML using the Boolean attribute "IsDriver : Boolean".

4.5.2.2 Binary associations

Binary associations are the most common type of associations occurring between classes and are shown as lines between classes. An association carries a name, which is usually a verb. For example, Class "Branch" and class "Car" are associated if some object of class "Branch" links with some object of class "Car". The association may be verbalised, as Branch owns Car. Arrows can be used to navigate through associations, and the arrow indicates that the association can be used only in the direction of the arrow [Eriksson, Penker, 1998]. Names can be added on both sides of the association, which in UML is indicated by a solid triangle. The solid triangle either precedes or follows the association names, depending upon the direction. The solid triangle in figure 8 reads the association as "Car returned to Branch". Figure 8 shows a simple binary association represented in UML.

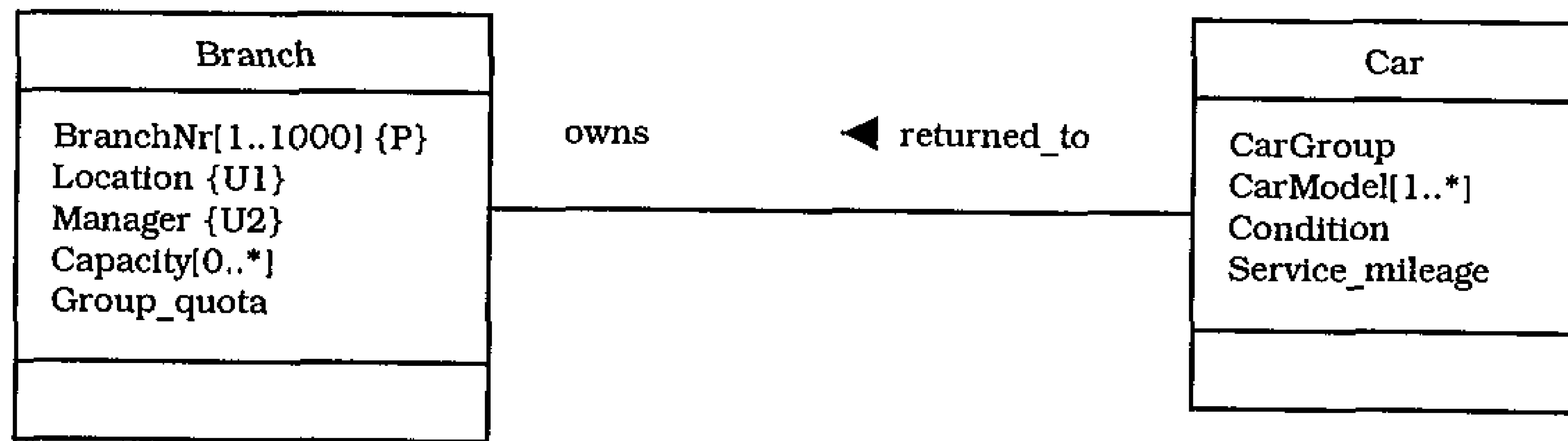


Figure 8: UML association

In Figure 8, names are added on both sides of the association. The association can be read as Branch owns Car or Car returned to Branch.

Role names may be added to the line endings of associations in order to specifically identify the role played by the classes participating in the association. Adding role names is optional, but once added, they may not be suppressed [Juric, 1998]. For example, Customer can play the role of driver in the association "Customer reserves Car". Roles are useful for representing the context of a class and its objects [Eriksson, Penker, 1998].

4.5.2.3 Ternary associations

Ternary association associates three classes, and in UML they are depicted by a large diamond connected to the classes by lines. Roles and multiplicities can be shown optionally on ternary associations, but since many lines are used to denote the association, directional verbalisation is ruled out, so the diagram cannot be used to communicate in terms of sentences [Halpin, 2001]. Figure 9 shows a ternary association in UML which can be read as "Lectures are booked for a given classroom and time".

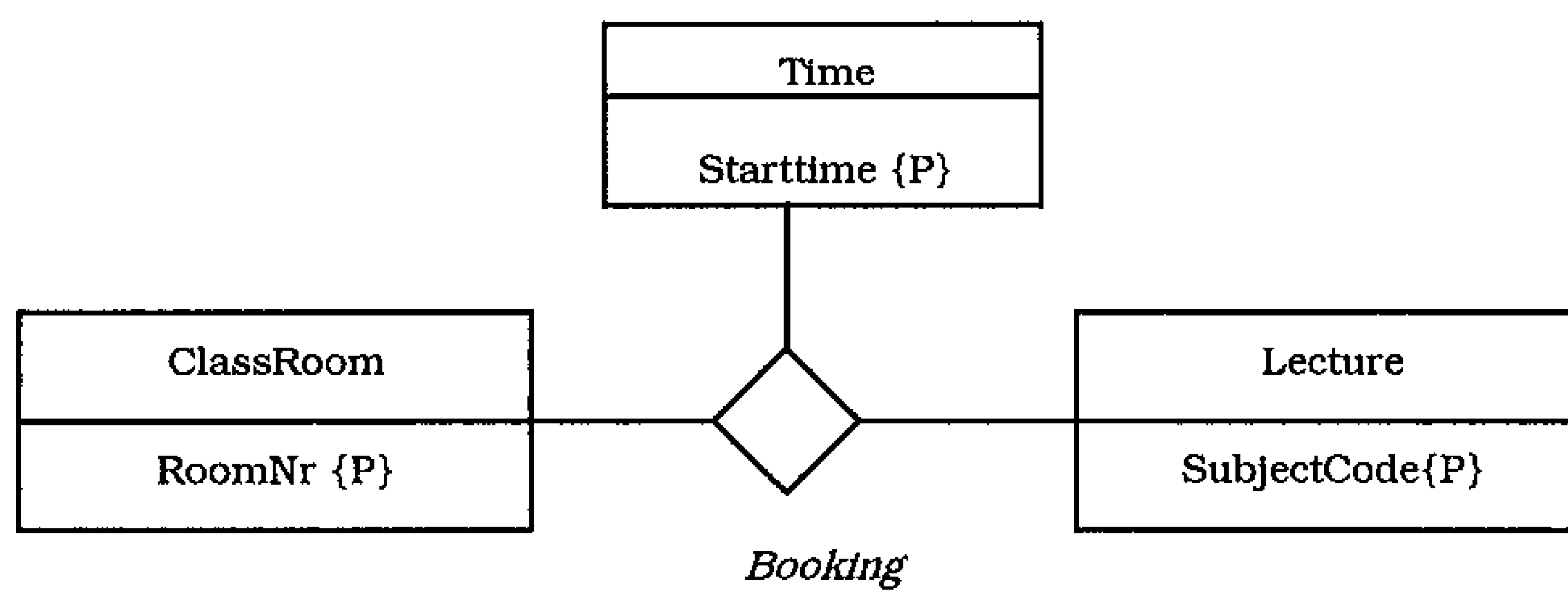


Figure 9: Ternary association in UML

The following section elaborates on how multiplicity constraints can be shown on associations.

4.5.3 Multiplicity constraints on associations

The previous section discussed how UML shows multiplicity constraints on attributes. The same technique is used to specify multiplicities on associations. The notion of multiplicity is used to specify a range of objects linked together in the association.

Multiplicity can be expressed as follows: zero-to-one as (0..1), zero-to-many as (0..* or *), one-to-many as (1.. *), two to seven as (2..7), and so on. If no range is specified, it means exactly one by default [Bennett et al., 2001]. Associations use a similar notation, where the relevant multiplicities are written beside the relevant roles. In Figure 10 the relevant multiplicity constraints are added to figure 8. UML places each multiplicity constraint on the far role, in the direction in which the association is read.

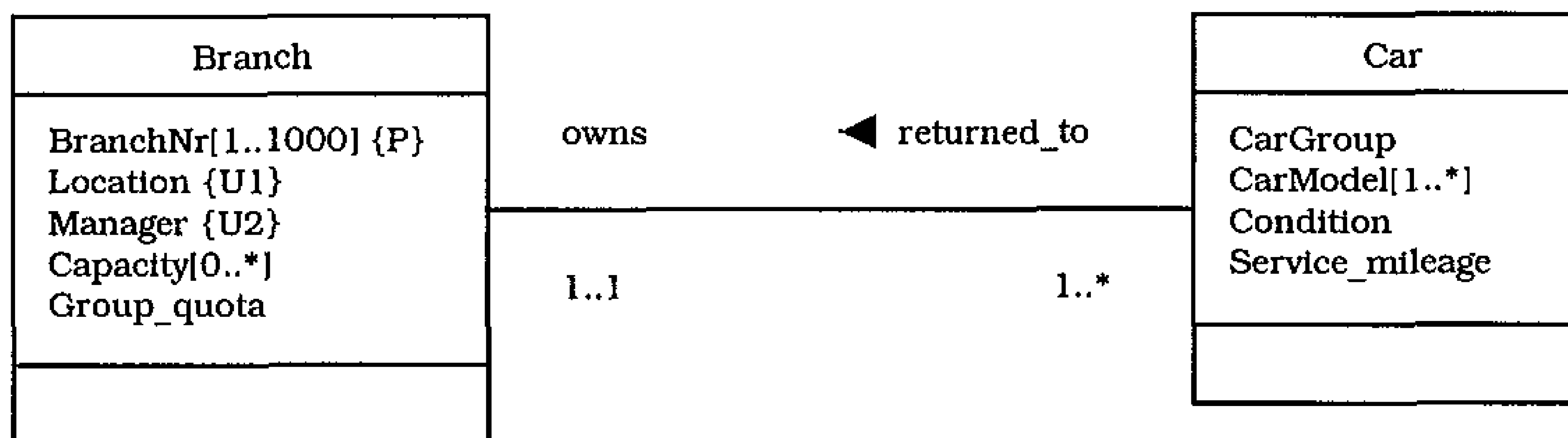


Figure 10: Multiplicity constraints in associations

The multiplicity constraints applied to figure 10 indicate that a branch owns one or more cars. Multiplicities add all the relevant meaning to the model. Figure 11 adds the multiplicity constraints to the ternary association depicted in figure 9.

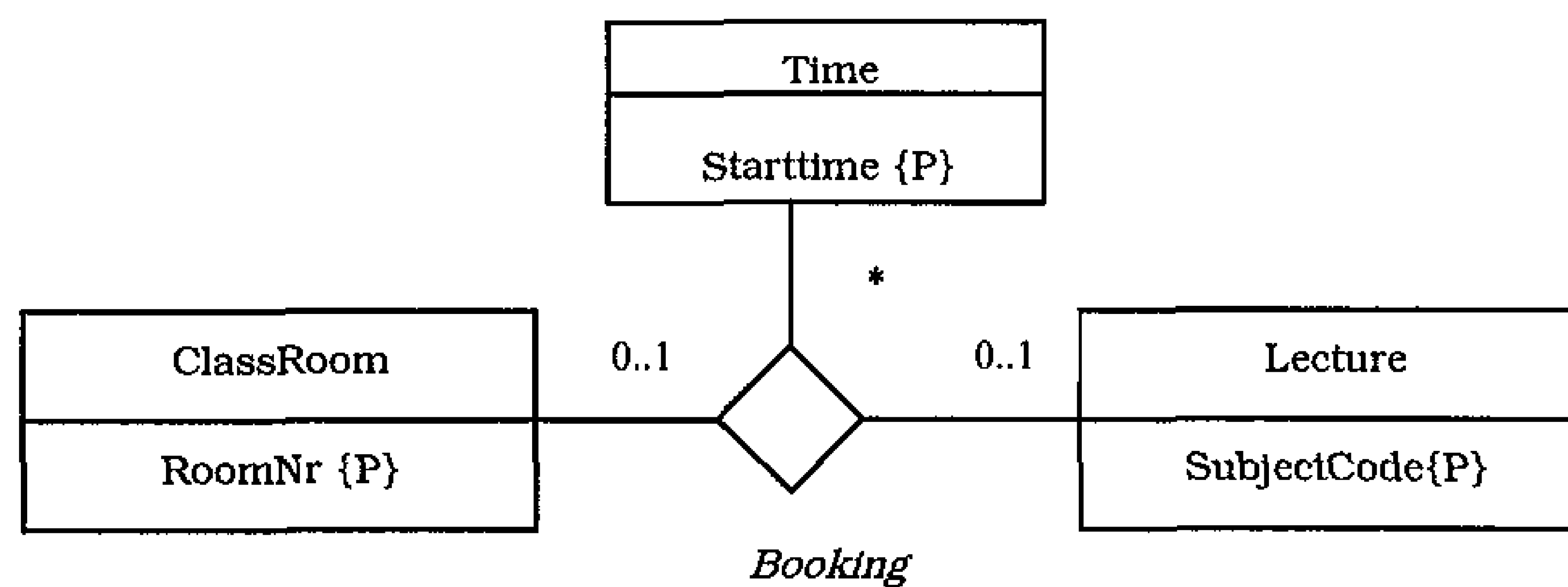


Figure 11: Multiplicity constraints on a ternary association in UML

Figure 11 indicates that for a given Classroom and time, at most one lecture is booked.

4.5.4 Association classes

A special class can be attached to an association called an association class. An association class is an association that also has class properties. It not only connects a set of classifiers (classes), but also defines a set of features that belong to the relationship itself and not only to the classifiers [Gogolla, Richters, 1999]. The association class can have attributes, operations and

other associations [Eriksson, Penker, 1998]. The main purpose of using an association class is to add additional information to the association. UML allows any association (binary and above) to be objectified into a class, regardless of its multiplicity constraints. An association class can be attached to an association by dotted lines as depicted in figure 12. Car is related to Customer through Reservation. Thus Reservation becomes an association class, which adds additional information to the association between Customer and Car.

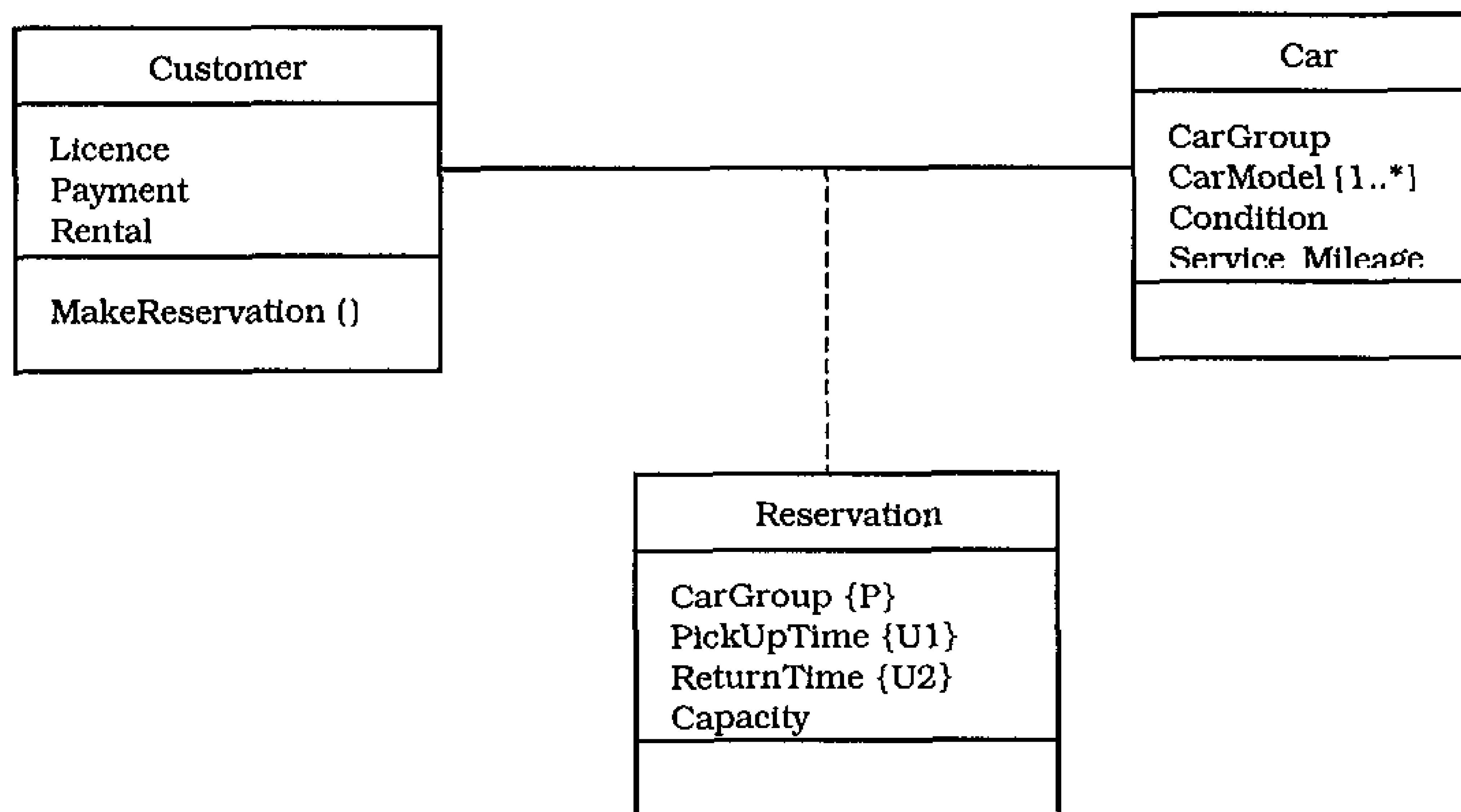


Figure 12: Association class in UML

4.5.5 Qualified associations

Qualified associations are used with one-to-many or many-to-many associations [Eriksson, Penker, 1998]. A qualifier is an attribute or list of attributes whose values serve to partition the set of objects associated with an object across an association. This means that qualifiers are attributes of the association [Gogolla, Richters, 1999]. A qualifier is depicted as a rectangular box enclosing its attributes. Figure 13 is an example of a qualified association in UML.

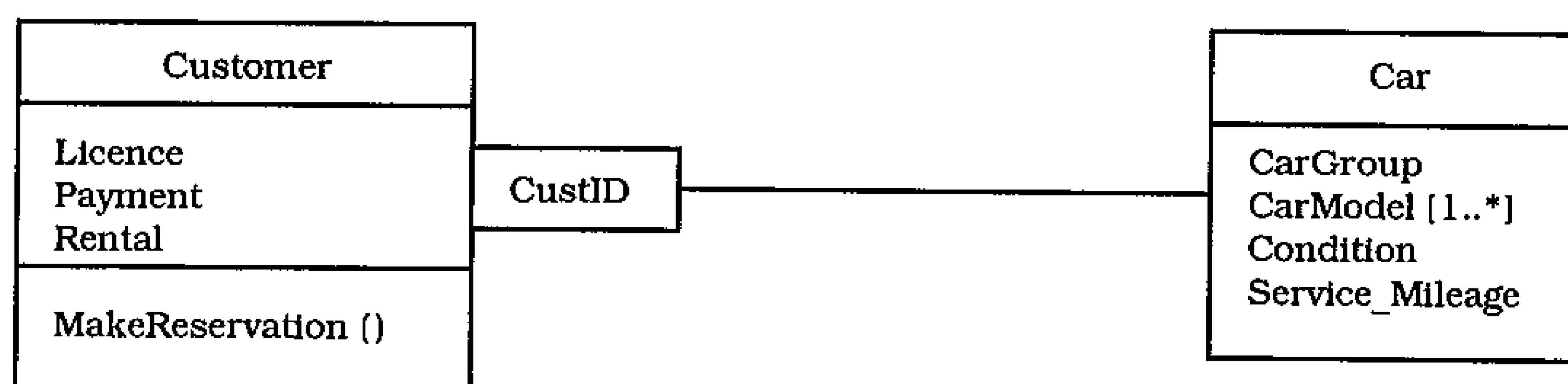


Figure 13: Qualified association

Here CustID serves as the qualifier for the association depicted in Figure 13. The qualifier CustID distinguishes a person belonging to the Customer class. The qualifier helps to identify a specific reservation made by a customer for a specific car.

4.5.6 OR associations

The term "or-association" is used in UML to represent one or many associations arising from a class. At any moment, each member in a class may participate in at most one of these associations. An or-association can be considered as a constraint on two or more associations.

To represent or-associations diagrammatically, UML uses an or-constraint between the associations, attaching the constraint string "{or}" to a dotted line connecting the relevant associations. Figure 14 is an example of an or-association.

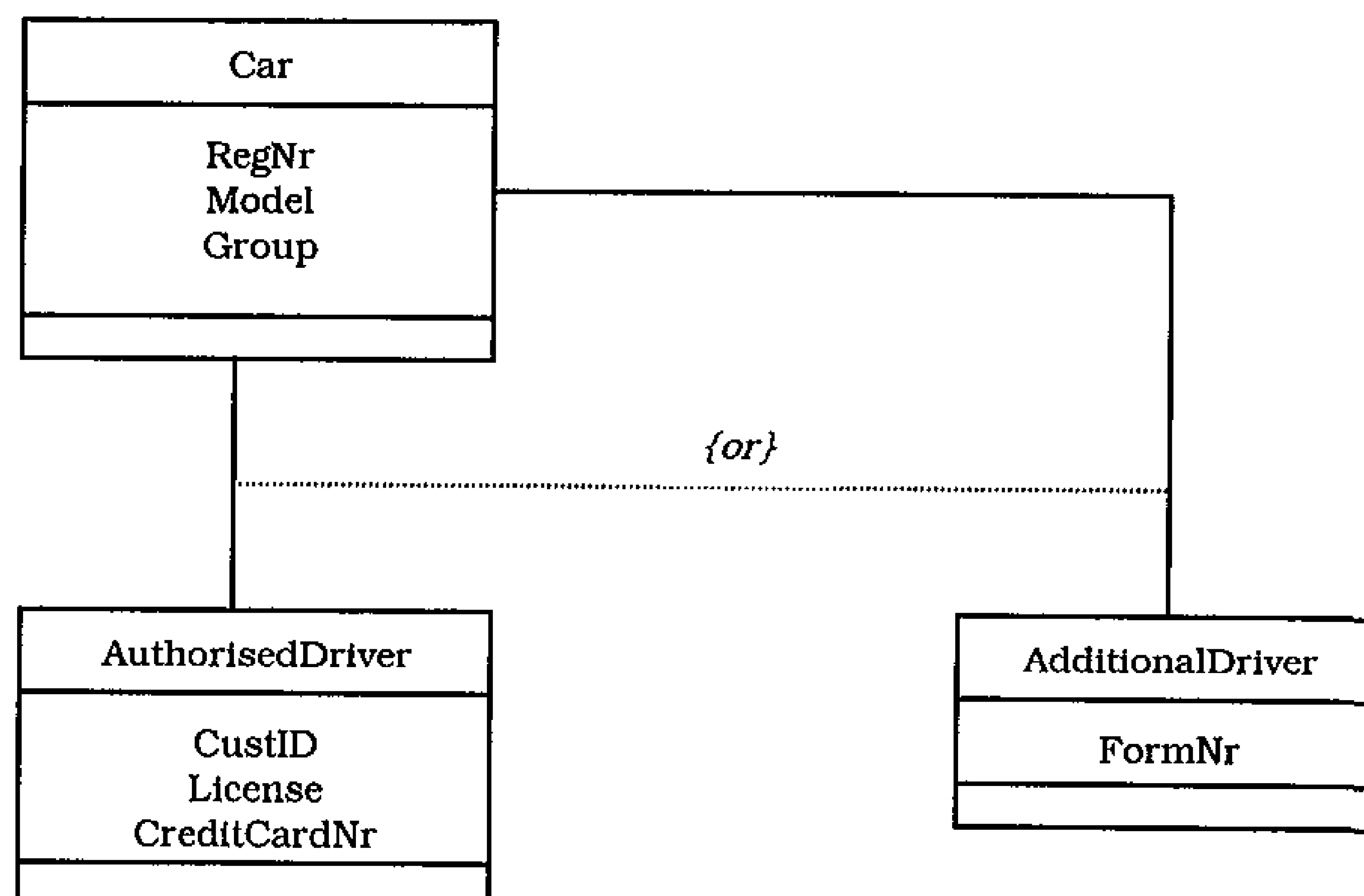


Figure 14: OR-association

In figure 14, the UML model may be verbalised as follows: A Car class is associated with authorised driver or an additional driver, but not both.

4.5.7 Aggregation and composition

Aggregation is a special form of association that specifies a part/whole relationship between the aggregate (whole) and a component part. Composition is a form of aggregation with strong ownership and coincident lifetimes of a part with the whole. For example, the LoyaltyScheme consists of Customers, so this membership may be modelled as an aggregation association between LoyaltyScheme and Customer. According to Eriksson [Eriksson, Penker, 1998], the keywords that can be used to identify aggregates are "consists of", "is part of", "contains", etc. Currently, UML associations are classified into one of three kinds: ordinary association (no aggregation); shared (or simple) aggregation; composite (or strong) aggregation. UML uses an unfilled diamond at the end of the association indicating an aggregation. The diamond is connected to the aggregate class. Aggregation associations are also referred to as "whole-part". Class instances at one end of the association are

called the "wholes" which are made up of "parts". The aggregation diamond is always at the whole end of the association [Bennett et al., 2001].

A shared aggregation is one in which the multiplicity on the whole side is other than 1 [Eriksson, Penker, 1998]. Shared aggregation is denoted in UML as a binary association, with a hollow diamond at the "whole" or "aggregate" end of the association. Figure 15 is an example of a shared aggregation.

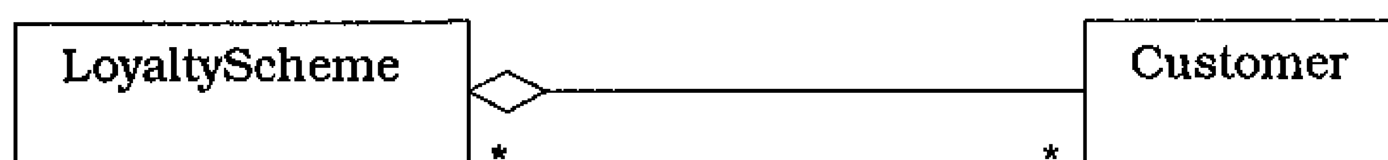


Figure 15: Shared aggregation in UML

Composition (composite aggregation) owns its parts and is depicted with a filled diamond. In this case, the part is completely owned by the whole. For example, Figure 16 shows a composition.

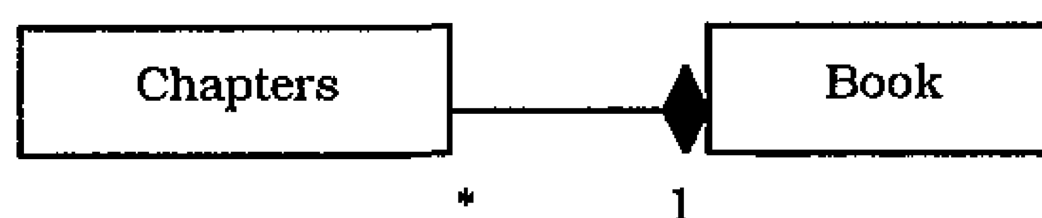


Figure 16: Composition in UML

A book contains many chapters. Therefore the association between "Book" and "Chapters" can be treated as a composition since the part (Chapters) is strongly owned by the whole (Book).

UML allows some alternative notations for aggregation. If a class is an aggregate of more than one class, the association lines may be shown joined to a single diamond (see Figure 17(a)). For composition, the part classes may be shown nested inside the whole by using role names, and multiplicities of components may be shown in the top right hand corners (see Figure 17(b)) [Halpin, 2001].

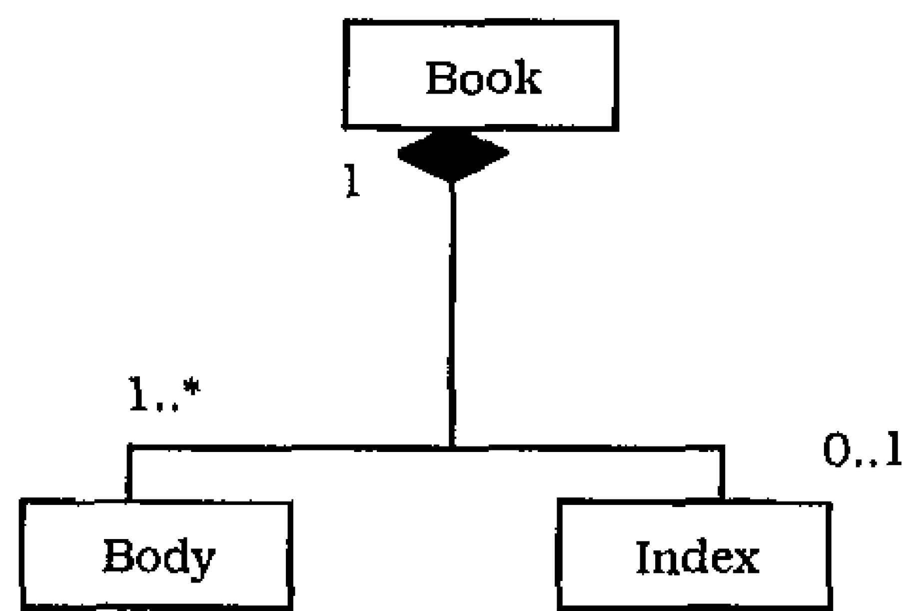


Figure 17: a (composition)

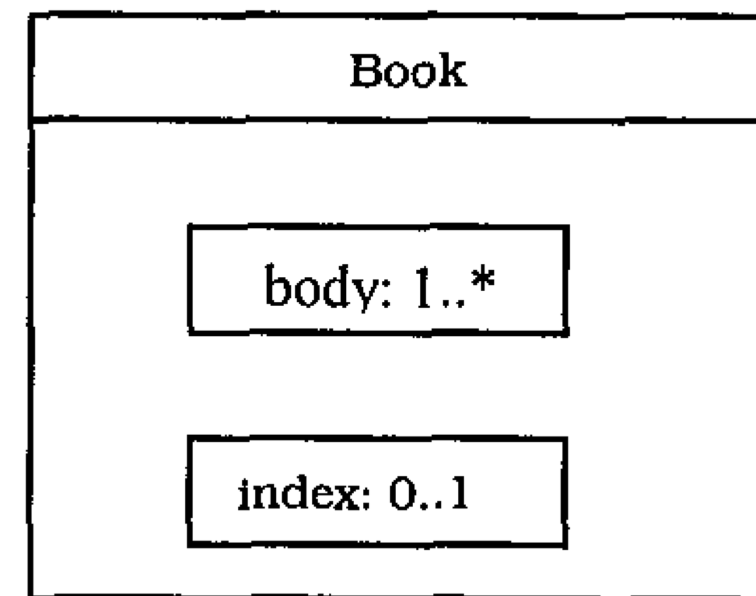


Figure 17: b (composition)

4.5.8 Generalisation

Generalisation is the taxonomic relationship between a more general element and a more specific element that is fully consistent with the first element and that adds information [Gogolla, Richters, 1999]. This means that generalisation is a relationship between a more general class and a more specific class, where the more specific class is fully consistent with the more general class with additional information [Eriksson, Penker, 1998]. Generalisation is called inheritance, where classes can be generalised into new classes that can be treated as separate classes of their own. This is sometimes called an "is-a" relationship where a new class is a generalisation of an existing class. In UML, specialisation and generalisation are inverse procedures; specialisation introduces subtypes and generalisation introduces supertypes.

Subtypes are used in object modelling to do at least one of the following:

1. Assert typing constraints

2. Encourage reuse of model components

3. Show a classification scheme (taxonomy)

In this context, typing constraints ensure that subtype-specific roles are played only by the relevant subtype. Subtyping supports reuse [Hay, 1999]. Since a subtype inherits the properties of its supertype(s), only its specific roles need to be declared when it is introduced. Apart from reducing code duplication, the more generic supertypes are likely to find reuse in other applications. Using subtypes to show taxonomy is of little use, since taxonomy is often more efficiently captured by predicates.

For example, the association "Customer is a driver" may be treated as a generalisation, where Customer is called a superclass and Driver is called a subclass that is derived from the Customer class. Generalisation is depicted using a solid line from the subclass ending with a large triangle at the superclass end. Figure 18 indicates a generalisation in UML.

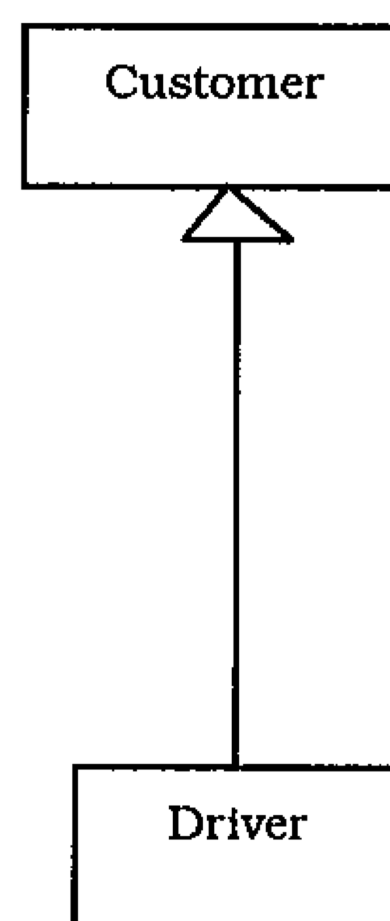


Figure 18: Generalisation

Generalisation may be extended using the following constraint names in order to add more meaning [Bennett et al. 2001]:

- Overlapping
- Disjoint
- Complete
- Incomplete

These names help to add more information on the generalisation and are represented in braces along with the generalisation. Overlapping generalisation means that any subclasses inheriting from the subclasses in the inheritance relationship can inherit more than one of the subclasses. This is a situation called multiple inheritance. Disjoint generalisation means that the subclasses are not allowed to be specialised into a common subclass. Complete generalisation is a special case where no more subclasses can be inherited from the superclass. Incomplete generalisation, as the term implies, indicates that subclasses can be added on in future [Eriksson, Penker, 1998].

UML also distinguishes between abstract and concrete classes. An abstract class cannot have any direct instances, and is shown by writing its name in italics or by adding "{abstract}" below the class name. Abstract classes are realised only through their descendants. Concrete classes may be directly instantiated. This means that it is possible to derive objects from a concrete class. Abstract and concrete classes have little relevance at the conceptual level, but are important in the code designing phase.

4.5.9 Dependency relationships

A dependency relationship can sometimes occur between two classes. Where one class is dependent on the other class, a change in the independent class will have an effect on the dependent class. The dependency relation is depicted using a dashed line with an arrow between the classes. A stereotype can be used as a label, which will identify the type of dependency. For example, a friend dependency in C++ explicitly gives a class access to the members of another class. Figure 19 is an example of a dependency relationship where the class Customer acts as a friend to the class Reservation. Therefore Customer can access all the data of the Reservation class.

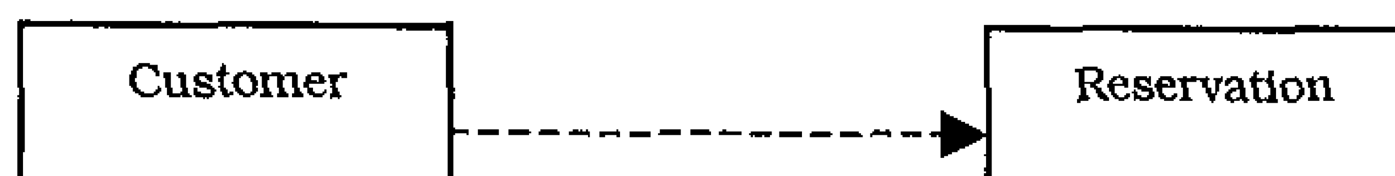


Figure 19: Dependency relationship

Having discussed the different types of relationships in UML, it is now important to identify the UML constraints.

4.5.10 UML constraints

Constraints help to add rules or restrictions to a model. Specifying the constraints in a conceptual model adds more meaning to the existing model and improves its ability to capture the requirements more completely. UML provides techniques that support the specification of different kinds of constraints; each of them is discussed in the following sections.

4.5.10.1 Subset constraints

UML does not support the possibility of inheriting the structure of a class and then redeclaring it [Podehl, Arnold, 1999]. However, UML allows subset constraints to be specified between associations by attaching the constraint label "{subset}" next to a dashed arrow between the associations. For example, the subset constraint in figure 20 indicates that many branches own many cars; a car returned to a branch is specifically owned by that branch.

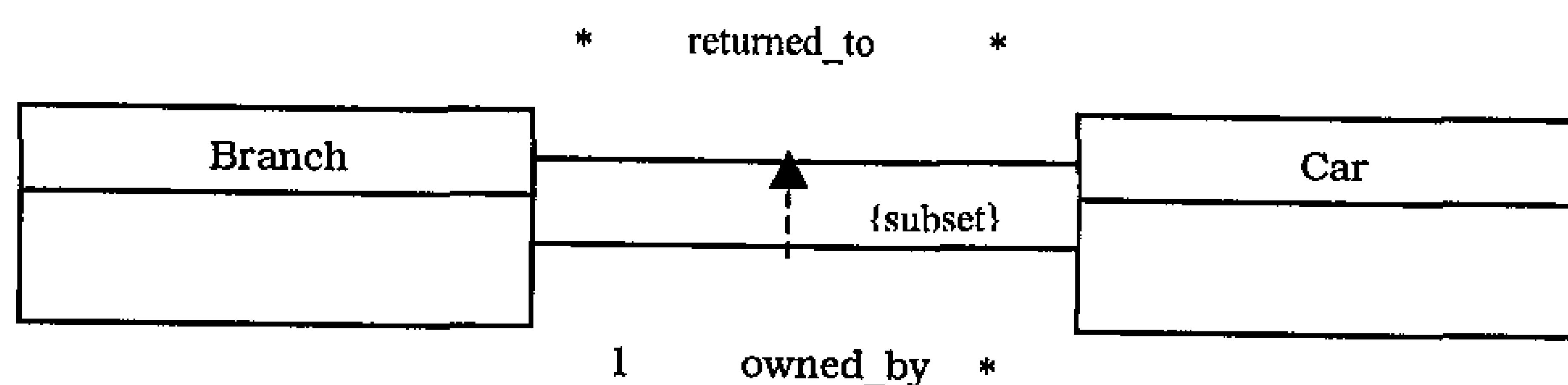


Figure 20: Subset constraint in UML

4.5.10.2 Textual constraints

UML enables the modeller to add textual notes to make the model more understandable and to convey special meaning related to the modelling element. Moreover, UML has no graphic notation for equality constraints. Textual constraints help to solve this problem of not having special notation for equality constraints by writing a note (in braced comments). For whole

associations, the use of two separate subset constraints would make the model too complicated [Halpin, 2001]. The following example in figure 21 illustrates the use of textual constraints.

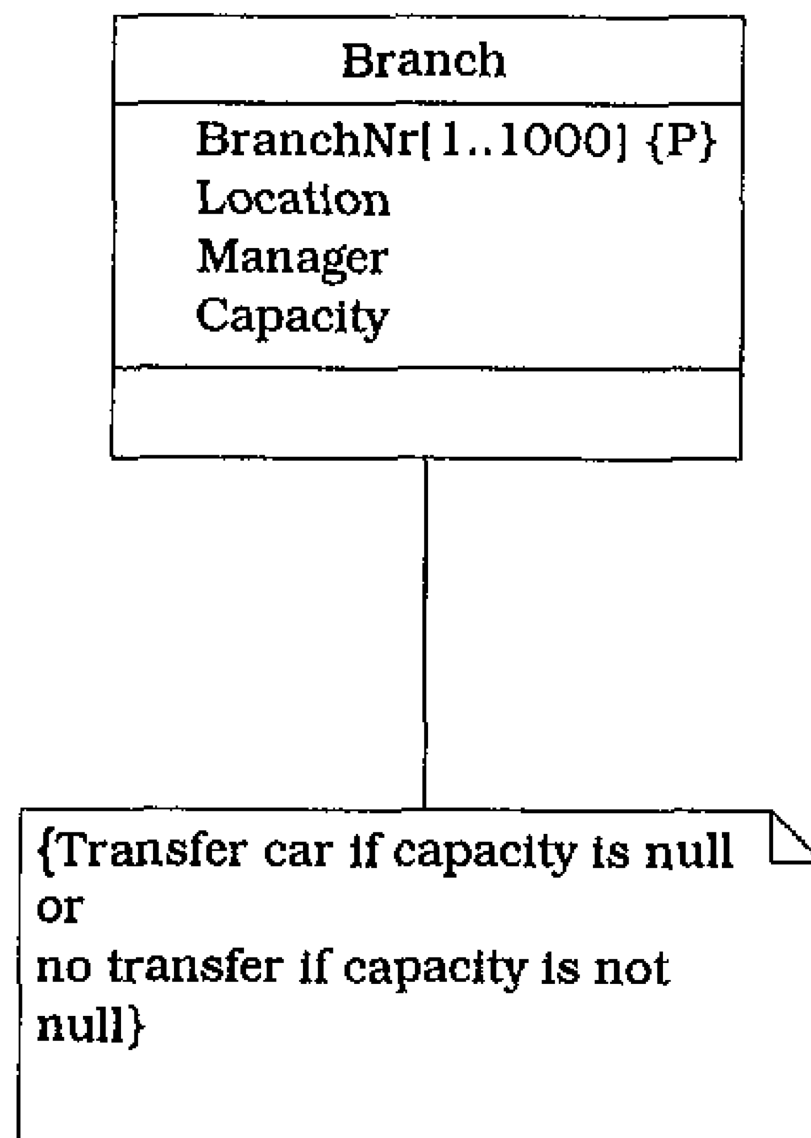


Figure 21: Textual constraint

Figure 21 indicates that a branch can transfer cars from other branches if there are no cars in that branch.

4.5.10.3 Value, ring and join constraints

Constraints may sometimes be value constraints, ring constraints or join constraints in UML. Value constraints are used to restrict the value of an object. Value constraints can be specified as enumerations. Enumeration types may be modelled as classes, stereotyped as enumerations, with their values listed as attributes [Halpin, 2001]. Declaring a textual constraint in braces, using any formal or informal language, may specify ranges and mixtures. In UML, value constraints other than enumeration, range and mixture may be declared as textual constraints. Figure 22 shows an enumeration type in UML.

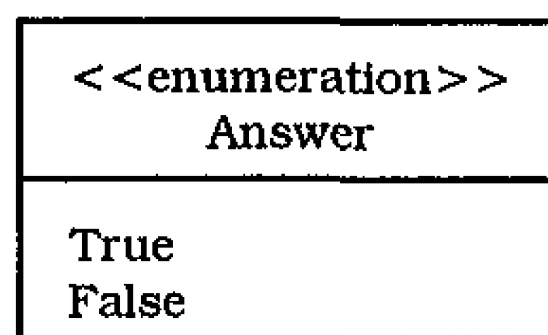
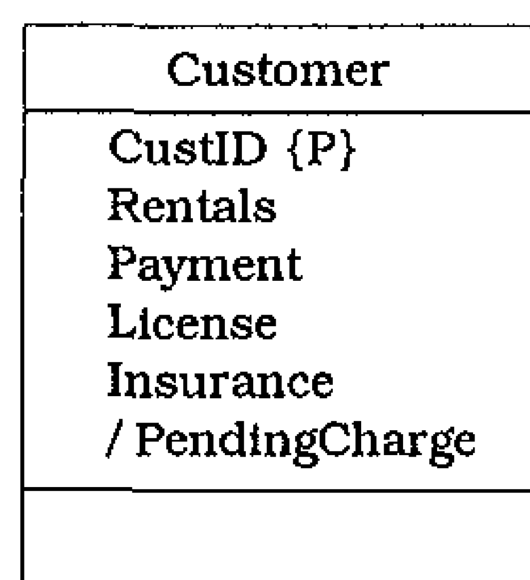


Figure 22: Enumeration type

UML does not provide built-in ring constraints, but allows the modeller to specify them as textual constraints. Although join constraints arise frequently in real applications, UML has no graphic symbol for them. However, they may be declared on UML diagrams by writing a textual constraint or comment in a note attached by a dashed line to the model elements involved.

4.5.11 Derivation rules

Derivation rules help to specify how something can be derived, and in UML these rules can be attached to a class. The rules are specified in braces near the class. The derivation rules can be used for attributes, associations, inheritance and roles [Eriksson, Penker, 1998]. A derived attribute begins with a slash. UML's capability of expressing derivation rules makes the model very powerful. This is illustrated in figure 23.



{PendingCharge = NormalCharge + DamageCharge}

Figure 23: UML derivation rule

The static structure of a system depicts the object classes that exist in the system as well as the associations that exist between the classes. The static

model indicates what the system contains and how the classes are related, but does not indicate how the classes communicate with each other to provide the required functionality of the system. The dynamics of a system explains how the objects communicate as well as the effects of such communications.

UML uses different diagrams (listed in section 4.3) to model the dynamics of the system. Each diagram serves a special purpose and they are explained in the sections below.

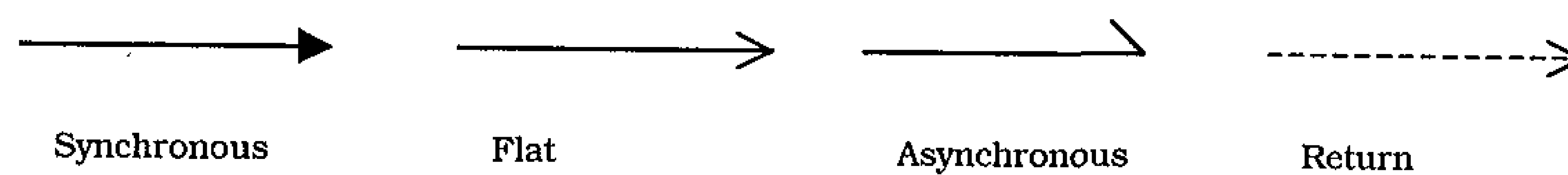
4.6 Sequence Diagrams

Dynamic models model the behaviour of objects that is necessary to implement the functionality of a system. In dynamic modelling, an interaction between two objects is depicted as a message sent from one object to the other and depicted by an arrow between the sender and receiver [Bennet et al., 2001]. Sequence diagrams focus on the time ordering of messages. A sequence diagram is drawn by placing the participating objects at the top of the diagram horizontally and the messages that the objects send vertically. All objects have instances represented in a sequence diagram which are indicated by vertical dashed lines. Such a line is called the lifeline of the object and represents the time during which an object exists. If an object exists before the commencement of the interaction and continues to exist after the interaction, then the lifeline is shown from top to bottom as shown in the sequence diagram of figure 24. According to Bennet [Bennet et al., 2001], sequence diagrams are useful to model the following situations:

- They model the high-level interaction between the active objects in a system.
- They model the interaction between object instances within collaboration realising a use case.
- They are used to model an interaction between objects within a collaboration that realises an operation.

- They can be used either to model generic interactions showing all possible paths through the interaction or specific instances of an interaction showing just one path through the interaction.

Objects communicate by means of horizontal message lines between the object lifelines. The message lines are labelled with the message that is sent from the sender object. Message lines are shown by different arrow styles, each representing a different type of message. The following types of message lines can be shown in a sequence diagram [Bennet et al., 2001]:



A synchronous message is sent from one object to the other and the sender object waits until the resulting action has been completed. A flat arrow shows a progression from one step to the next in a sequence. An asynchronous message is sent from the sender object to the receiver object where the sender object does not wait for the completion of the resulting action. Return flow is used to show the return of control from the receiver object to the sender object.

Figure 24 indicates a sequence diagram starting with a "change customer information" message. The sequence diagrams for the use cases identified in the car rental case study can be found in Appendix C.

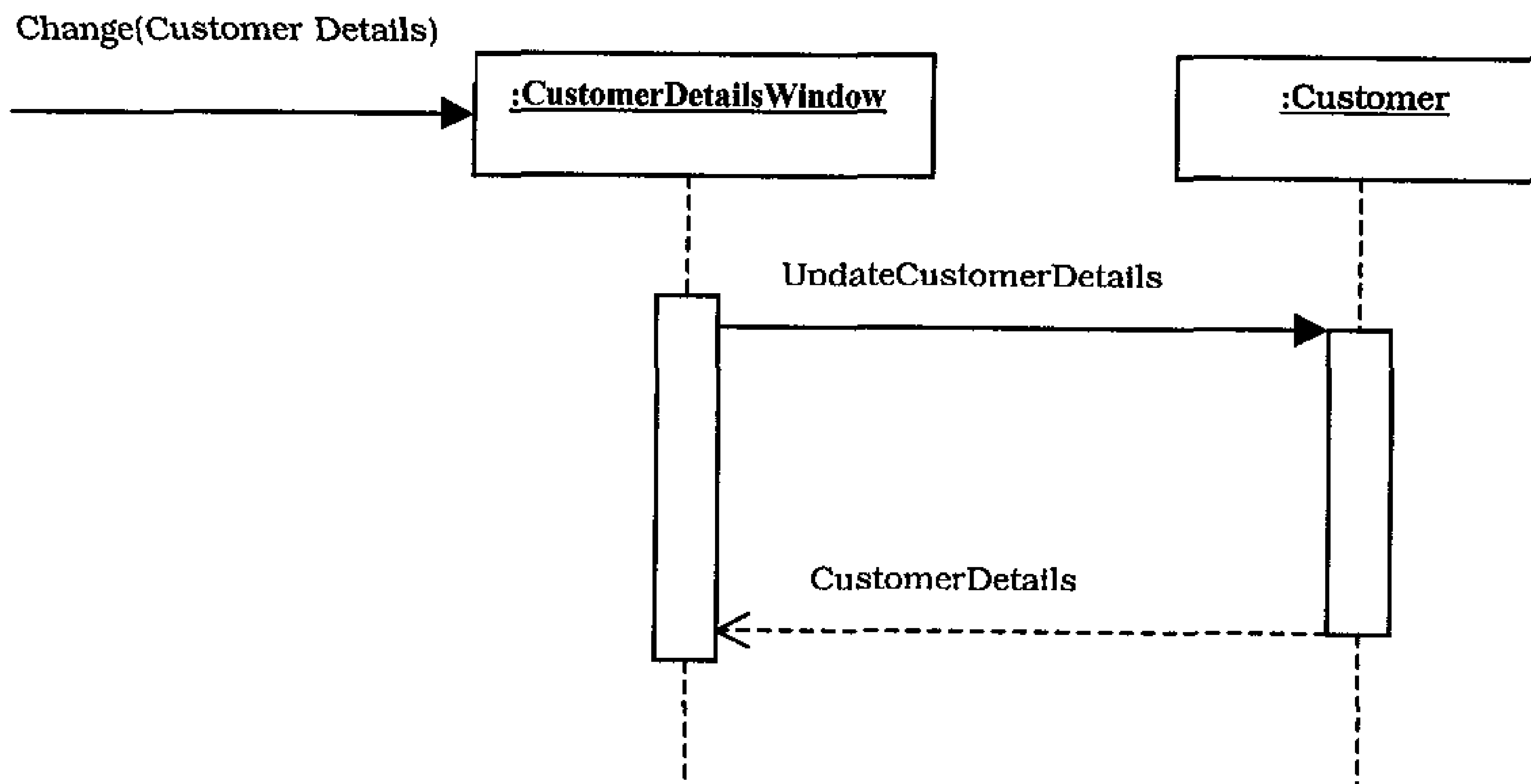


Figure 24: Sequence diagram

Sequence diagrams can depict the destruction of an object using an 'X' at the end of a lifeline of an object as shown in figure 25.

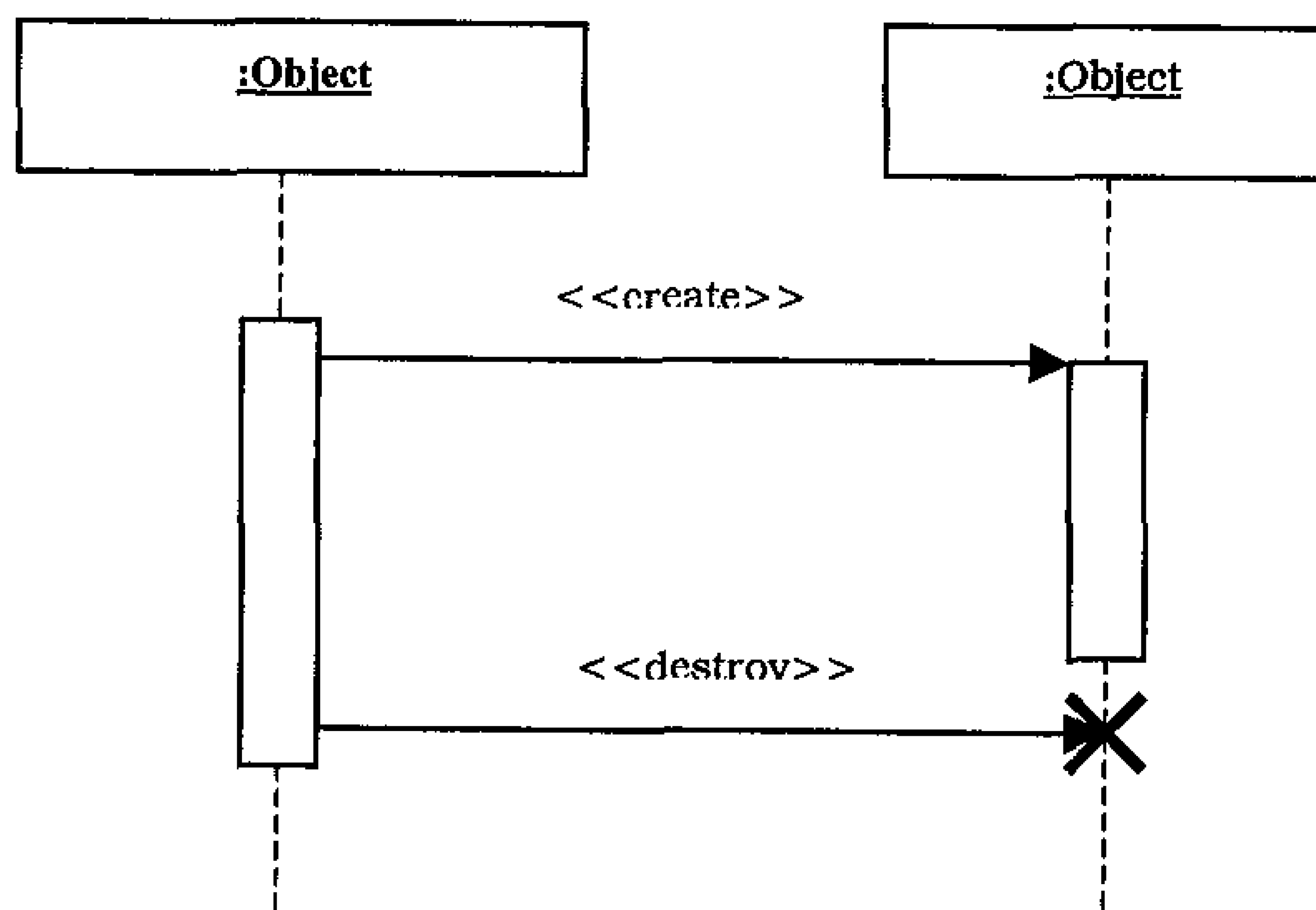


Figure 25: Sequence diagram where object is destroyed

4.7 Collaboration diagrams

Collaboration diagrams model the interaction between the objects rather than the timing of the interaction, as in sequence diagrams [Sturm, 1999]. Like the sequence diagram, the collaboration diagram shows how messages are passed between the objects and actors. The diagram can represent a use case completely.

The primary step in building a collaboration diagram is to identify the actors and objects that participate in the interaction. The actors and objects are identified from the use case, and the interactions between the objects and actors are represented by lines called "links" [Booch et al., 1999a].

Messages sent between the objects and actors are depicted as named arrows pointing from the sender to the receiver of the message and are numbered in order of occurrence. The entire use case may be modelled using separate collaboration diagrams, each depicting a particular scenario. Figure 26 shows the outline of a collaboration diagram. The collaboration diagrams developed for the use cases can be found in Appendix B.

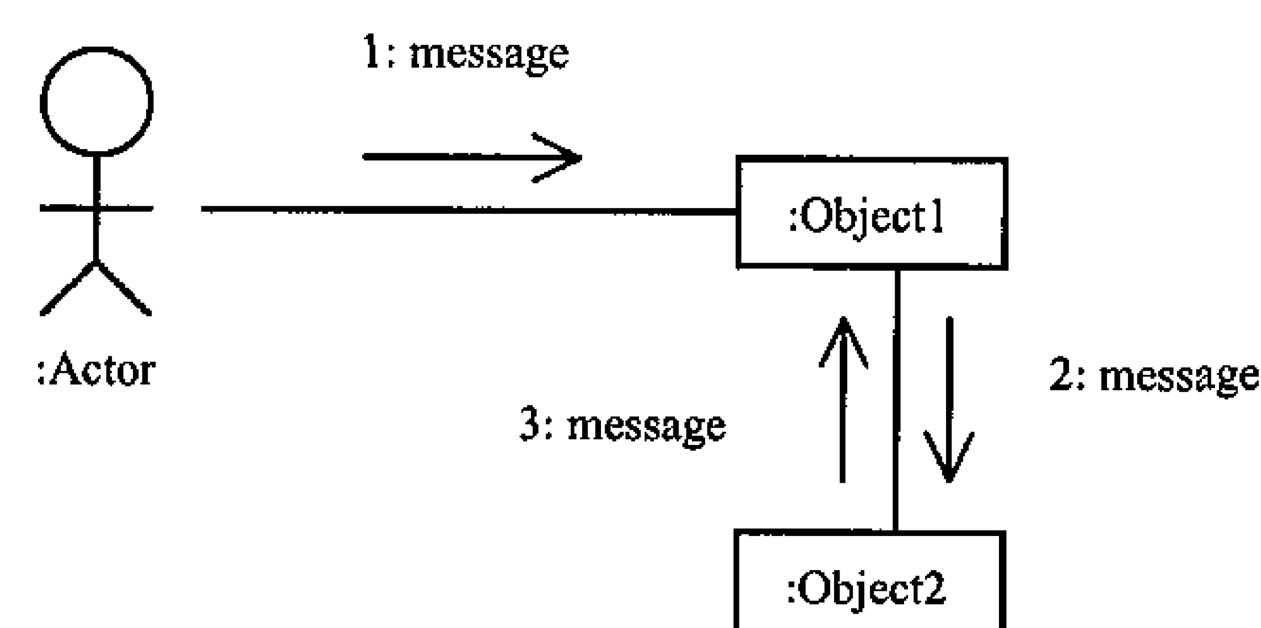


Figure 26: Collaboration diagram

4.8 Deriving the Domain Model Class Diagram From the Use Cases

The domain model class diagram is derived through a series of iterations from use-case diagrams to interaction diagrams and then the domain model class diagram. The starting point of deriving the domain model class diagram is the use-case model. The second step is to build interaction diagrams from the identified use cases. Sequence diagrams or collaboration diagrams may be used for this purpose. Both these interaction diagrams serve the same purpose: sequence diagrams focus on the temporal sequence of events involved in an interaction, whereas collaboration diagrams focus on the structural organisation of objects. The domain model class diagram is a structural organisation of object classes and their associations. For this reason, this study uses the collaboration diagram for modelling the interactions involved in the use cases for the case study.

Collaboration diagrams are drawn by identifying objects from the use cases. In order to identify the objects, each noun involved in the use case is considered as an object. The domain model class diagram includes different classes, where a class is a description of a set of objects with attributes and operations. Classes are extracted from the various objects identified in the collaboration diagram. The iterative UML process helps to identify missing classes in the domain model class diagram; this is the reason for modelling the collaborations and refining the use cases iteratively. The complete collaboration diagrams for the use cases can be found in Appendix B, from which the domain model class diagram is derived.

4.9 Conclusion

Use cases are important to capture and document the requirements of a system. A UML domain model class diagram consists of classes that are linked by associations and generalisations. Multiplicity constraints can be depicted on associations. An association may be labelled with a name called a role name. An association that has class properties becomes an association class.

Attributes of associations can be shown as qualifiers. A class can be generalised into a superclass and subclass, where a superclass denotes a generalising class and subclass denotes a specialising class. Additional constraints can be placed on objects by the relationships that exist between classes. UML constraints may be value constraints, ring constraints or join constraints. A semantic model should describe the object set assigned to a class and the constraints enforced on them. The subtyping capabilities of UML are described in this chapter. Finally, aggregation and composition, special forms of association, are explained with examples.

The static modelling of a system is done with the help of the UML class diagram. UML supports modelling the dynamic structure of a system by providing the state, sequence, collaboration and activity diagrams. Sequence diagrams and collaboration diagrams model the interaction between the objects and actors, but the difference between the two diagrams is that sequence diagrams focus on the interaction timing whereas the collaboration diagram does not represent the passage of time but shows the links and messages between the objects and actors. The domain model class diagram is constructed through a series of iterations from the use cases to interaction diagrams. This helps to identify missing classes in the domain model class diagram.

In this study, use-cases, collaboration diagrams and the domain model class diagrams were used. Sequence diagrams were discussed to provide more background on dynamic modelling but were not used in the case study because of its focus on the timing of the interaction rather than the nature of the object interaction.

Chapter 5

OBJECT ROLE MODELLING

5.1 Introduction

Object Role Modelling (ORM) is a technique used to construct the conceptual model of a software system. The root of ORM is the elementary fact [Becker, 2000]. A fact represents a true statement about the system. An elementary fact (discussed in section 5.2) is one that cannot be broken down into smaller facts that can be combined to yield the original fact. ORM models a software system as a set of objects that plays roles or takes part in relationships. The term object in ORM does not mean the objects in object-oriented technology; in ORM terms, an object refers to any physical thing (nouns) that exists in the system, and objects in a fact are connected by means of relationships. It is called "fact-based" modelling because ORM verbalises data as facts [Halpin, 2001]. The technique originated in the mid-70s in Europe and came to be known as Natural Language Information Analysis Method (NIAM). ORM has since been used in different parts of the world and has been studied and extended by many researchers. In the 1980s, Halpin formalised this technique for the first time and added many extensions to the technique, which came to be known as Formal ORM (FORM) [Halpin, 2001]. The essential feature of this ORM technique is that the underlying conceptual model is free of any attributes, which are the descriptors of objects, and it has an extensive collection of notations for expressing business rules. Attributes of objects are modelled as relationships in ORM. In essence, ORM is a conceptual modelling technique that helps modellers develop, evolve and validate their conceptual models.

Microsoft Corporation provides a software tool called VisioModeler that supports the creation of ORM models. It is an integrated collection of design tools that allows one to create a conceptual model. It has an enhanced user interface that allows the discovery and documentation of business rules and a fact editor that helps to define, edit and add constraints to the fact types in the ORM diagram. This assistance eliminates human errors to a great extent.

The ORM technique and the different notations that are supported by the technique are studied in detail in the following sections.

5.2 Components of ORM diagrams

The ORM conceptual model is built on elementary facts, which are declarations or statements that an object plays a particular role. A simple example of an elementary fact may be "Customer rents car". In this fact, the object (Customer) plays a role (rents). An elementary fact does not include logical connectives such as "not, and, or, if, all or some". ORM considers an object as an entity or value, both of which are special types of objects. An entity refers to a separate thing that physically exists in the system and is referenced by relating it to other objects. Example of an entity is Car has CarGroup 'A'. A value refers to an object, which is unchangeable and has a constant value. An example of a value is Car has RegNr. Here RegNr is a fixed value. When simply stated as object, entity or value, it refers to instances. A set of possible instances of entities or values is referred to as entity types or value types.

Object types are depicted as named ellipses, with solid lines being used for entity types and dashed lines for value types [Halpin, 2001]. An entity is identified by a reference scheme. For example, cars can be identified by registration numbers; the registration number acts as a reference scheme for the entity Car. The reference scheme is shown in parenthesis with the entity types. An entity type can have more than one primary reference, and one of them can be declared as a primary reference in the underlying model.

ORM uses relationship types to represent a relationship between object types. Since ORM models do not use attributes, only the relationship types are used to connect objects and their roles. A relationship type between object types is depicted as a named sequence of one or more role boxes connected to the object type that plays it. Each role is represented as a box connected to the object type that plays the role. Each relationship type has at least one predicate name. Both forward and inverse predicates (separated by a "/") may be shown for relationships. If only one predicate is necessary for a relationship type, the other may be ignored. A predicate describes the relationship types. In figure 27, "rented_to" and "rent" are called predicate names. ORM supports relationships of any arity (binary, ternary etc.) that plays any number of roles [Hay, 1999]. Figure 27 represents a binary relationship and Figure 28 represents a ternary relationship. In the ORM representation, both forward and inverse predicate names may be shown. Sometimes showing the predicate name in one direction is enough to verbalise a fact. Role names may be added in the ORM diagram.

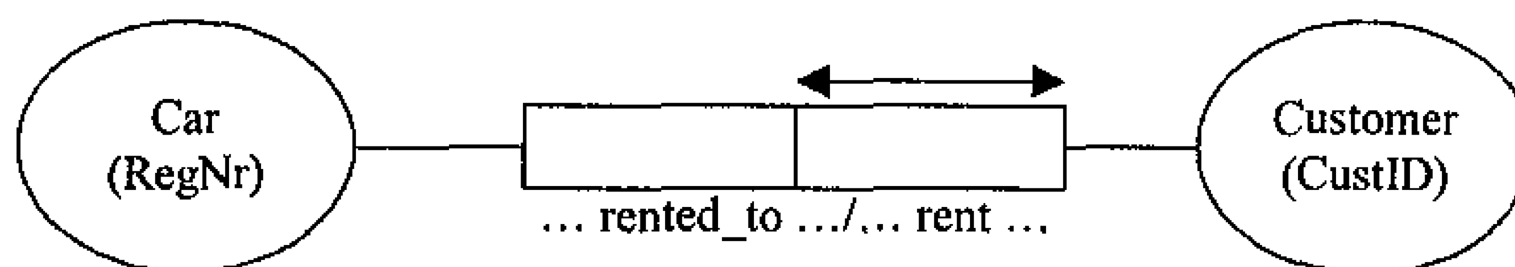
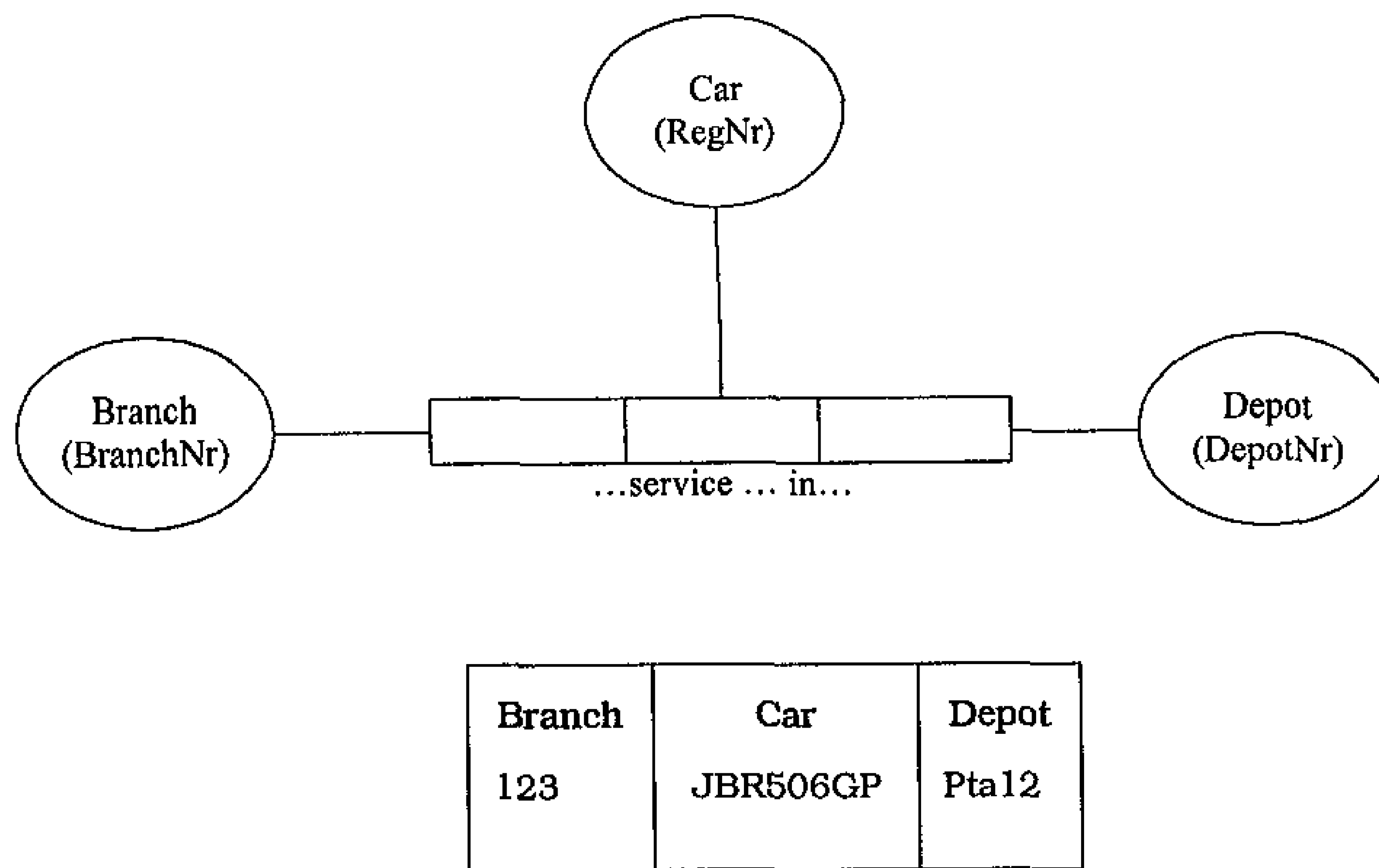


Figure 27: Binary relationship in ORM

In figure 27, Car and Customer are the two object types (entity types in this case). "RegNr" and "CustID" are the reference schemes for the object types. The predicate "rented_to" is used as a forward predicate and "rent" as an inverse predicate. The relationship may be verbalised as "Car rented to customer" or "Customer rent car". The uniqueness constraint that is indicated by the double-headed arrow on the right-hand role of figure 27 is explained in section 5.4.3.

A ternary relationship is represented in ORM as shown in Figure 28. In a conceptual schema, the ternary fact type is depicted as a sequence of three role boxes, each of which is attached to an object type as in figure 28. The relationship may be verbalised as "Branch service car in Depot". For each

relationship, a fact table as depicted in figure 28 may be provided to populate sample data. Each column of the fact table is associated with a single role. The fact table helps to validate facts with domain experts.



Branch	Car	Depot
123	JBR506GP	Pta12

Figure 28: A ternary association in ORM

Relationship types are of two types, namely reference types and fact types. These are discussed in the following sections.

5.2.1 Reference types

A relationship type that is used only for primary referencing is called a reference type, where every instance of the entity type is related to a unique value that identifies it. Any other relationship type is called a fact type. The number of roles is called the "arity" of the relationship type. Figure 29 indicates a reference type in ORM.

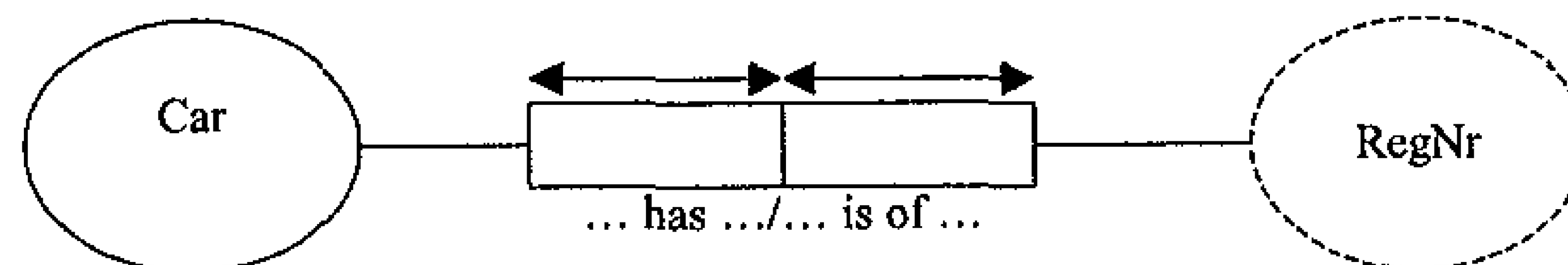


Figure 29: Reference type in ORM

In figure 29, "Car" is an entity type and "RegNr" is a value type. This reference type may be verbalised as each car has exactly one registration number.

The building block of the ORM model is the fact. Since facts must be verbalised correctly in order to build a correct conceptual schema, the different fact types are discussed next.

5.2.2 Fact types

A fact associates many objects that take part in relationships and is a combination of entities, attributes and relationships [Hay, 1999]. Facts can be simple elementary, compound or derived facts. For example, "Branch owns Car" is an elementary fact where the fact cannot be split into sub-facts without information loss. A compound fact combines many facts, for example, "Branch owns Car and Car can be returned to any Branch". A derived fact is a fact whose value is derived from other facts. For example, "The Rental_Charge is calculated from the rental_rate times the number of days" becomes a derived fact. Derived fact types are indicated by an asterisk.

A set of instances of facts is collectively called a fact type. For example, "Each Branch owns at most one Car" is a fact type. Figure 30 shows an ORM representation of an elementary fact type.

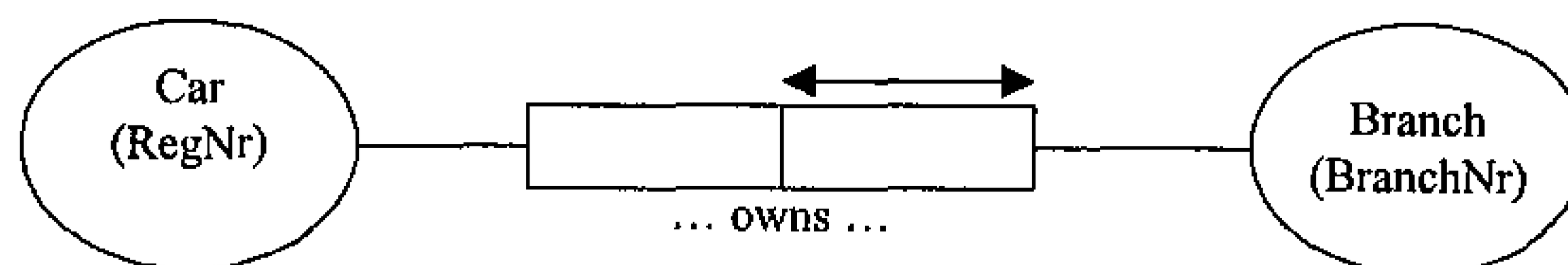


Figure 30: ORM simple fact type

Figure 31 shows a derived fact type. In figure 31, the derivation rule is Charge = daily_rate + local_tax.

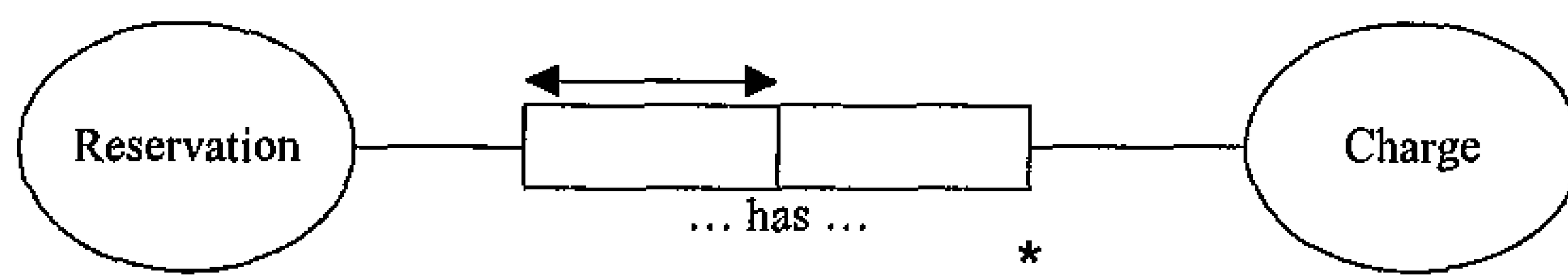


Figure 31: Derived fact type

Objects in a fact type are connected together by means of relationship types. Relationship types added to object types make a complete fact type. The relationship types in ORM can be objectified, where an objectified relationship treats a relationship between objects as an object itself, as explained below.

5.3 Objectified Relationship Types

ORM allows relationship types to be objectified. In ORM, a verb phrase is objectified by a noun phrase. This facilitates natural verbalisation for the constructs, and different names can be used for both [Halpin, 2001]. Relationships that are objectified in ORM must have at least two roles. The relationship then becomes a nested fact type [Hay, 1999]. This objectified relationship type can be considered as an entity having other entities related to it. An ORM model depicts the objectified association with an object type frame, as illustrated in figure 32.

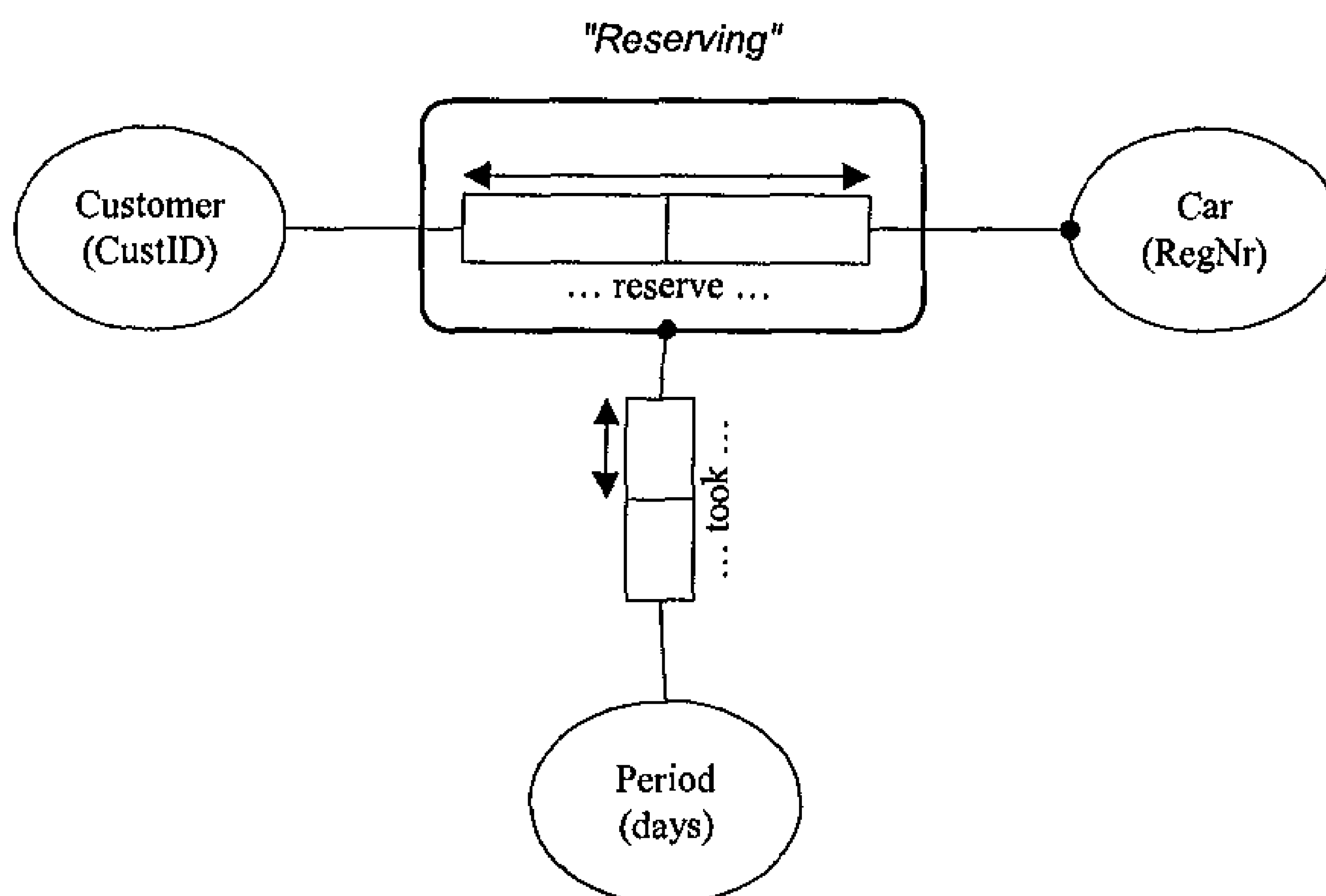


Figure 32: Objectified relationship types in ORM

In figure 32, the objectified relationship type "Reserving" is unique for a Car and Customer combination.

The conceptual schema of a software system must be capable of capturing the fact types and express all the constraints correctly on the fact types. Only then can we say that the schema captures all the relevant business rules. Therefore representing constraints on a conceptual schema is vital to the successful completion of a model. ORM provides a rich collection of constraints, which are explained in the following section.

5.4 Representing Constraints in ORM

ORM is capable of representing different types of constraints that are likely to occur in complex business rules. This section identifies and discusses the different ORM constraints and how they can be depicted in the conceptual model.

5.4.1 Mandatory role constraints

A role appearing in an ORM conceptual schema may be either mandatory or optional. A mandatory role constraint in the ORM model indicates that every entity in the model must have an associated value and is depicted as a solid dot. For example, in figure 33 the four mandatory role constraints indicate that every Branch must have a Branch Number, Manager, Location and Capacity. This may be verbalised as each branch has at least one branch number, each branch has at least one manager, each branch has at least one location and each branch has some capacity.

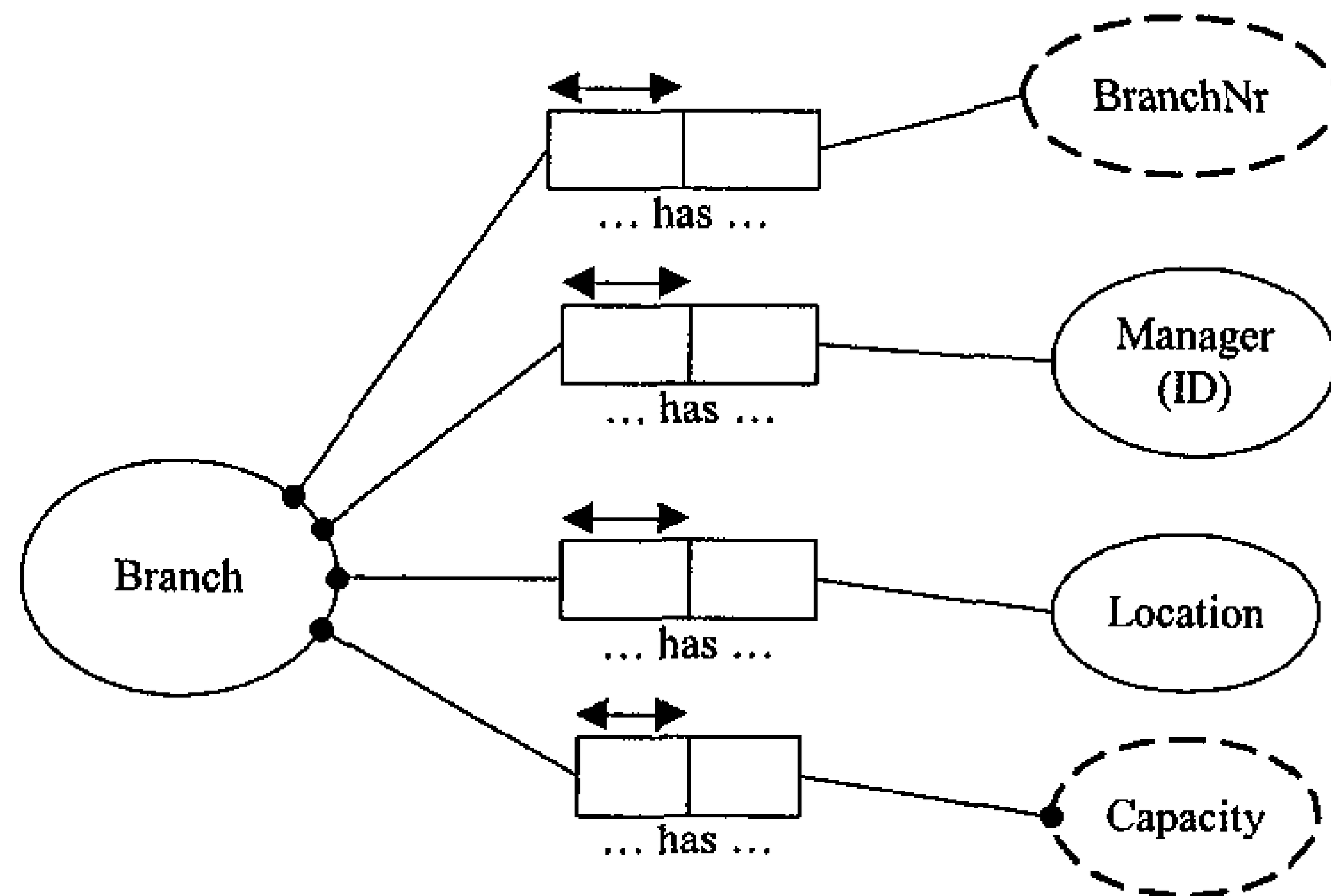


Figure 33: Mandatory role constraints in ORM

5.4.2 Disjunctive mandatory role constraints

ORM facilitates the use of disjunctive mandatory role constraints, which means that the disjunction of two or more roles is mandatory. Figure 34a and 34b depicts the disjunctive mandatory role constraint, which can be verbalised as Car has CarGroup or CarModel or both.

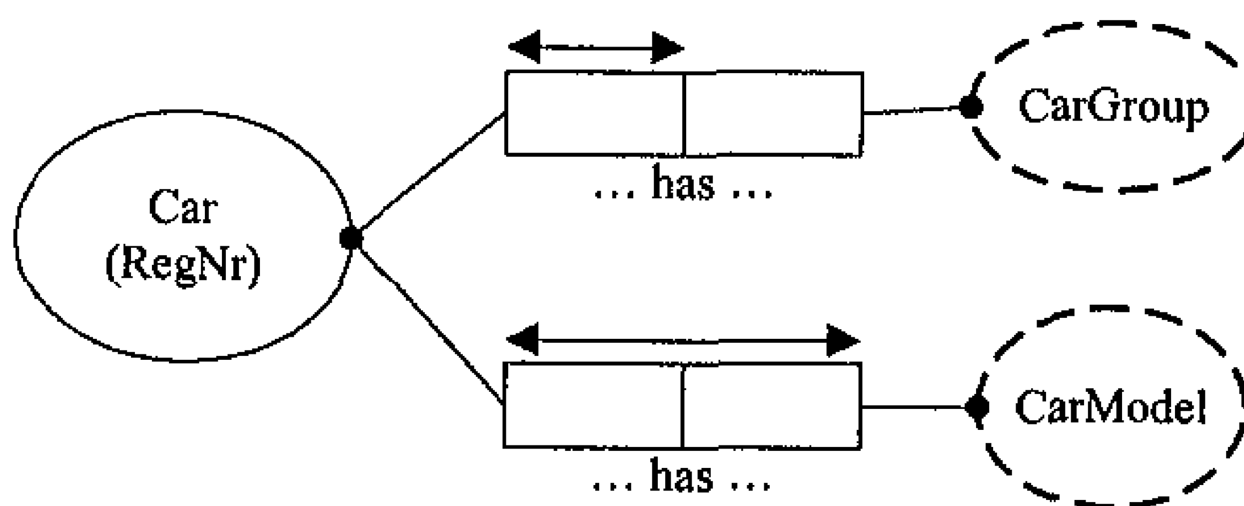


Figure 34a: Disjunctive mandatory role constraint

At least one of the two roles must be played, even though the two roles are optional individually. A disjunctive mandatory role is shown explicitly by linking the individual roles to the disjunctive mandatory role dot. Disjunctive mandatory roles have an alternate notation as shown in figure 34b.

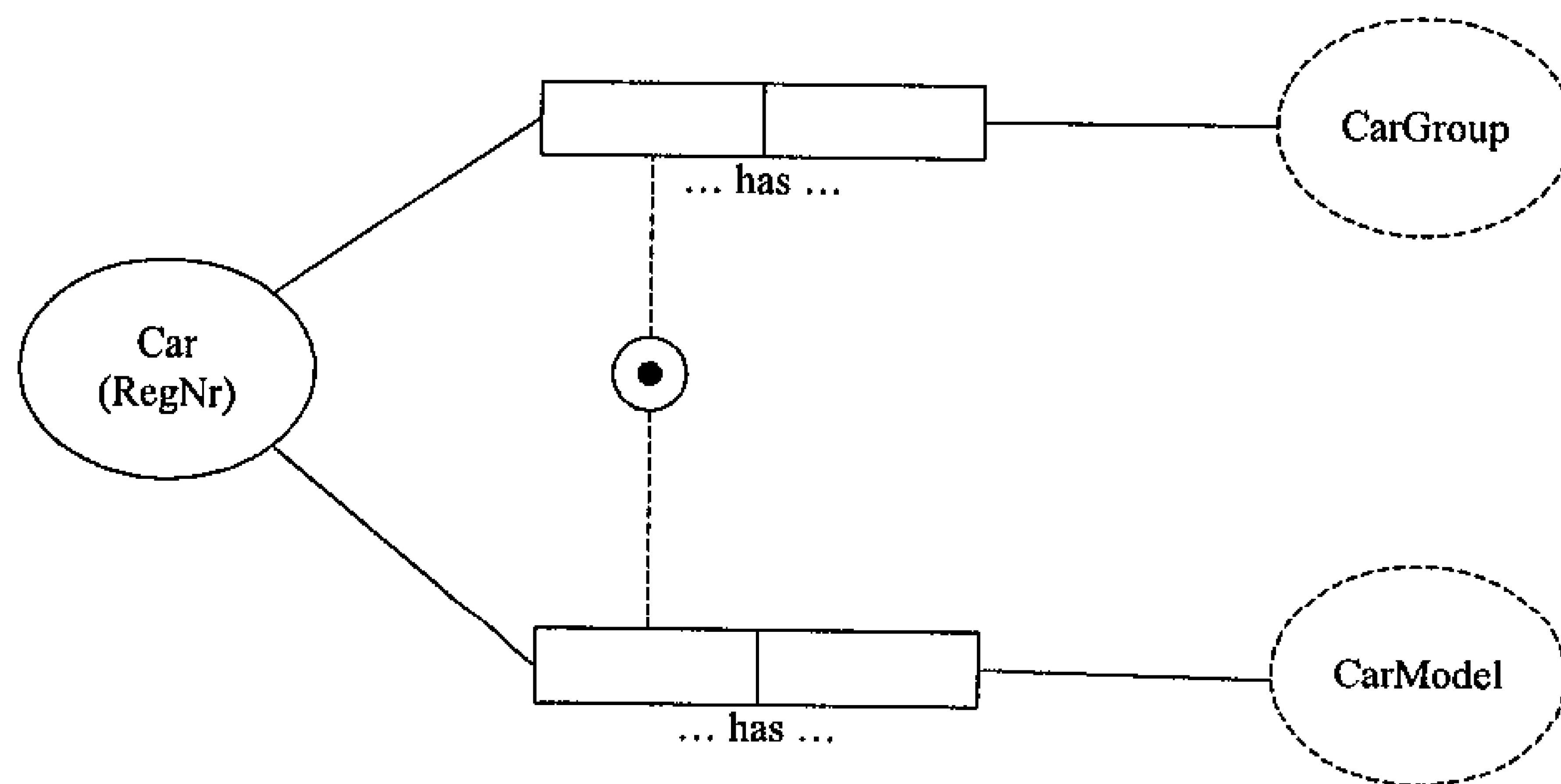


Figure 34b: Disjunctive mandatory role constraint

5.4.3 Uniqueness constraint

Cardinality in ORM is determined by the uniqueness of occurrences of a fact. A uniqueness constraint identifies a unique occurrence of an entity. Uniqueness constraints can be **internal** or **external**. **Internal uniqueness constraints** can be applied to one or more roles of a given fact type when every occurrence of the fact type is unique. An entity's uniqueness with respect to a relationship is represented in ORM by a double-headed arrow [Halpin, 2001]. The arrow shown over a predicate(s) in a fact type is called the uniqueness constraint of the fact type, and indicates that each data value entered in the role's fact column must be unique. The uniqueness constraint over a unary predicate indicates that all the entries in the role's fact column must be unique. For a binary predicate, the uniqueness constraint depicted over one predicate indicates that each data in the fact table for that particular predicate is unique, whereas duplicates are allowed for the other predicate which does not have a uniqueness constraint. Uniqueness constraints shown separately for both predicates indicate that each entry in the two role's fact column is unique. If the uniqueness constraint spans both predicates, this indicates that each row is unique. Examples of the three cases for a binary fact type are shown in figures 35a, 35b and 35c. The same rule for uniqueness can be extended to ternary and higher order relationships.

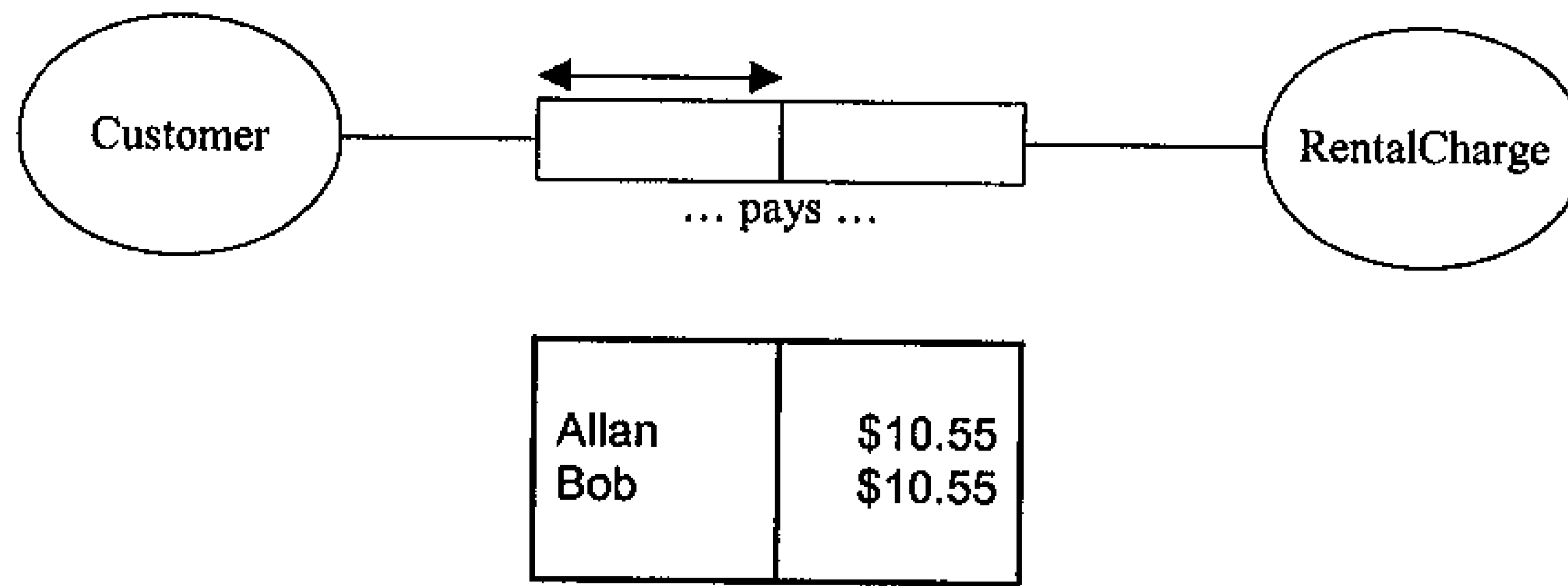


Figure 35a: Uniqueness on left predicate

The uniqueness constraint on the left-hand role of figure 35a indicates that every occurrence of Customer in the fact table is unique. This has the limitation that Customer names must be unique.

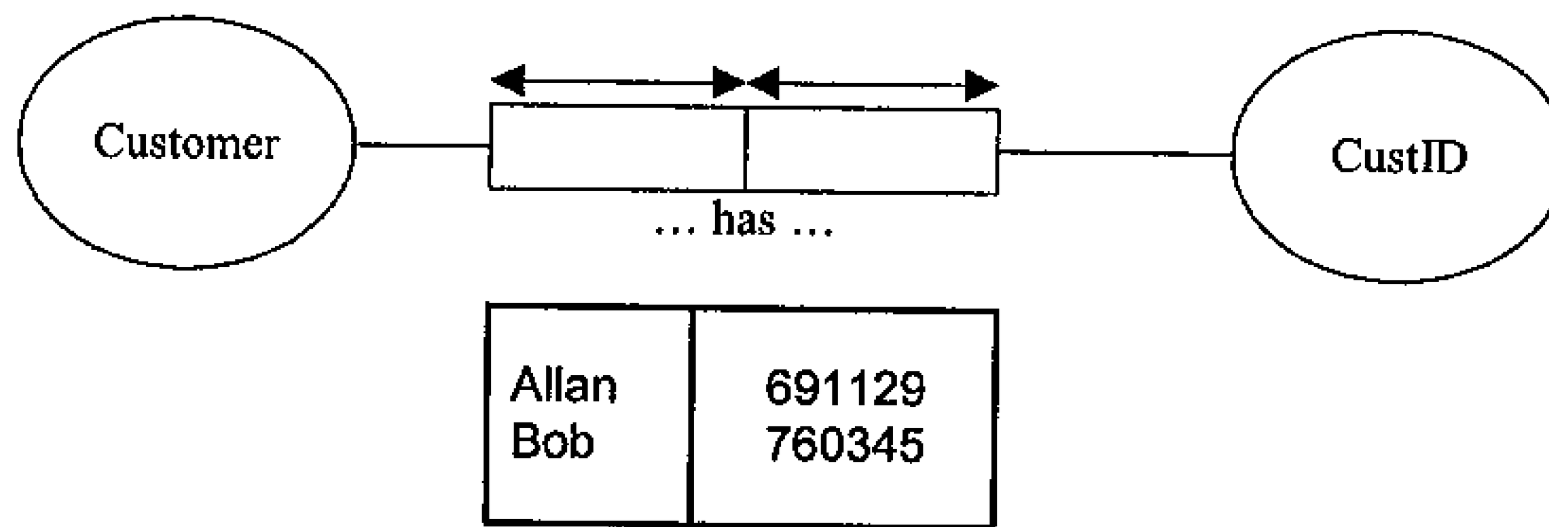


Figure 35b: Uniqueness shown separately on both predicates

The uniqueness constraint shown separately on both roles of figure 35b indicates that each entry for Customer and CustomerID is unique.

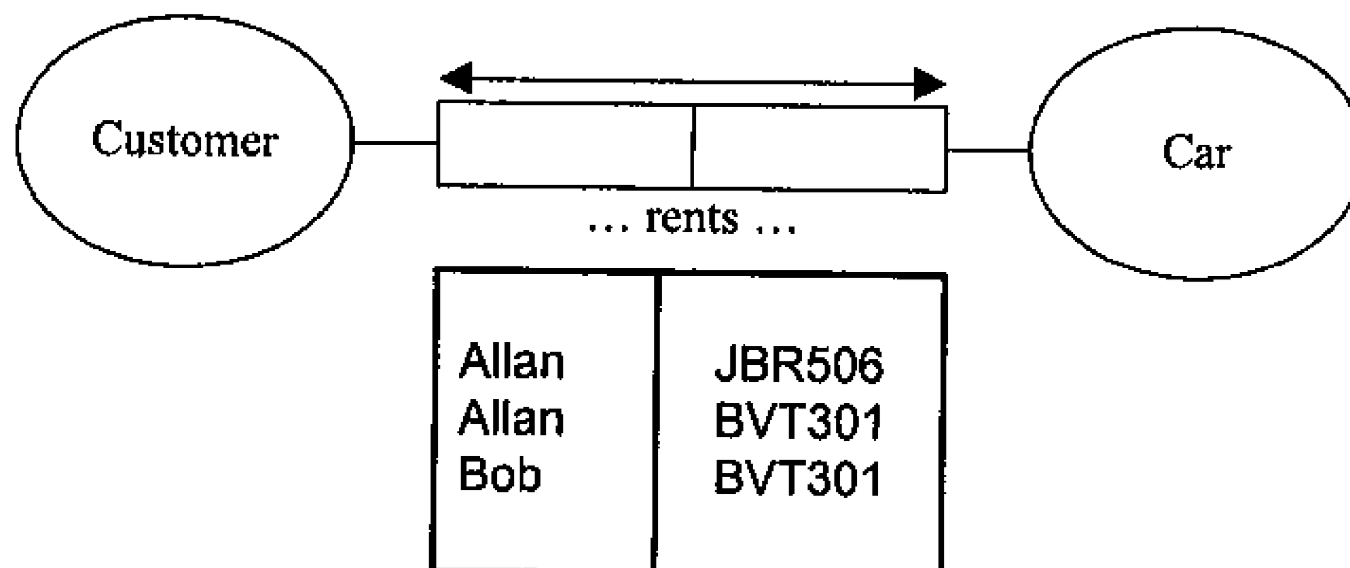


Figure 35c: Uniqueness shown spanning both predicates

The uniqueness constraint spanning both roles in figure 35c indicates that the same Customer can rent different Cars and the same Car can be rented by different Customers.

External uniqueness constraints can be applied to two or more roles in different fact types when the fact types share the same object types. The external uniqueness constraint depicted by the circled U in figure 36 indicates that any Customer on a specific date is given at most one Car.

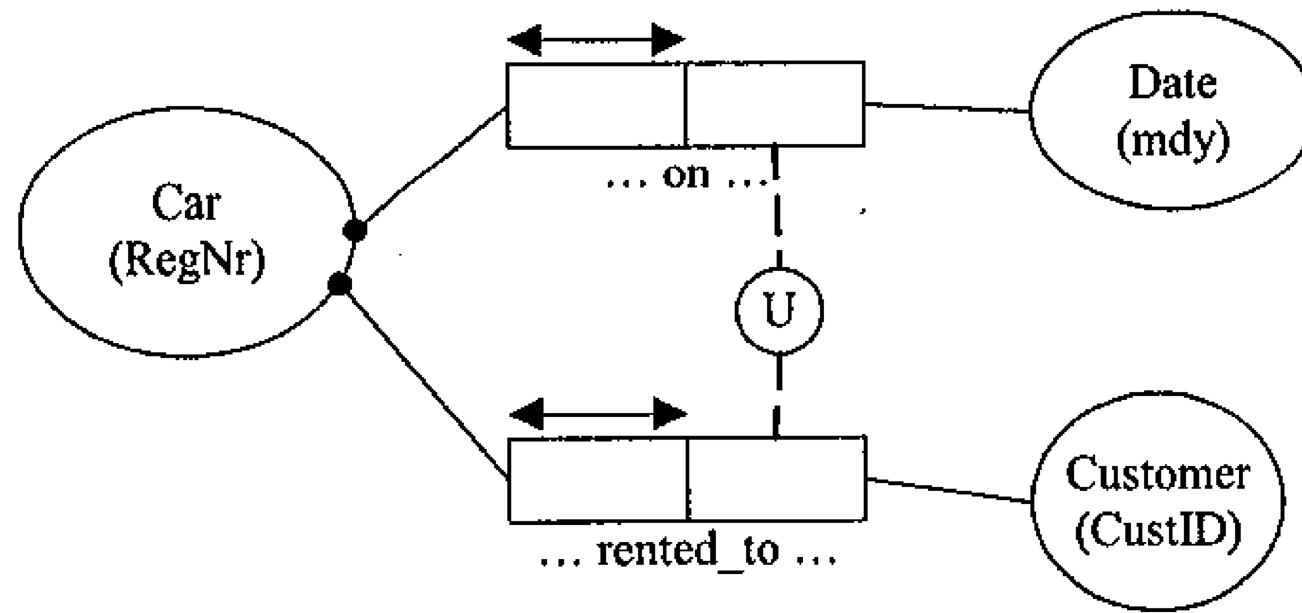


Figure 36: External Uniqueness Constraint

5.4.4 Frequency constraint

ORM frequency constraints are used to indicate the exact number or range of numbers of occurrences of the instance of the associated role. For example, consider the fact type, "A customer must have had at least 4 rentals in order to qualify as a Loyalty Scheme member". The ORM model captures this rule with the help of a frequency constraint as indicated in figure 37.

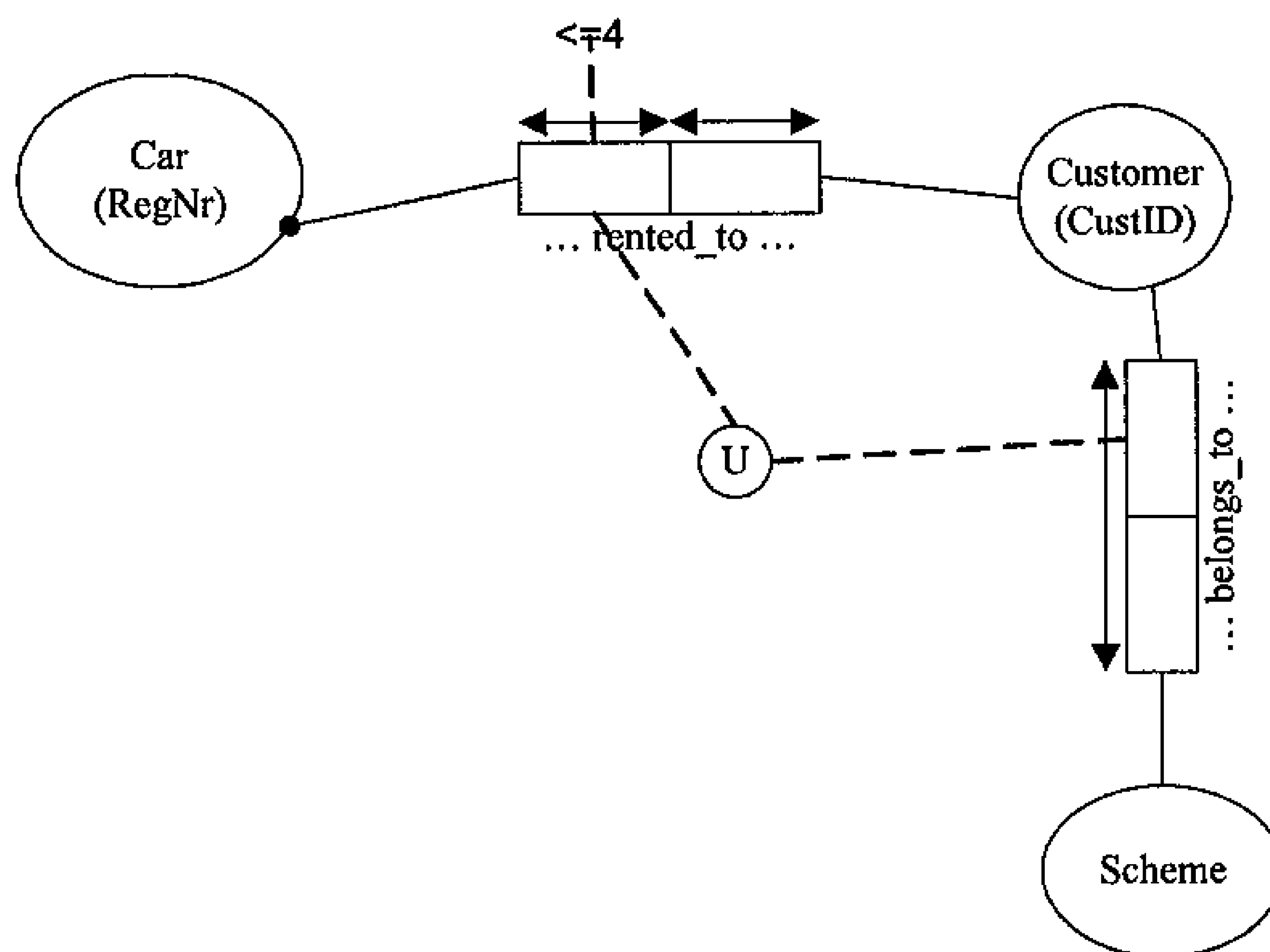


Figure 37: ORM frequency constraint

5.4.5 OR constraint

An ORM Or-constraint can be applied when an object type plays two or more role sequences. For example, a Customer can be an AdditionalDriver or AuthorizedDriver, but not both at the same time. ORM captures such a business rule with the help of the Or-constraint depicted in figure 38.

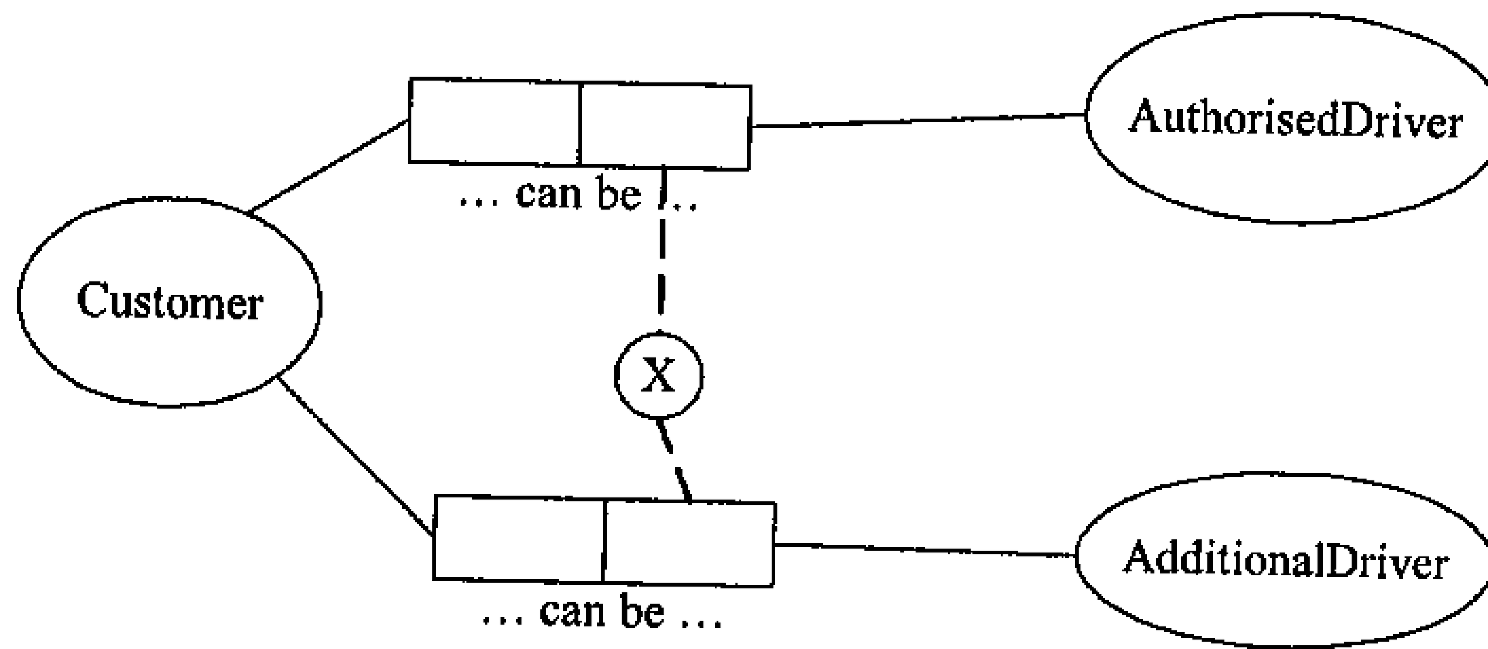


Figure 38: Or - Constraint in ORM

5.4.6 Subset constraints

Subset constraints can be used when two or more role sequences are played by the same object type and the roles are optional. For example, if there is a business rule which states that a branch owns a car only if it is returned to the branch, then this business rule is captured in ORM with the help of the subset constraint represented in figure 39.

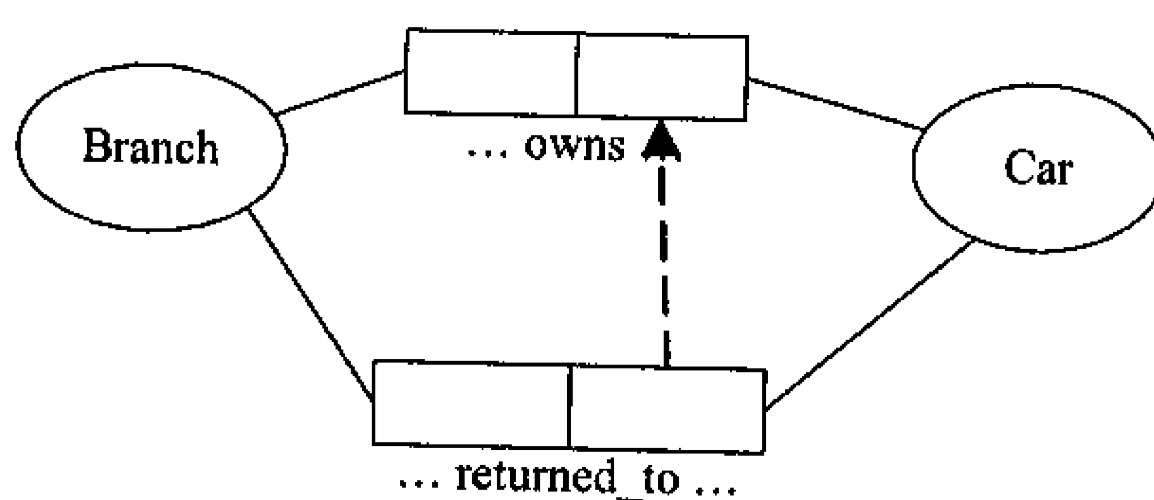


Figure 39: Subset constraint

5.4.7 Equality constraints

Equality constraints can be applied between two or more role sequences played by an object type where the population of the first role sequence must be equal to the population of the second role sequence. Figure 40 indicates that Customer is a member of Incentive Scheme only if he/she has 4 or more rentals. This business rule can be depicted in the ORM model using the equality constraint as shown in figure 40.

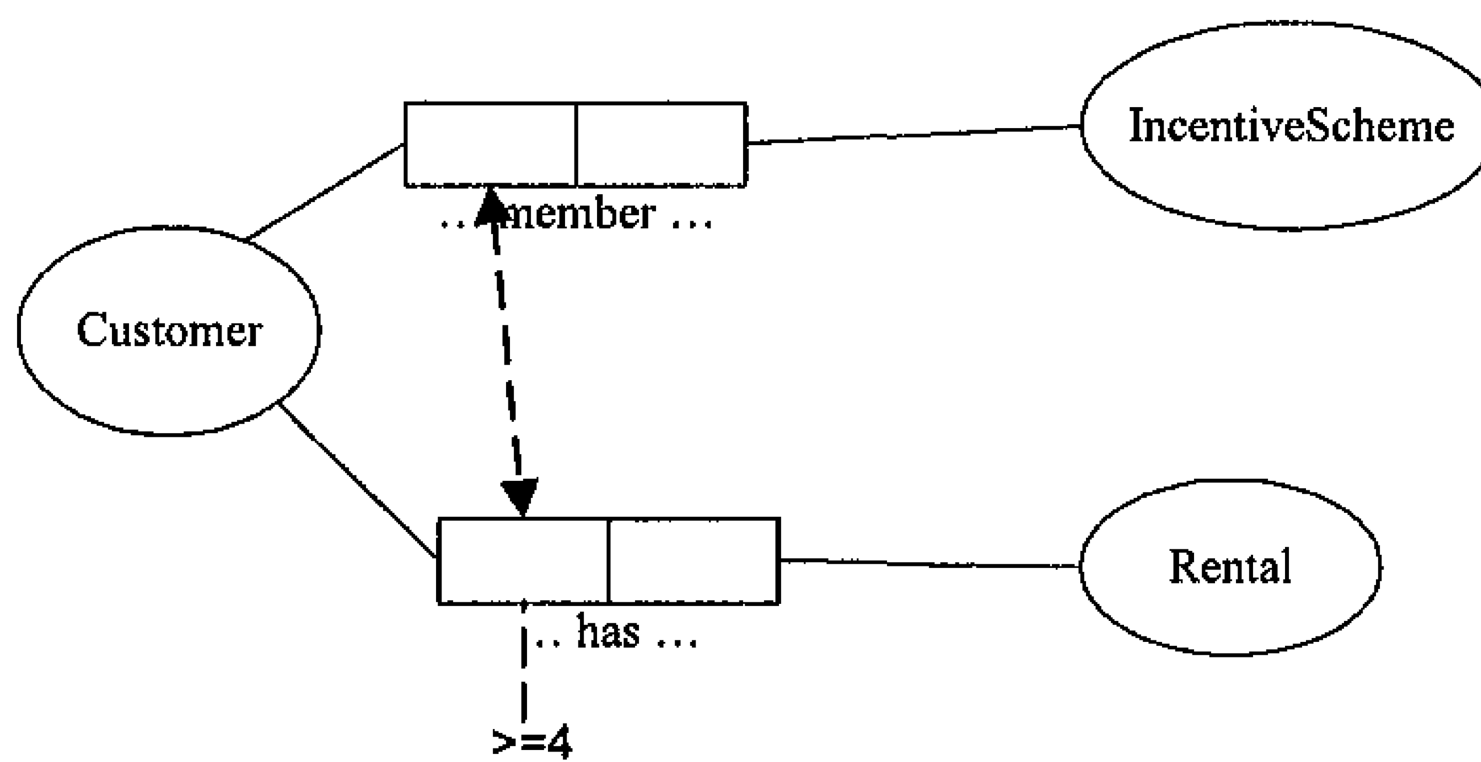


Figure 40: Equality constraints in ORM

5.4.8 Ring constraints

ORM supports the application of ring constraints to a pair of roles connected to the same object type. If the same object type plays at least two roles either directly or indirectly through a supertype, then the constraint becomes a ring. ORM allows ring constraints to be applied to roles and provides six types of ring constraints : antisymmetric (o_{ans}), asymmetric (o_{as}), acyclic (o_{ac}), irreflexive (o_{ir}), intransitive (o_{it}) and symmetric (o_{sym}).

5.5 Subtyping

ORM technique supports subtyping, where a subtype is an object type that is contained in another object type called the supertype. ORM supports single and multiple inheritance, where a subtype has more than one direct supertype. In ORM, a subtype inherits all the roles of its supertypes.

ORM subtypes are shown by means of arrows to their supertypes. ORM allows mutually exclusive subtypes to be depicted by means of a circled X connected to the relevant subtypes with dotted lines. Figure 41 indicates the representation of subtypes in ORM. In this example, Additional Drivers and Authorised Drivers are subtypes of Customers. Scheme Member has more than one supertype, supporting multiple inheritance.

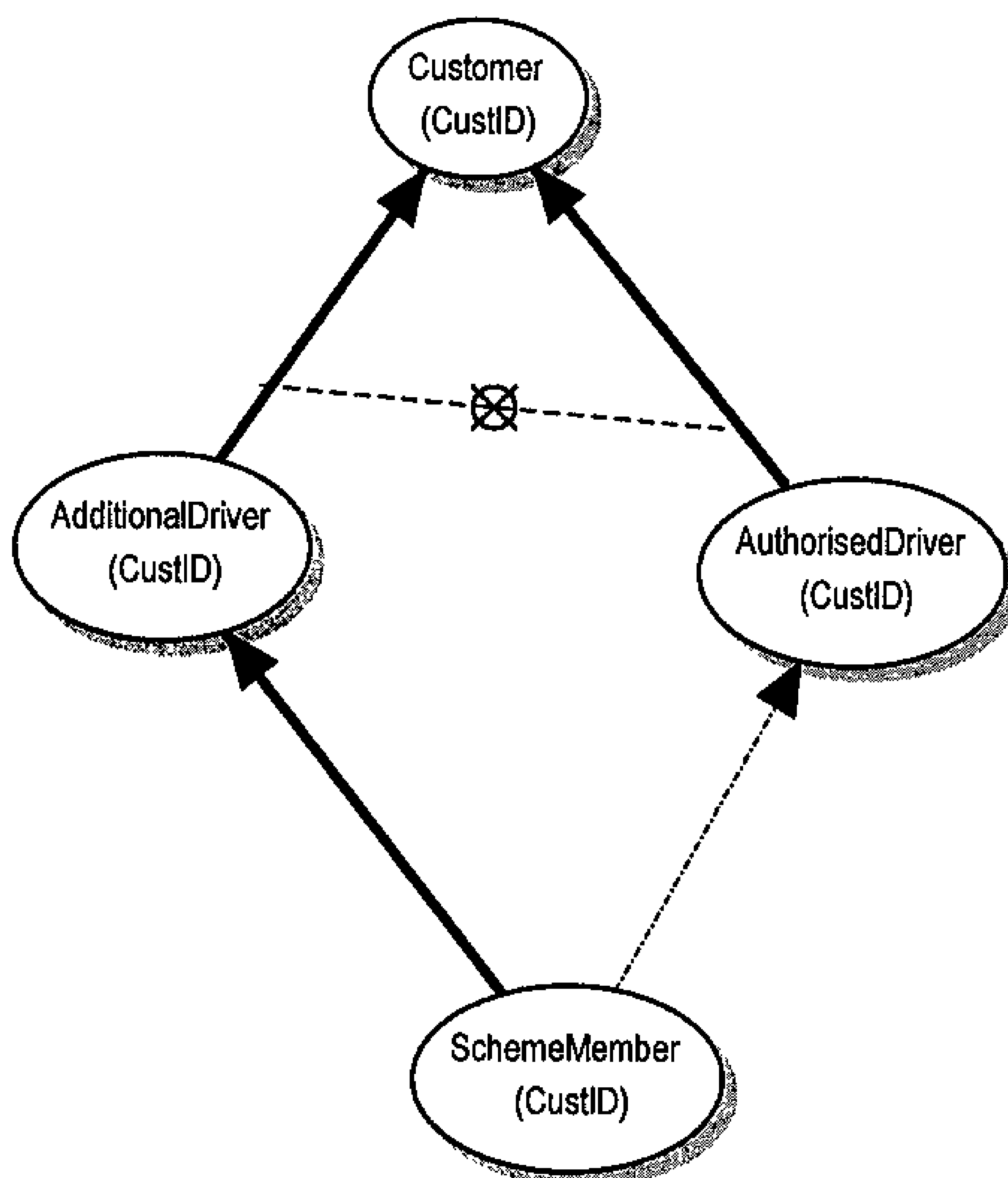


Figure 41: ORM subtypes

The data structures and constraints of ORM discussed in the previous sections are applied to generate a complete conceptual model of the system. If one decides to use UML for design purposes after generating the ORM schema, then this can be done by creating a method-free class diagram from the ORM schema. Once the ORM conceptual schema is built, it is necessary to generate a class diagram from the ORM model, in order to compare the models at the same level. The mapping rules from ORM to UML as discussed by Halpin [Halpin, 2001] are given in the following section.

5.6 Generating Classes from the ORM Model

Halpin [Halpin, 2001] gives the following transformation rules in order to transfor an ORM model into a class diagram. Object types and value types in the ORM model map to object classes and attributes in UML respectively, where relationships between object types in ORM are replaced by attributes in UML. Object cardinality represented as uniqueness constraint in ORM maps to class multiplicity. Binary relationships between object types are best mapped into binary associations between classes. It is at the discretion of the modeller to decide whether relationships between object types may be mapped as attributes or associations between UML classes. Objectified relationship types in ORM map to association classes in UML.

Internal uniqueness constraints in ORM map to a maximum multiplicity of 1 in UML. External uniqueness constraints in ORM map to textual constraints or qualified associations in UML. A simple mandatory role constraint in ORM maps to a minimum multiplicity of 1 in UML. A disjunctive mandatory role constraint in ORM maps to a textual constraint in UML. Frequency constraints in ORM map to multiplicity constraints in UML, value constraints in ORM map to enumeration or textual constraints in UML. Subset constraints in ORM map to either subset constraints or textual constraints in UML. Subtyping in ORM is directly mapped to a subtype in UML. Ring constraints and join constraints map to textual constraints in UML.

With the set of transformation rules given, it is possible to transform an ORM model to a class diagram.

5.7 Conclusion

ORM models data using facts, where a fact represents the objects contained in the application domain, how the objects are referenced by values and the object's relationships. This chapter describes the ORM technique, focusing on the referencing scheme of data as well as the constraints that are supported by the technique. ORM fact types may be populated with sample data in order to validate facts with domain experts. ORM facts may be verbalised both in the forward and reverse directions. The difference between reference types and fact types is identified, namely that reference types are used for primary referencing and all other relationship types are called fact types. ORM supports the representation of a collection of constraints, namely mandatory role, uniqueness, frequency, Or, subset, equality and ring constraints. Each of these is explained using illustrations. ORM subtypes and their benefits, such as the ability to declare a constraint that a specific role is played only by that subtype, are discussed. Finally, the transformation of the ORM model to a class diagram is discussed. This is necessary to fulfill the secondary objective of the research namely to determine whether the ORM model is suitable only for requirements analysis or can also be used to construct classes in order to build up the requirements model into the implementation model.

Chapter 6

Evaluation

6.1 Introduction

This chapter deals with the evaluation of the results obtained from testing the benchmark domain model class diagram derived via the Unified Process and the class diagram derived from the ORM conceptual schema. The testing was carried out on end users using the questionnaire provided in Appendix E. The evaluation was carried out from two perspectives, that of the end users and that of the author as the domain expert. The evaluation by the end users is now discussed. The objectives of the end user evaluation were the following:

- To identify the level of user satisfaction with the two conceptual class diagrams
- To find possible factors that influenced the results provided by the users
- To answer the research question, namely whether ORM can be proposed as an alternate technique to the use-case-driven UML technique.

In order to distinguish clearly between the two conceptual class diagrams, the class diagram derived by means of the Unified Process (see appendix B) will be referred to as the benchmark class diagram and the class diagram derived from the ORM schema (see appendix D) will be referred to as ORM-class diagram in this chapter. Collectively, they will be referred to as conceptual class diagrams.

The following procedure was used to test the two conceptual class diagrams. A group of twenty students enrolled for the course Software Engineering IV[1] at

[1] Software Engineering 1V is a course offered at Technikon Pretoria as part of the software development specialisation offered by the Information Technology department. The students who are enrolled for this course have a background of the development of and processes involved in software systems development.[1]

Technikon Pretoria was used as end users for testing the conceptual class diagrams. At this level, the students are familiar with the software development process and the importance of requirements modelling. Furthermore, specific notations for software development, such as the UML, are introduced at this level, where a student is expected to put his theoretical knowledge into practice on any software project. However, the testing of the two conceptual class diagrams was performed on this specific group of students in the initial lecture, before they were formally introduced to the UML notation.

The twenty students were divided into ten groups of two students each, as group work provides communication support among the end users, allowing them to exchange ideas and work on the same documents. The students were provided with the following documentation:

- The problem statement (case study in appendix A)
- The benchmark class diagram (appendix B)
- The ORM-class diagram (appendix D)
- Questionnaire (appendix E)
- Use cases (in appendix B)
- ORM facts (in appendix C)

Both conceptual class diagrams were developed to model the requirements of the selected case study. The use cases and ORM facts represent the requirements of the system. In order to understand and evaluate each question correctly, the requirements were given to the end users together with the two conceptual class diagrams. The collaboration diagrams and the ORM schema were made available for the students to view the complete process by which the two conceptual class diagrams were derived. The benchmark class diagram was evaluated first, followed by the ORM-class diagram.

The number of students used was too small to draw statistically significant conclusions from the results, but the test nevertheless provides some insight into the way that end users will experience the conceptual class diagrams and evaluate them. The implication of the author's evaluation is that if the ORM-

class diagram captures the requirements as effectively as the benchmark class diagram, then ORM can be proposed as an alternate technique for requirements modelling, and UML can be used for further design. The results obtained from the end users influence the author's evaluation because they must be satisfied with the ORM-class diagram before it can be proposed as an alternative to the use-case-driven UML technique.

The steps for mapping an ORM model to the UML class diagram can be found in section 5.6, and the actual mapping of the ORM model to a conceptual class diagram is shown in appendix D. The results of the questionnaire survey are summarised in a table in appendix E, with individual ratings for each question. The majority rating for each specific question provided by the end users is now considered in order to evaluate the two conceptual class diagrams.

6.2 Evaluation of the Questionnaire Results

Questions 1, 2, 3, 13, 14 and 15 are based on the completeness principle, listed in section 3.4. Questions 4, 5, 16 and 17 are based on the accuracy principle, 6, 7, 18 and 19 refer to the clarity principle, 8, 9, 20 and 21 refer to the simplicity principle, 10, 11, 22 and 23 are based on the uniqueness principle and finally 12 and 24 refer to the process by which the two conceptual class diagrams were derived.

6.2.1 Evaluation for completeness

A conceptual class diagram is said to be complete if it can model all the object structures and constraints that are relevant in the system. Regarding the completeness principle, the end users were more satisfied with the object structures and constraints that were captured by the benchmark class diagram than with those captured by the ORM-class diagram. They identified attributes such as BranchDriver, InspectionOfficer and FinancialClerk as being captured in the benchmark class diagram, but not in the ORM-class diagram. The classes DepotReception, BranchReception, PaymentDetails for Servicing, CustomerList and Carlist were also identified only in the benchmark class diagram. In terms of the completeness principle, therefore, the end-user evaluation is that the benchmark class diagram is more complete than the ORM-class diagram in capturing the requirements.

Having identified the end user's view of the completeness of the two conceptual class diagrams, it is now important to discuss the factor(s) that influenced the result from the author's point of view. The reason why the end users preferred the benchmark class diagram for completeness is that the benchmark class diagram captured not only the static aspects of the system, but also its behaviour by means of collaboration diagrams; as a result, more classes were identified. The ORM schema did not capture the behavioural aspects of the system; hence fewer classes and therefore fewer requirements are captured in the ORM-class diagram.

It must be noted that ORM does have special behavioural techniques that can work in conjunction with the elementary facts to capture behaviour. But none of these techniques are standardised yet, and therefore they are not used in this study. It is possible that if a behavioural technique is used to construct an ORM schema from which the ORM-class diagram is then generated, the ORM-class diagram might capture more requirements and hence be more complete.

6.2.2 Evaluation for accuracy

Overall, the end users expressed a satisfactory response regarding the one-to-one correspondence between a business rule and its assertion depicted in both conceptual class diagrams. However, they were not enthusiastic about the fact that a business rule cannot be read the same way in either of the conceptual class diagrams as they could be read in the problem statement. However, the end users were satisfied with the fact that there were no ambiguities in either the benchmark class diagram or the ORM-class diagram; both were regarded as accurate.

From the author's point of view, it must be noted that the end users were not familiar with the UML notation, and they had to study the meaning of the symbols (e.g. those used for aggregation and inheritance) used in both conceptual class diagrams before verifying the accuracy of the models. Once the meanings of the symbols used in the conceptual class diagrams had been interpreted, the end users could proceed with the test for accuracy. The end users could have easily verified the accuracy if the business rules had been depicted in the conceptual class diagrams in the same way as they read in the problem statement. It is found that the conceptual class diagrams add complexity from an end user's perspective, but also add more richness from the implementation point of view. The use of text in ORM promotes easy readability for an end user and thus increases the simplicity of the model, but at the same time behavioural aspects are lost. This is a case of a trade-off between completeness and simplicity.

6.2.3 Evaluation for clarity

The clarity of a conceptual class diagram is a measure of how clearly the requirements are depicted in it. The end users were satisfied with the clarity and dissatisfied with the verbalisation of business rules in both conceptual class diagrams. The implication of this end-user evaluation is that the

benchmark class diagram as well as the ORM-class diagram have good clarity with poor verbalisation.

From the author's point of view, the conceptual class diagrams became clearer to the end users after they had interpreted the meanings of the symbols used in them. The unsatisfactory response from the end users regarding the verbalisation of business rules in the conceptual class diagrams is due to of the fact that the requirements were not expressed in such a way that they could be read in simple sentences. It must also be noted that the problem was not due to the nature of the conceptual class diagrams alone, but also to the fact that the users were novices.

6.2.4 Evaluation for simplicity

Simplicity is a measure of the ease with which the conceptual class diagram can be interpreted. The end users could interpret the two conceptual class diagrams, with a moderate difficulty being presented by the need to interpret the symbols used. Otherwise, their response was positive regarding the simplicity of the models. This evaluation shows that the end user must have prior knowledge of the UML notation in order to interpret the conceptual class diagram.

In the author's experience, the major factor that affects simplicity is verbalisation. If the conceptual class diagrams provide easy verbalisations, then it becomes much simpler for the end user to interpret the conceptual class diagram. It must be noted that the ORM schema is simple because it allows for almost mechanical translation from the problem statement to the facts, without forcing the user to consider the interaction between objects. This apparent advantage in terms of simplicity unfortunately hides the fact that the interaction between objects is something that has to be dealt with sooner or later, and if this aspect is ignored it leads to incompleteness, as found in the ORM-class diagram.

6.2.5 Evaluation for uniqueness

The uniqueness principle states that no business rule may be duplicated in the conceptual class diagram. The end users were satisfied that no duplication was identified in either of the conceptual class diagrams and that they were concise. According to this evaluation, both the benchmark class diagram and the ORM-class diagram adhere to the uniqueness principle and are brief and comprehensive.

The positive response from the end users regarding the conciseness of the conceptual models is due to the presence of attributes in the models. Attributes are useful to condense a conceptual diagram, even though their presence does not facilitate the verbalisation of business rules in simple sentences. For example, consider a class "Branch" which has an attribute "Branchnumber". If an end user who is not familiar with the UML notation tries to interpret the class with the listed attribute, he cannot read the business rule as a simple statement that "Branch has Branchnumber" or "Branchnumber belongs to a Branch". In the author's experience, the expressiveness of a conceptual class diagram is affected by the presence of attributes in it, while the attributes become useful to condense the conceptual class diagram.

6.2.6 Evaluation for validation

Finally, questions 12 and 24 concern the processes by which the two conceptual class diagrams were derived. The end users were keen to study these processes. At this stage, they observed the collaboration diagrams and the ORM schema, thus understanding the two processes separately. The end users were of the opinion that iterating through the Unified Process is a more painstaking effort than writing simple facts from the problem statement and then modelling them. They also found that the fact tables depicted in the ORM Schema Design Process helps to validate the business rules correctly as well as to check the uniqueness of the facts. Once the processes were understood, the end users studied the conceptual class diagrams and came to the conclusion

that it was difficult to validate both conceptual models against the problem statement.

In the author's experience, both the conceptual class diagrams derived by means of the two different processes have poor validation facilities. The provision of fact tables in the ORM schema building process provides the support for validating business rules by populating the schema with multiple instances. The use of specific validation techniques in UML must be considered.

The end user's evaluation was useful to check whether the two conceptual class diagrams were complete, accurate, clear, simple, unique and validatable from the end-users point of view. The level of user satisfaction with the benchmark class diagram and the ORM-class diagram could be identified from the end users' evaluation.

The end-user evaluation having been completed, the following section compares the two conceptual class diagrams in order to decide whether ORM can be proposed as an alternate technique to the use-case-driven technique of UML for requirements modelling.

6.3 Evaluation of the Domain Model Class Diagram and the Class Diagram Derived from the ORM Model

Having developed the two conceptual models for the case study, the author has a deeper understanding of the problem statement and the user requirements than the end users. Therefore this discussion focuses on the author's comparison of the two conceptual models in order to determine whether, if the ORM-class diagram captures the requirements that were captured by the benchmark class diagram and is as effective in other aspects such as accuracy, clarity, simplicity, uniqueness and validation, this technique proves to be useful

for requirements modelling. The user satisfaction also has an effect on the author's evaluation.

It appears from the end users' evaluation for completeness that the benchmark class diagram provides more classes than the ORM-class diagram. This implies that the benchmark class diagram captures more requirements than the ORM-class diagram. The reason for the loss of requirements is the loss of behaviour in the ORM schema. The ORM schema did not model the behavioural aspects of the system, and therefore only entity classes were identified in this diagram. Although it is worth noting that the missing classes in the ORM-class diagram, namely DepotReception and BranchReception, are more like system service classes and do not deal with the information that is recorded in the system, these missing classes nevertheless possess functionality stated by the requirements that must be captured in the conceptual class diagram. Entity classes alone, as provided by the ORM technique, cannot capture these additional requirements. Consequently, the ORM classes would not be able to model the system with all the functionality specified by the requirements.

At this stage of the research, it is noted that the ORM-class diagram is not as complete as the benchmark class diagram. As regards accuracy, no ambiguities were found in the ORM-class diagram; this is also true of the benchmark class diagram. The assertions in the ORM-class diagram reflected its own business rule statement, as was the case in the benchmark class diagram, implying that the ORM-class diagram is as accurate as the benchmark class diagram. As regards the clarity of the ORM-class diagram when compared with the benchmark class diagram, the requirements were clearly depicted in both diagrams. However, verbalisation of business rules in the ORM-class diagram was poor, as it was in the benchmark class diagram. The ORM-class diagram is simple to understand for end users who are familiar with the UML notation. The same applies to the benchmark class diagram, implying that an end user finds it difficult to interpret the conceptual class diagrams without prior knowledge of the notation. The ORM-class diagram did not duplicate the requirements, and it was as concise as the benchmark class diagram. Finally,

neither diagram could be validated against the problem statement because sample populations could not be used for validation purpose in the conceptual class diagrams.

The result of this evaluation is that for the selected case study, the ORM-class diagram does not represent the requirements as completely as the benchmark class diagram; as for the other principles, the ORM-class diagram compares well with the benchmark class diagram. Completeness is one of the most essential principles of a conceptual model, because a conceptual model will not fulfil its purpose if the requirements cannot be captured completely. According to this study the ORM-class diagram was found not to be as complete as the benchmark class diagram.

6.4 Conclusion

The aim of the end-user evaluation was to identify the user satisfaction with the two conceptual class diagrams. The purpose of the author's evaluation was to identify factors that influenced the findings of the end users and to compare the two conceptual class diagrams in order to provide an answer to the research question, namely whether ORM could be proposed for requirements modelling as an alternate technique to the use-case-driven technique of UML.

The end user's evaluation led to the identification of missing classes in the ORM-class diagram, which in turn led to the conclusion that the ORM-class diagram did not capture the requirements as completely as the benchmark class diagram.

Regarding the other principles, namely accuracy, clarity, simplicity, uniqueness and validation, the benchmark class diagram and the ORM-class diagram compare favourably with each other. The favourable factors that can be highlighted here are that both the conceptual class diagrams are simple and concise with no duplicated business rules. However, the unfavourable factors that can be highlighted here are that the interpretation of the conceptual class

diagrams required prior study of the notation used and that it was not possible to validate and verbalise business rules in either of the two conceptual class diagrams. It would have been possible for the end users' reaction to the ORM-class diagram to be influenced by their familiarity with the problem statement [Olivier, 1999]. However, the fact that the ORM-class diagram was not found significantly clearer and simpler indicates that the students were not influenced by the previous exposure to the problem, or that the benchmark class diagram is indeed not less clear and simple than the ORM-class diagram.

This is where the ORM schema has the advantage that it is possible to read the business rules in the schema. But when the conceptual class diagram is derived from the ORM schema, this verbalisation is lost. The same applies to validating the facts in the ORM schema with fact tables; this advantage is also lost when the conceptual class diagram is derived from the schema.

The study was carried out up to the level of the domain model class diagram without considering the dynamics of the ORM model. This study approached the study of requirements modelling from an object-oriented and data perspective using separate processes. It identified the favourable and unfavourable aspects of the conceptual class diagrams derived using the Unified Process and the ORM Schema Design Process. Taking into consideration that The study concludes that ORM cannot be used as an alternate technique for conceptual modelling up to the domain model class diagram (benchmark) level.

Chapter 7

Conclusions and Recommendations

7.1 Introduction

The objective of this dissertation was to investigate the use of ORM as a conceptual modelling technique with the UML domain model class diagram as benchmark. The approach followed was to compare the benchmark class diagram with the class diagram derived from the ORM schema (ORM-class diagram) with the aim of proposing ORM as an alternate conceptual modelling technique to the use-case-driven technique of UML.

This chapter discusses the main conclusions of the study and identifies possible aspects that can be refined in the study as well as aspects that need further research.

7.2 Conclusions

The study not only provided an in-depth understanding of requirements modelling from two perspectives, namely object modelling and data modelling, but also the separate processes used to derive the requirements models, namely the Unified Process, the ORM Schema Design Process and the ORM-UML class diagram mapping process. Sets of criteria that can be used to evaluate a conceptual model were studied from different perspectives and integrated into one set of principles. From this integrated set, those principles that could be tested on end users were identified and used in the end-user evaluation. The conclusions drawn from the study help a novice software system modeller to choose the process and the techniques needed for effective requirements modelling of the system.

At this stage of the research it must be noted that the benchmark class diagram was the result of the initial iteration. With additional iterations, more classes may be added or classes may be refined by combining existing classes. It is possible to refine the ORM-class diagram by behavioural techniques that are designed to work in conjunction with ORM's fact-based techniques as mentioned in section 6.2.1 of chapter 6, where initially the essential processes (services) are defined and linked to elementary facts. This is a possible refinement that can be added to this research. Based on what has been done in this study, the following conclusions and recommendations are drawn up.

- ORM cannot be proposed as an alternate technique for the use-case-driven technique of UML, which is used for requirements modelling, because the class diagram that was derived from the ORM schema failed to capture the requirements that were captured by the benchmark class diagram derived using the Unified Process. The dissatisfaction among the end users regarding the completeness of the ORM-class diagram when compared with the benchmark class diagram supports this conclusion.

  It is recommended that behavioural techniques be used in conjunction with the different stages of the ORM Schema design process to capture the behavioural aspects of the system. If this is done effectively, then the ORM Schema Design and Mapping Processes will identify service classes together with the entity classes. The implication is that in such a situation, ORM would prove to be useful for requirements modelling and the subsequent generation of a more complete conceptual class diagram from the schema. The ORM schema could then be used to compensate for the dissatisfaction among the end users about the poor verbalisation and validation of the conceptual class diagrams.
- Several authors, e.g. Halpin [Halpin, 2001], Hay [Hay, 1999] and Post [Post 1999], refer to a domain model class diagram as a "data model". As the study shows, the domain model class diagram is obtained by means of use-

case modelling and dynamic modelling, where not only the "data" but also the "behaviour" of the system is captured. Therefore it is inappropriate to refer to a domain model class diagram as a "data model". It is recommended that a domain model class diagram be referred to as an "object model" instead of a "data model".

- As regards the Unified Process, the author found that writing use cases for the case study followed by the dynamic modelling using collaboration diagrams was time-consuming and tedious. The use cases had to be refined many times during the process of dynamic modelling in order to arrive at the use-cases given in appendix B. On the other hand, writing elementary facts and then modelling the facts together with fact populations using diagrams that were intuitive was a more natural way of requirements modelling, and the process was less time-consuming. Every step involved in the ORM schema design process was a mechanical process, and at every stage the fact table provided a way of testing the correctness and uniqueness of facts that were modelled. The ORM-UML Mapping Process was straightforward and involved less mental strain. Furthermore, in the author's experience the Unified Process is more mentally strenuous than the ORM Schema Design and Mapping Processes because the former requires more modelling through behaviour capture.

Having discussed the major conclusions from the research, it is now important to consider the aspects that need further research.

7.3 Further Research

It was stated clearly that if ORM could capture the behaviour of the system, then the ORM-class diagram could capture requirements more comprehensively. This capacity to capture behavioural aspects needs further research. At this stage there is no standard dynamic modelling technique associated with ORM. One way of capturing the dynamics in ORM is to use task structures; other dynamic modelling techniques are BizTalk, the DEMO approach and the CAP theory of Jan Dietz, where behavioural modelling is done in stages, starting with the conceptual and ending with the logical/physical stage [Halpin, 2002].

The case study used for the research is typical of many information systems. In case studies with more behavioural aspects, the difference found between the two conceptual class diagrams would be even clearer. Other case types with more behavioural aspects than the one used in the study need to be investigated in order to determine ORM's capacity to deal with behavioural aspects in greater detail.

Concerning the principles identified for evaluating a conceptual model, it was found that not all the identified principles could be tested in the conceptual models. For a more complete evaluation of the conceptual class diagrams, further investigation to determine ways of testing all of the listed principles is needed.

The study is thus concluded with an answer to the research question, possible refinements that can be made to the study as well as possibilities of future research.

Bibliography

1. ALHIR, S., 1998, *UML in a Nutshell: A Desktop Quick Reference,* O'Reilly & Associates, Inc.
2. PODEHL G., AND ARNOLD, F., 1999, Best of Both Worlds- A Mapping from EXPRESS-G to UML. *In:* J. BEZIVIN AND P. MULLER eds. *The unified modeling language: UML '98: beyond the notation: First International Workshop,* June 3-4, 1998, Mulhouse, France. Berlin, New York: Springer-Verlag, LNCS, 1618.
3. BAHRAMI, A., 1999. *Object Oriented Systems Development.* McGraw Hill.
4. BECKER, S. A., 2000, Building a better data model. *The Journal of Conceptual Modeling* [online]. Available from: http://www.tdan.com/ [Accessed on 09 June 2000].
5. BECKER, S., A., 2001. Conceptual data modeling in an object-orirnted process. *The Journal of Conceptual Modeling,* vol. 18.
6. BENNETT, S., SKELTON, J., AND LUNN, K., 2001. *Schaum's Outline of UML.* UK: McGraw-Hill International.
7. BEZIVIN, J. AND MULLER, P., 1999. UML: The Birth and Rise of a Standard Modeling Notation. *In:* J. BEZIVIN AND P. MULLER eds. *The unified modeling language: UML '98: beyond the notation: First International Workshop,* June 3-4, 1998, Mulhouse, France. Berlin, New York: Springer-Verlag, LNCS, vol 1618, pp. 1-19.
8. BOOCH, G., RUMBAUGH, J. AND JACOBSON, I., 1999a. *The Unified Modeling Language User Guide.* USA: Addison Wesley Longman.
9. BOOCH, G., RUMBAUGH, J. AND JACOBSON, I., 1999b. *The Unified Software Development Process.* Massachusetts: Addison Wesley Longman.
10. CLAXTON, J. C., MCDOUGALL, P. A., 2000. Measuring the quality of models. *The Data Administration Newsletter* [online]. Available from: http://www.tdan.com/i014ht03.htm [Accessed 06 November 2000].
11. DEITEL, H. M., AND DEITEL, P. J., 2001. *C++ How to Program.* 3rd ed. New Jersey: Prentice-Hall.
12. ERIKSSON, H., PENKER, M., 1998. *UML Toolkit.* Canada: John Wiley & Sons.

13. EVANS, A., 1999. Reasoning with UML class diagrams. *In:* R.FRANCE AND B. RUMPE eds. *WIFT'98 Proceedings of IEEE,* 20-23 October Florida. Springer-Verlag, LNCS 1723.
14. FISHER, R. AND SCHMIDT, B., 1998. The Language of Data Modeling. *DM Review* [online]. Available from: http://www.dmreview.com/ [Accessed on 23 March 2001].
15. FOWLER, M., 1997. *Analysis Patterns,* Addison-Wesley.
16. FRANCE, R., EVANS, A., AND LANO, K., 1999. UML as a formal modeling notation *In:* J. BEZIVIN AND P. MULLER eds. *The unified modeling language: UML '98: beyond the notation: First International Workshop,* June 3-4, 1998, Mulhouse, France. Berlin, New York: Springer-Verlag, LNCS, vol 1618.
17. GOGOLLA, M., RICHTERS, M., 1999. Transformation Rules for UML Class Diagrams. *In:* J. BEZIVIN AND P. MULLER eds. *The unified modeling language: UML '98: beyond the notation: First International Workshop,* June 3-4, 1998, Mulhouse, France. Berlin, New York: Springer-Verlag, LNCS, vol 1618. pp. 92-105.
18. GULLA, J. A., 2000, A general explanation component for conceptual modeling in CASE environments. *ACM transctions on Information Systems,* Volume 14, Issue 3.
19. HALPIN, T.A., AND BLOESH, A.C., 1999. Data modeling in UML and ORM: a comparison. *Journal of Database Management,* vol.10, USA: Idea Group Publishing Co.
20. HALPIN, T.A., 2001. *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design.* Morgan Kaufmann Publishers.
21. HALPIN, T.A., (TessryHa@microsoft.com). (5 February 2002). RE>> *MSc Research.* E-mail to M. John (manjuj@techpta.ac.za).
22. HAY, D., 1999. A Comparison of Data Modeling Techniques. *The Database Newsletter,* vol. 23, Number 3, Essential Strategies, Inc.
23. HOFSTEDE, TER A.H.M., PROPER, H. A., AND VAN DER WEIDE TH. P., 1997. Data Modeling in Complex Application Domains. *In:* P. LOUCOPOULOS, ed. *Fourth International Conference on Advanced Information Systems Engineering,* Lecture Notes in Computer Science, UK: Springer-Verlag, vol. 593, pp 364-377.
24. JURIC, R., 1998. The UML Rules. *Software Engineering Notes,* vol.23 no.1, pp

92-97.

25. OLIVIER, M. S., 1999. *Information Technology Research, A Practical Guide.* Johannesburg.
26. OMG-UML 1.3 draft. OMG-UML Revision Task Force website [online]. Available from: http://uml.systemhouse.mci.com/ [Accessed on 12 February 2001].
27. PAIGE, R.F., OSTROFF, J.S., AND BROOKE, P.J., 2000. Principles for modeling language design, *Information and Software Technology,* 42(2000) 665 - 675, Elsevier Science, Canada.
28. POST, G.V., 1999. *Database Management Systems,* USA: McGraw Hill.
29. RAMESH, V., PARSONS, J., AND BROWNE, G. J., 1999, What is the role of cognition in conceptual modeling?. *In:* P.P. CHEN et al. eds. *International conference on conceptual modeling,* Springer-Verlag, LNCS 1565, pp. 272 – 280.
30. ROB, P., CORONEL, C., 2000, *Database Systems Design, Implementation & Management.* USA: Course Technology.
31. RUMBAUGH, J., BLAHA, M., PREMERLANI, W., 1994. *Object-oriented modeling and design,* Prentice Hall.
32. SHELLY, G. B., CASHMAN, T. J., AND ROSENBLATT, H. J., 2001. *Systems Analysis and Design.* 4ed. USA: Thomson Learning.
33. SIMSION, G., 1994. *Data Modeling Essentials.* International Thomson Publishing.
34. STURM, J., 1999. *VB6 UML Design and Development.* Canada: Wrox Press Ltd.
35. UML Partners. 1997. UML summary, version 1.1, Available from www.omg.org [Accessed on 03 March 2001].
36. WHITTEN, J. L., BENTLEY, L. D., DITTMAN, K. C., 2000. *Systems Analysis and Design Methods.* 5th ed. New York : McGraw Hill Companies.

Appendix A

Case Study: EU-Rent Car Rentals

Acknowledgements

A project like this could not have been undertaken without the support of Model Systems Ltd., who provided me with the EU-Rent car rental case study. I was also fortunate to have the support and assistance of practitioners at the Business Rules Group, since the case studied has the typical characteristic of complex business rules - which makes it all the more suitable to satisfy the main objective of this study.

Introduction

The case study is first modelled using UML and then with ORM. Appendix B features the domain model class diagram derived using the Unified Process. Appendix C features the ORM conceptual model and Appendix D consists of the class diagram derived from the ORM model.

Problem statement

EU-Rent Car Rentals

EU-Rent is a car rental company owned by EU-Corporation. It is one of three businesses, the other two being hotels and an airline, all of which have their

own business and IT systems, but share the customer base. Many of the car rental customers also fly with EU-Fly and stay at EU-Stay hotels.

EU-Rent business

EU-Rent has 1000 branches in several countries. At each branch, cars (classified by car group) are available for rental. Each branch has one manager and booking clerks who handle rentals.

Rentals

Most cars are reserved in advance; the rental period and the car group are specified at the time of reservation. EU-Rent will also accept immediate ("walk-in") rentals, if cars are available.

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

Returns

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

Servicing

EU-Rent also has service depots, each serving several branches. Cars may be booked for maintenance at any time, provided that the service depot has capacity on the day in question. For simplicity, only one booking per car per day is allowed. A rental or service may cover several days.

Customers

A customer can have several reservations, but only one car rented at a time. EU-Rent keeps records of customers, their rentals and bad experiences such as late return, problems with payment and damage to cars. This information is used to decide whether to approve a rental.

EU-Rent Business Rules

External constraints

- Each driver authorised to drive the car during a rental must have a valid driver's licence.
- Each driver authorised to drive the car during a rental must be insured to the level required by the law of each country that may be visited during the rental.
- Rented cars must meet local legal requirements regarding technical condition and emissions for each country that may be visited during the rental.
- Local tax must be collected (at the drop-off location) on the rental charge.

Acceptance of reservations

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

Car allocation for advance reservations

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

- Only cars that are physically present in EU-Rent branches may be assigned.
- If a specific model has been requested, a car of that model should be assigned if one is available. Otherwise, a car in the same group as the requested model should be assigned.
- If no specific model has been requested, any car in the requested group may be assigned.
- The end date of the rental must be before any scheduled booking of the assigned car for maintenance or transfer.
- After all assignments within a group have been made, 10% of the group quota for the branch (or all the remaining cars in the group, whichever number is lower) must be reserved for the next day's walk-in rentals. Surplus capacity may be used for upgrades.

- If there are not sufficient cars in a group to meet demand, a one-group free upgrade may be given (i.e. a car of the next higher group may be assigned at the same rental rate) if there is capacity
- Customers in the loyalty incentive scheme have priority for free upgrades.

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

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

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

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

Walk-in rentals

- The end date of the rental must be before any scheduled booking of the assigned car for maintenance or transfer.
- If there are several available cars of the model or group requested, the one with the lowest mileage should be allocated.

Handover

- Each driver authorised to drive the car during a rental must be over 25 and have held a driver's licence for at least one year.
- The credit card used to guarantee a rental must belong to one of the authorised drivers, and this driver must sign the rental contract. Other drivers must sign an 'additional drivers authorisation' form.
- The driver who signs the rental agreement must not currently have a EU-Rent car on rental.
- Before releasing the car, a credit reservation equivalent to the estimated rental cost must be made against the guaranteeing credit card.
- The car must not be handed over to a driver who appears to be under the influence of alcohol or drugs.
- The driver must be physically able to drive the car safely - must not be too tall, too short or too fat; if disabled, must be able to operate the controls.
- The car must have been prepared -- cleaned, full tank of fuel, oil and water topped up, tires properly inflated.
- The car must have been checked for roadworthiness - tyre tread depth, brake pedal and hand brake lever travel, lights, exhaust leaks, windscreen wipers.

No-shows

- If an assigned car has not been picked up within 90 minutes after the scheduled pick-up time, it may be released for walk-in rental, unless the rental has been guaranteed by credit card.
- If a rental has been guaranteed by credit card and the car has not been picked up by the end of the scheduled pick-up day, one day's rental is charged to the credit card and the car is released for use the following day.

Return from rental

- At the end of a rental, the customer may pay by cash, or by a credit card other than the one used to guarantee the rental.
- If a car is returned to a location other than the agreed drop-off branch, a drop-off penalty is charged.
- The car must be checked for wear (brakes, lights, tyres, exhaust, wipers etc.) and damage, and repairs scheduled if necessary.
- If the car has been damaged during the rental and the customer is liable, the customer's credit card company must be notified of a pending charge.

Early returns

- If a car is returned early, the rental charge is calculated at the rate appropriate to the actual period of rental (e.g. daily rate rather than weekly).

Late returns

- If the car is returned late, an hourly charge is made up to 6 hours' delay; after 6 hours a whole day is charged.

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

Car maintenance & repairs

- Each car must be serviced every three months or 10,000 kilometres, whichever occurs first.
- If there is a shortage of cars for rental, routine maintenance may be delayed by up to 10% of the time or distance interval (whichever was the basis for scheduling maintenance) to meet rental demand.
- Cars needing repairs (other than minor body scratches and dents) must not be used for rentals.

Car purchase and sale

- Only cars on the authorised list can be purchased.
- Cars are to be sold when they are one year old or have clocked 40,000 kilometres, whichever occurs first.

Car ownership

- A branch cannot refuse to accept a drop-off of a EU-Rent car, even if a one-way rental has not been authorised.

- When a car is dropped off at a branch other than the pick-up branch, the ownership of the car (and, hence, responsibility for it) switches to the drop-off branch when the car is dropped off.
- When a transfer of a car is arranged between branches, ownership switches to the 'receiving' branch when the car is picked up.
- In each car group, if a branch accumulates cars to take it more than 10% over its quota, it must reduce the number back to within 10% of quota by transferring cars to other branches or selling some cars.
- In each car group, if a branch loses cars to take it more than 10% below its quota, it must increase the number back to within 10% of quota by transferring cars from other branches or buying some cars.

Loyalty incentive scheme

- To join the loyalty incentive scheme, a customer must have made 4 rentals within a year.
- Each paid rental in the scheme (including the 4 qualifying rentals) earns points that may be used to buy 'free rentals.'
- Only the basic rental cost of a free rental can be bought with points. Extras, such as insurance, fuel and taxes must be paid by cash or credit card.
- A free rental must be booked at least fourteen days before the pick-up date.
- Free rentals do not earn points.
- Unused points expire three years after the end of the year in which they were earned.

Examples of "rules for running the business"

- Performance targets must be set for each branch - number of rentals, utilisation of cars, turnover, profit, customer satisfaction etc.
- Where performance requirements conflict (e.g. profit versus customer satisfaction, when a customer requests a reduction in charges after an unsatisfactory rental) heuristics must be provided to guide branch staff.
- Performance data must be captured.

If performance targets are not met, control action must be taken. Control action may include:

- changing the resources at branches (e.g. numbers of cars, quotas of cars within each group, number of staff),
- changing responsibilities (e.g. having transfers of cars managed by groups of branches, rather than by negotiation between individual branch managers),
- changing operational guidance (e.g. what proportion of cars should be kept for walk-in rentals), but not external constraints (e.g. legal requirements) or company policies (e.g. rentals must be guaranteed by a credit card, a customer may have only one car at a time).

Appendix B (UML Domain Model Class Diagram)

Requirements Analysis of EU-Car Rental system using UML

Requirements Analysis

Using the Unified Process, the initial step in requirements analysis is to define the use cases. Use-cases are explained in Chapter 4. An actor communicates with the system. Relationships between actors and use cases are depicted in a use case diagram by arrows. Use-case modelling helps to identify the functionality that is expected from the system as well as the external actors who use the system. According to Eriksson [1998], use cases must have the following characteristics:

- A use case must be initiated by an actor.
- A use case must provide value to an actor.
- A use case must be complete.

Identification of Actors in the Car Rental System

The initial task in developing a use-case diagram is to identify the actors who use the system. In the case study, the following actors are identified:

1. Customer
2. Booking clerk
3. Depot manager
4. Branch manager
5. Branch driver
6. Inspection officer
7. Financial clerk
8. Buyer

Using the identified actors, a use-case diagram is drawn as shown in figure 42. The use cases are documented in tables 2-6.

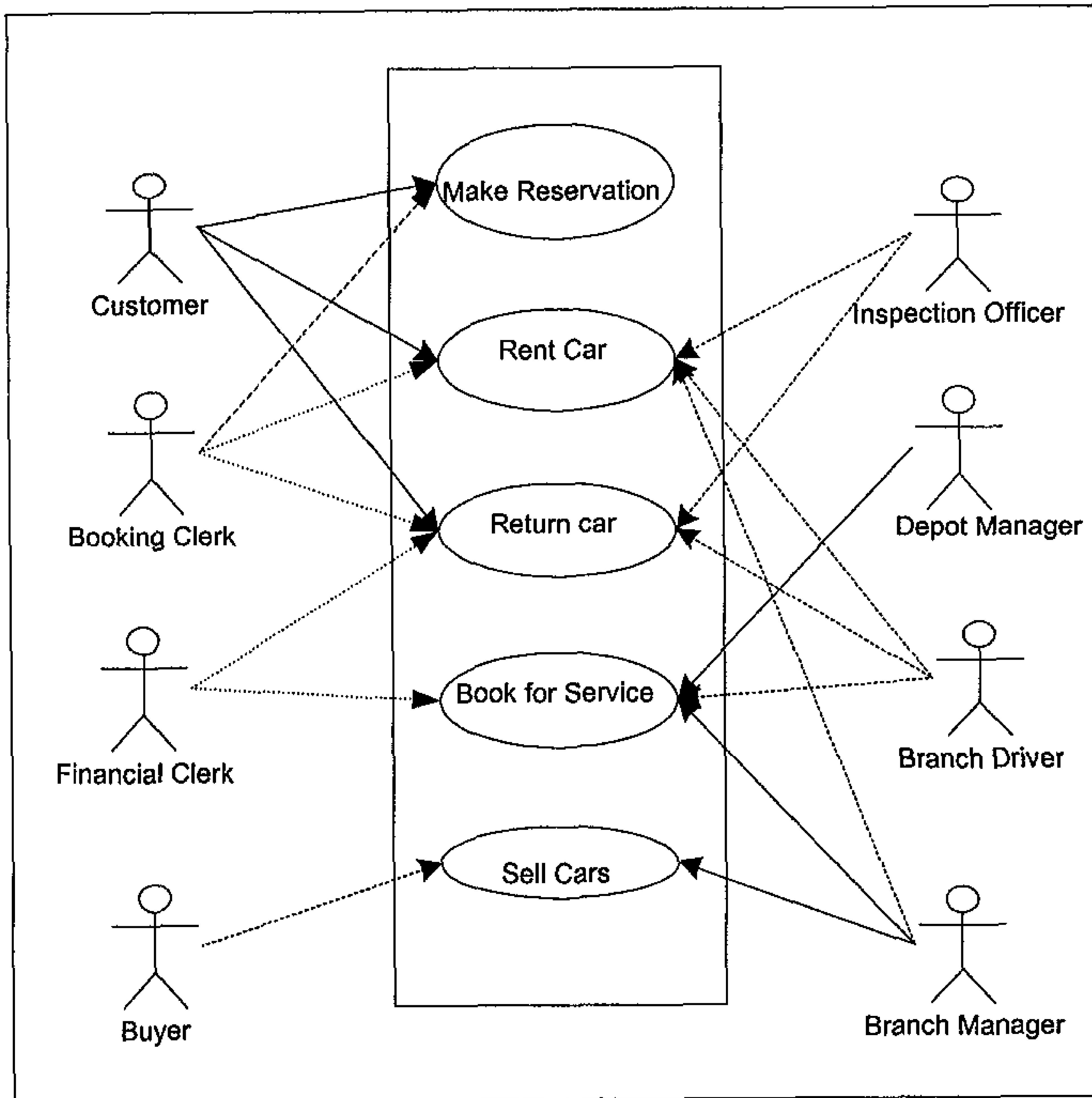


Figure 42: Use-case Diagram

Primary Actors – Customer, depot manager, branch manager.

Secondary Actors – Booking clerk, buyer, branch driver, inspection officer, financial clerk.

Table 2: Use case: Make Reservation

USE CASE: 01	Make Reservation.	
Context	Customer makes a reservation for a car from a branch.	
System	Car Rental Company.	
Preconditions	The customer must complete an application form.	
Success End Condition	Customer's reservation request is accepted.	
Failed End Condition	Customer's reservation request is rejected.	
Primary, Secondary Actors	Customer Booking clerk, branch driver.	
Trigger	Customer submits a reservation request.	
NORMAL COURSE	**Step**	**Action**
	1	The customer submits an advance reservation request.
	2	The booking clerk requests the 'reservation form' option from the system.
	3	The system displays a reservation form.
	4	The booking clerk enters the customer details in the reservation form.
	5	The system verifies the customer details.
	6	The system displays the message that the customer is new and prompts the booking clerk to 'add customer' to the customer list.
	7	The booking clerk adds the customer to the existing list of customers.
	8	The booking clerk requests the 'save customer' option from the system.
	9	System saves customer information including Customer ID, age, address, licence details, insurance details, credit card details, rental period and requested car group and model.
	10	The system prompts the booking clerk to continue with a new reservation for the customer.
	11	Booking clerk requests the 'check car availability' option from the list of cars.
	12	System verifies the availability of the requested car group and model from the list of cars.
	13	System notifies the booking clerk about the car availability.
	14	Booking clerk selects the requested car group and model and requests the 'save car' option from the system.
	15	System saves the selected car group and model details for the customer as the customer reservation.
	16	Booking clerk terminates the session.
ALTERNATE COURSE	**Step**	**Branching Action**

	1a	The customer submits an immediate walk-in reservation request. 1a1: Continue from 2 in normal course.
	6a	The system displays the message that the customer already exists. 6al: The booking clerk requests the 'display customer' option from the system. 6a2: The system displays the customer information including previous rental, payment history, late returns and damages if any. 6a3: The system prompts the booking clerk to edit any incorrect information. 6a4: Continue from 10 in normal course.
	9a 9b 9b2a	Customer does not request a particular car group or model. 9a1: Perform 10 - 11 in the normal course. 9a2: System verifies the car availability in the default group A. 9a3: System gives the lists of cars in default group A. 9a4: Continue from 13 in normal course. Customer specifies a particular model only. 9b1: Perform 10 – 11 in normal course. 9b2: System verifies if a car of the requested model is available. 9b3: System gives the list of cars of the requested model. 9b4: Continue from 13 in normal course. System notifies that a requested model is not available. 9b2a1: System prompts the booking clerk to assign a car in the same group as the requested model. 9b2a2: Booking clerk enters a car in the group. 9b2a3: Continue from 14 in normal course.
	10a	Customer makes multiple future reservations. 10a1: Booking clerk requests the 'save future reservation' option from the system. 10a2: System displays the requested option. 10a3: Booking clerk enters the future reservations for the customer. 10a4: System saves the future reservation details for the customer. 10a5: Perform 16 of the normal course.
	13a 13a1a	System informs that cars are not physically available at the particular branch. 13a1: System prompts the booking clerk with the following options: 'Allocate a car from next day's walk-in', ' upgrade', 'downgrade', 'transfer from another branch', and 'use a car scheduled for service'. Booking clerk requests the option 'Allocate_a car from next day's walk-in' 13a1a1: System displays the list of cars allocated for next day's walk-in. 13a1a2: System prompts the booking clerk to select a car from the list allocated for walk-in.

		13a1a3: Booking clerk selects a car for rental from the displayed list. 13a1a4: Continue from 14 in normal course.
	13a1b	Booking clerk requests the 'Upgrade' option from the system. 13a1b1: System verifies if the customer is a member of the incentive scheme. 13a1b2: System confirms that the customer is a member of the incentive scheme. 13a1b3: System displays the list of upgradable cars and prompts the booking clerk to select a car from the list displayed. 13a1b4: Booking clerk selects a car from the list. 13a1b5: Continue from 14 in normal course.
	13a1c	Booking clerk requests the 'downgrade' option from the system. 13a1c1: Booking clerk requests the system to select a car in a lower group than the requested one. 13a1c2: Continue from 13 in normal course.
	13a1d	Booking clerk requests the 'transfer from another branch' option from the system. 13a1d1: System verifies the availability of cars at other branches. 13a1d2: System displays the list of available cars for transfer. 13a1d3: Perform 14 in normal course. 13a1d4: System saves transfer information such as the branch from which a car needs to be transferred, car group and car model requested. 13a1d5: System confirms the transfer of the car from the identified branch and changes the ownership of the car to the requested branch. 13a1d6: Booking clerk notifies the branch driver to transfer the car from the identified branch to the branch that requested the transfer. 13a1d7: Branch driver transfers the car to the requested branch. 13a1d8: Perform 16 in normal course.
	13a1e	Booking clerk requests the 'use a car scheduled for service' option from the system. 13a1e1: Booking clerk verifies with the customer if the rental would take more than 10% of the service mileage. 13a1e2: Customer confirms the rental mileage with the booking clerk. 13a1e3: Booking clerk requests the 'rental mileage' option from the system. 13a1e4: System prompts the booking clerk to enter the rental mileage in the rental mileage form. 13a1e5: Booking clerk enters the rental mileage and requests the system to save the rental mileage. 13a1e6: System saves the rental mileage for the customer and verifies the availability of cars in the service list.

		13a1e7: System displays the list of serviceable cars and prompts the booking clerk to select a car from the 'car service list' of the system. 13a1e8: Booking clerk selects a car from the service list. 13a1e9: System reschedules the selected car for rental. 13a1e10: System confirms the rental acceptance. 13a1e11: Perform 16 in normal course.
EXCEPTIONS	**Step**	**Branching Action**
	6a2a	System displays the message that the customer is blacklisted. 6a2a1: The system prompts the booking clerk to terminate the reservation request. 6a2a2: Perform 16 in normal course.
	9a2a	System confirms that no cars are available in default group A. 9a2a1: Perform 16 in normal course.
	13a1a1a	System confirms that a car is not available for the next day's walk-in. 13a1a1a1: Perform 16 in normal course.
	13a1b2a	System confirms that the customer is not a member of incentive scheme. 13a1b2a1: System refuses to perform an upgrade for the customer. 13a1b2a2: Perform 16 in normal course.
	13a1c2a	System notifies that car in a lower group is not available. 13a1c2a1: Perform 16 in normal course.
	13a1d2a	System notifies that cars are not available for transfer at the other branches. 13a1d2a1: Perform 16 in normal course.
	13a1e2a	Customer confirms that the rental will take more than 10% of the service mileage. 13a1e2a1: Perform 16 in normal course.
RELATION TO OTHER USE CASES	Rent Car, Return Car, Book for Service	

Table 3: Use case Rent Car

USE CASE: 02	Rent Car	
Context	Customer expects to rent a car from a branch.	
System	Car Rental Company.	
Preconditions	The customer's reservation request has been accepted by the system and a car is allocated to him.	
Success End Condition	Customer rents a car from the branch where the application is processed.	
Failed End Condition	Customer makes no-show and a car is not rented out.	
Primary, Secondary Actors	Customer. Inspection officer, booking clerk.	
Trigger	The customer has a valid reservation.	
NORMAL COURSE	**Step**	**Action**
	1	The customer has a valid reservation and the booking clerk verifies the customer reservation on the system.
	2	Booking clerk requests the inspection officer to check if the available car is prepared and checked for roadworthiness to be rented out to the customer.
	3	Inspection officer prepares the car for rental and informs the booking clerk that the car is ready for pick-up.
	4	Booking clerk verifies the customer's driver's licence and CustomerID with the customer and with the system.
	5	Booking clerk enters the rental mileage of the car into the system at pick-up and requests the system to save the rental mileage.
	6	System saves the rental mileage at pick-up.
	7	Customer rents the car from the branch.
	8	Booking clerk terminates the session.
ALTERNATE COURSE	**Step**	**Branching Action**
		No alternate courses.
EXCEPTIONS	**Step**	**Branching Action**
	4a	Customer's driver's licence is not valid. 4a1: Booking clerk informs the customer that he cannot rent a car without a driver's licence. 4a2: Perform 8 in normal course.
RELATION TO OTHER USE CASES	Make Reservation, Return Car, Book for Service	

Table 4: Use case Return Car

USE CASE: 03	Return Car	
Context	Customer returns the rented car to the branch.	
System	Car Rental Company.	
Preconditions	The customer has rented a car from a branch.	
Success End Condition	Customer returns the rented car to a branch. If car is returned to a non-pick-up branch, ownership of car changes to the non-pick-up branch. Customer pays either using credit card or in cash. Credit card belongs to authorised driver and all additional drivers must pay in cash.	
Failed End Condition	Customer does not return the car and rental charge to the branch.	
Primary, Secondary Actors	Customer. Inspection officer and financial clerk.	
Trigger	Customer returns the rented car back to a pick-up branch at the drop-off time.	
NORMAL COURSE	**Step**	**Action**
	1	Customer returns the car to the inspection officer at the pick-up branch at the drop-off time.
	2	Inspection officer checks the car for any car damage caused during the rental period.
	3	Inspection officer requests the 'car condition on return' option from the system.
	4	System displays a form in which the condition of car at return time is captured as well as the rental mileage of the car.
	5	Inspection officer completes the form and requests the system to save the details of rented car for the particular customer.
	6	System saves the submitted details for the customer.
	7	Inspection officer requests the financial clerk to prepare an invoice for the customer.
	8	Financial clerk requests the 'rental details for customer', 'branch details option' and 'car condition' option from the system.
	9	System displays the rental details for customer including pick-up time, drop-off time, branch details including pick-up branch, drop-off branch and car condition including damage caused if any.
	10	Financial clerk requests the invoice form from the system.
	11	System displays the invoice form on which the financial clerk enters the pick-up time, drop-off time, pick-up branch, drop-off branch and damage caused.
	12	Financial clerk requests the system to calculate the rental charge.
	13	System calculates the rental charge for the customer.
	14	Financial clerk hands over the printed invoice to the customer.
	15	Customer pays the rental charge to the financial clerk

		using credit card.
	16	Financial clerk enters credit card details into the system.
	17	The system verifies the credit card details of customer.
	18	Financial clerk enters the rental charge paid by the customer and requests the system to save the payment details.
	19	System saves the payment details of the customer.
	20	Financial clerk terminates the session.
ALTERNATE COURSE	**Step**	**Branching Action**
	1a	Customer returns the rented car to a non-pick-up branch at the specified time. 1a1: Continue from 2 to 12 in normal course. 1a2: System calculates the rental charge for the customer by adding a drop-off penalty charge to the rental charge. 1a3: Continue from 14 in normal course.
	1b	Customer returns to a non-pick-up branch earlier than the drop-off time. 1b1: Continue from 2 to 12 in normal course. 1b2: System calculates a rental charge at a rate appropriate to the actual period of rental at a daily rate. 1b3: Continue from 14 in normal course.
	1c	Customer returns to a non-pick-up branch later than the drop-off time. 1c1: Continue from 2 to 12 in normal course. 1c2: System calculates a rental charge according to a 6-hourly delay. If the drop-off is after 6 hours, a full day's rental is charged. 1c3: Continue from 14 in normal course.
EXCEPTIONS	**Step**	**Branching Action**
	1ca	Customer does not return car after 3 days since the drop-off time. 1ca1: Booking clerk informs the police.
ASSUMPTION	Authorised drivers sign a contract and additional drivers sign an authorisation form.	
RELATION TO OTHER USE CASES	Make Reservation, Rent, Book for Service.	

Table 5: Use case Book for Service

USE CASE: 04	Book for Service	
Context	Car has been rented for three months or reached a service mileage of 10000km.	
System	Car Rental Company.	
Preconditions	A car has reached 10000kms as service mileage.	
Success End Condition	Car is serviced and returned from depot.	
Failed End Condition	Car cannot be serviced.	
Primary, Secondary Actors	Depot manager Branch manager, financial clerk and branch driver	
Trigger	Depot manager requests the 'car service list' from the system.	
NORMAL COURSE	**Step**	**Action**
	1	The depot manager requests the 'car service list' option from the system.
	2	The system verifies the list of cars to pick up the ones with mileage of 10000km or more.
	3	System displays the list of cars that need service.
	4	The depot manager requests the 'depot capacity' option from the system.
	5	System displays the capacity of all the depots servicing the branches.
	6	The depot manager requests the 'book for service' option from the system.
	7	The system displays a service form that captures the service date and time as well as return date and time for a particular car.
	8	The depot manager enters the details in the service form and requests the system to save the service details.
	9	System saves the service details for a car.
	10	The Depot manager notifies the branch manager that a certain number of cars are booked in for service.
	11	The branch manager notifies the branch driver to drop off the serviceable cars at the depot.
	12	The branch driver drives the cars to the depot.
	13	The depot services the car from a branch.
	14	The depot manager calculates the service charge and notifies the financial clerk of the charge.
	15	Financial clerk prepares the cheque for the requested amount and pays the amount to the depot manager.
	16	Depot manager enters the cheque details into the system including payment method, amount paid and service date and request the system to save the payment details.
	17	System saves the payment details.
	18	Depot manager terminates the session.
ALTERNATE COURSE	**Step**	**Branching Action**

	2a	System verifies cars running without service for a period of three months. 2al: Continue from 3 in normal course.
	15a	Financial clerk pays the service charge in cash. 15a1: Continue from 16 in normal course.
EXCEPTIONS	**Step**	**Branching Action**
	5a	System notifies that all depots are full and a new service cannot be accepted until a car becomes free. 5a1: Perform 18 in normal course.
RELATION TO OTHER USE CASES	Return Car, Rent Car.	

Table 6: Use case Sell Cars

USE CASE: 05	Sell Cars.	
Context	Cars reach 40000 km or age of one year.	
System	Car Rental Company.	
Preconditions	The mileage of a car is 40000 km.	
Success End Condition	Car is sold.	
Failed End Condition	Car cannot be sold.	
Primary, Secondary Actors	Branch manager Buyer	
Trigger	Branch manager requests the 'authorised sale list' from the system.	
NORMAL COURSE	**Step**	**Action**
	1	Branch manager requests the 'authorised sale list' from the system.
	2	System verifies the car list to pick up cars that reached a mileage of 40000 km and above.
	3	System displays the list of cars for sale.
	4	Branch manager advertises the list for sale including car model and sale price.
	5	Buyers contact the branch manager for sale negotiations.
	6	Branch manager sells a car to the buyer.
	7	Branch manager requests the 'remove car' option from the system.
	8	System prompts the branch manager to remove the sold car from the authorised list.
	9	Branch manager removes the car.
	10	Branch manager terminates the session.
ALTERNATE COURSE	**Step**	**Branching Action**
	2a	System checks for cars running for one year. 2a1: Continue from 3 in normal course.
EXCEPTIONS	**Step**	**Branching Action**
	5a	No buyers contact the branch manager for sale. 5a1: Perform 10 in normal course.
RELATION TO OTHER USE CASES	Rent Car, Book for Service	

Realisation of Use Cases Using Collaboration Diagrams

Once the use cases have been written, the next step is to draw the interaction diagram for the use cases. The interaction diagram chosen for this case study is the collaboration diagram (discussed in section 4.7). In order to draw the collaboration diagrams for the developed use cases, the first step is to identify all the objects in the use case, which is done by identifying the nouns in the use cases.

Identifying Objects from the Use Cases

Sturm [1999] gives the following rules for identifying objects from use cases.

- Objects are things - usually nouns, but sometimes they can be expressed as verbs.
- Objects contain some type of information (they have state).
- Objects have some type of behaviour (they do something).
- Any component that contains something that has already been identified as an object is also an object.

Objects identified from use case 01

Table 7: Objects Identified for Use-case 01

Normal Course	
Object	**Component of Object**
Customer	Customer Details
Customer Details	(Customer ID, age, address, licence details, insurance details, credit card details)
Customer List	
Advance Reservation	Reservation, Reservation Form, Reservation Details
Reservation Details	(Rental Period)
Booking clerk	
System	(Branch Reception)
Car	Car Model
Car Group	
Car List	

Additional Object(s) Identified for Alternative 6a	
Customer Information	Previous Rental, Payment History, Late Returns and Damages
Additional Object(s) Identified for Alternative 10a	
Advance Reservation	Future Reservation
Additional Object(s) Identified for Alternative 13a1a	
Branch	(BranchCapacity)
Additional Object(s) Identified for Alternative 13a1b	
Scheme	
Additional Object(s) Identified for Alternative 13a1d	
Branch Driver	Does not interact with the system and is therefore not modelled in the collaboration
Additional Object(s) Identified for Alternative 13a1e	
Service Mileage	
Rental Mileage	
Rental	
Service List	
Additional Object(s) Identified for Exception 6a2a	
Blacklisted Customer	Part of Customer object and not modelled separately

Noun: Booking clerk

Booking clerks only interact with the system as actors and are not themselves recognised within the system. Therefore, Booking Clerk will be modelled as an actor in the collaboration diagram.

Nouns: Customer, Customer List

A customer is identified within the system and is an object relevant to the system. Therefore Customer will be modelled as an object. Customer details such as Customer ID, age, address, licence details, insurance details and credit card details are contained in the Customer object and therefore Customer Details is not shown as a separate object. Customer List needs to be recorded in a separate object, and this object is modelled as Customer List.

Noun: System

The system as such does not accept or reject the users' requests; rather, an object representing a system function will process the request. This system object is represented by Branch Reception.

Noun: Advance Reservation

Reservation Details (rental period) is part of the Advance Reservation object and therefore not shown as a separate object.

Nouns: Car, Car Group

The noun Car Group has sets of cars contained inside. Therefore Car Group and Car are modelled as different objects. Car model belongs to each car and it is modelled together with the Car object.

Nouns: Future Reservation, Service Mileage and Rental Mileage

Future Reservation is part of Advance Reservation and is not shown as a separate object. Service Mileage is part of the Car object and Rental Mileage is part of the Rental object.

The collaboration diagrams for use case 01 are shown in the figures 43-60.

Normal Course: Customer submits an advance reservation

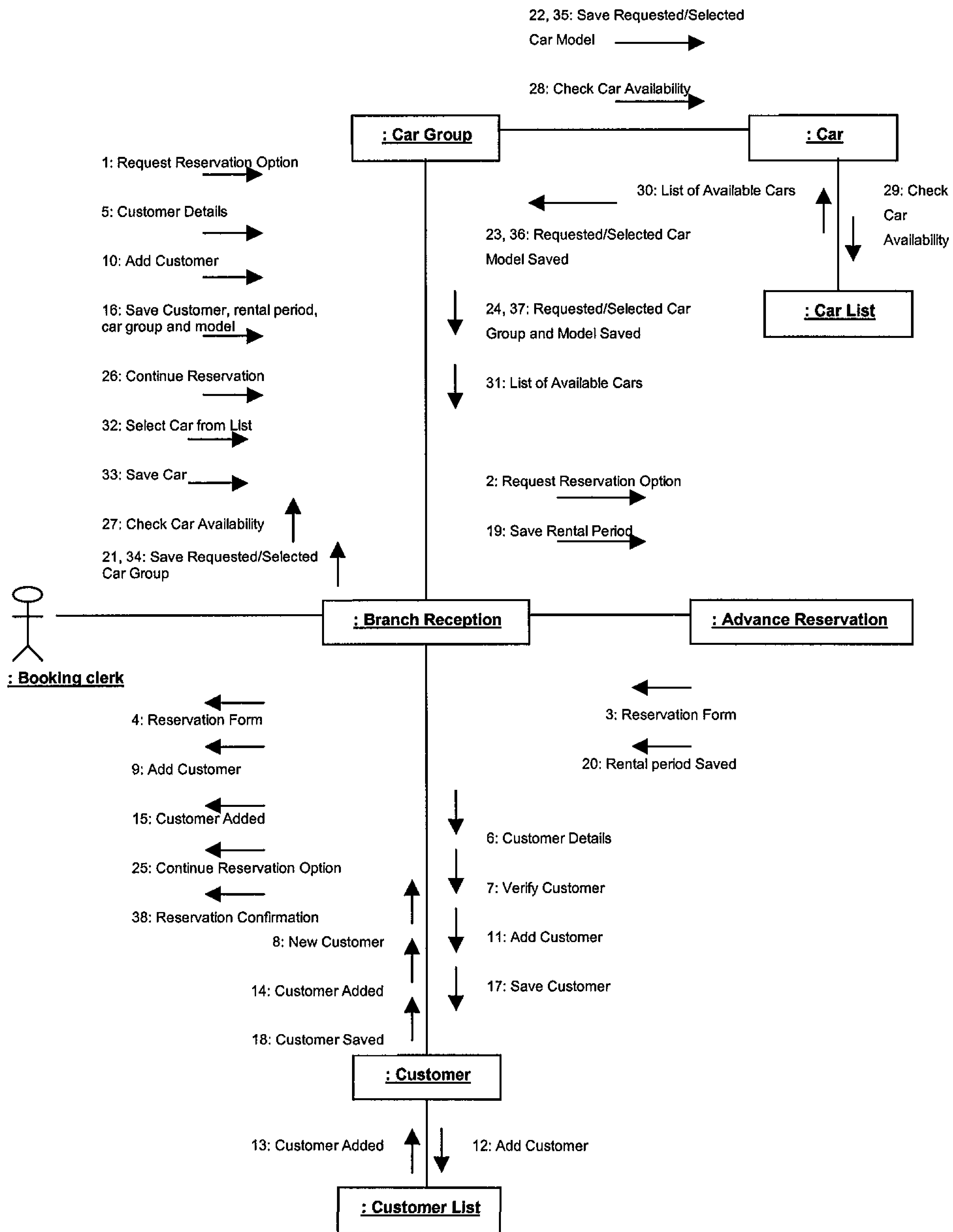


Figure 43: Collaboration for use case 01: Make reservation (normal course)

Alternative 1a: Customer submits an immediate walk-in reservation

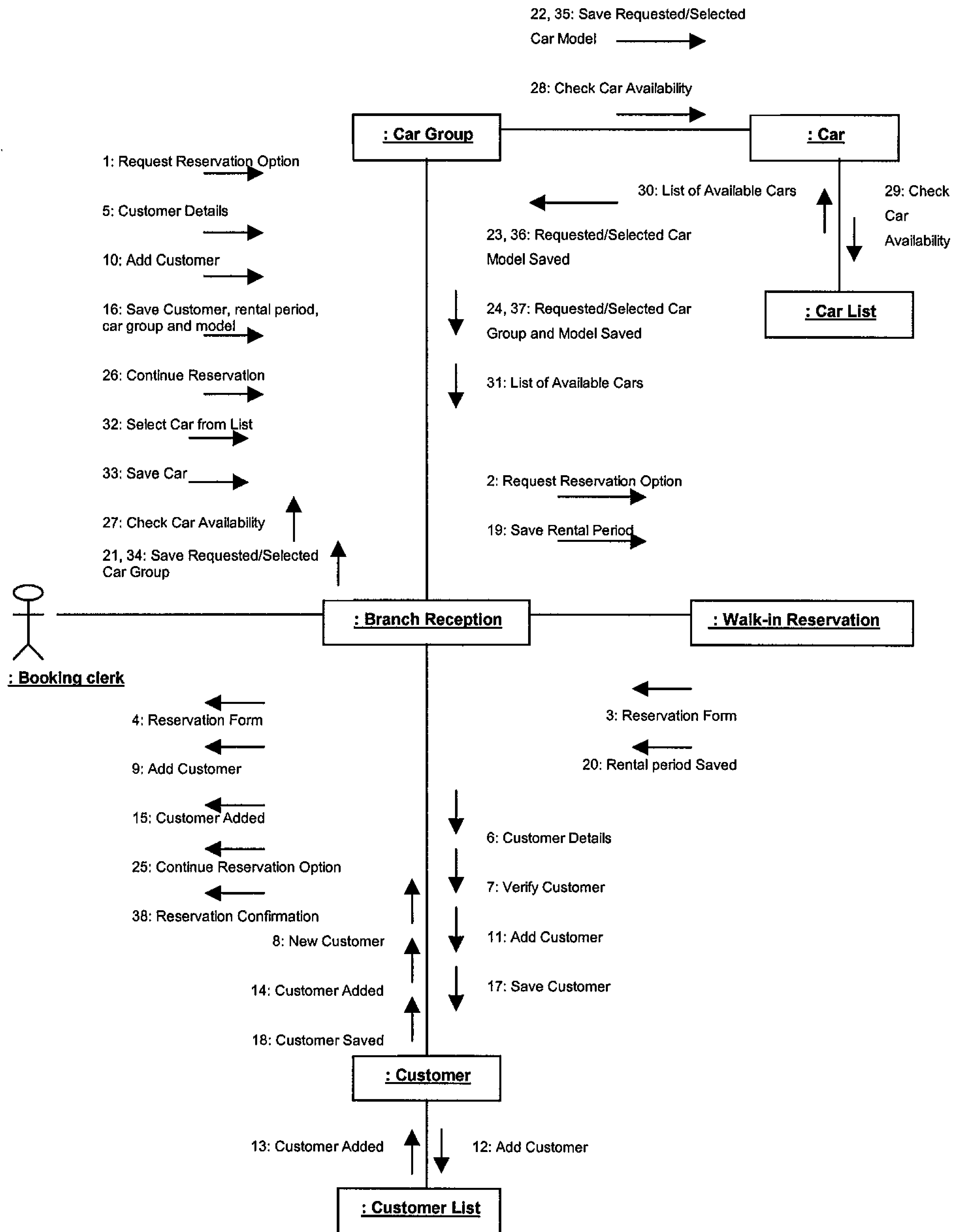


Figure 44: Collaboration for use case 01: Make reservation (Alternative 1a)

Alternative 6a: System informs that customer exists

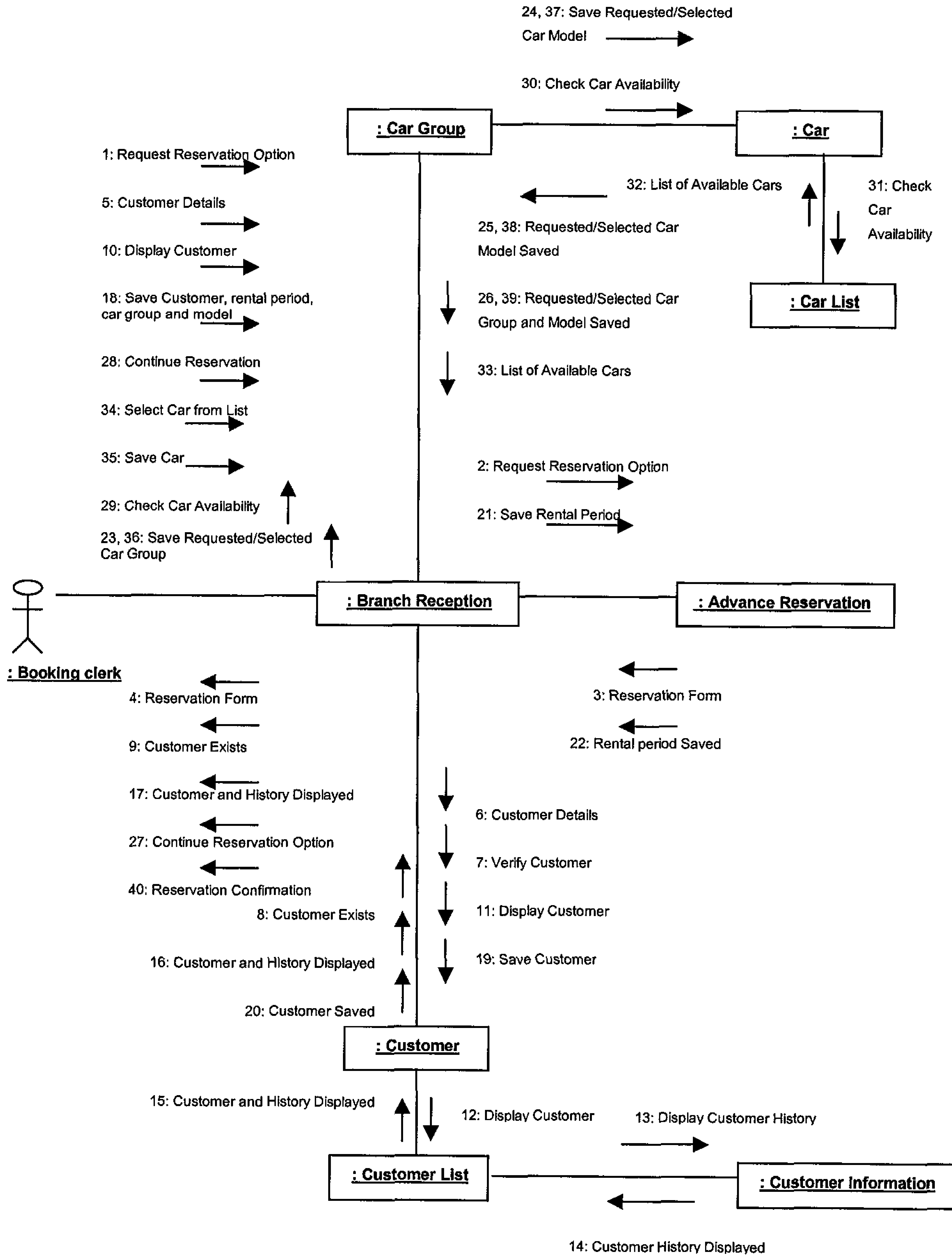


Figure 45: Collaboration for use case 01: Make reservation (Alternative 6a)

Alternative 9a: Customer does not request a particular car group and model

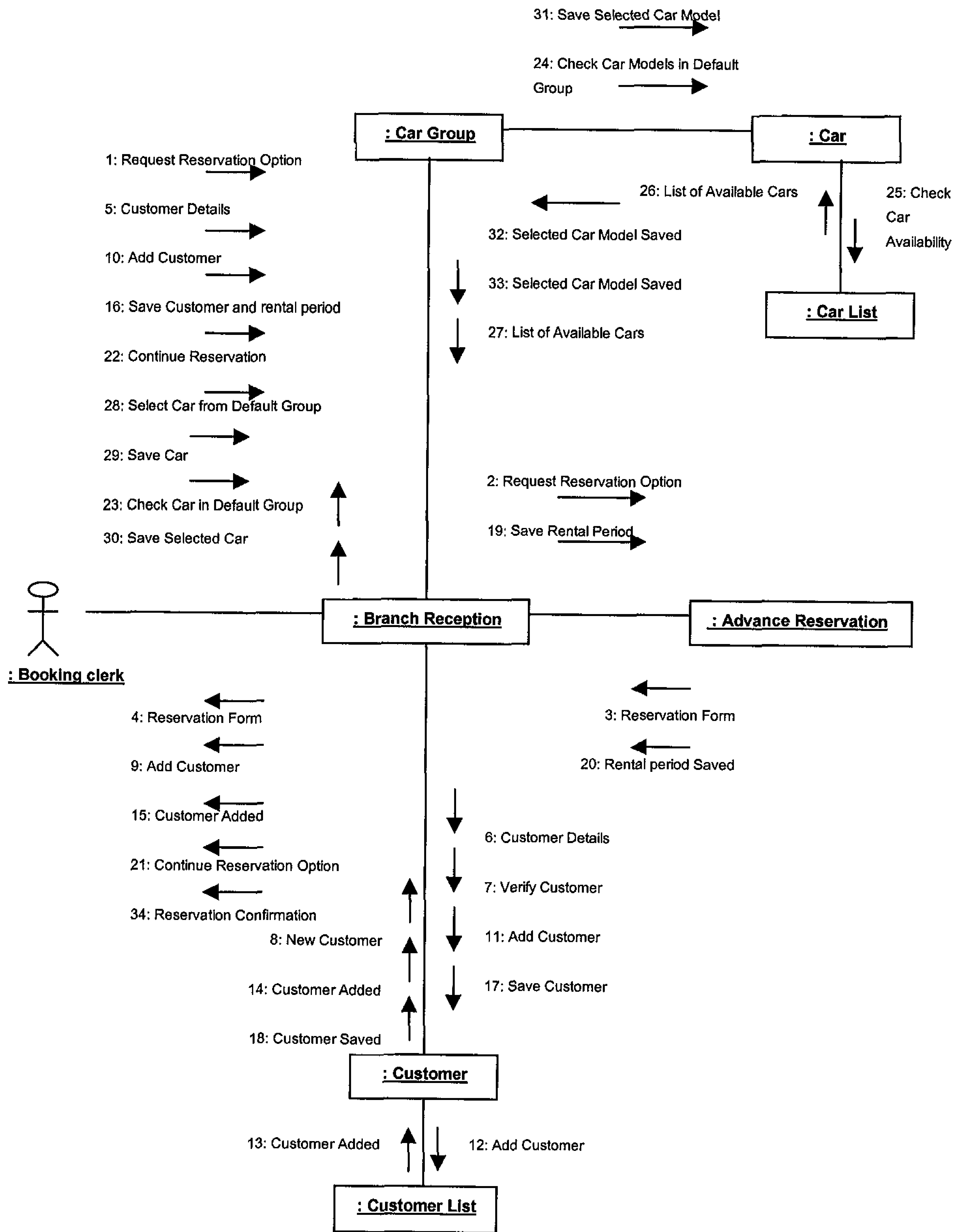


Figure 46: Collaboration for use case 01: Make reservation (Alternative 9a)

Alternative 9b: Customer specifies a particular model only

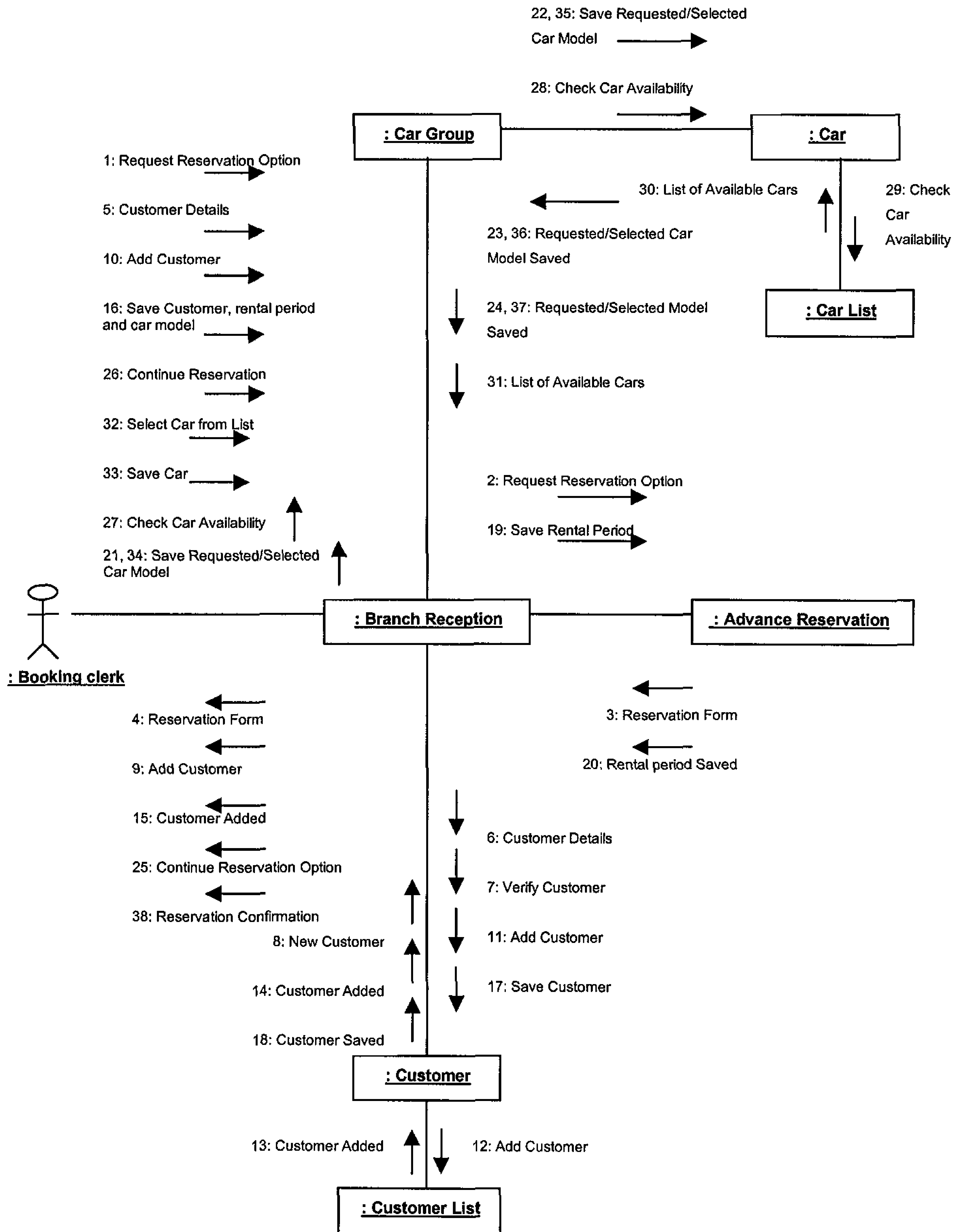


Figure 47: Collaboration for use case 01: Make reservation (Alternative 9b)

Alternative 9b2a: System notifies that a requested model is not available

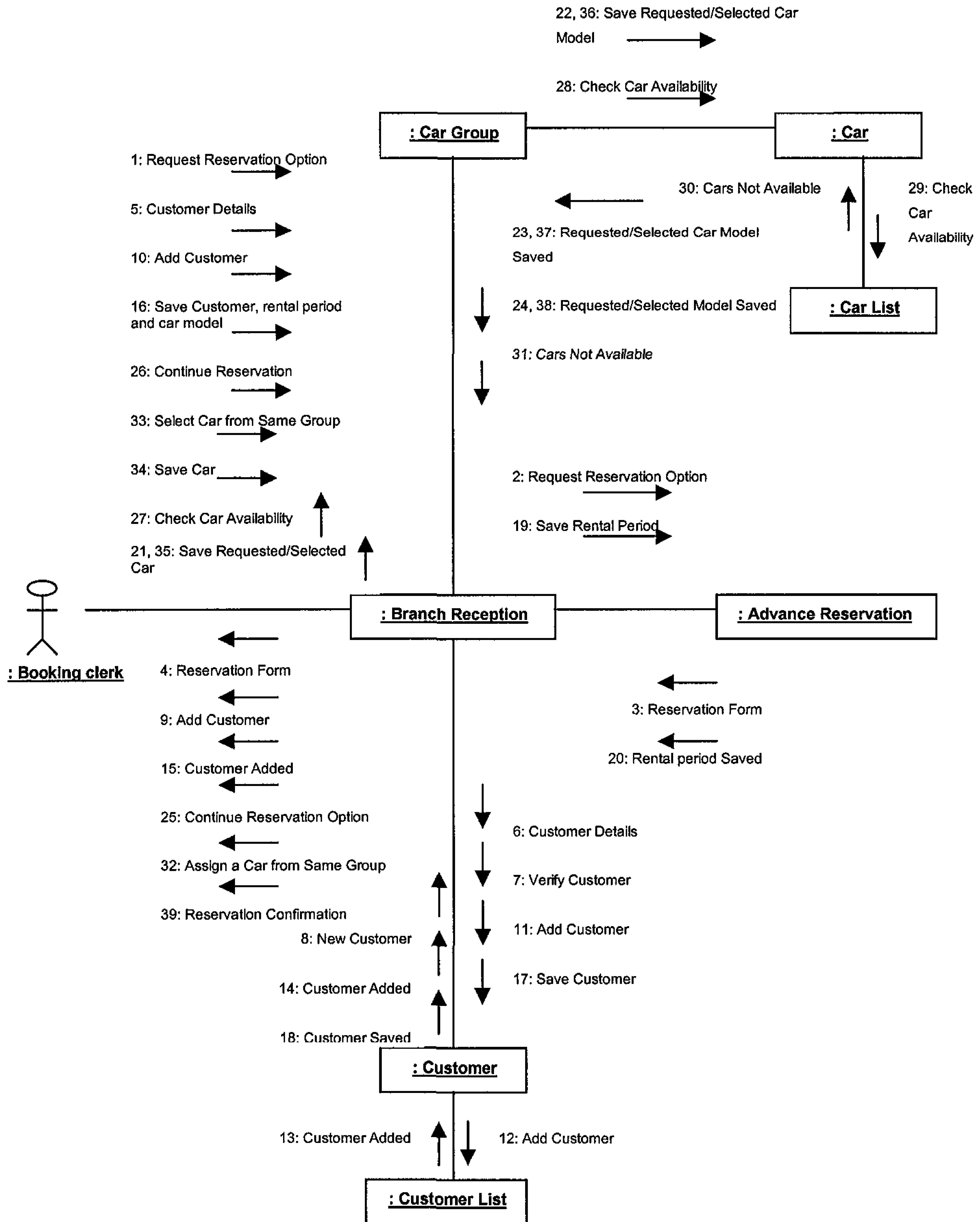


Figure 48: Collaboration for use case 01: Make reservation (Alternative 9b2a)

Alternative 10a: Customer makes multiple future reservations

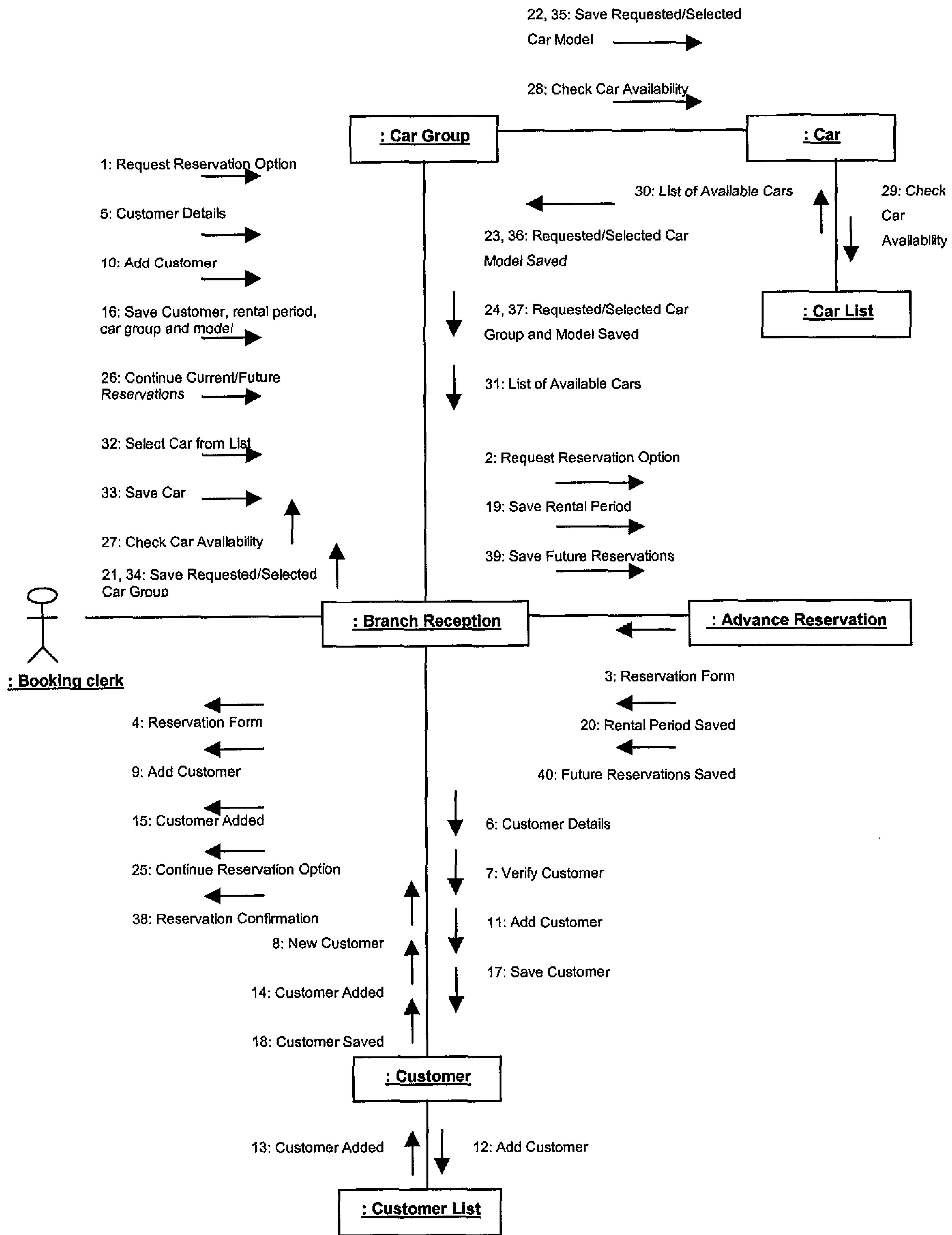


Figure 49: Collaboration for use case 01: Make reservation (Alternative 1oa)

Alternative 13a1a: System informs that cars are not available at the branch and gives the option to allocate a car from next day's walk-in.

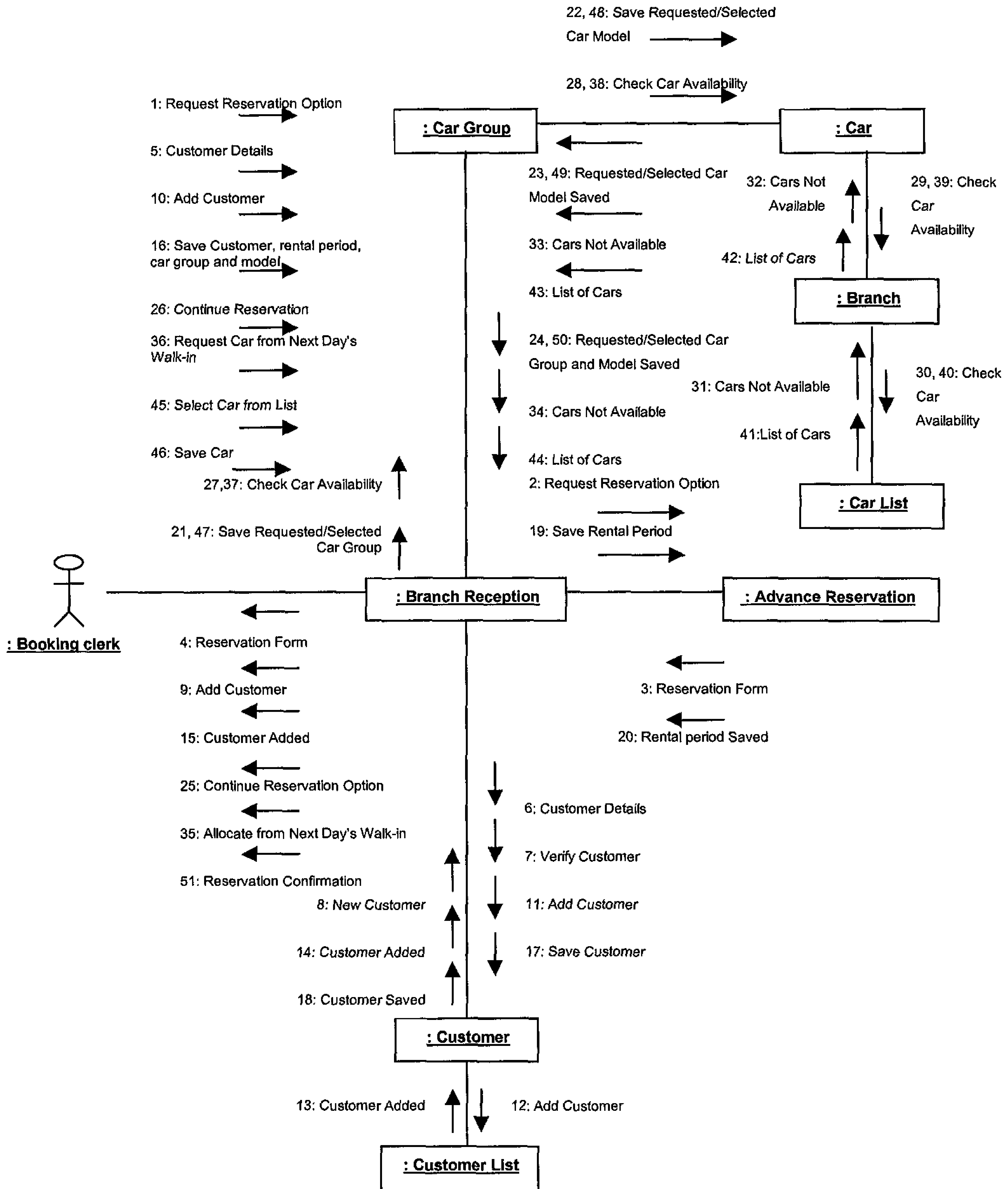


Figure 50: Collaboration for use case 01: Make reservation (Alternative 13a1a)

Alternative 13a1b: Booking clerk requests the upgrade option from the system

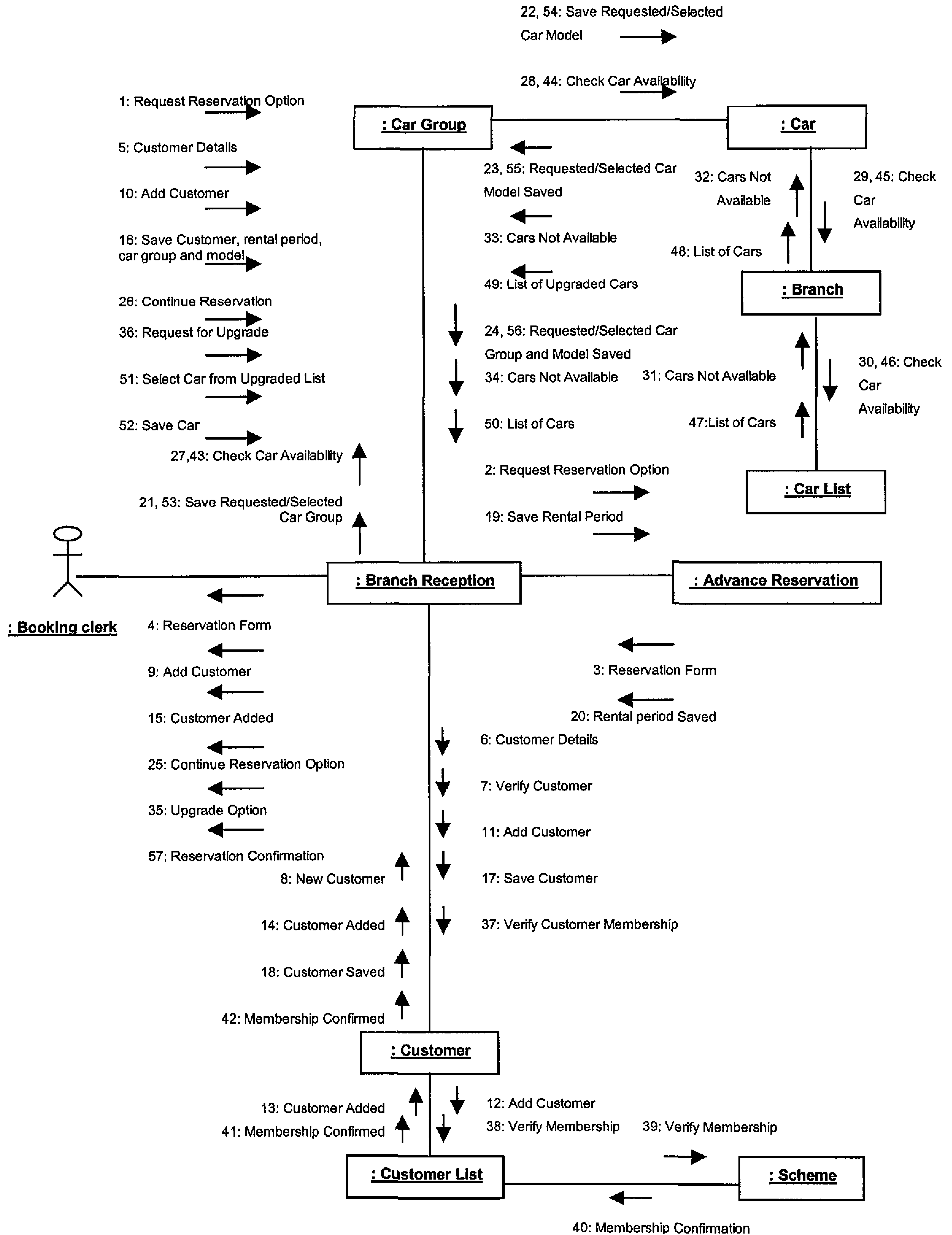


Figure 51: Collaboration for use case 01: Make reservation (Alternative 13a1b)

Alternative 13a1c: Booking clerk requests the downgrade option from the system

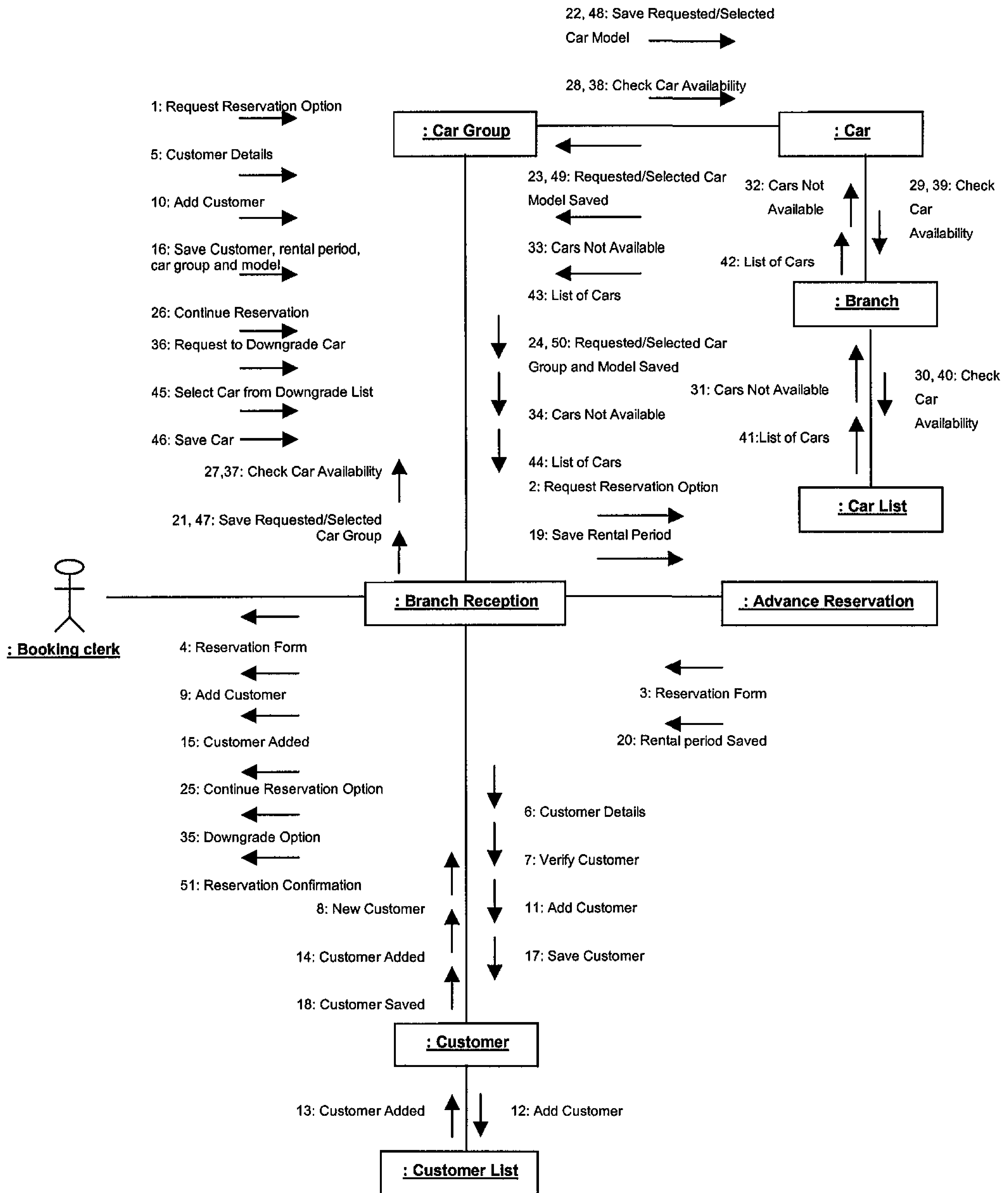


Figure 52: Collaboration for use case 01: Make reservation (Alternative 13a1c)

Alternative 13a1d: Booking clerk requests the transfer car option from the system

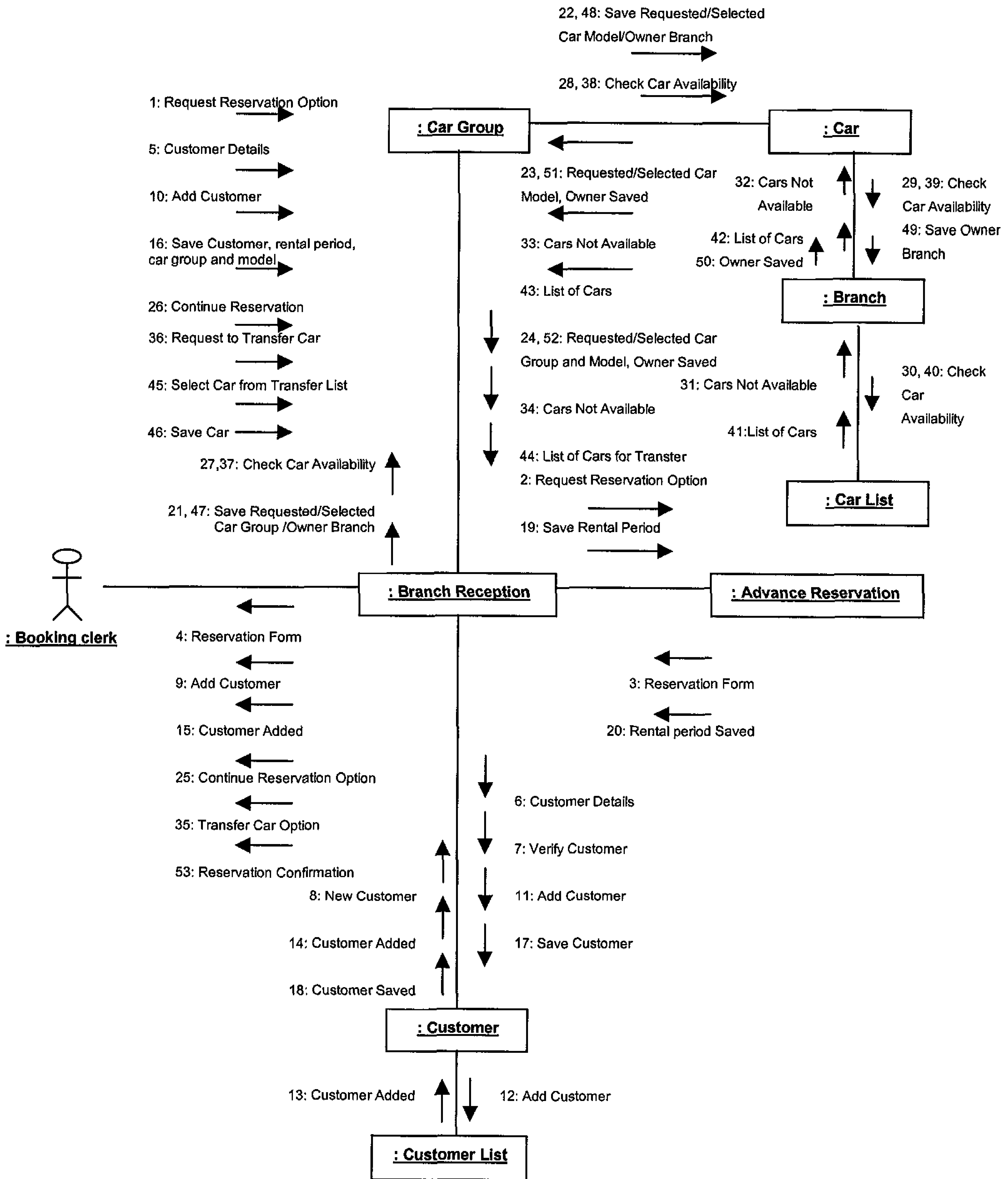


Figure 53: Collaboration for use case 01: Make reservation (Alternative 13a1d)

Alternative 13a1e: Booking clerk requests the 'use a serviceable car' option from the system

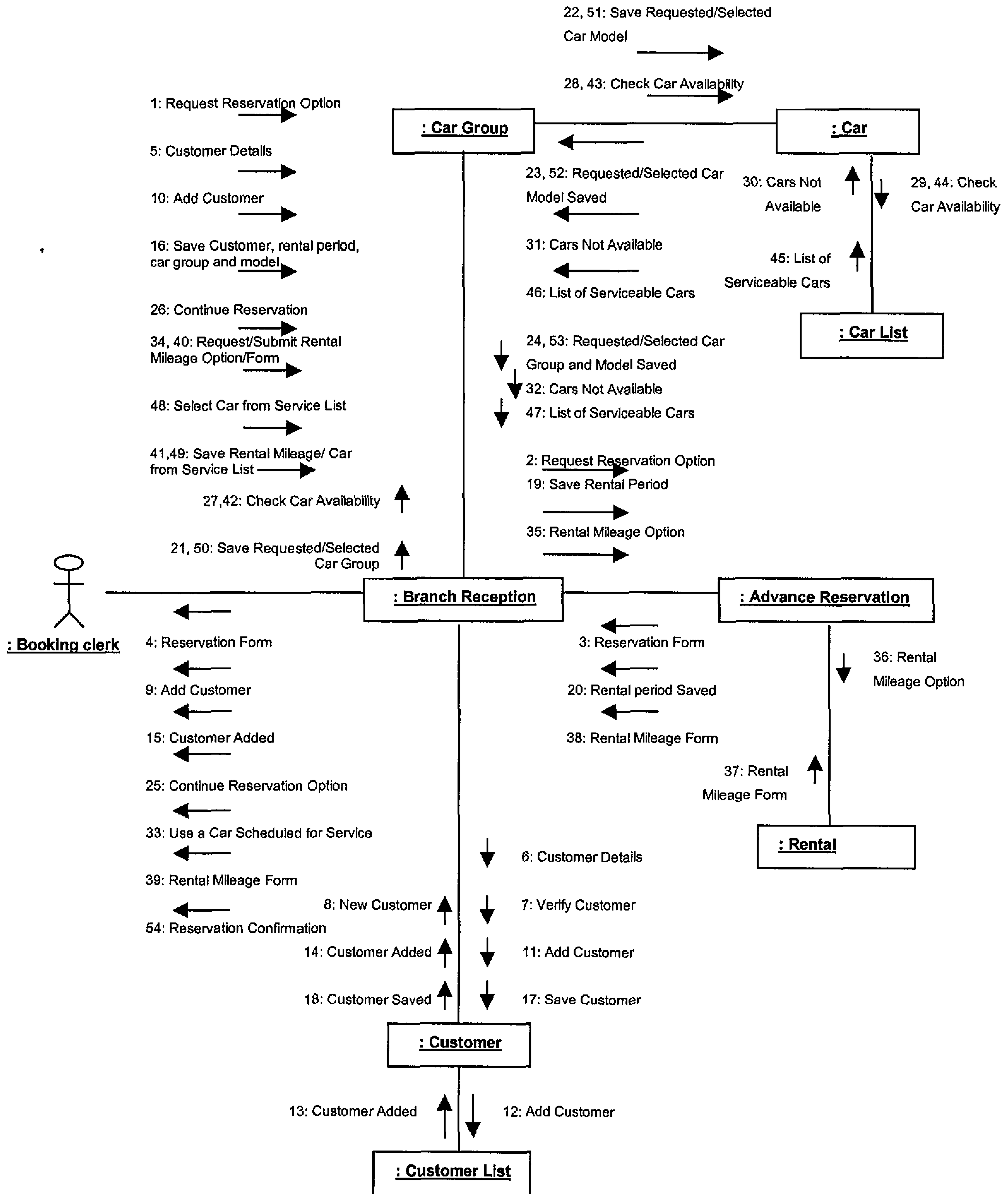


Figure 54: Collaboration for use case 01: Make reservation (Alternative 13a1e)

Exception 6a2a: System informs that customer is blacklisted

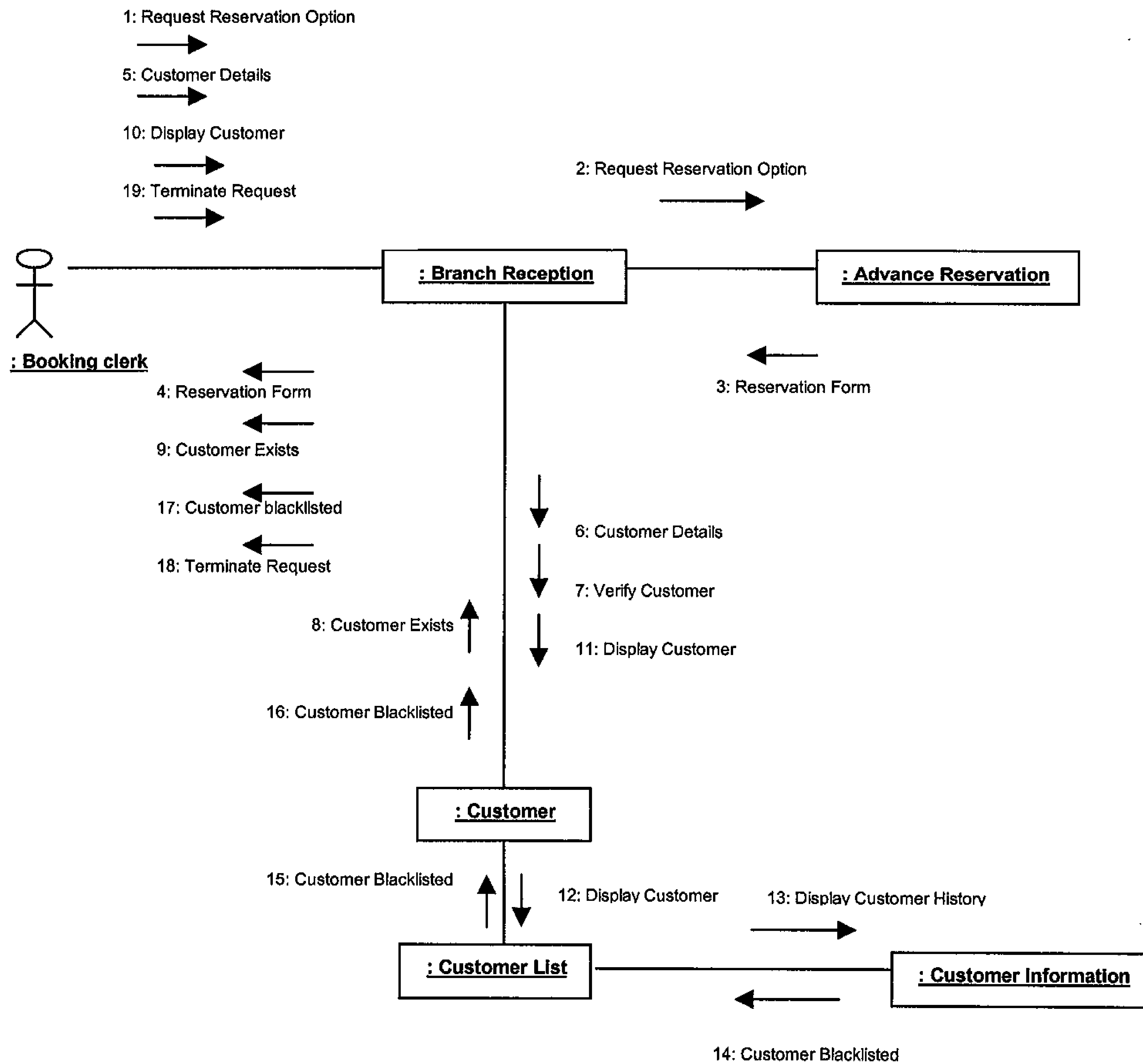


Figure 55: Collaboration for use case 01: Make reservation (Exception 6a2a)

Exception 9a2a: System informs that there are no cars available in the default group

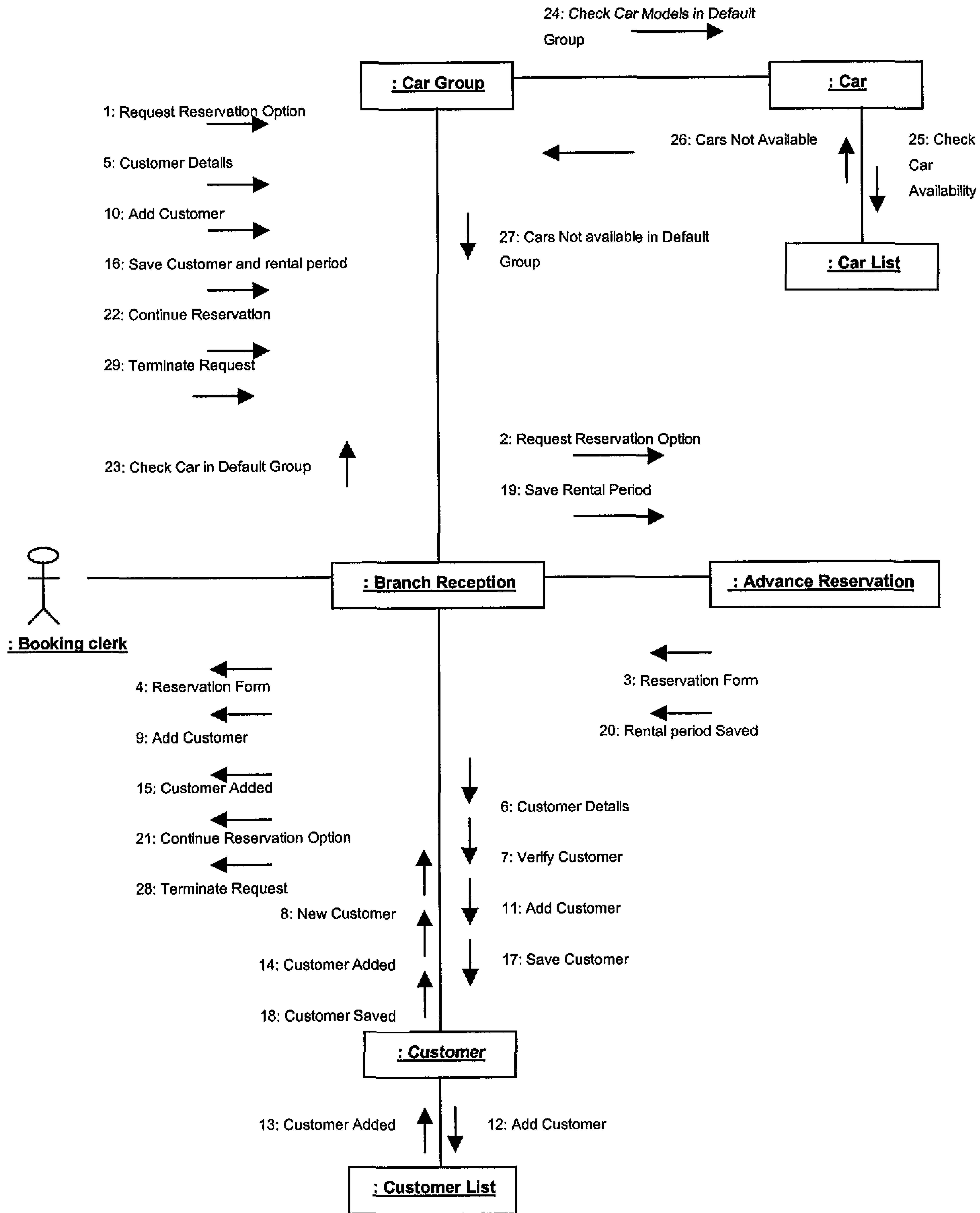


Figure 56: Collaboration for use case 01: Make reservation (Exception 9a2a)

Exception 13a1a1a: System informs that there are no cars available from the next day's walk-in

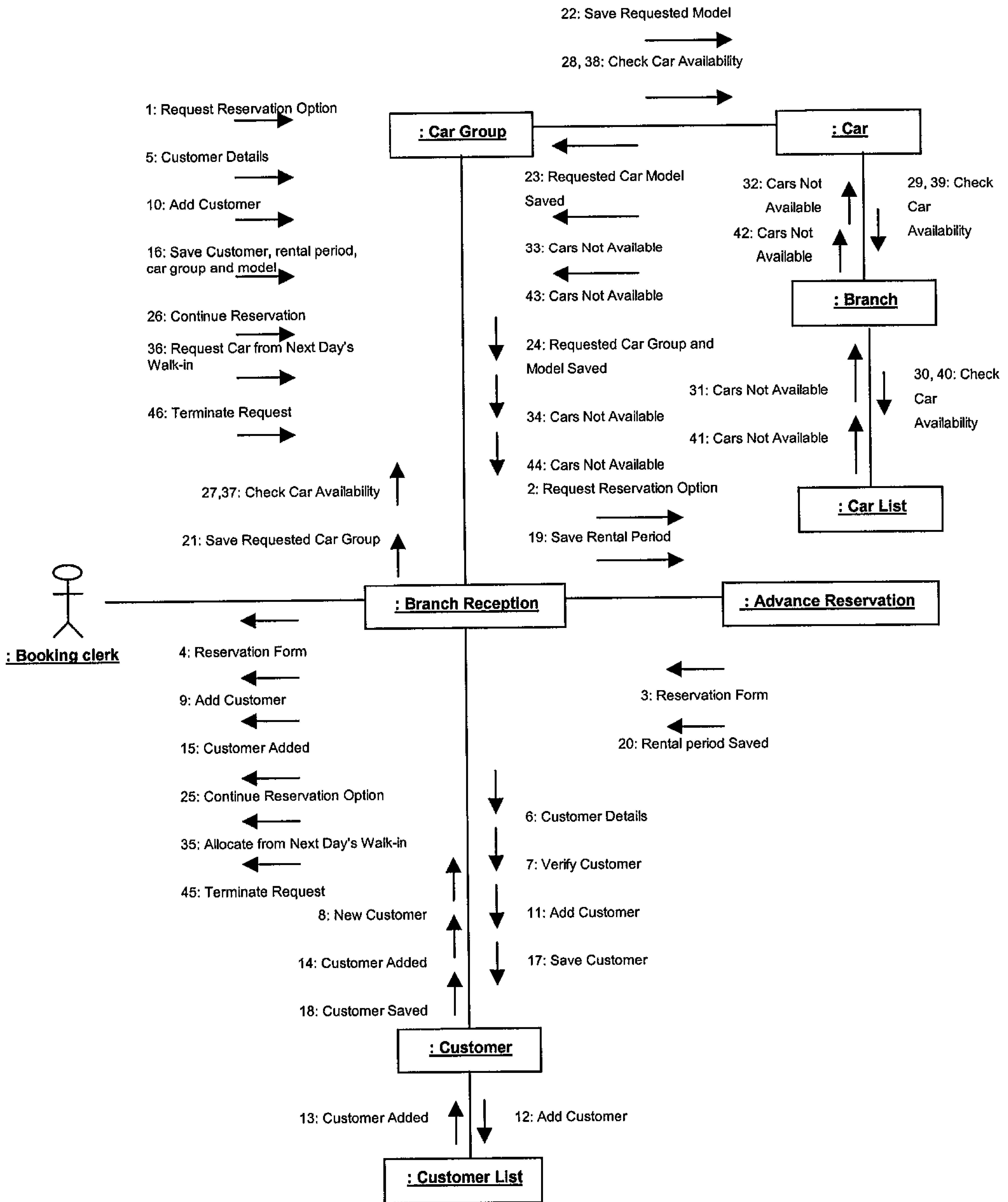


Figure 57: Collaboration for use case 01: Make reservation (Exception 13a1a1a)

Exception 13a1b2a: System informs that customer is not a scheme member

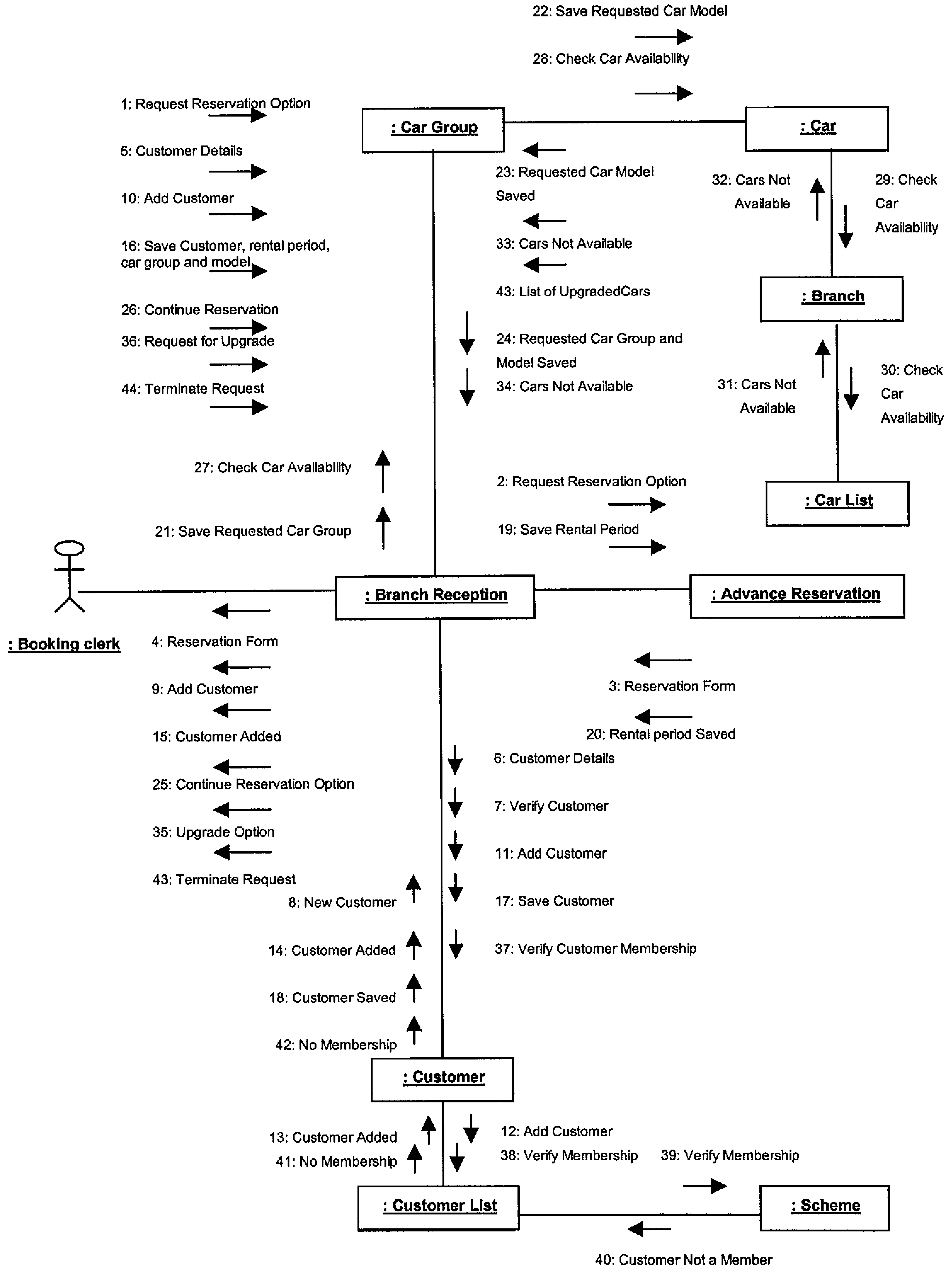


Figure 58: Collaboration for use case 01: Make reservation (Exception 13a1b2a)

Exception 13a1c2a: System informs that a car in the lower group is not available

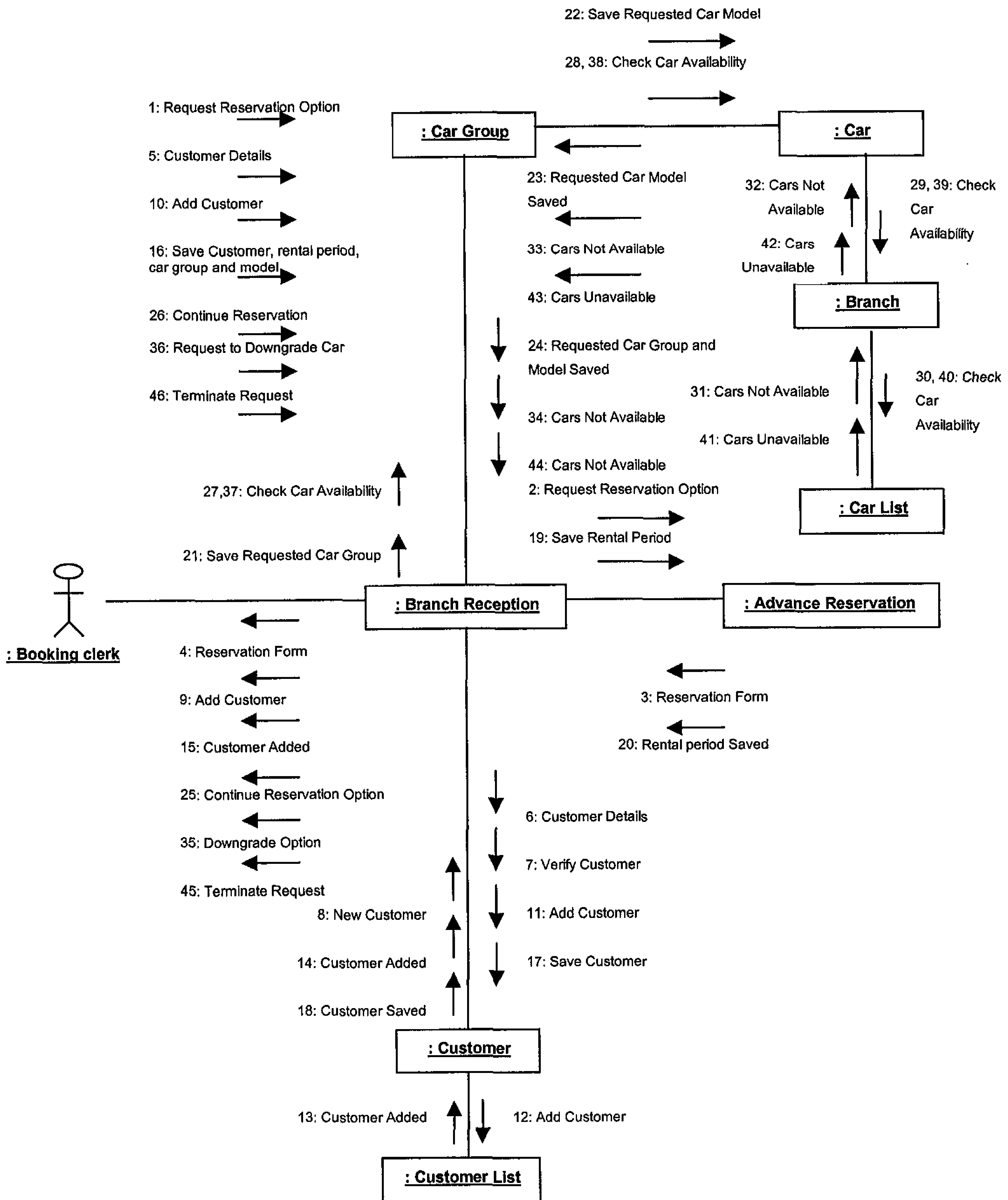


Figure 59: Collaboration for use case 01: Make reservation (Exception 13a1c2a)

Exception 13a1d2a: System informs that cars are not available for transfer

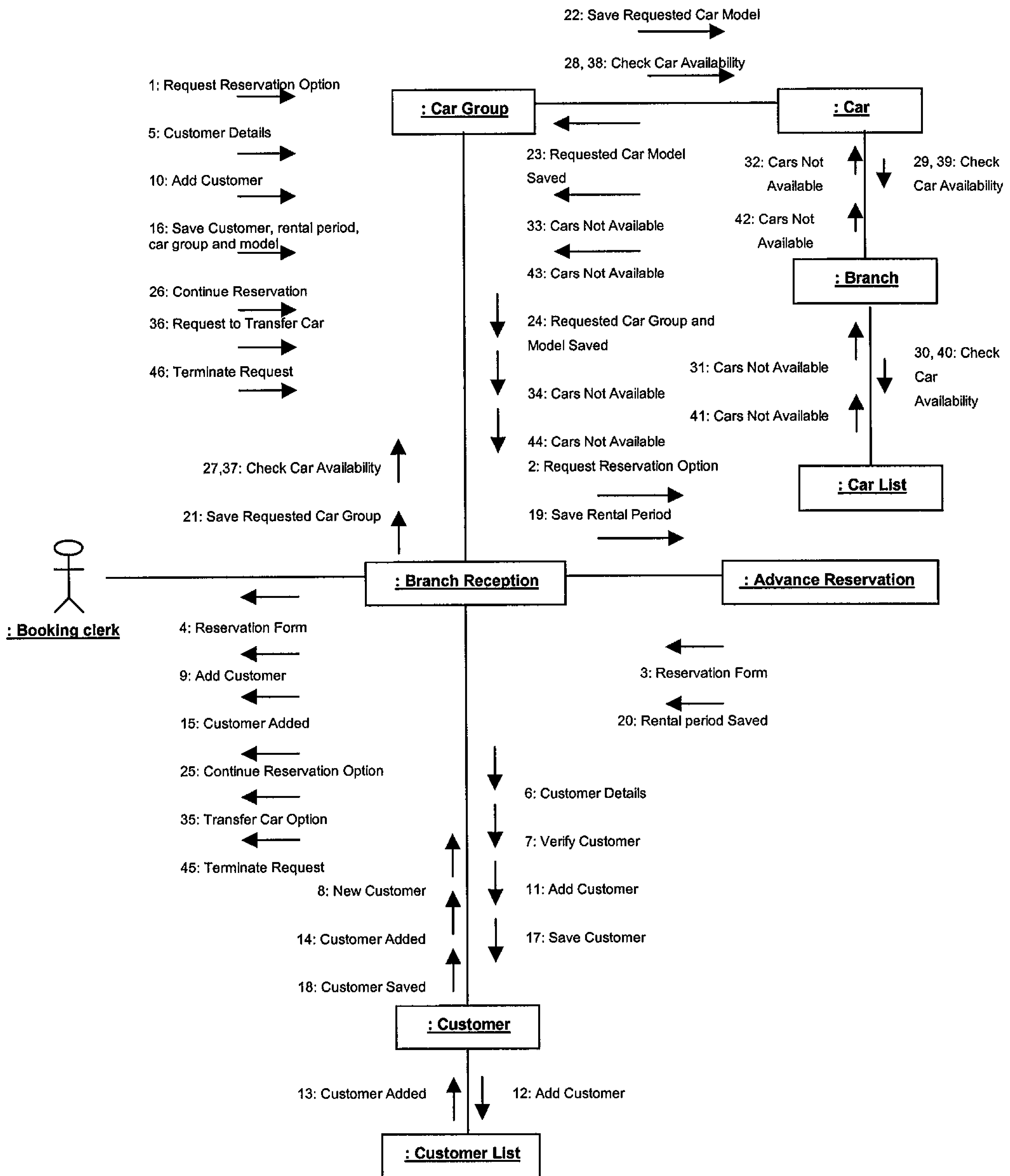


Figure 60: Collaboration for use case 01: Make reservation (Exception 13a1d2a)

Exception 13a1e2a: Customer confirms that the rental will take more than 10% of the service mileage

In this case, there is no interaction between the booking clerk and the system and hence no collaboration diagram involved with exception 13a1e2a.

Objects identified from use case 02

Table 8: Objects Identified for Use-case 02

Normal Course	
Object	**Component of Object**
Customer	Driver Licence, CustomerID
Customer Details	Part of Customer object
Reservation	
Booking clerk	
Inspection Officer	
Car	
Rental	Rental Mileage
Branch	
System	Branch Reception

Normal Course: Use case Rent Car

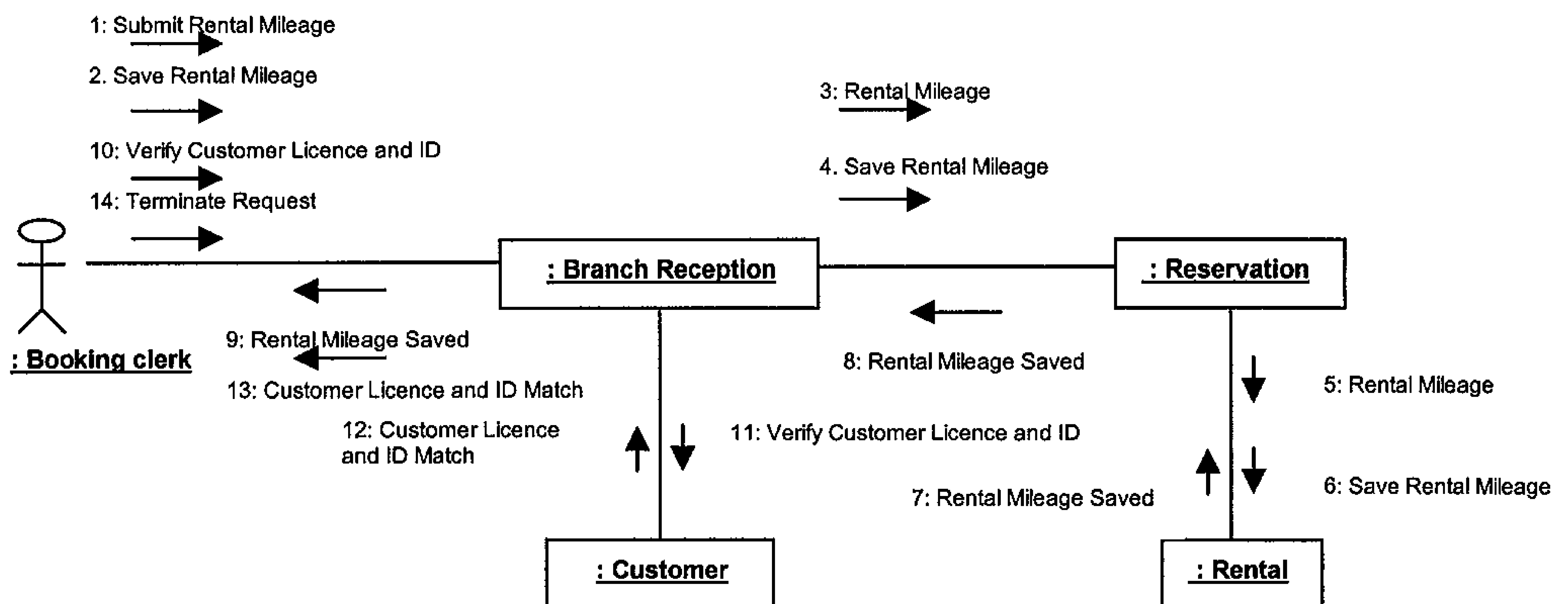


Figure 61: Collaboration for use case 02: Rent Car (Normal Course)

Exception: Customer's driver's licence is not valid

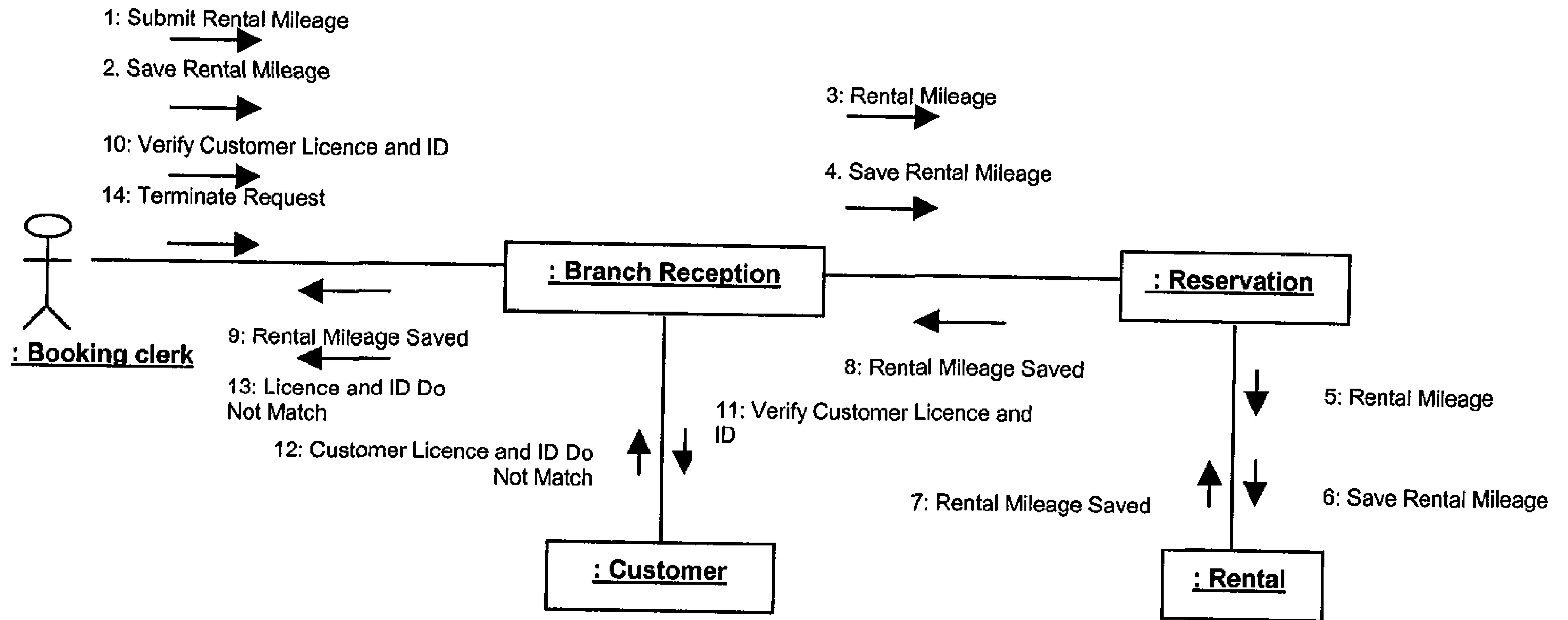


Figure 62: Collaboration for use case 02: Rent Car (Exception 4a)

Objects identified from use case 03

Table 9: Objects Identified for Use-case 03

Normal Course	
Object	**Component of Object**
Customer	
Inspection Officer	
Financial Clerk	
Branch Reception	Car Condition on Return Form, Invoice Form
Car	Condition
Rental	Rental Mileage, Rental Period, Rental Charge
Rental Details	Pick-up Time, Drop-off Time
Branch	Pick-up Branch, Drop-off Branch
Payment Details	

Normal Course: Use case Return Car

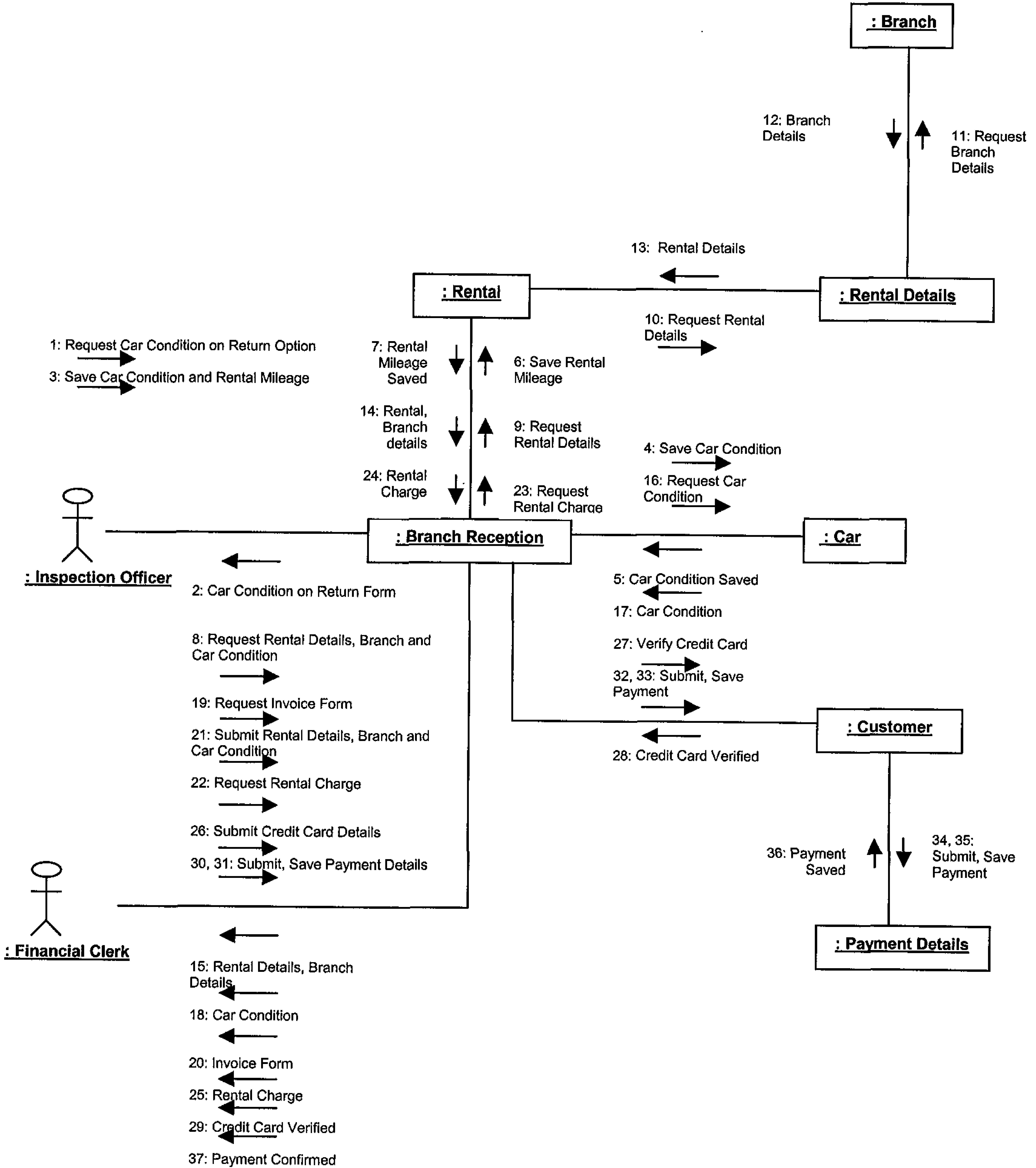


Figure 63: Collaboration for use case 03: Return Car (Normal Course)

Alternative 1a: Customer returns the car to a non-pick-up branch at the drop-off time

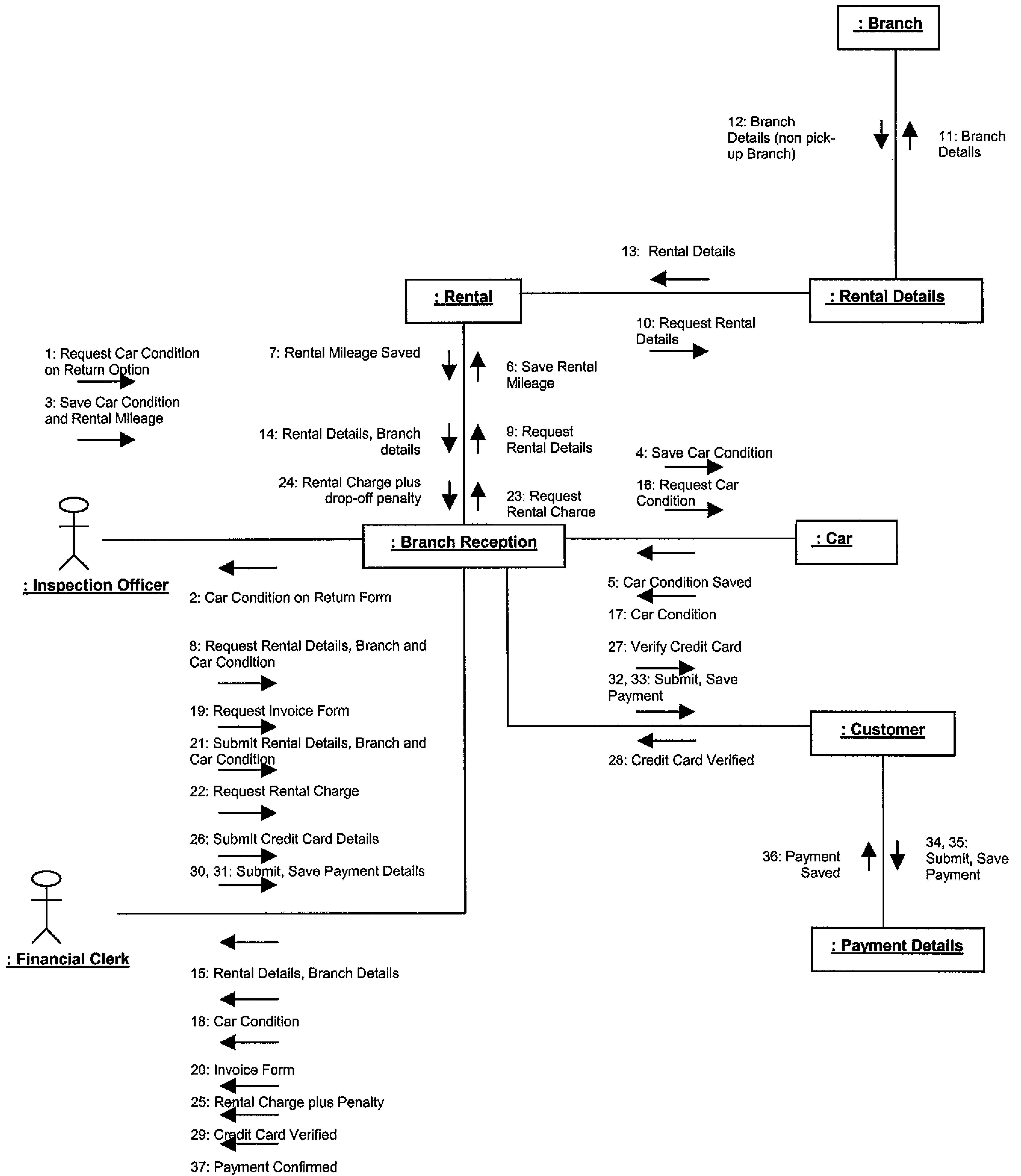


Figure 64: Collaboration for use case 03: Return Car (Alternative 1a)

Alternative 1b, c: Customer returns the car to a non-pick-up branch earlier than/ later than the drop-off time

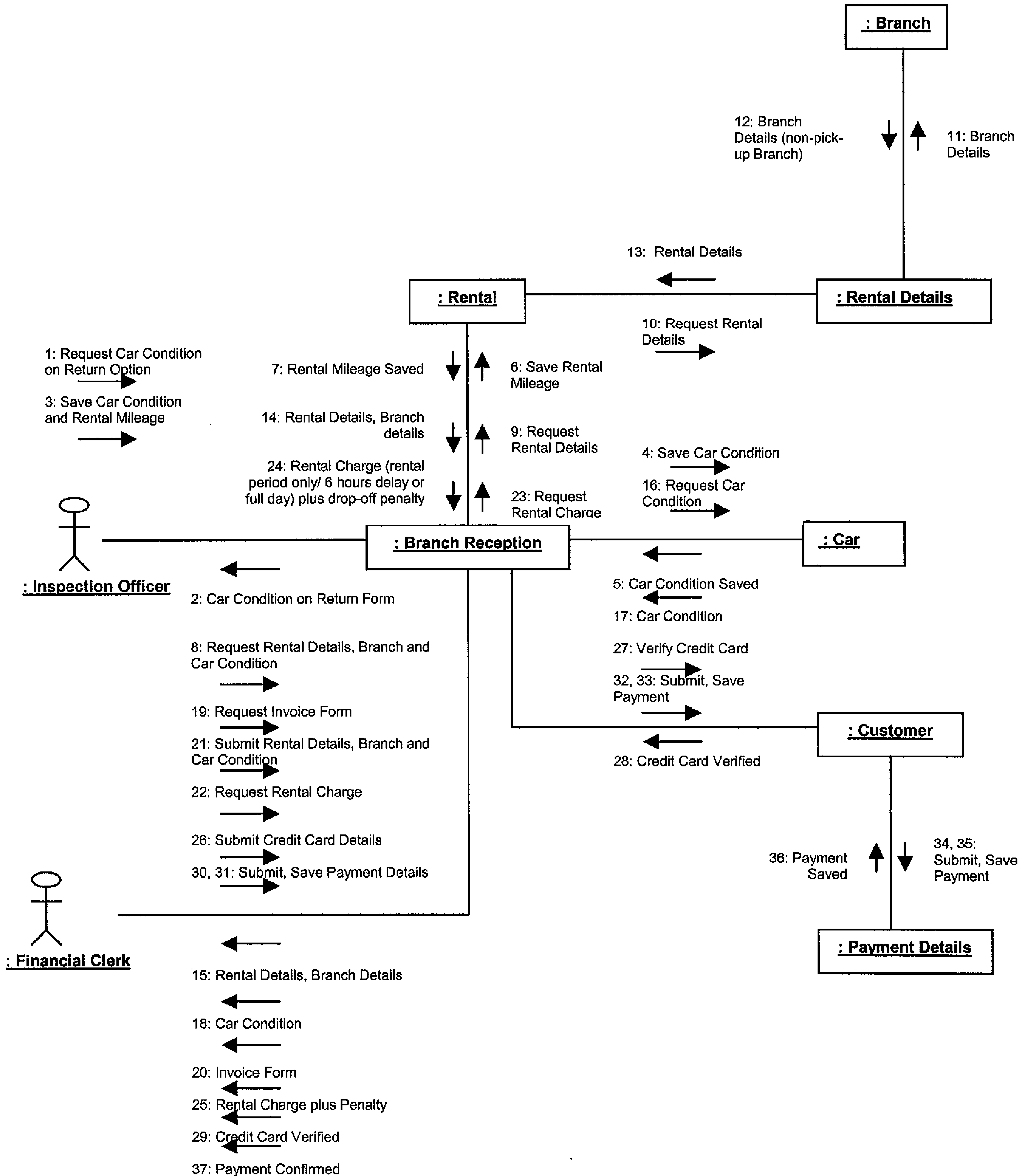


Figure 65: Collaboration for use case 05: Return Car (Alternative 1b, c)

Objects identified from use case 04

Table 10: Objects Identified for Use-case 04

Normal Course	
Object	**Component of Object**
Depot Manager	
Car	Mileage
Car List	
Depot	
Depot Capacity	
Branch	
System	Depot Reception
Service	Service date, time, return date and time, form, charge
Branch Manager	Does not interact with the system and therefore not modelled.
Branch Driver	Does not interact with the system and therefore not modelled.
Financial Clerk	Does not interact with the system and therefore not modelled.
Payment Details	Payment method, amount paid, service date and car mileage

Normal Course: Use case Book for Service

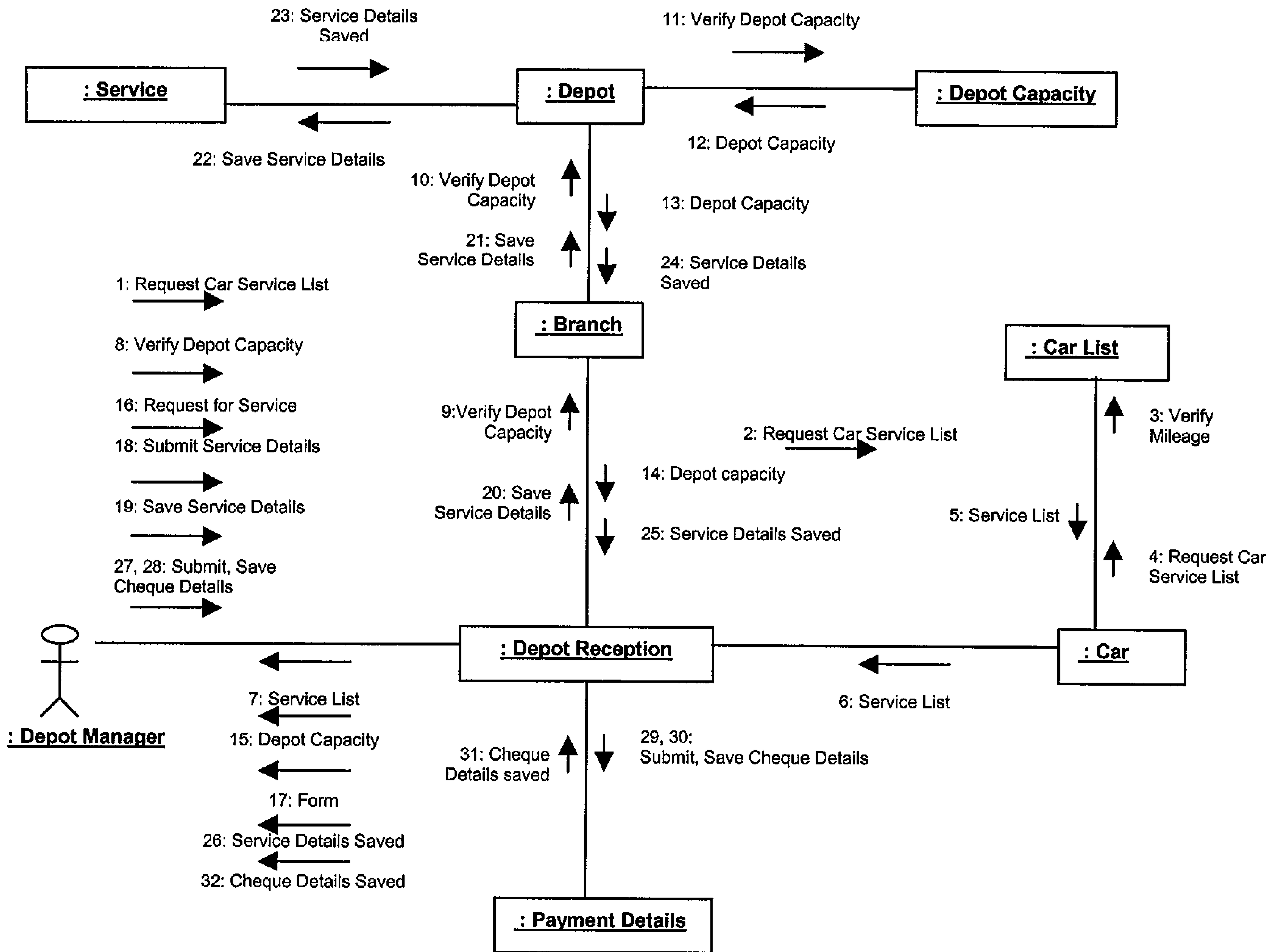


Figure 66: Collaboration for use case 04: Book for Service (Normal Course)

Alternative 2a: Use case Book for Service

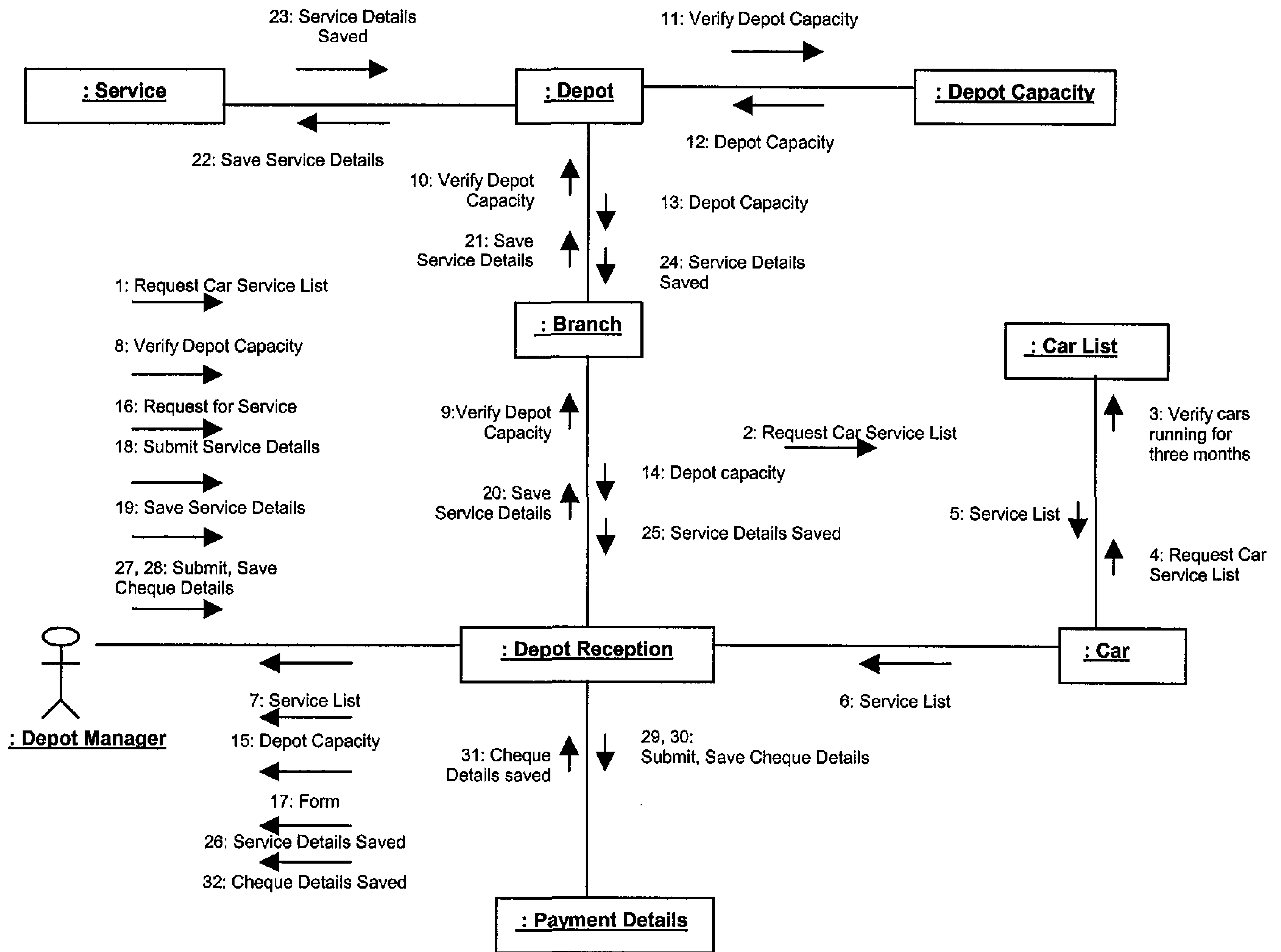


Figure 67: Collaboration for use case 04: Book for Service (Alternative 2a)

Alternative 15a: Use case Book for Service

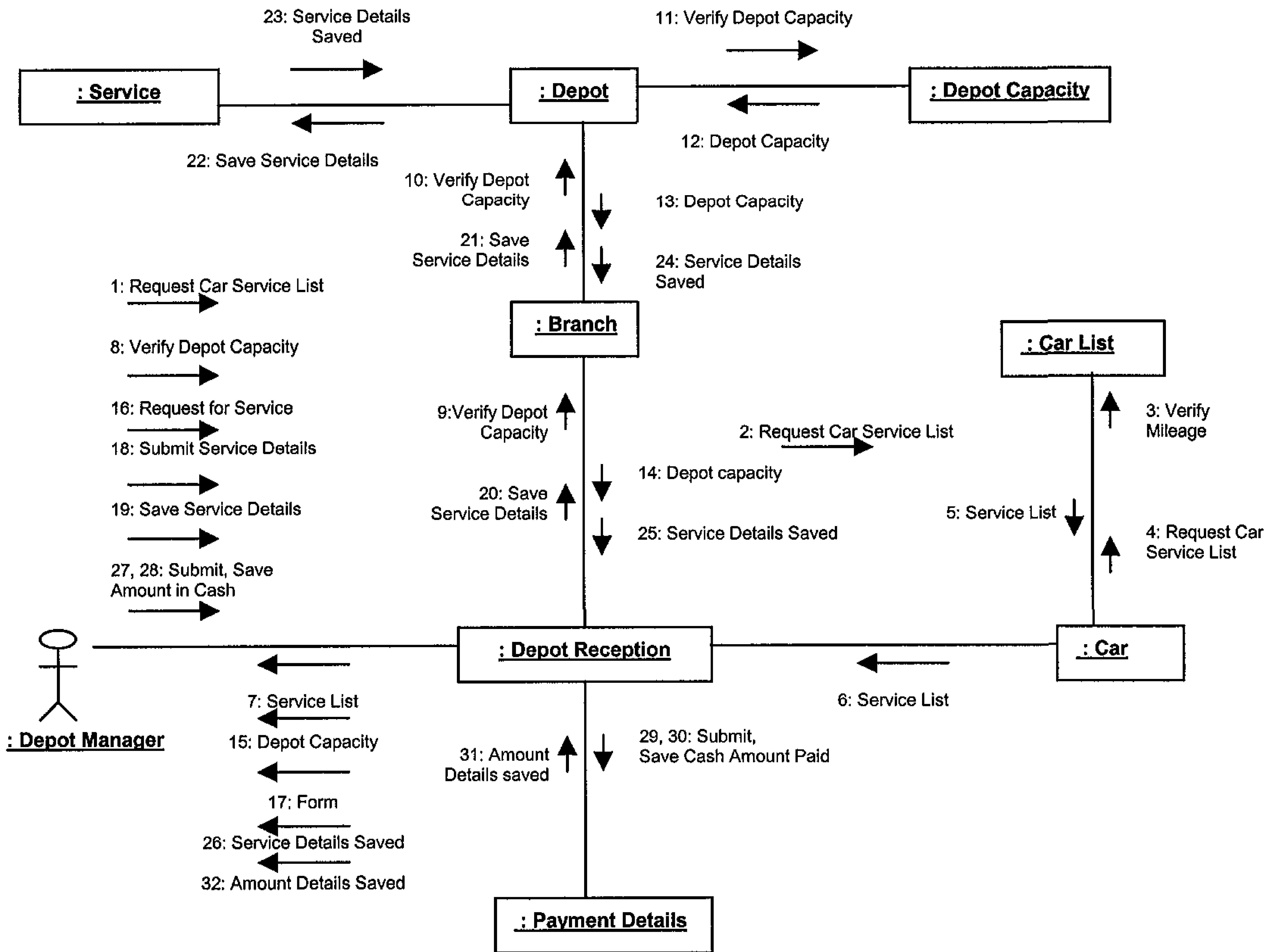


Figure 68: Collaboration for use case 04: Book for Service (Alternative 15a)

Exception: Use case Book for Service

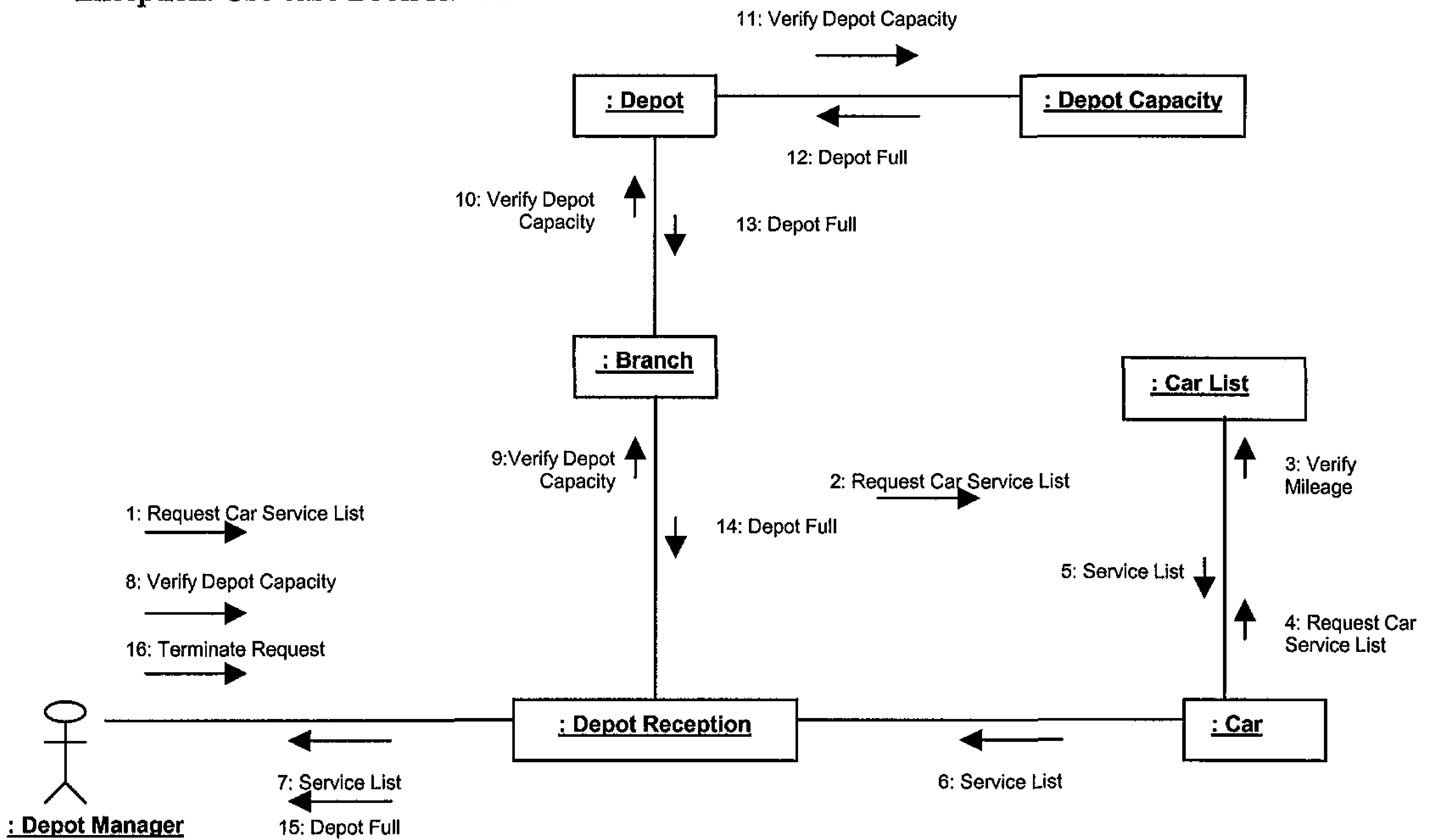


Figure 69: Collaboration for use case 04: Book for Service (Exception)

Objects identified from use case 05

Table 11: Objects Identified for Use-case 05

Normal Course	
Object	**Component of Object**
Branch Manager	
Car	Mileage
Authorised List	Model, Price
Branch Reception	
Buyer	Does not interact with the system and therefore not modelled

Normal Course: Use case Sell Cars

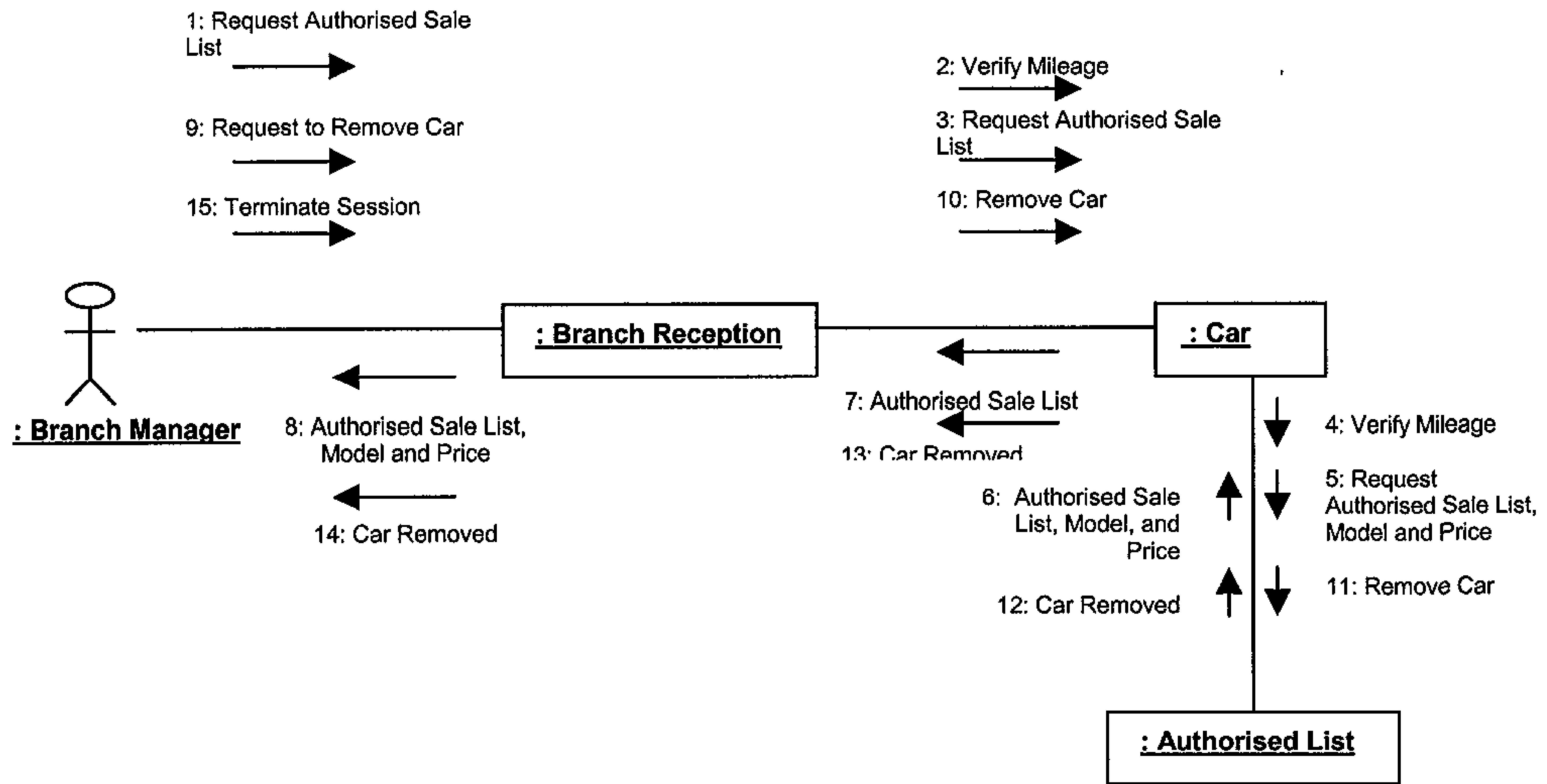


Figure 70: Collaboration for use case 05: Sell Cars (normal course)

Alternative 2a: Use case Sell Cars

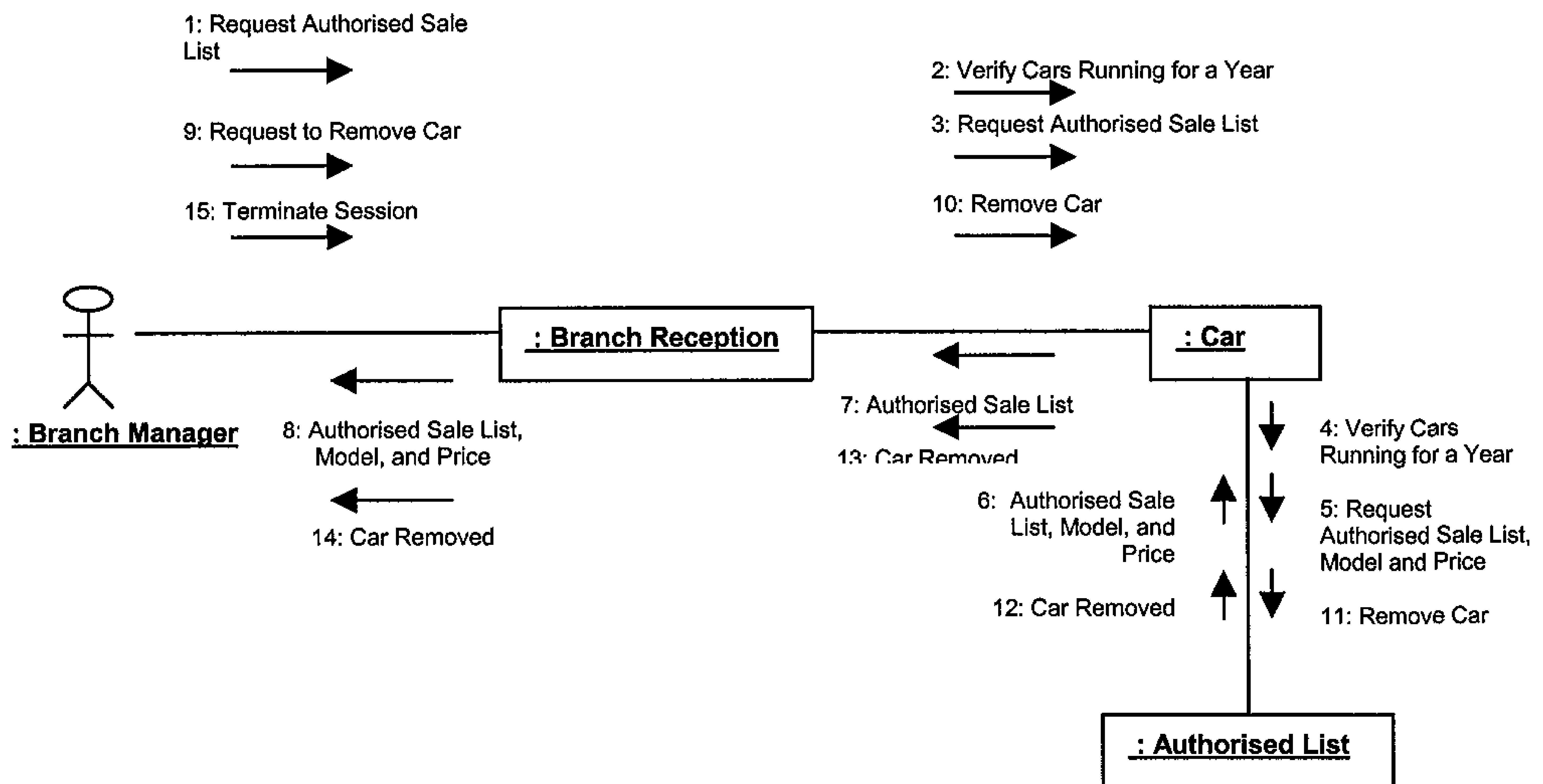


Figure 71: Collaboration for use case 05: Sell Cars (Alternative 2a)

Exception: Use case Sell Cars

There is no system interaction involved in the exception. Hence collaboration is not provided for this exception.

Having drawn the collaboration diagrams, the next step is to draw the domain model class diagram using the objects identified in the collaborations. This is done in the following section.

Initial list of candidate classes

Bahrami [1999] lists certain guidelines that are useful in the identification of classes. They are the following:

- Examine sources of documents to identify **things** that the organisation uses to maintain data such as CARS, CUSTOMERS etc.,
- Examine **business events** that the organisation must remember, such as accepting a rental request, servicing a car etc.,
- Examine the **roles** played by people in the system such as customer, manager etc.,
- Examine the **physical locations** that are relevant to the system such as branches, service depots etc.

Considering the above-mentioned guidelines, the following **classes** are identified from the collaboration diagrams.

DepotReception	AdvanceReservation
Depot	WalkinReservation
BranchReception	Scheme
Branch	Rental
Customer	PaymentDetails
CustomerInformation	Servicing

CustomerList	AuthorisedList
BlacklistedCustomer	Buyer
AuthorisedDriver	Car
AdditionalDriver	CarGroup
	CarList

Evaluation of classes

The candidate classes have to be evaluated to identify and eliminate irrelevant classes as well as to identify the class instances.

In the next iteration, CustomerInformation class can be combined with the Customer class because these two classes do not have separate behaviour. All the classes are significant, and they convey stand-alone meaning. Therefore these classes are included in the model.

Identification of Attributes

Attributes are unique identifiers of each class, and it is therefore important to identify them in the class diagram. Table 12 summarises the classes in the case study together with their identified attributes.

Table 12: List of Classes

Class	Attributes
Customer	CustomerID, Age, Address, LicenceNr, InsuranceNr
CustomerInformation	PreviousRentals, Payment, LateReturns, Damages
AuthorisedDriver	CreditCardNr, ContractNr
AdditionalDriver	AuthorisationFormNr
Branch	BranchNr, BranchCapacity, PickUpBranch, DropOffBranch, BranchManagerID, BranchDriverID, InspectionOfficerID, FinancialClerkID, BookingClerkID
Depot	DepotNr, DepotManagerID, DepotCapacity

Servicing	ServiceNr, ServiceDate, ServiceTime, ReturnDate, ReturnTime, ServiceFormNr
PaymentDetails	AmountPaid, PaymentType
Car	RegNr, CarModel, Condition
AuthorisedList	CarModel, CarPrice
Buyer	BuyerID
Rental	RentalMileage, RentalPeriod, RentalCharge, PickUpTime, DropOffTime
AdvanceReservation	ResNr, FutureReservationNr
WalkinReservation	ResNr
Scheme	SchemeNr

Associations in the case study

The following three types of relationships between classes have to be identified and depicted in the data model.

- **Associations** between classes are relationships that describe a set of links among the objects of the classes.
- **Generalisation** is a relationship between a superclass and a subclass.
- **Aggregation** is a simple association between two classes that represents the part/whole relationship.

Table 13 summarises the associations for the identified classes and their cardinalities.

Table 13: List of Associations

Class	Related Class(s)	Association Name	Cardinality
Branch	BranchReception	has	One to one
Depot	DepotReception	has	One to one
Branch	Branch	request_transfer	One to many

Depot	Branch	service	One to many
Servicing	PaymentDetails	includes	One to one
Buyer	AuthorisedList	purchase_from	One to many
Car	Branch	returned_to	One to one
Branch	Car, Rental	classifies	Many on all associations
Customer	Rental	pays	One to one
Rental	AdvanceReservation	has	One to one
Rental	PaymentDetails	includes	One to one
Customer	AdvanceReservation	make	One to many
Customer	AdvanceReservation, Scheme	upgraded_for	Customer (many), Scheme (one), AdvanceReservation (one)
Customer	WalkinReservation	make	One to many
Customer	CarGroup	request	One to one

Generalisations in the case study

Table 14 summarises the superclasses and related subclasses.

Table 14: List of generalisations

Superclass	Subclass
AdvanceReservation	WalkinReservation
Customer	AuthorizedDriver AdditionalDriver

After identifying the classes and their attributes, associations between related classes and generalisations, the UML class diagram can now be constructed. Figure 72 represents the domain model (benchmark) class diagram for the case study.

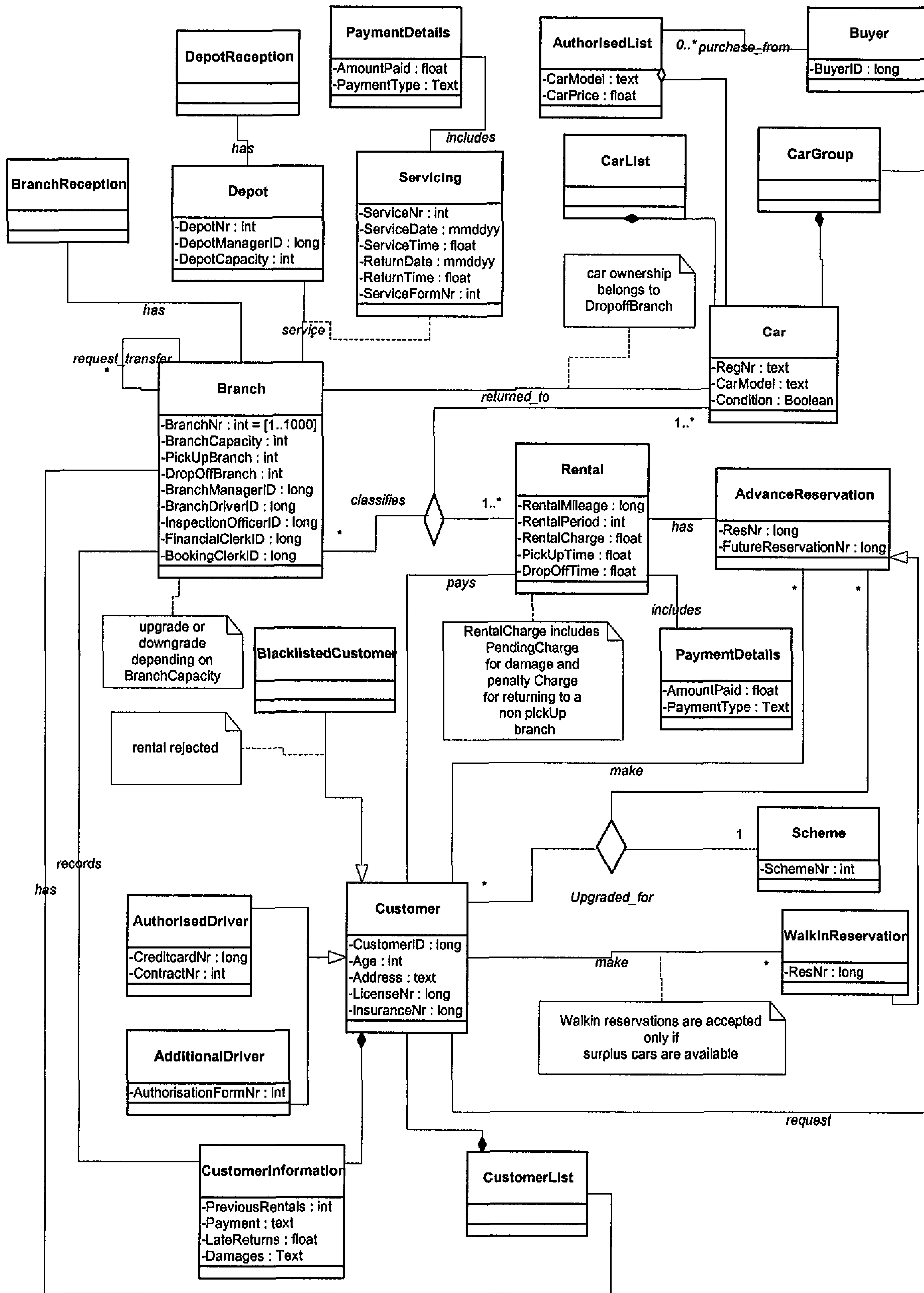


Figure 72: UML Domain Model Class Diagram

Appendix C (ORM Conceptual Schema)

Requirements Analysis Using ORM

The conceptual schema of the EU Car Rental case study is developed using the Conceptual Schema Design Process for ORM listed by Halpin [2001].

Conceptual Schema Design Process

This section describes the process followed to design a conceptual schema in ORM. The main steps are given, followed by the application of the technique to the Car Rental case study. The Conceptual Schema Design Process starts by identifying the fact types in the system, followed by the addition of constraints to the fact types. Checking is done to identify derived facts. The schema is built by the verbalisation of elementary facts, population checks and careful analysis of business rules.

Step 1
Transform familiar information examples into elementary facts and apply quality checks.

Step 2
Draw the fact types and apply a population check.

Step 3
Check for entity types that should be combined and note any arithmetic derivations.

Step 4
Add uniqueness constraints to fact types.

Step 5

Add mandatory role constraints and check for logical derivations.

Step 6

Add value and subtyping constraints.

Step 7

Add other constraints (if any) and perform final checks.

Step 1 is very important, as facts form the backbone of the ORM schema. An elementary fact cannot be split into smaller facts that collectively provide the same meaning as the original fact. Elementary facts do not use logical connectives like **not, and**, **or** and **if** or logical quantifiers like **all / some.** Applying this rule to the case study, the following elementary facts are drawn up. An elementary fact must assert that a particular object has a property, or that one or more objects participate in a relationship, where that relationship cannot be expressed as a conjunction of simpler facts without introducing new object types. F1 – F43 represent elementary facts. Object terms are enclosed in square brackets, with reference schemes in brackets and predicates shown in bold.

F1 [Branch *(branchNr)*] **classifies / classified by** [Car *(regNr)*] **for** [Rental *(rentalNr)*]

F2 [Branch *(branchNr)*] **has / belongs to** [Manager *(managerID)*]

F3 [Branch *(branchNr)*] **has / belongs to** [Clerk *(clerkID)*]

F4 [Rental *(rentalNr)*] **made by / belongs to** [Reservation *(resNr)*]

F5 [Reservation *(resNr)*] **identified by / specified for** [Car *(regNr)*]

F6 [Car *(regNr)*] **has / belongs to** [CarGroup]

F7 [Car *(regNr)*] **has / belongs to** [CarModel]

F8 [Manager *(managerID)*] **request** [Branch *(branchNr)*] **for** [Car *(regNr)*]

F9 [Car *(regNr)*] **returned to / accepts** [Branch *(branchNr)*]

F10 [Branch *(branchNr)*] **owns / belongs to** [Car *(regNr)*]

F11 [Branch *(branchNr)*] **has / serves** [Depot *(depotNr)*]

F12 [Depot (*depotNr)*] **has / belongs to** [DepotCapacity]

F13 [Customer *(custID)*] **make / belongs to** [Reservation *(resNr)*]

F14 [Branch *(branchNr)*] **records / captured at** [Customer *(custID)*]

F15 [Customer *(custID)*] **has / belongs to** [CustomerDetails]

F16 [CustomerDetails] **include** [Rental *(rentalNr)*]

F17 [CustomerDetails] **include** [LateReturns]

F18 [CustomerDetails] **include** [PaymentProblem]

F19 [CustomerDetails] **include** [CarDamage]

F20 [Customer *(custID)*] **has / belongs to** [License *(licenseNr)*]

F21 [Customer *(custID)*] **has / belongs to** [Insurance *(insuranceNr)*]

F22 [Customer *(custID)*] **has / belongs to** [Age]

F23 [Car *(regNr)*] **meet / checked for** [Condition]

F24 [Customer *(custID)*] **requests** [Car *(regNr)*] **with** [CarModel]

F25 [Reservation *(resNr)*] **accepted up to** [BranchCapacity *(branchNr)*]

F26 [Branch *(branchNr)*] **upgrade** [Reservation *(resNr)*]

F27 [Branch *(branchNr)*] **reserves / reserved by** [Car *(regNr)*] **for** [WalkinReservation *(resNr)*]

F28 [Rental (rentalNr)] **includes / included in** [RentalCharge]

F29 [Customer *(custID)*] **pays** [RentalCharge]

F30 [Rental *(rentalNr)*] **refused for** [Customer *(blacklisted)*]

F31 [Customer *(custID)*] **belongs to** [Scheme *(schemeNr)*] **upgraded for** [Reservation *(resNr)*]

F32 [Branch *(branchNr)*] **downgrade** [Reservation *(resNr)*]

F33 [Rental *(rentalNr)*] **has** [PickupTime]

F34 [Rental *(rentalNr)*] **has** [DropoffTime]

F35 [Customer *(custID)*] **has** [CreditCard *(cardNr)*] **sign** [Contract *(contractNr)*]

F36 [Customer *(custID)*] **sign** [AuthorisationForm *(formNr)*]

F37 [Car *(regNr)*] **checked for** [Damage] **from** [Rental *(rentalNr)*]

F38 [Rental *(rentalNr)*] **adds** [Penalty] **for** [Branch *(branchNr)*]

F39 [Rental *(rentalNr)*] **adds** [Pending] **for** [Damage]

F40 [RentalCharge] **calculated at** [RentalPeriodRate]

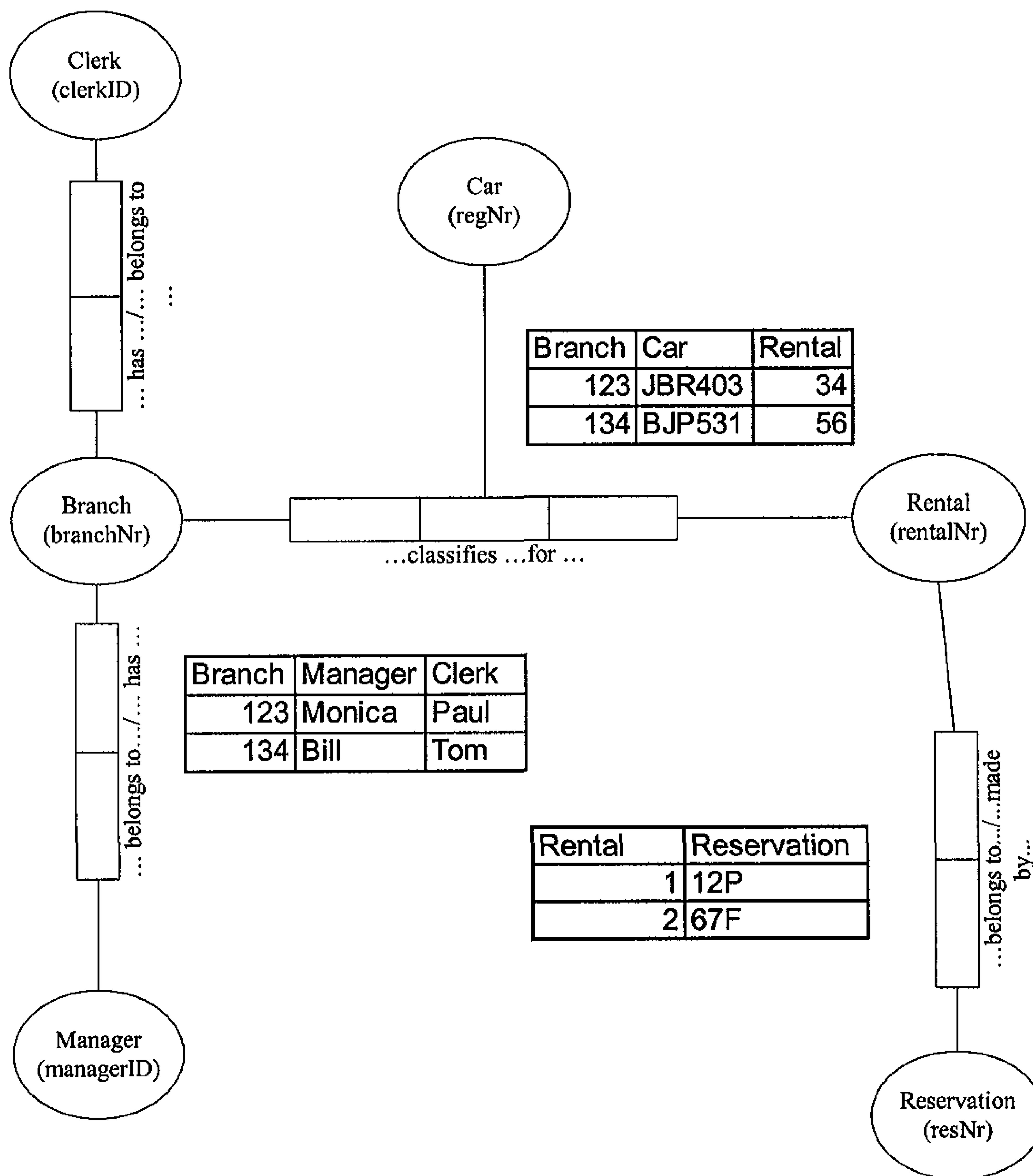
F41 [Branch *(branchNr)*] **contact** [Customer *(custID)*] **on** [LateReturn]

F42 [Sale *(saleNr)*] **arranged for** [Car *(regNr)*]

F43 [Buyer*(BuyerID)*] **purchase** [Car *(regNr)*] **from** [AuthorisationList *(ListNr)*]

Step 2 of the Schema Design Process is to draw the fact types that are identified and apply a population check. Population checking helps to check the correctness of a schema, and is done by adding a fact table of each fact type and entering values in the appropriate columns. It is not necessary to depict a fact table for each and every fact, and it may be omitted from the schema if it is not necessary to show one. Verbalising a fact from a fact table ensures that the schema is logically correct.

Facts 1, 2, 3 and 4 yield the following schema diagram where the constraints are omitted.



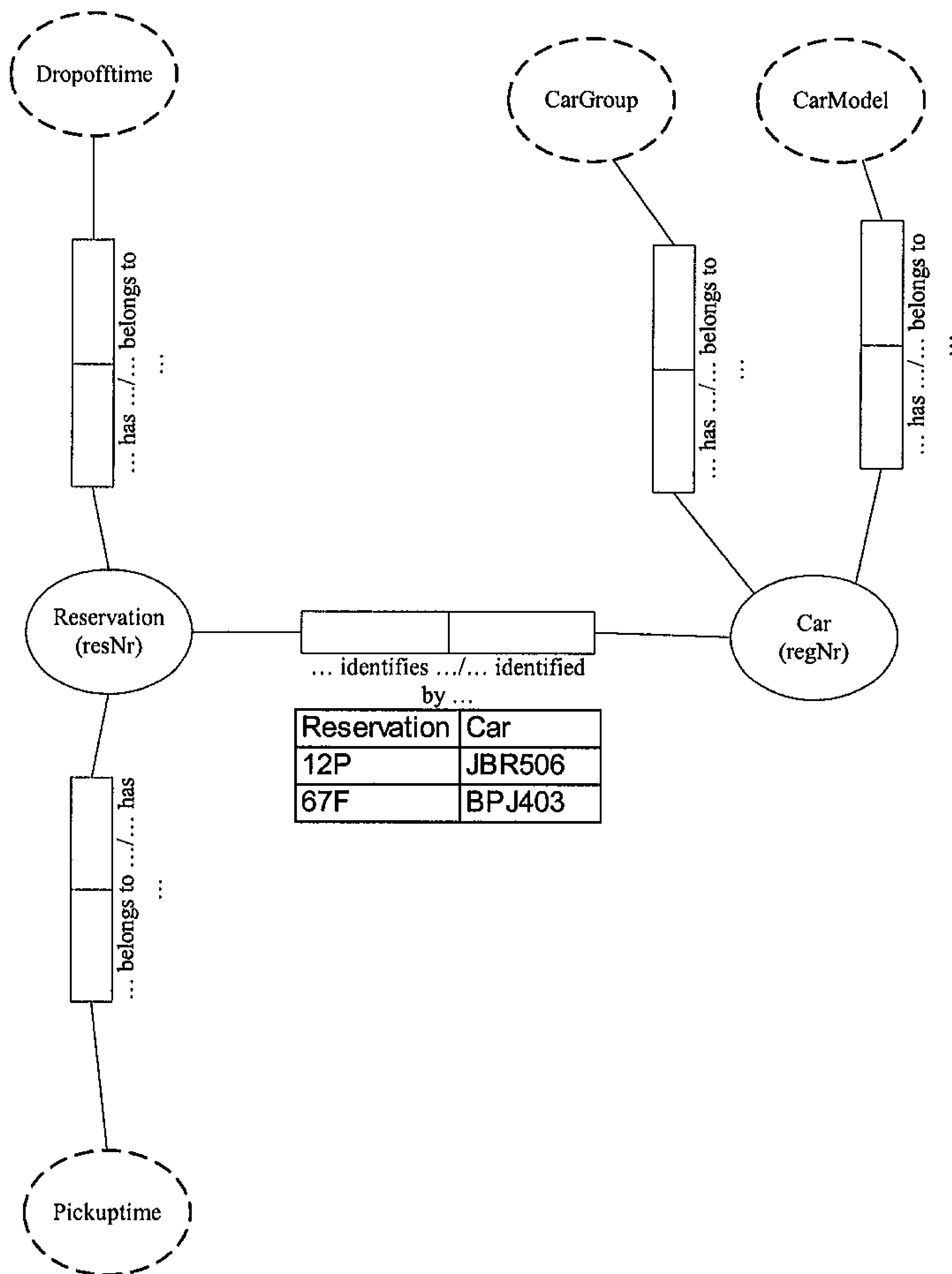
Branch	Car	Rental
123	JBR403	34
134	BJP531	56

Branch	Manager	Clerk
123	Monica	Paul
134	Bill	Tom

Rental	Reservation
1	12P
2	67F

Figure 73: Facts 1, 2, 3 and 4

Facts 5, 6, 7, 33 and 34 yield the following schema diagram:



Reservation	Car
12P	JBR506
67F	BPJ403

Figure 74: Facts 5, 6, 7, 33 and 34

Facts 8, 9, 10, 11 and 12 yield the following schema:

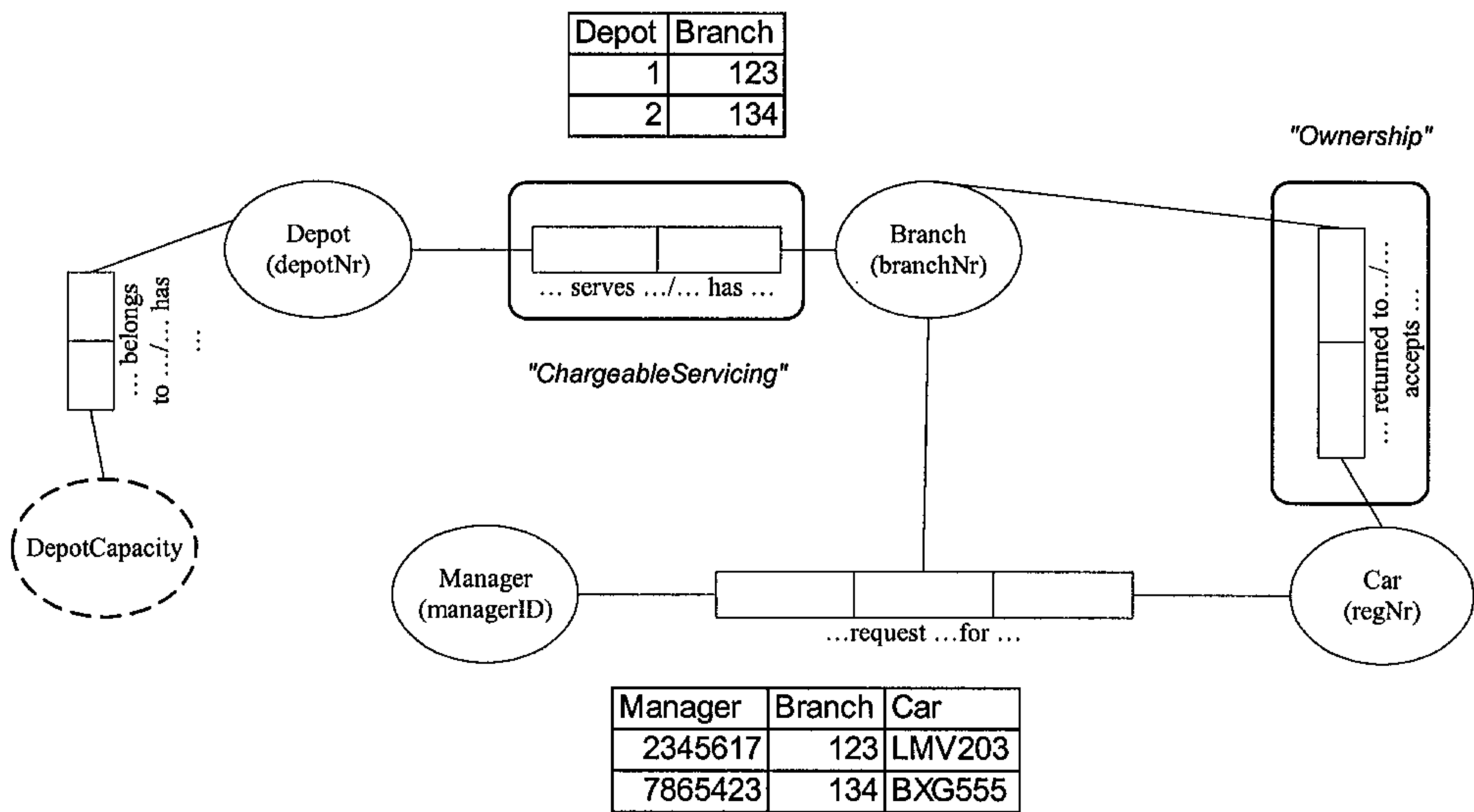


Figure 75: Facts 8, 9, 10, 11 and 12

Facts 13, 14 and 15 yield the following schema:

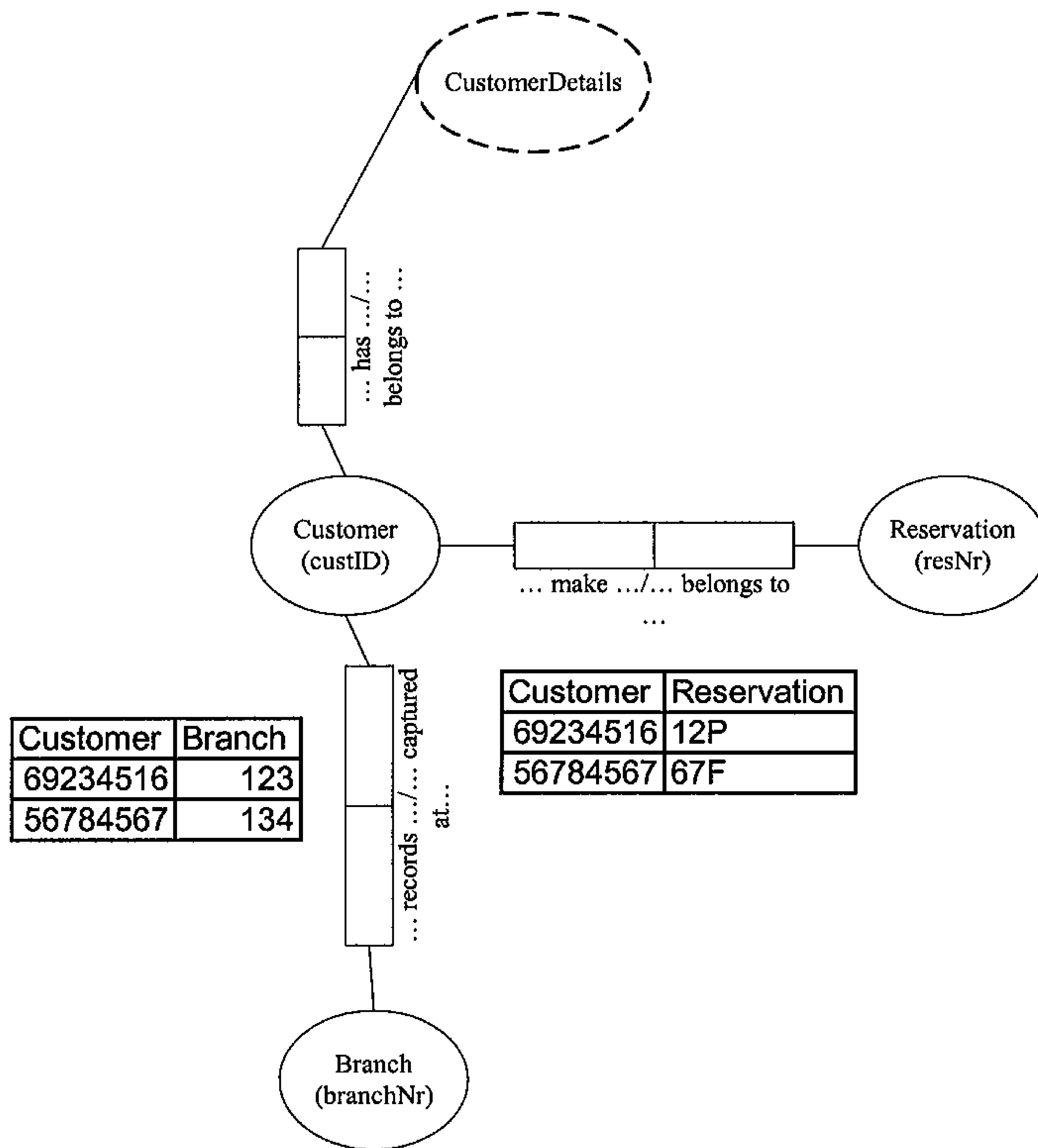


Figure 76: Facts 13, 14 and 15

Facts 16, 17, 18, 19, 20, 21 and 22 yield the following schema:

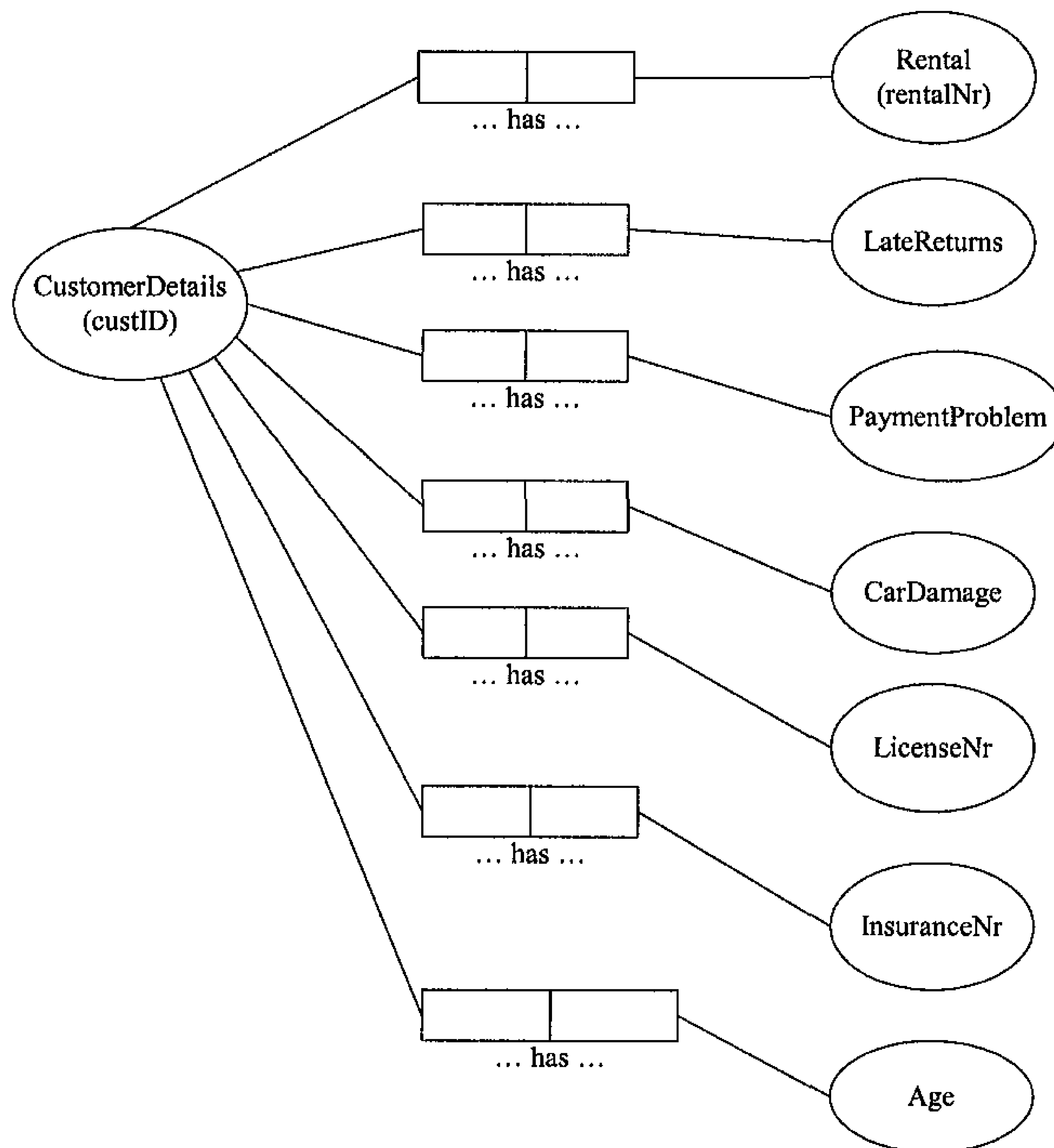
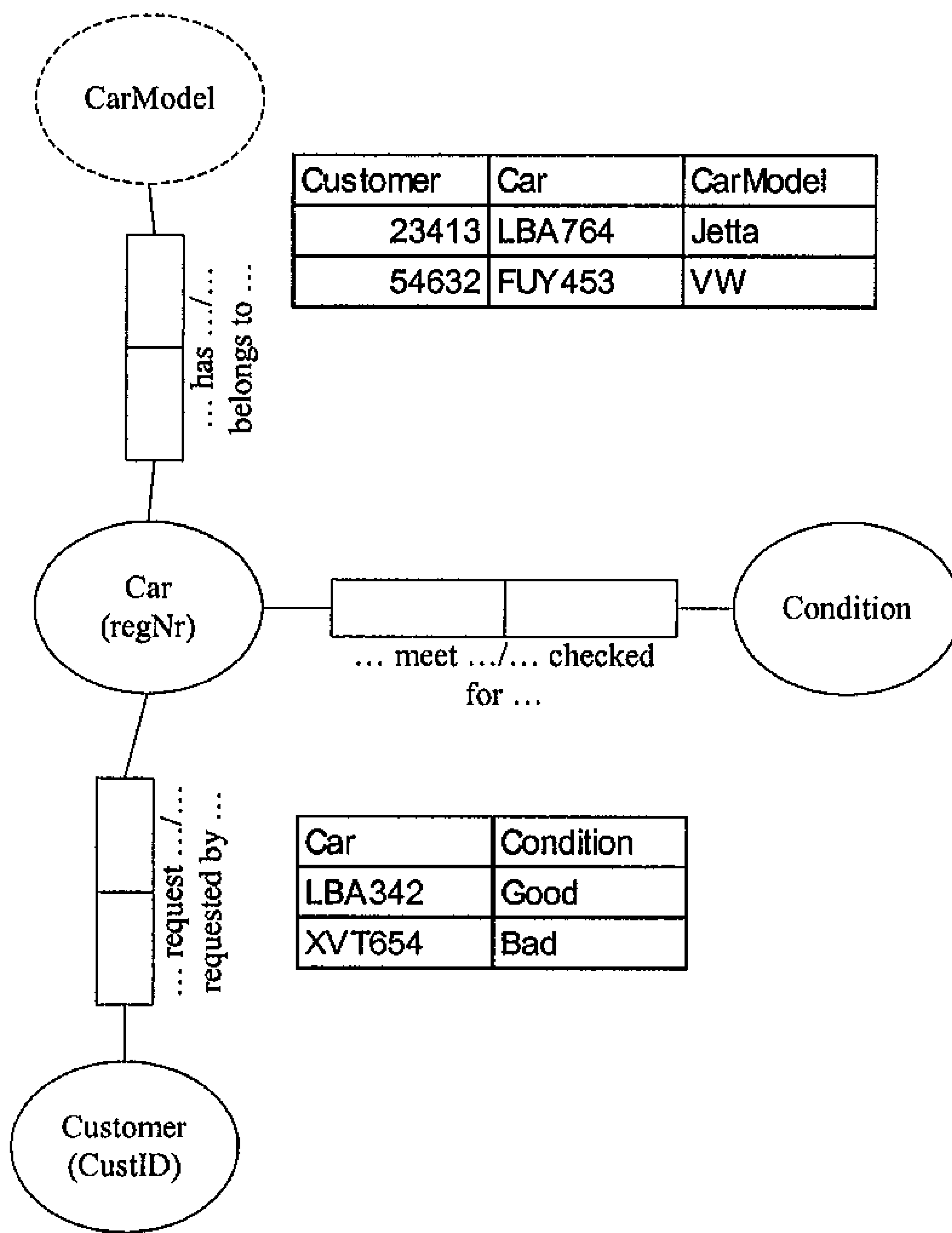


Figure 77: Facts 16, 17, 18, 19, 20, 21 and 22

Facts 23 and 24 yield the following schema:



Customer	Car	CarModel
23413	LBA764	Jetta
54632	FUY453	VW

Car	Condition
LBA342	Good
XVT654	Bad

Figure 78: Facts 23 and 24

Facts 25, 26 and 27 yield the following schema:

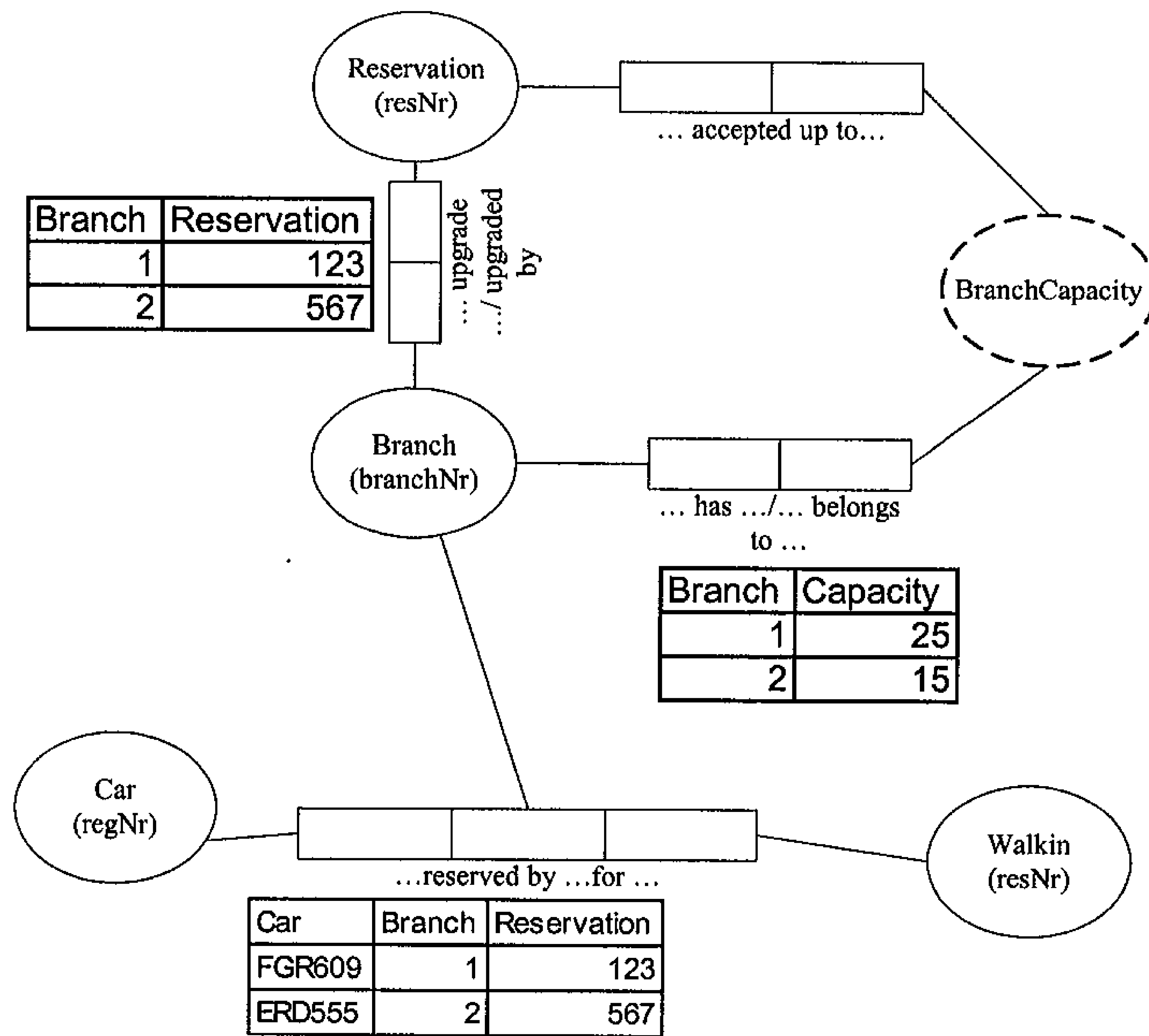
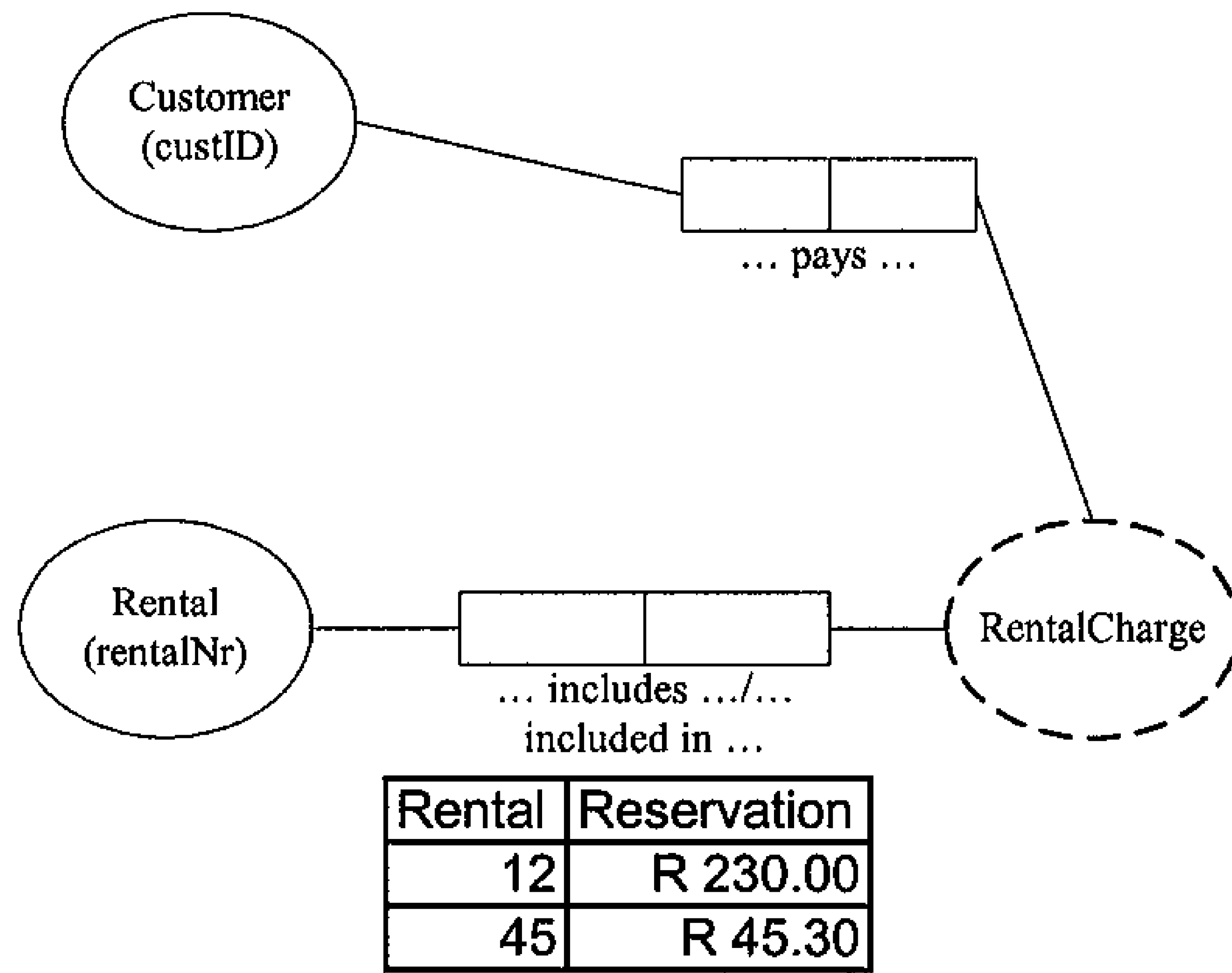


Figure 79: Facts 25, 26 and 27

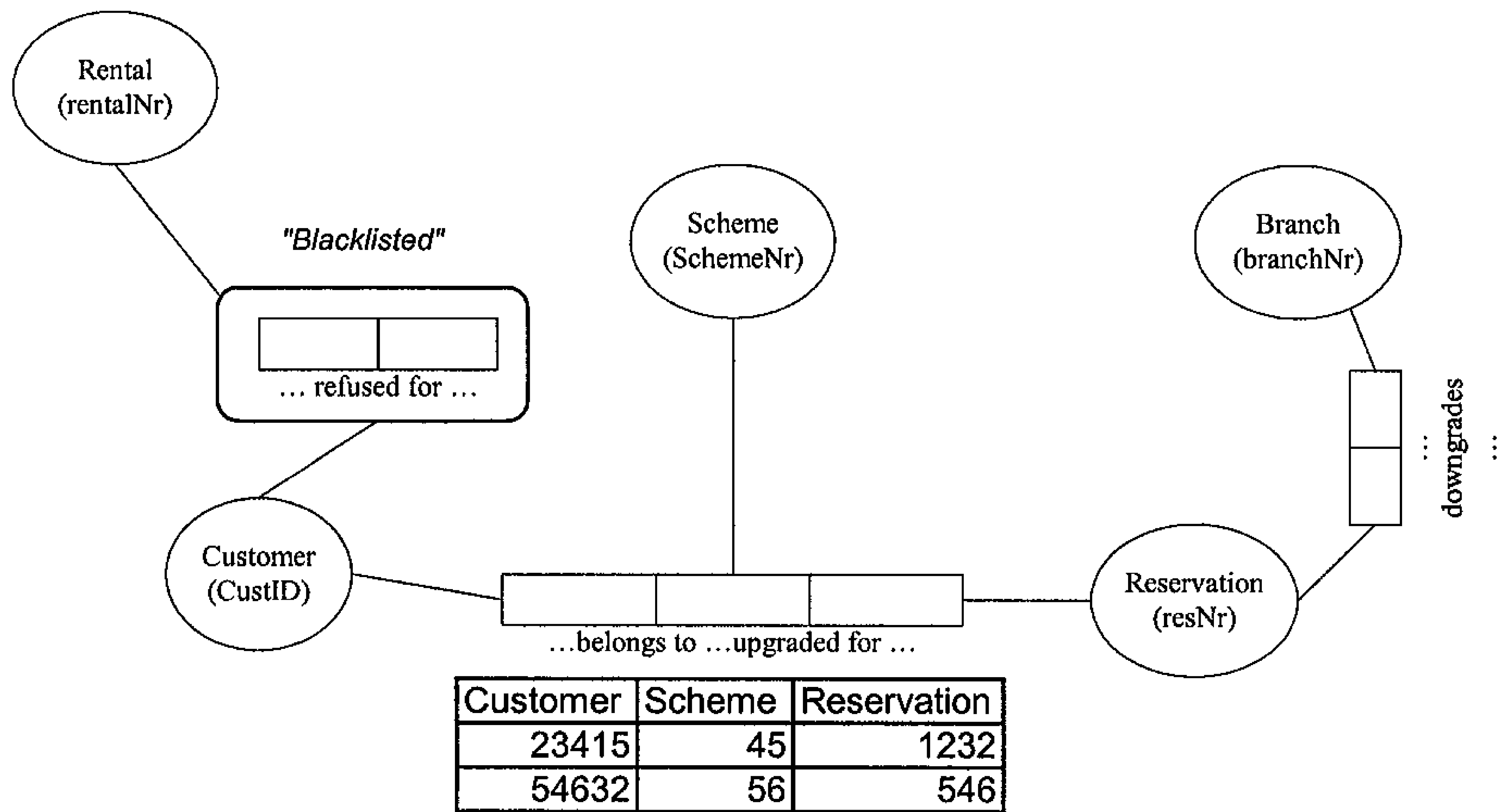
Facts 28 and 29 yield the following schema:



Rental	Reservation
12	R 230.00
45	R 45.30

Figure 80: Facts 28 and 29

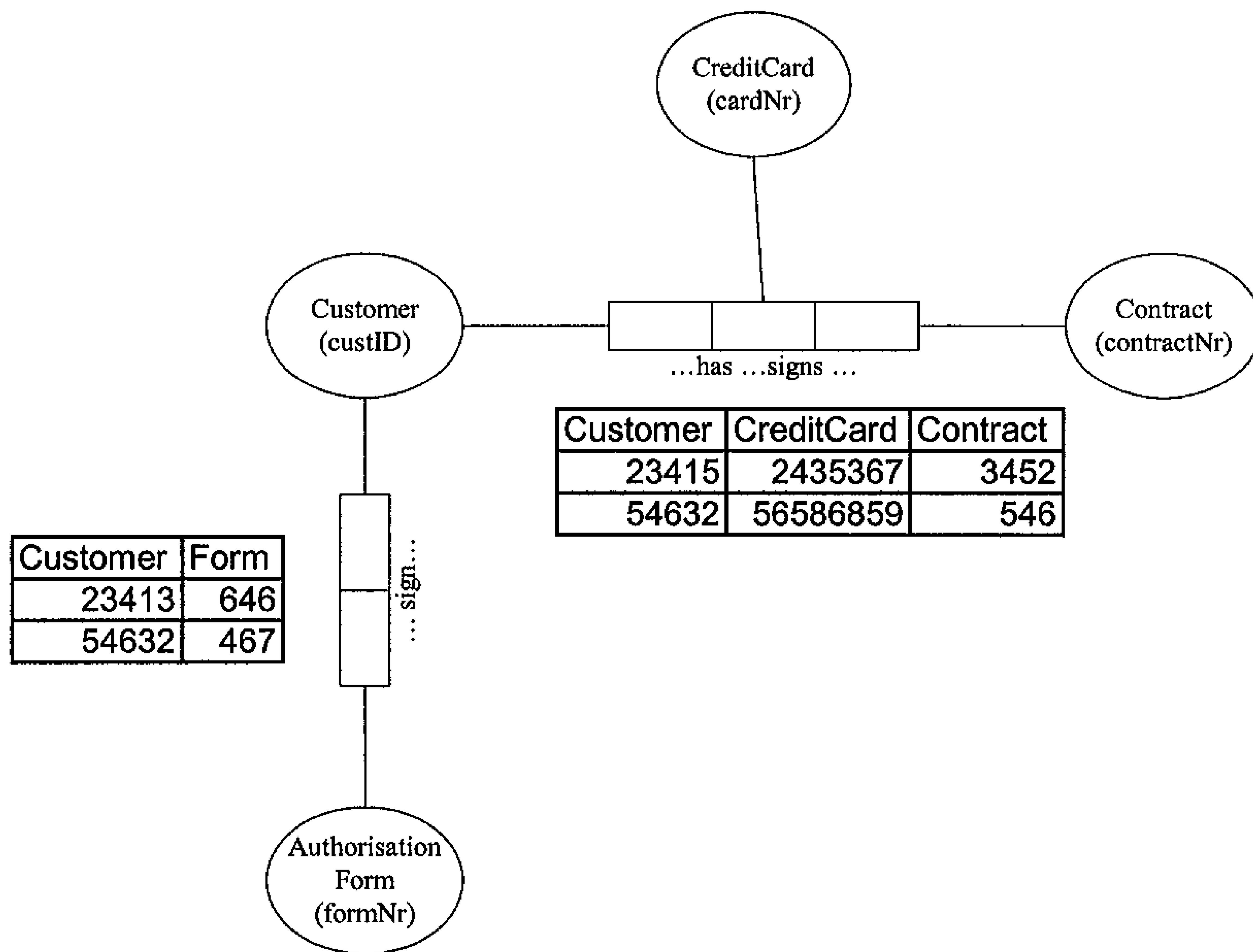
Facts 30, 31 and 32 yield the following schema:



Customer	Scheme	Reservation
23415	45	1232
54632	56	546

Figure 81: Facts 30, 31 and 32

Facts 35 and 36 yield the following schema:

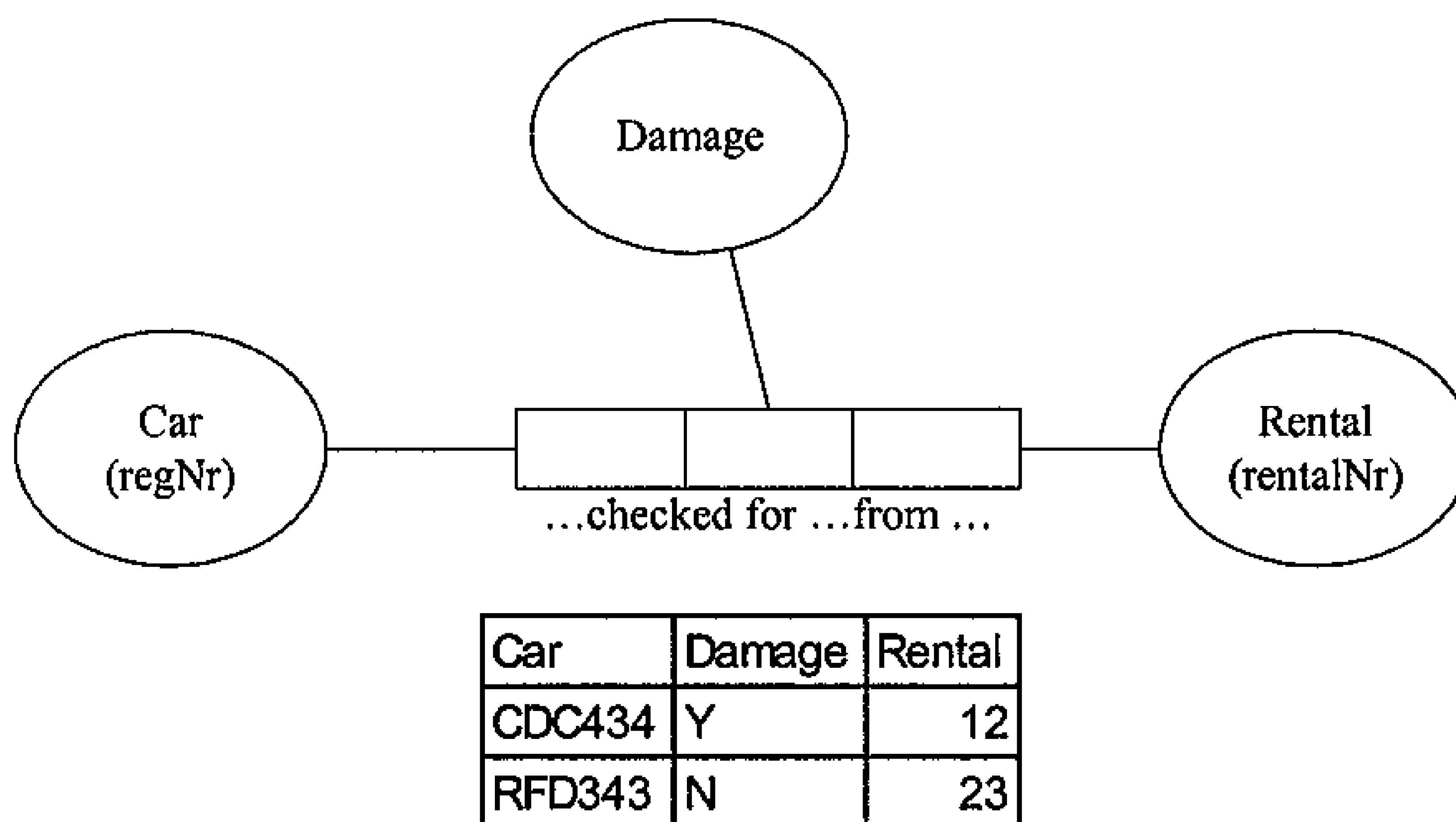


Customer	CreditCard	Contract
23415	2435367	3452
54632	56586859	546

Customer	Form
23413	646
54632	467

Figure 82: Facts 35 and36

Fact 37 is captured in figure 83.

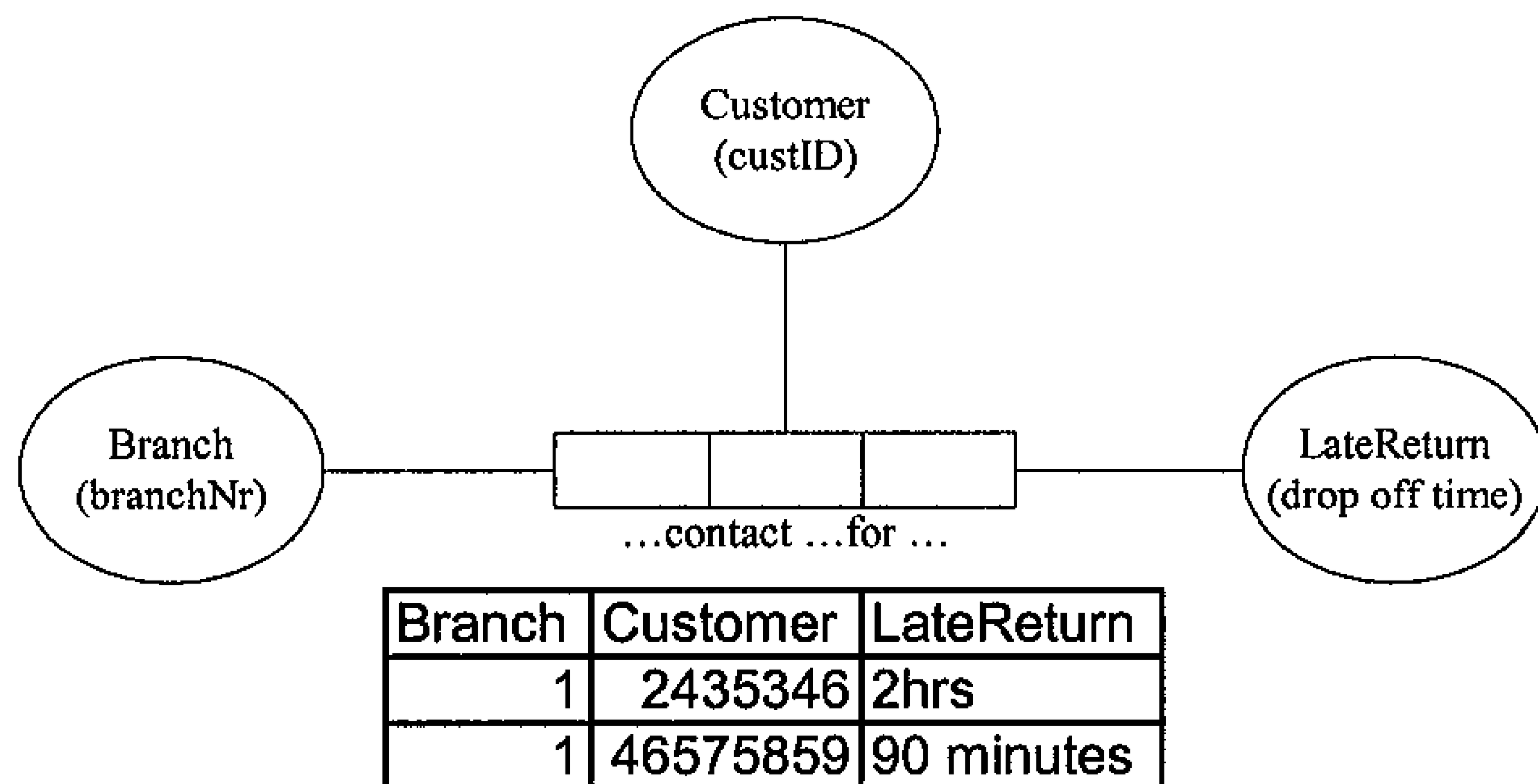


Car	Damage	Rental
CDC434	Y	12
RFD343	N	23

Figure 83: Fact 37

Facts 38, 39 and 40 are derivations.

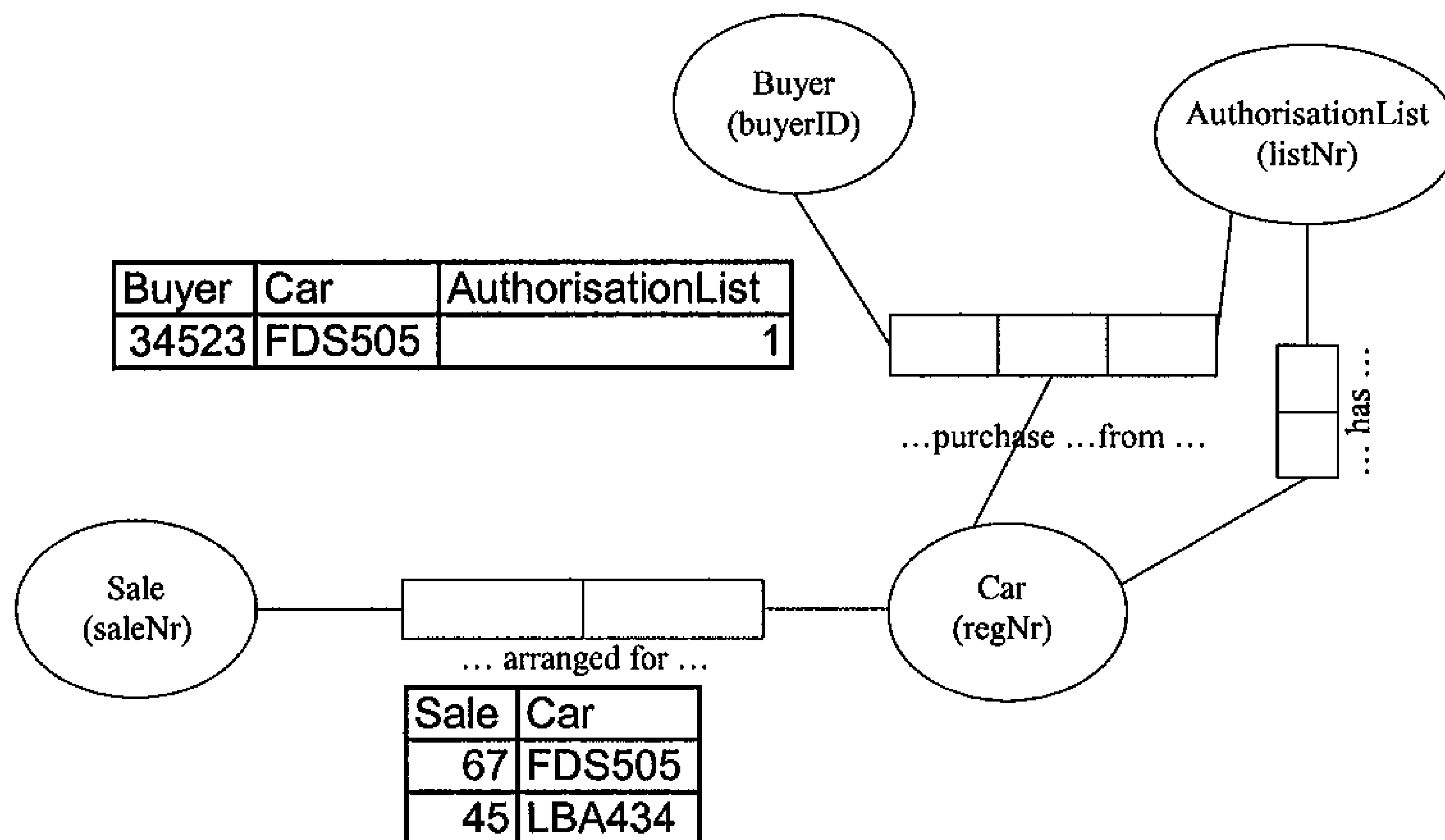
Fact 41 yields the following schema:



Branch	Customer	LateReturn
1	2435346	2hrs
1	46575859	90 minutes

Figure 84: Fact 41

Facts 42 and43 yield the following schema:



Buyer	Car	AuthorisationList
34523	FDS505	1

Sale	Car
67	FDS505
45	LBA434

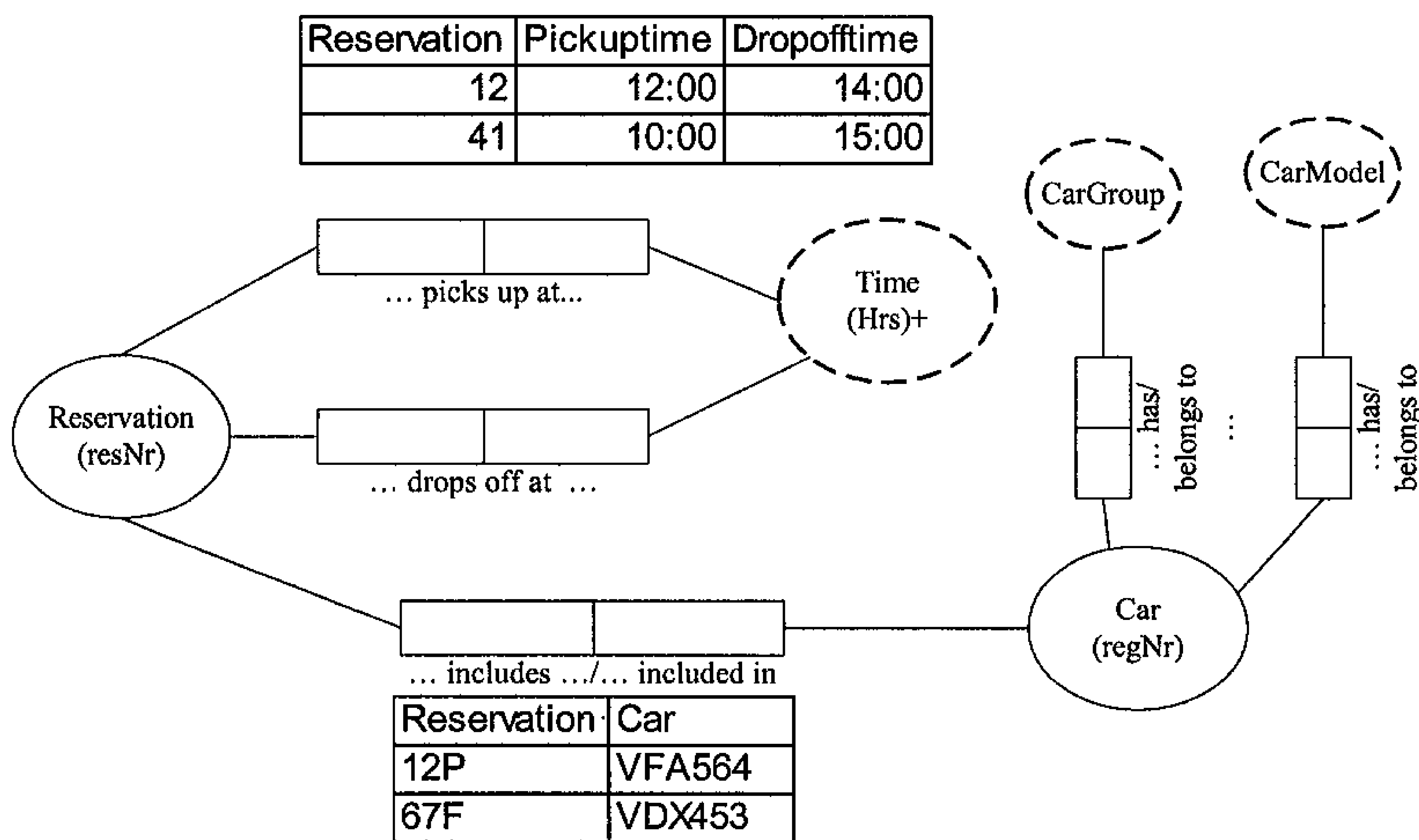
Figure 85: Facts 42 and43

Step 3 is to check for entity types that must be combined and to note any arithmetic derivations. Derivation rules are written as text below the schema diagram. Step 3 performs the following individual operations:

- If the same entity is a member of two entity types, then combine the entity types into one.
- If the same kind of information is recorded for different entity types, then combine the entity types in such a way that information is not lost.
- If a fact type can be arithmetically derived from other facts, then the derivation rule must be added and indicate the derivation with an asterisk "*".

Step 3 is applied to figure 73. No entity types can be combined in this case. Manager and Booking Clerk have different tasks and are mutually exclusive. Furthermore, manager initiates a car transfer, and his role is uniquely distinguished from that of a booking clerk.

Figure 74 changes to figure 86 after applying step 3. In figure 74, Reservation has both pickupTime and Dropofftime; they can be combined to yield the schema in figure 86.

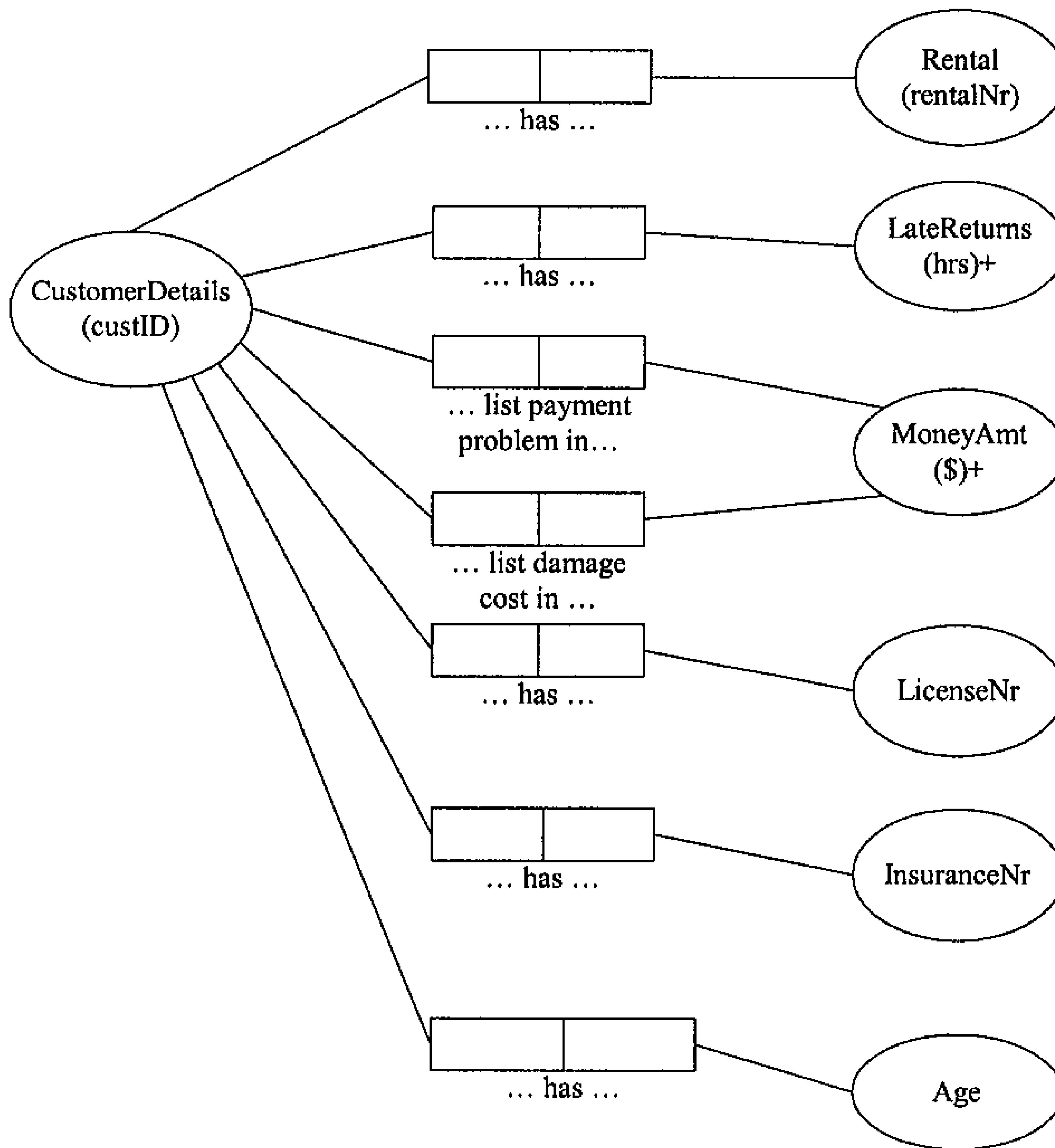


Reservation	Pickuptime	Dropofftime
12	12:00	14:00
41	10:00	15:00

Reservation	Car
12P	VFA564
67F	VDX453

Figure 86: Facts 5, 6, 7, 33 and 34 combined

No entity types can be grouped in figures 75 and 76. Applying step 3 to figure 77 yields the schema in figure 87.

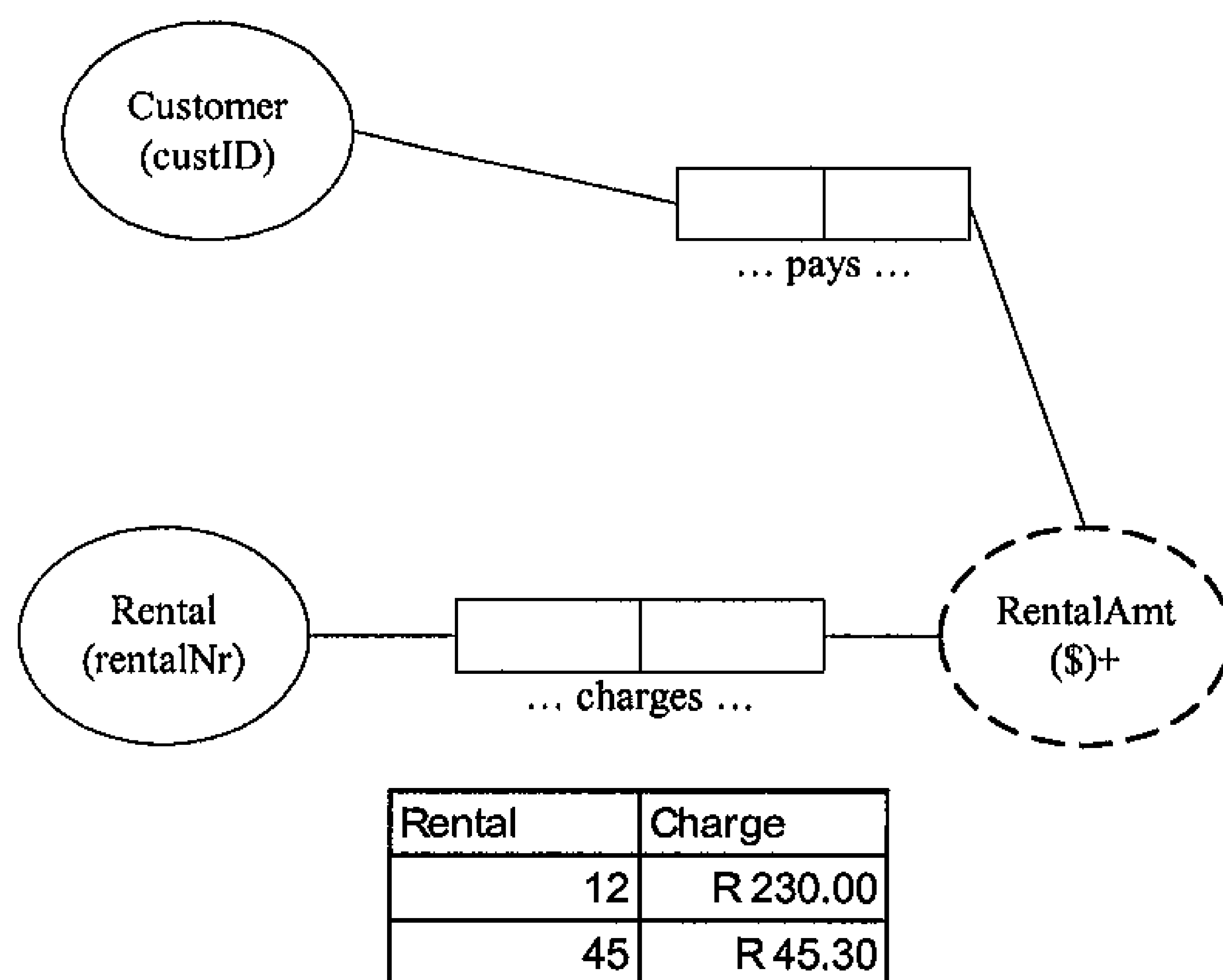


*{rental_charge = day_ charge + pending_charge}
Rental has pending_charge **iff** Car is damaged
during rental

Figure 87: Facts 16, 17, 18, 19, 20, 21 and 22 combined

The derivation rule is written as text below the schema and iff abbreviates "if and only if".

Figures 78, 79 remain the same after applying step 3. Figure 80 adds the derivation rule after applying step 3 and the changed schema depicted in figure 88.



Rental	Charge
12	R 230.00
45	R 45.30

*{rentalAmt = day_charge + pending_charge + penalty_charge}

pending_charge is added **iff** a damage is caused **and** penalty_charge is added if car dropped off at a non pickup branch

Figure 88: Facts 28 and 29 combined

Figures 81,82,83,84,85 remain the same after applying step 3.

Step 4 is to add uniqueness constraints and check the arity of fact types. Applying step 4 to figure 73 gives the schema depicted in figure 89. Forward and reverse verbalisations of facts are given in each case; although this adds clarity when reading through the schema, it is not necessary to verbalise all the facts in both directions.

Forward and reverse verbalisation of fact 1 (F1)

Each branch classifies **at most one** car for **at most one** rental.

Each rental **classified by** at most one car for **at most one** branch.

Each branch-rental combination is unique. The same branch has different cars classified for different rental numbers, but different branches have different cars classified for the same rental numbers. This is indicated in the fact table of figure 89.

Forward and reverse verbalisation of fact 2 (F2)

Each branch has **at most one** manager.

Each manager belongs to **at most one** branch.

Forward and reverse verbalisation of fact 3 (F3)

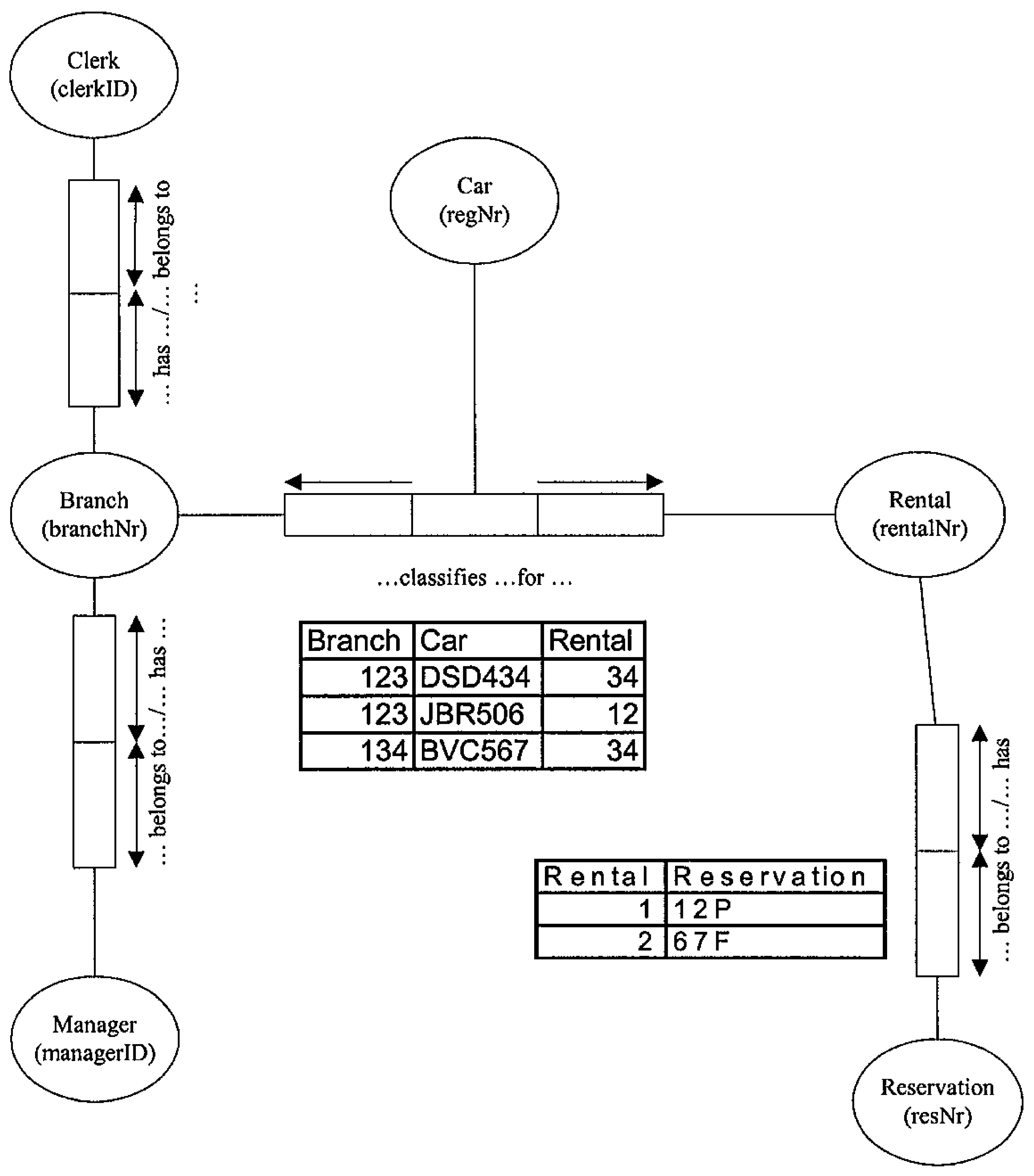
Each branch has **at most one** clerk.

Each clerk belongs to **at most one** branch.

Forward and reverse verbalisation of fact 4 (F4)

Each rental is made by **at most one** reservation.

Each reservation belongs to **at most one** rental.



Branch	Car	Rental
123	DSD434	34
123	JBR506	12
134	BVC567	34

Rental	Reservation
1	12P
2	67F

Figure 89: Uniqueness constraints added to figure 73

The verbalisations of facts in figure 86 are given below. The uniqueness constraints are added to figure 86 and the schema shown in figure 90.

Forward and reverse verbalisation of fact 5 (F5)

Each reservation identified by **at most one** car.

Each car identified in **at most one** reservation.

Forward verbalisation of facts 6 and 7 (F6, F7)

Each car has **at most one** cargroup.

Each car has **at most one** carmodel.

Forward verbalisation of facts 33 and 34 (F33, 34)

Each rental has **at most one** pickup time.

Each rental has **at most one** dropoff time.

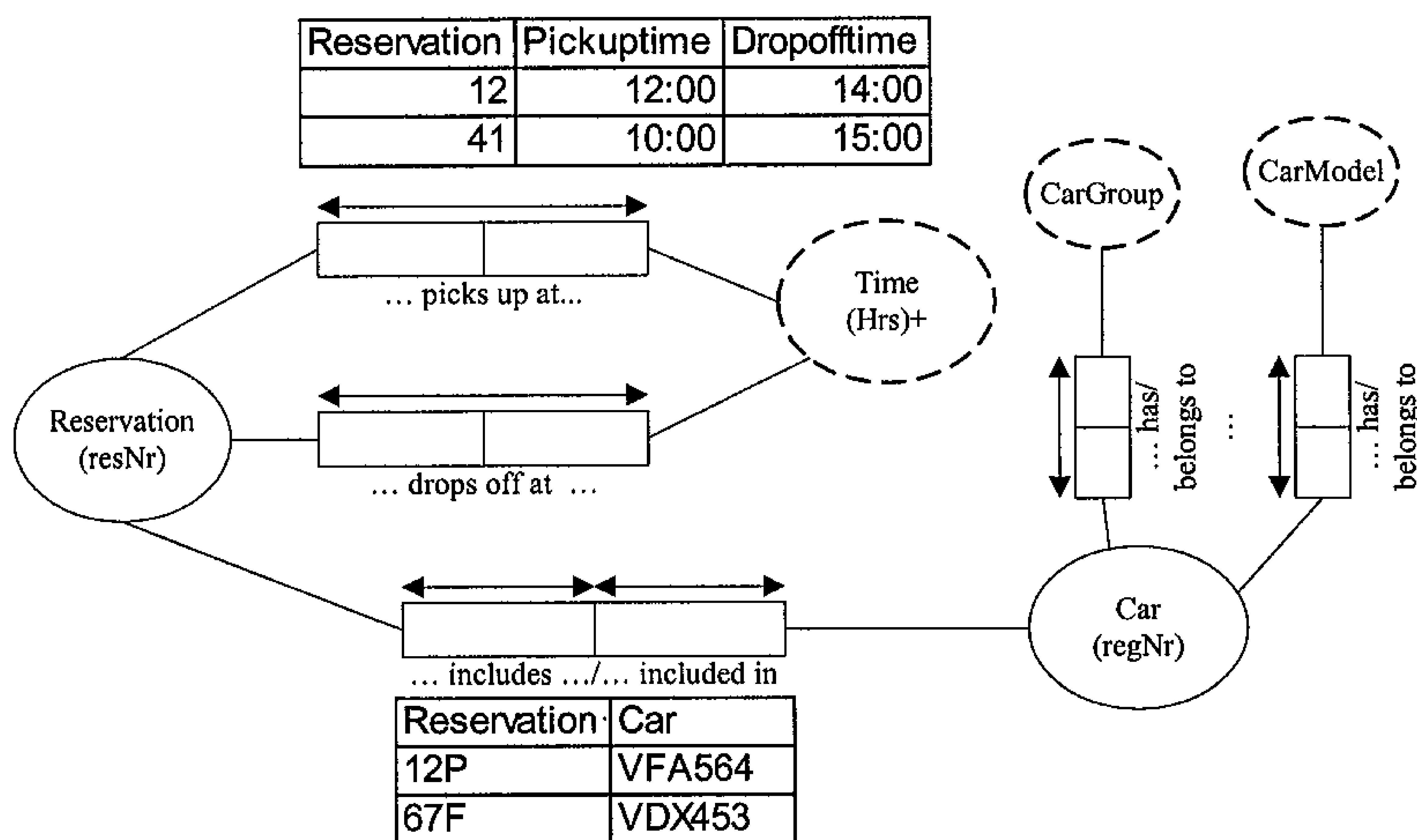


Figure 90: Uniqueness constraints added to figure 86

Step 4 is applied to figure 75, yielding the schema in figure 91. The verbalisation of facts is given below:

Forward verbalisation of fact 8 (F8)

Each manager requests **some** branch for **some** car. The same manager may request the same branch for a different car. Here the manager-branch pair is unique and the manager-car pair is unique.

Forward and reverse verbalisation of fact 9 (F9)

Each car returned to **at most one** branch.

Each branch accepts **at most one** car.

Forward and reverse verbalisation of fact 10 (F10)

Each branch owns **at most one** car.

Each car belongs to **at most one** branch.

Forward and reverse verbalisation of fact 11 (F11)

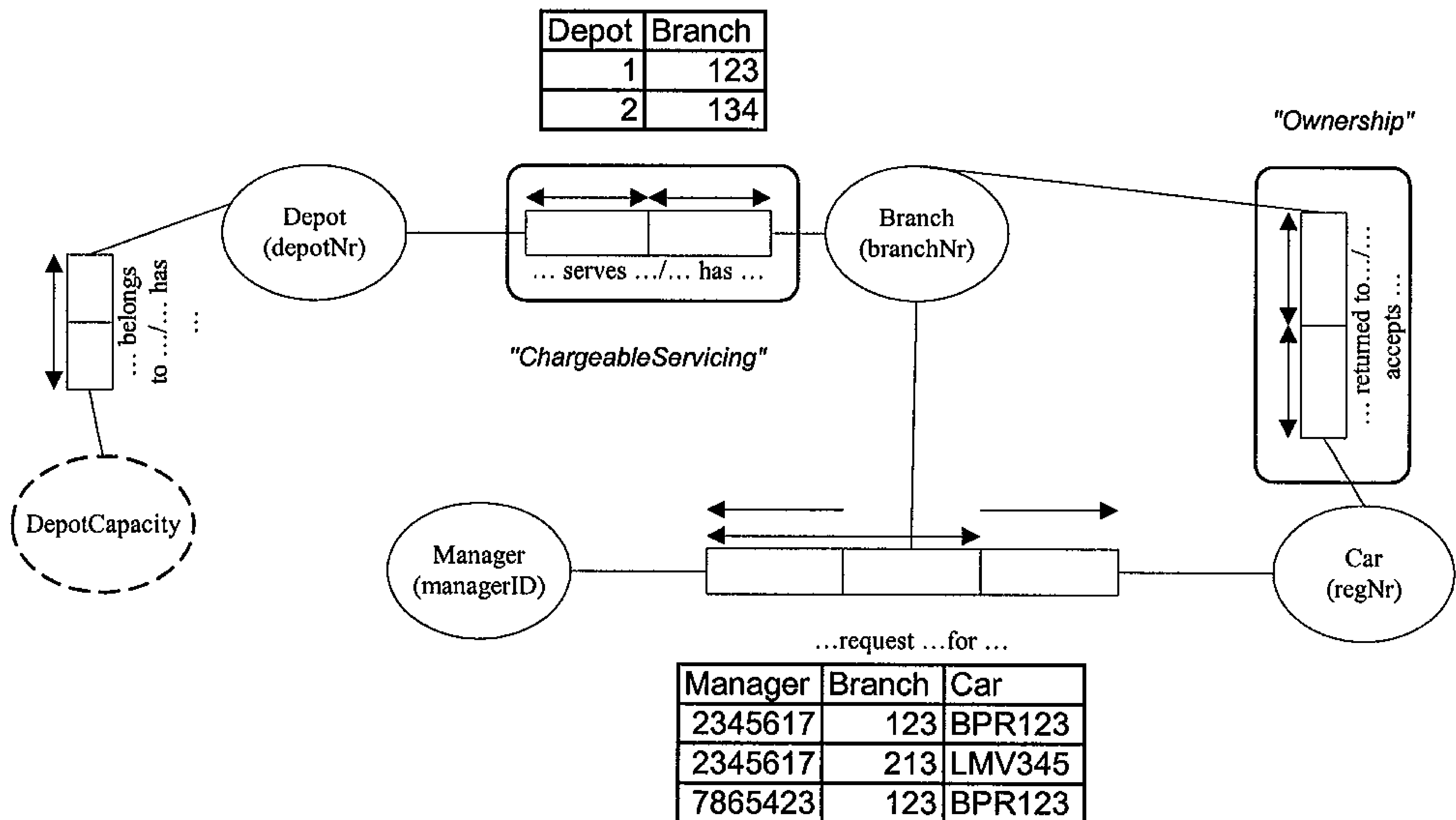
Each branch has **at most one** depot.

Each depot serves **at most one** branch.

Forward and reverse verbalisation of fact 12 (F12)

Each depot has **some** capacity.

Each capacity belongs to **at most one** depot.



Depot	Branch
1	123
2	134

Manager	Branch	Car
2345617	123	BPR123
2345617	213	LMV345
7865423	123	BPR123

Figure 91: Uniqueness constraints added to figure 75

Uniqueness constraints are added to figure 76, giving the schema in figure 92. The verbalisation of facts 13, 14 and 15 is given below:

Forward and reverse verbalisation of fact 13 (F13)

Each customer makes **at most one** reservation.

Each reservation belongs to **at most one** customer.

Forward and reverse verbalisation of fact 14 (F14)

Each branch records **some** customerID.

Each customerID is captured **at some** branch.

Forward and reverse verbalisation of fact 15 (F15)

Each customer has **some** customer details.

Each customer detail belongs to **at most one** customer.

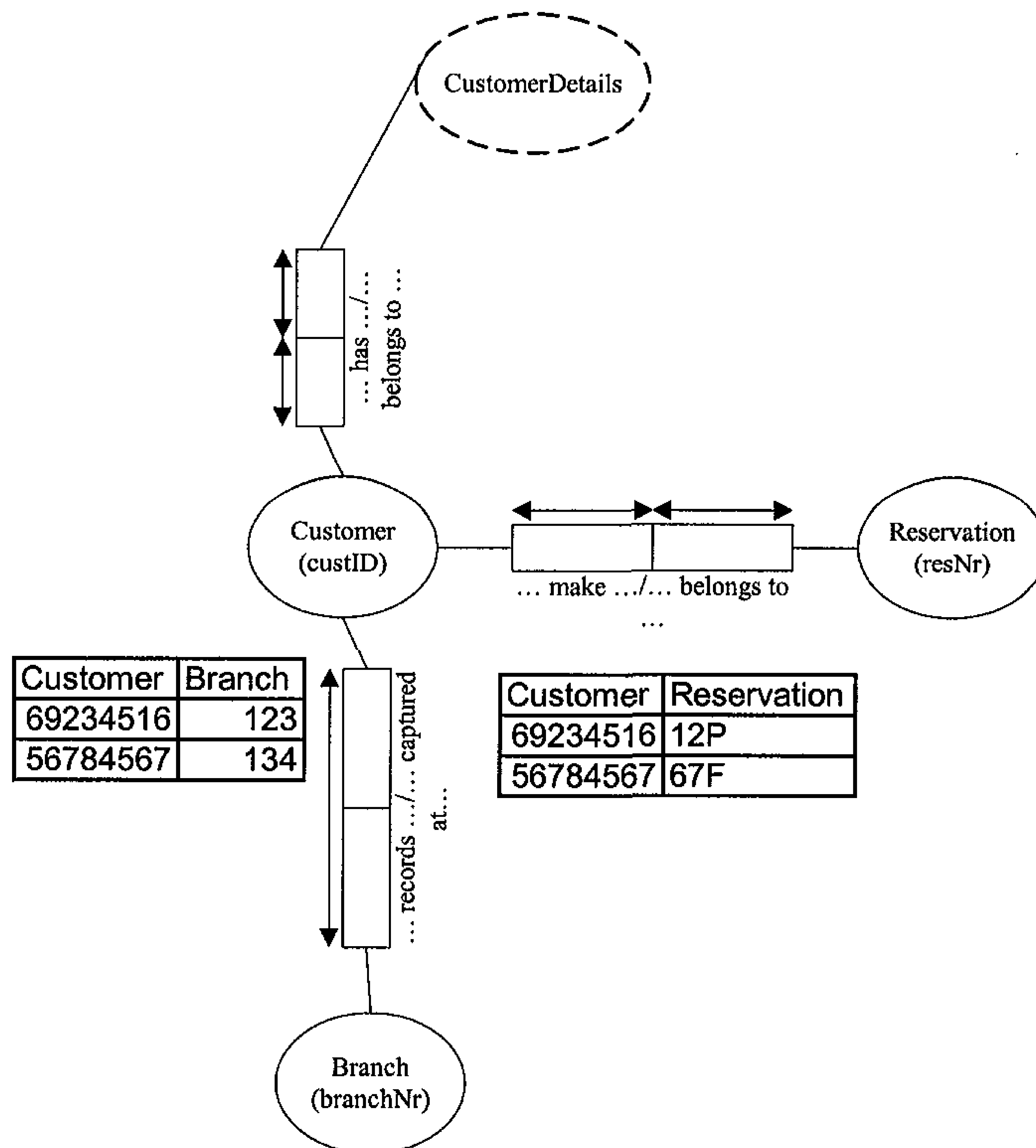


Figure 92: Uniqueness constraints added to figure 76

Uniqueness constraints are added to figure 87, yielding the schema in figure 93. The fact verbalisations are given below:

Forward verbalisation of fact 16 (F16)

Each customer detail includes **at most one** rental.

Uniqueness constraints cannot be applied to facts 17, 18 and 19 because a customer may not have at least one late return, payment problem or car damage.

Forward and reverse verbalisation of facts 20, 21, 22 (F20, F21, F22)

Each customer has **at most one** licence number.

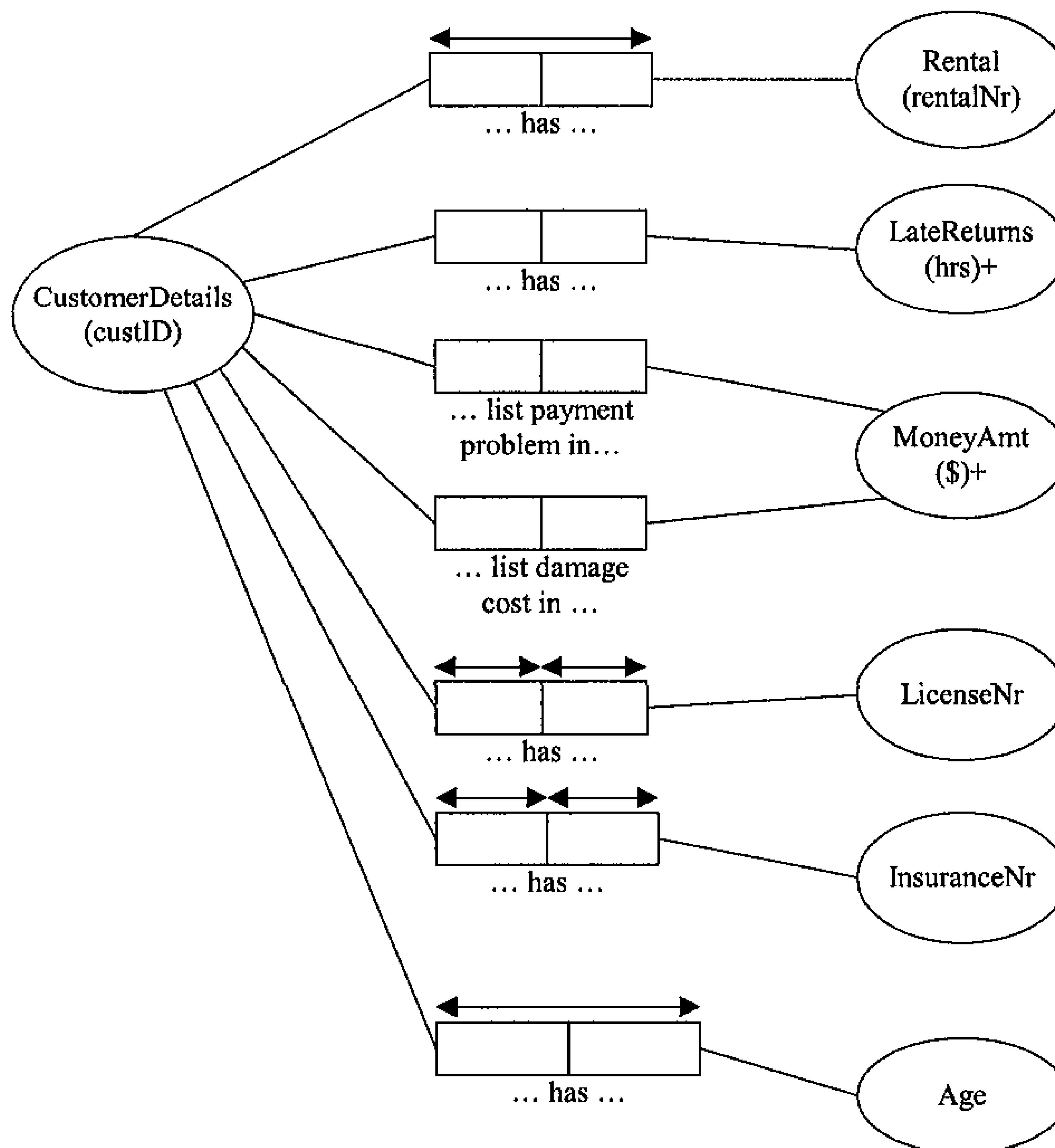
Each licence number belongs to **at most one** customer.

Each customer has **at most one** insurance number.

Each insurance number belongs to **at most one** customer.

Each customer has **some** age.

Each age belongs to **at most one** customer.



*{rental_charge = day_ charge + pending_charge}
Rental has pending_charge **iff** Car is damaged
during rental

Figure 93: Uniqueness constraints added to figure 87

Uniqueness constraints are added to figure 78, yielding the schema in figure 94. The fact verbalisations are given below:

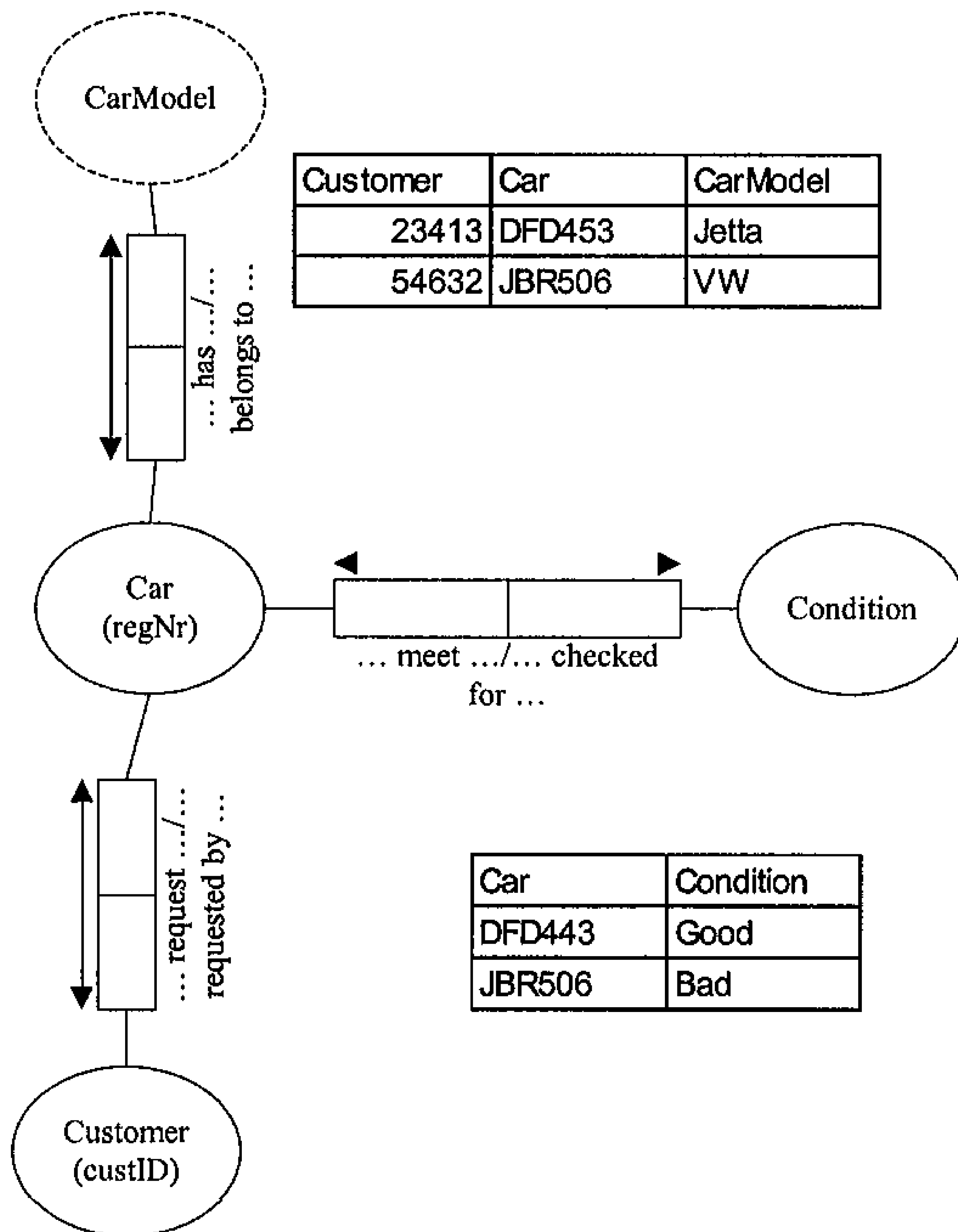
Forward and reverse verbalisation of fact 23 (F23)

Each car meets **some** condition.

Each condition checked for **at most one** car.

Forward verbalisation of fact 24 (F24)

Each customer requests **at most one** car with **at most one** model.



Customer	Car	CarModel
23413	DFD453	Jetta
54632	JBR506	VW

Car	Condition
DFD443	Good
JBR506	Bad

Figure 94: Uniqueness constraints added to figure 78

Uniqueness constraints are added to figure 79, yielding the schema in figure 95. The fact verbalisations are given below:

Forward verbalisation of fact 25, 26 (F25, F26)

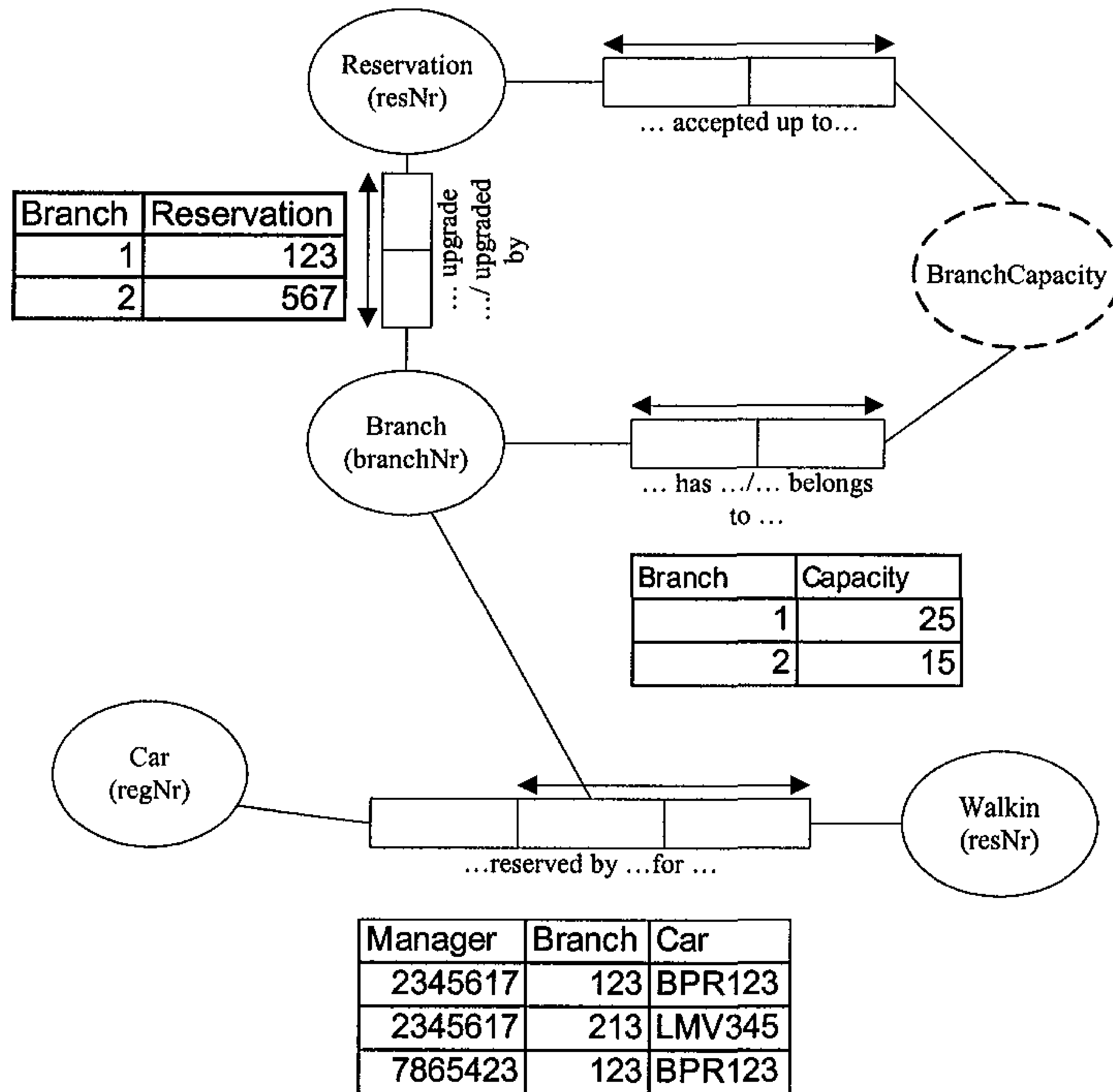
Each reservation is accepted up to **some** branch capacity.

Each branch upgrades **at most one** reservation.

Forward and reverse verbalisation of fact 27 (F27)

Each branch reserves **at most one** car for **at most one** walkin reservation.

Each walkin reservation is reserved by **at most one** car for **at most one** branch. The branch-walkin pair is unique.



Branch	Reservation
1	123
2	567

Branch	Capacity
1	25
2	15

Manager	Branch	Car
2345617	123	BPR123
2345617	213	LMV345
7865423	123	BPR123

Figure 95: Uniqueness constraints added to figure 79

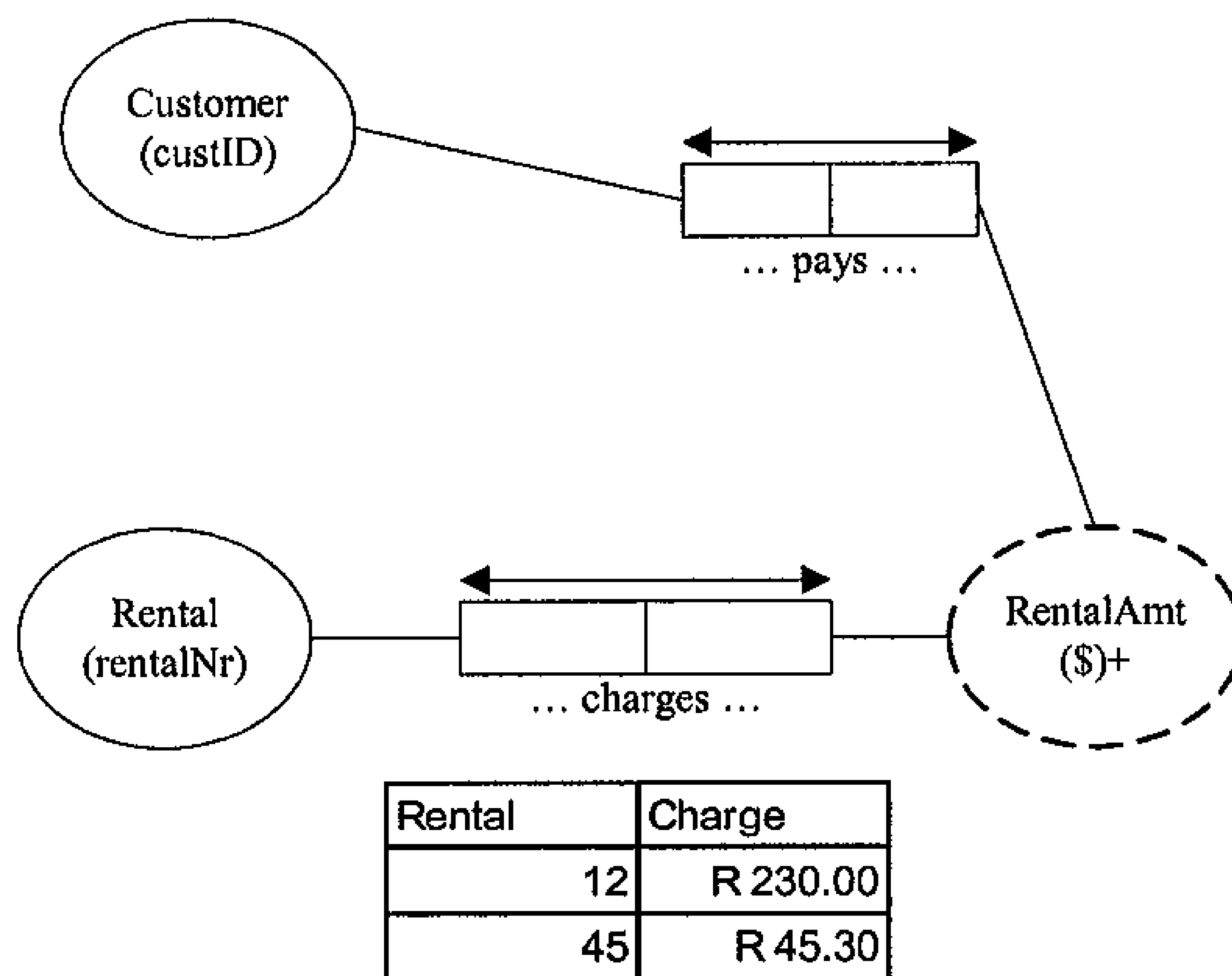
Uniqueness constraints are added to figure 88, delivering figure 96. The fact verbalisations are given below:

Forward and reverse verbalisation of fact 28, 29 (F28, F29)

Each rental includes **at most one** rental charge.

Each rental charge is included in **at most one** rental.

Each customer pays **at most one** rental charge.



Rental	Charge
12	R 230.00
45	R 45.30

*{rentalAmt = day_charge + pending_charge + penalty_charge}

pending_charge is added **iff** a damage is caused **and** penalty_charge is added if car dropped off at a non pickup branch

Figure 96: Uniqueness constraints added to figure 88

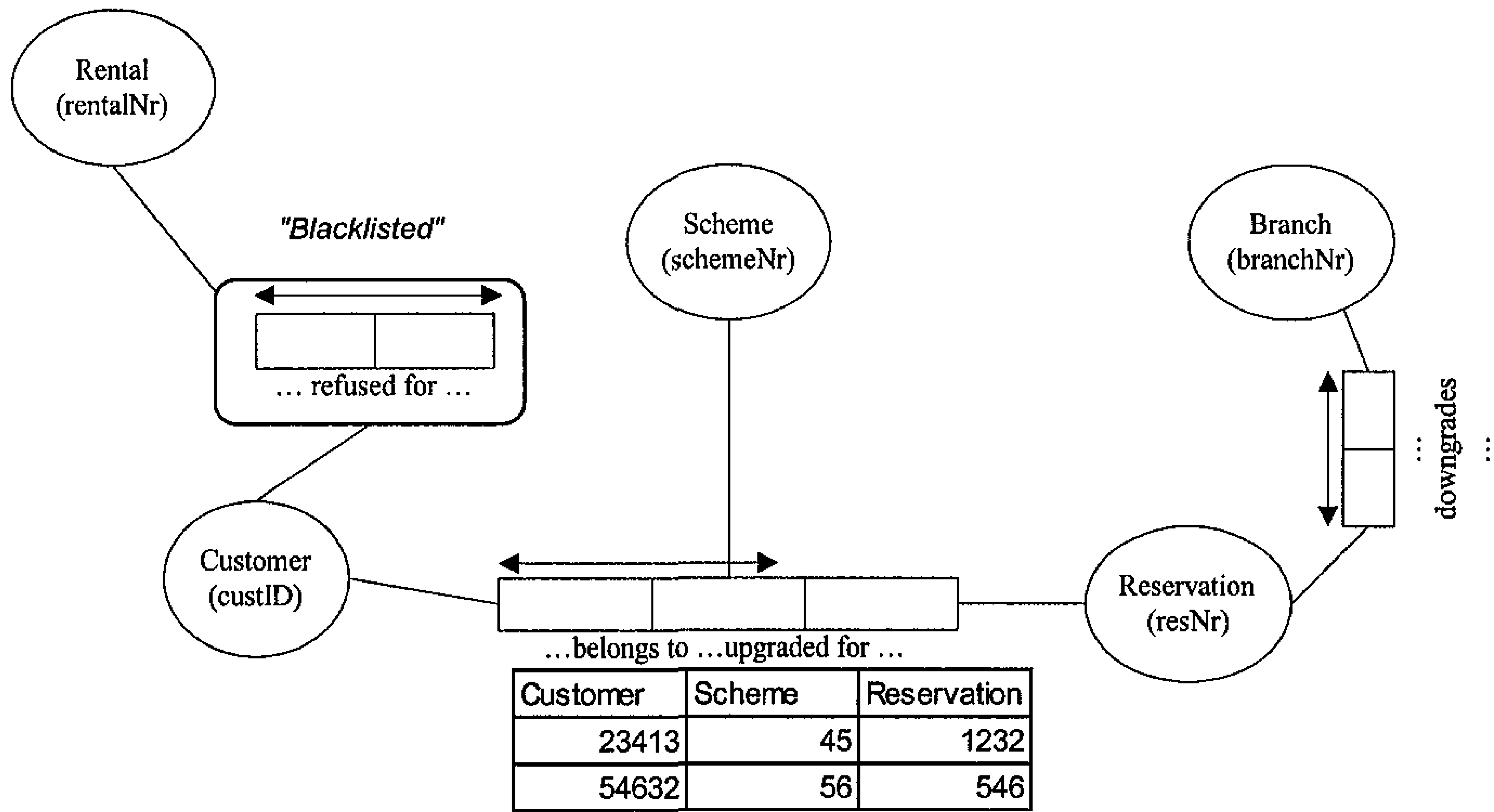
Figure 81 changes to figure 97 after applying step 4. The verbalisation of facts is given below:

Forward verbalisation of facts 30, 31, 32 (F30, F31, F32)

Each blacklisted customer is refused for **all** rentals.

Each customer belonging to **at most one** scheme is upgraded for **at most one** reservation. The customer-scheme pair is unique.

Each branch downgrades **some** reservation.



Customer	Scheme	Reservation
23413	45	1232
54632	56	546

Figure 97: Uniqueness constraints added to figure 81

Figure 82 changes to figure 98 after applying step 4. The verbalisation of facts is given below:

Forward verbalisation of fact35, 36 (F35, F36)

Each customer having **at most one** credit card number signs **at most one** contract.

Each customer signs **at most one** authorisation form.

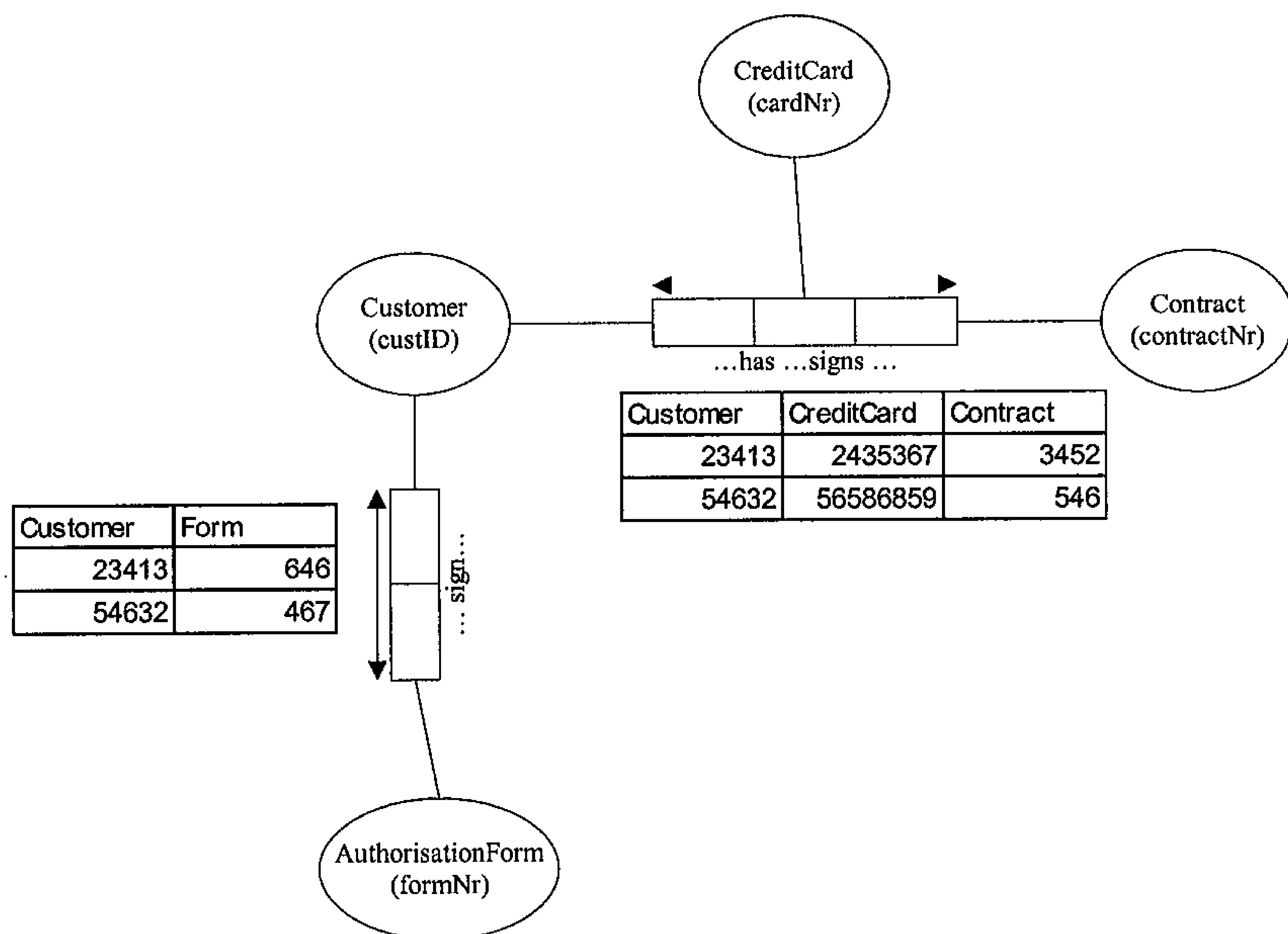


Figure 98: Uniqueness constraints added to figure 82

Figure 83 changes to figure 99 after applying step 4. The verbalisation of facts is given below:

Forward verbalisation of fact37 (F37)

Each car in group is checked for **some** damage from **at most one** rental. The car-damage-rental is unique.

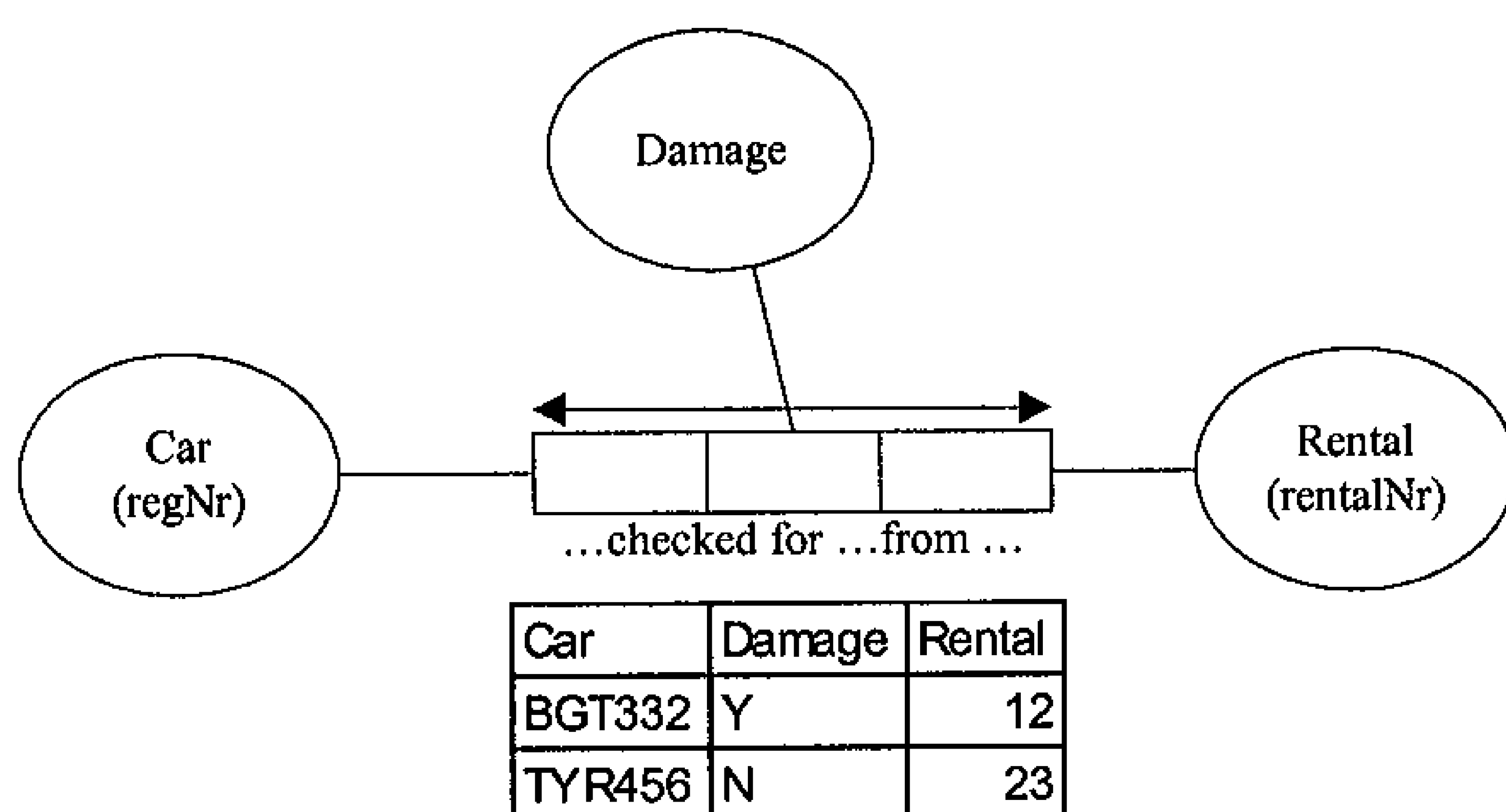
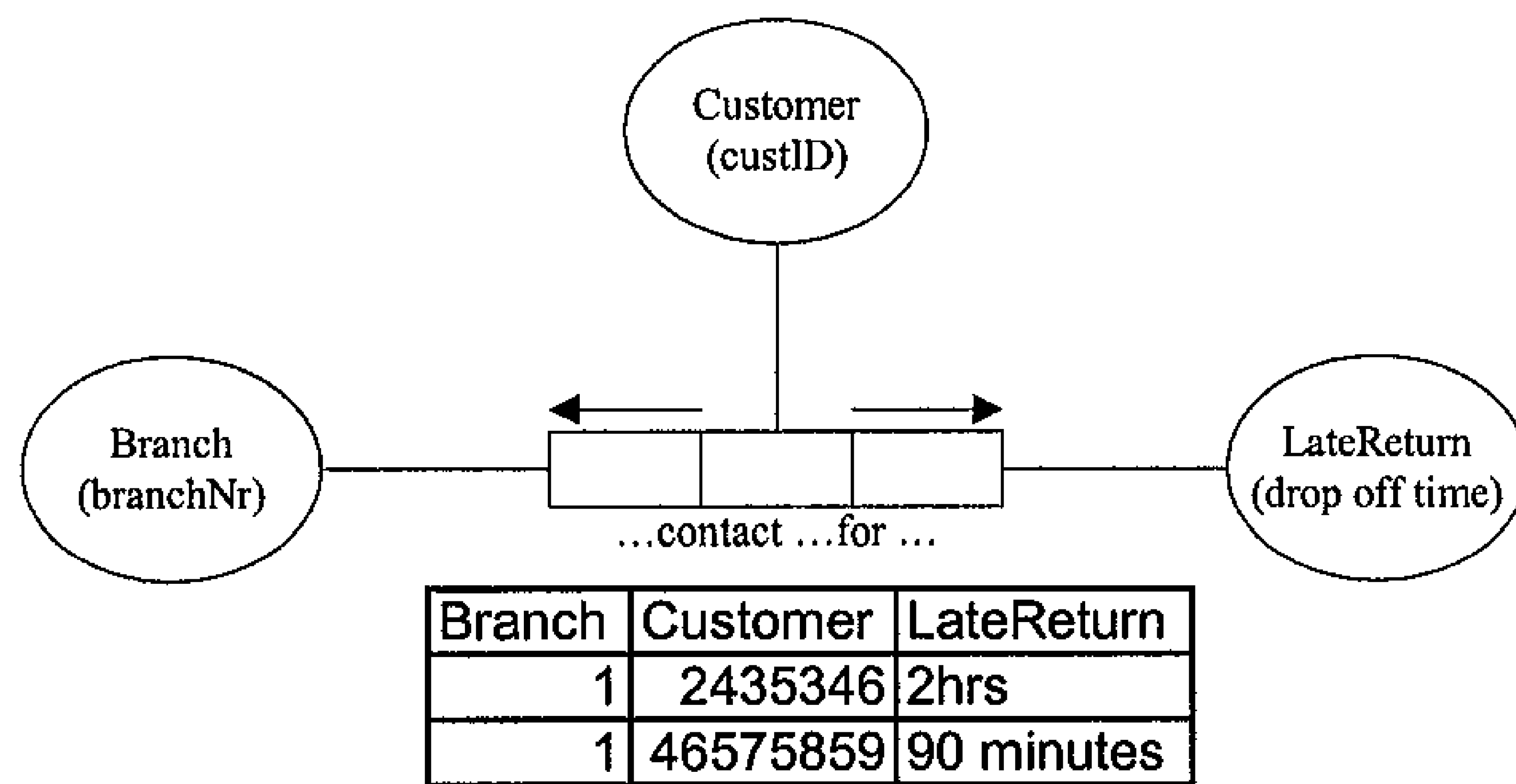


Figure 99: Uniqueness constraints added to figure 83

Figure 84 changes to figure 100 after applying step 4. Facts 38, 39 and 40 are derivations and captured in figure 96. The verbalisation of fact 41 is given below:

Forward verbalisation of fact 41 (F41)

Each branch contacts **at most one** customer on **at most one** late return. Here the branch-latereturn pair is unique.



Branch	Customer	LateReturn
1	2435346	2hrs
1	46575859	90 minutes

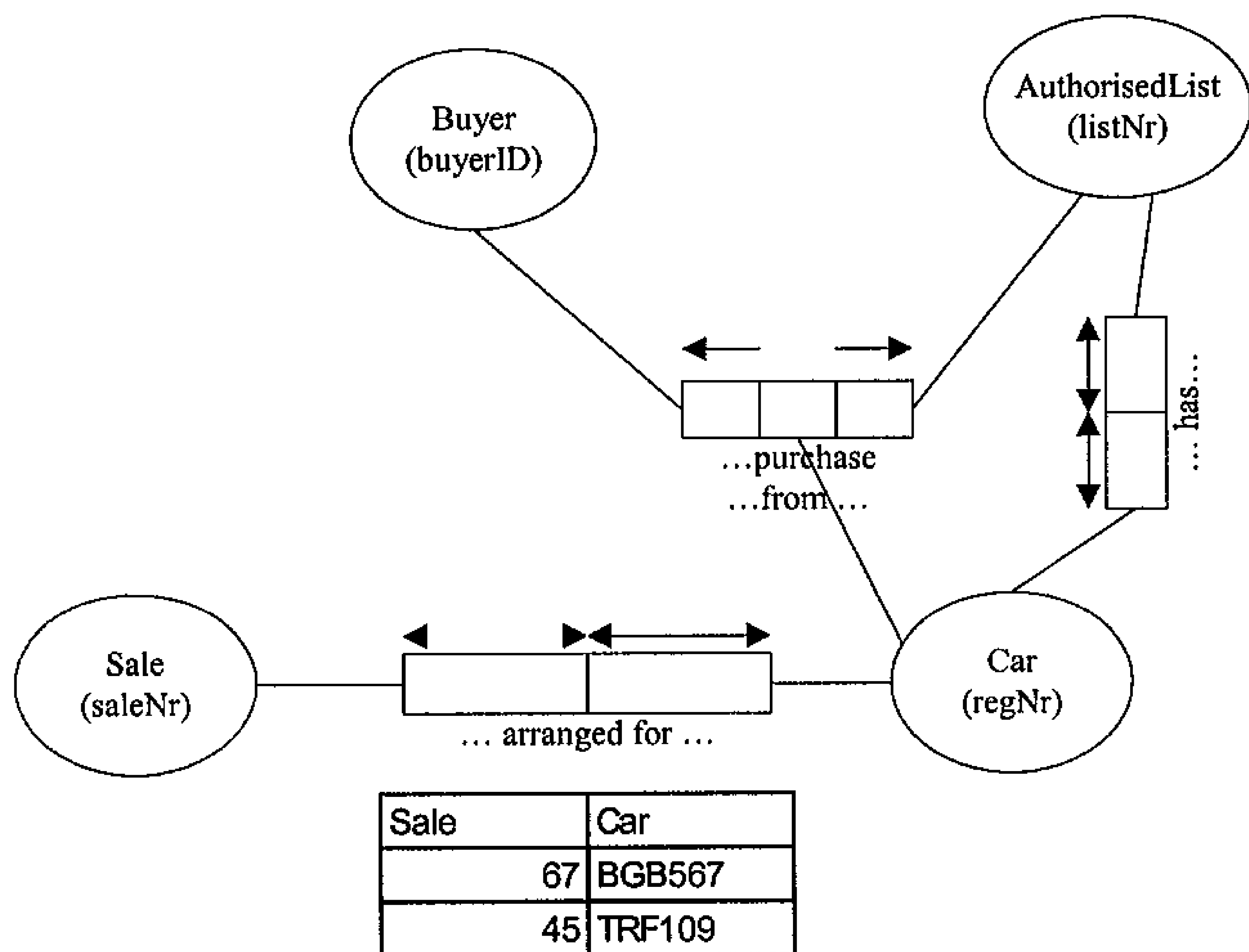
Figure 100: Uniqueness constraints added to figure 84

Figure 85 changes to figure 101 after applying step 4. The verbalisation of fact 42 and 43 is given below:

Forward verbalisation of fact 42, 43 (F42, F43)

Each car is arranged for **at most one** sale.

Each buyer purchases **some** car from **at most one** authorisation list.



Sale	Car
67	BGB567
45	TRF109

Figure 101: Uniqueness constraints added to figure 85

Figures 89 - 101 are combined into one schema and depicted in figure 102. In figure 102, object types (Car, Customer, Branch and Rental) are duplicated. Duplication of object types is allowed in ORM schemas and is represented by a shadow together with the duplicated object type. Since ORM schemas are large, duplication of object types sometimes gives more clarity to the schema.

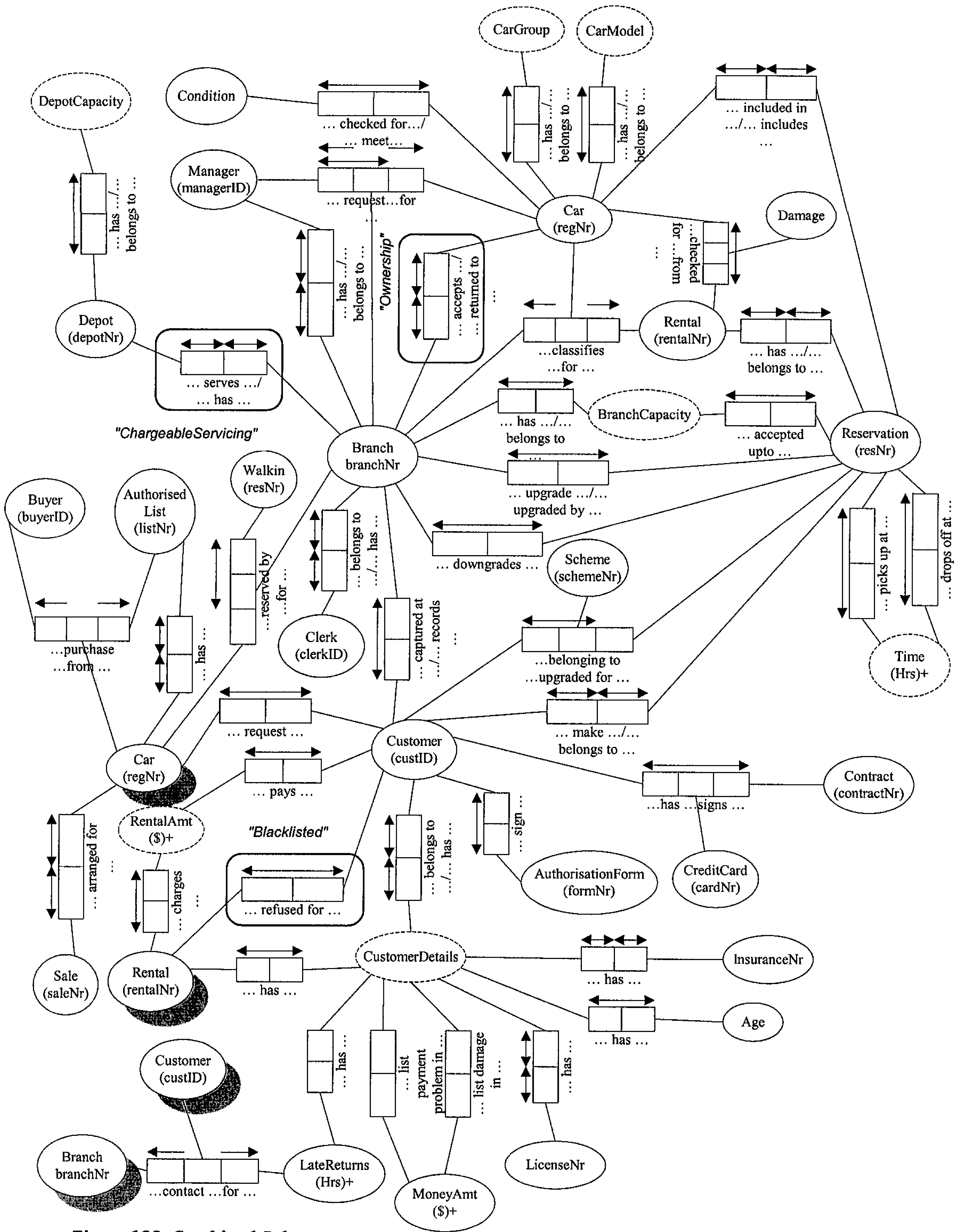


Figure 102: Combined Schema

Step 5 is to add mandatory role constraints and check for logical derivations. A role is mandatory if, and only if; every member of the population of its object type must play the role. Roles that are not mandatory are optional. The following list identifies the mandatory roles in the case study.

Mandatory roles

The mandatory role constraint of the facts 1- 43 verbalises as follows:

Each branch classifies **at least one** car group for **at least one** rental.

Each branch has **at least one** manager.

Each branch has **at least one** clerk.

Each rental is made by **at least one** reservation.

Each reservation is identified by **at least one** car.

Each car has **at least one** cargroup.

Each car has **at least one** carmodel.

Each reservation has **at least one** pickup time.

Each reservation has **at least one** dropoff time.

Fact 8 is optional, because each manager requests a car only if a transfer is necessary.

Each car is returned to **at least one** branch.

Each branch owns **at least one** car.

Each branch has **at least one** depot.

Each depot has **at least some** capacity.

Each customer makes **at least one** reservation.

Each branch records **at least one** customerID.

Each customer has **at least one** customer detail.

Each customer details includes **at least one** rental.

Facts 17, 18 and 19 are optional because a customer may not have late returns, payment problems or car damage.

Each customer has **at least one** license number.

Each customer has **at least one** insurance number.

Each customer has **only one** age.

Each car group meets **at least one** condition.

Fact 24 is optional because a customer may not request any specific car model.

Each reservation is accepted up to **some** branch capacity.

Fact 26 is optional because a branch may not upgrade a reservation. Fact 27 is optional because a branch may not have any walk-in reservations for a day.

Each rental includes **at least one** rental charge.

Each customer pays **at least one** rental charge.

Each blacklisted customer is refused **at least one** rental.

Each customer belonging to **at least one** scheme is upgraded for **at least one** reservation.

Fact 32 is optional because a branch may not downgrade a reservation.

Each customer having **at least one** credit card number signs **at least one** contract.

Fact 36 is optional because it is possible that not a single customer signs an authorisation form.

Each car in group is checked for **at least one** damage from **at least one** rental.

Each branch contacts **at least one** customer on **at least one** late return.

Each car is arranged for **at least one** sale.

Fact 43 is optional because a buyer may not make a purchase.

Adding the above mandatory role constraints to figure 102 yields figure 103.

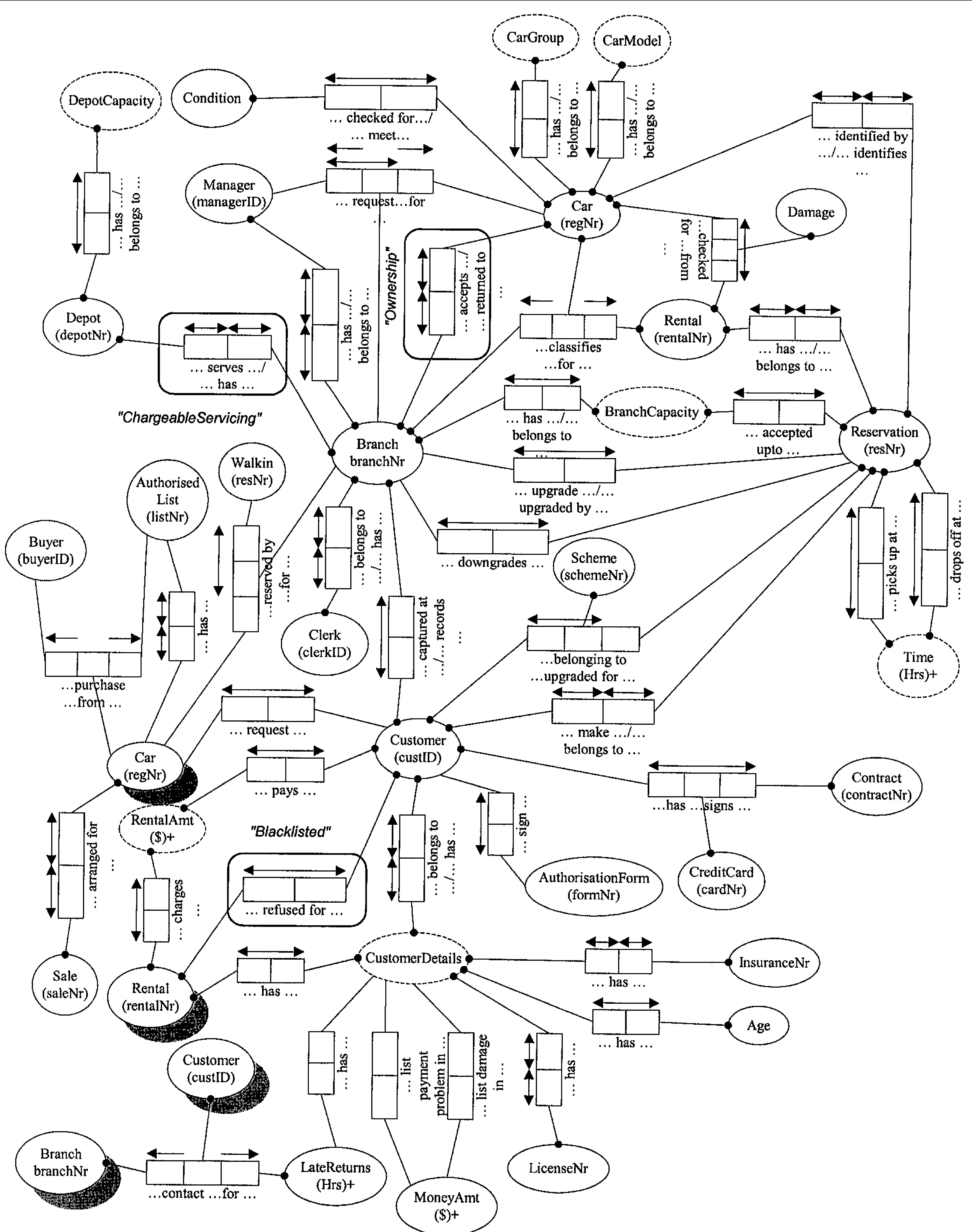


Figure 103: Combined schema with mandatory roles

The second stage of step 5 is to check for logical derivations. They are fact types that can be derived from others without the use of arithmetic. Derived facts are identified in the schema by the "*" added to the predicate of the derived fact. The derivation is written below the schema. Applying step 5 to the case study, the following logical derivations are derived from facts 1-43. In the following derivations, iff stands for if and only if.

- *Branch classifies car **iff** Car in a group is available for rental.
- *Manager request other branches for car **iff** the requesting branch runs out of cars.
- *Branch owns a car **iff** the car is returned to the branch.
- *Car is booked for service **iff** depot has capacity.
- *Branch records customer **iff** customer makes reservation at the branch.
- *CustomerDetails include LateReturns **iff** a car is returned late by the customer.
- *CustomerDetails include PaymentProblem **iff** the customer had problems with the payment.
- *CustomerDetails include CarDamage **iff** the car has been damaged by the customer.
- *Branch upgrades Reservation **iff** the requested group is not available.
- *Branch downgrades Reservation **iff** the requested group not available.
- *Customer signs Contract **iff** he has credit card.
- *Branch contacts Customer **iff** car is not returned at drop-off time.
- *Sale arranged for car **iff** car has been running for a year or reached 40,000 kms.

The above-mentioned logical derivations are added to figure 103 and yield figure 104.

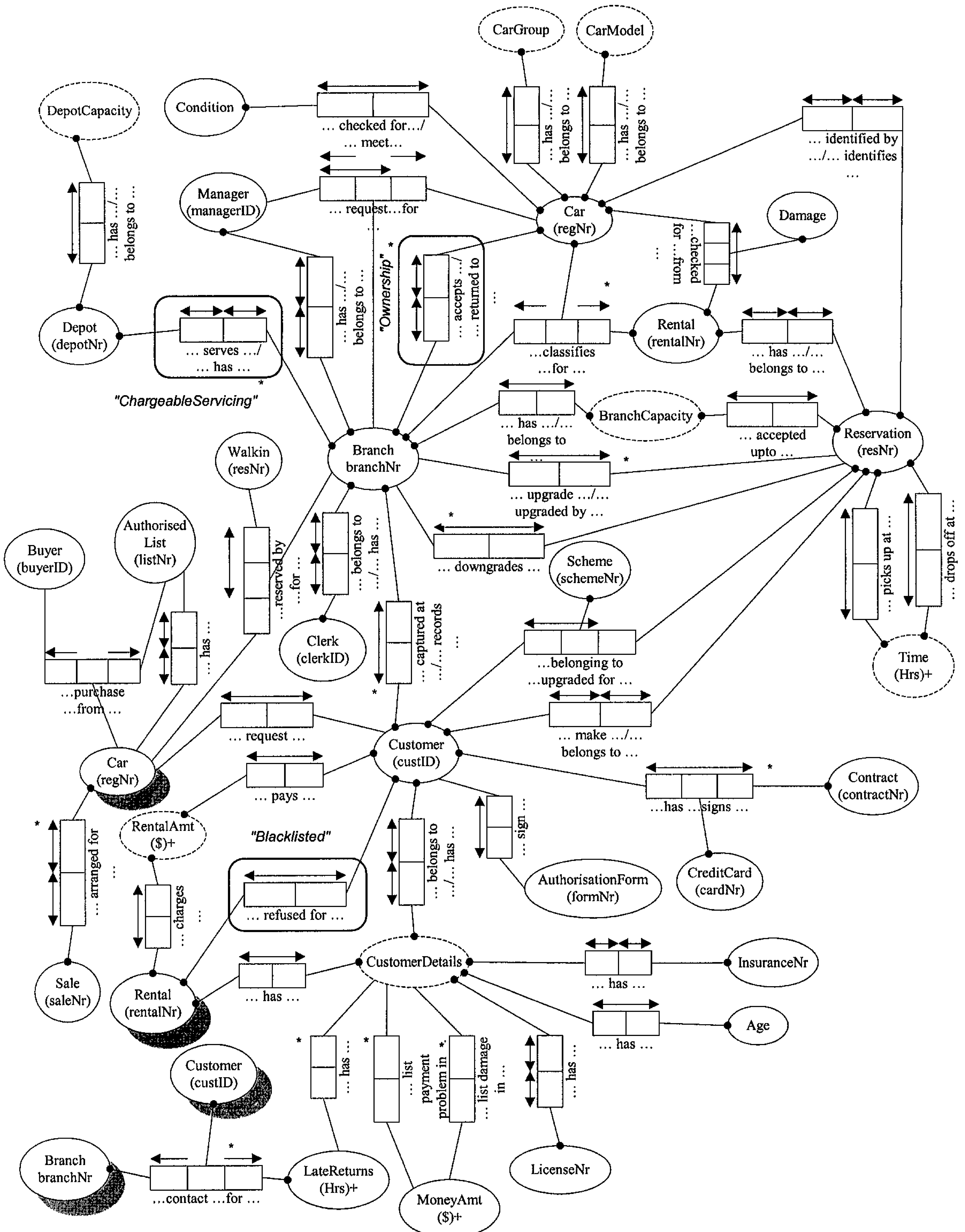


Figure 104: Combined schema with logical derivations

Step 6 is to add any value and subtyping constraints. A value constraint indicates the values in a value type. A value type may be indicated in a schema if and only if the value is stable.

AuthorisedDriver and AdditionalDriver are subtypes of Customer, and this is depicted in the schema by an arrow pointing from the subtype to the supertype. Here, AuthorisedDriver and AdditionalDriver are the subtypes and Customer is the supertype. Facts 1-43 are now checked for any stable value types that could be indicated in the schema. AuthorisedDriver and AdditionalDriver are subtypes of Customer. DepotCapacity and BranchCapacity can be limited from 1 to 30. Pickup and dropoff time can be limited from 08:00 to 17:00. CarModel value type is unstable, since models may change at any time or new models may be added. These value constraints are depicted in the schema. Subtypes are indicated in the schema if subtype – supertype relations can be identified. Other constraints that can be added are the equality constraints and exclusion constraints. Customer can be additional driver or authorised driver, but not both. Also, reservation can be advance or walkin, but not both at the same time. This is indicated by the exclusion constraint depicted by a circled dot in figure 105.

Step 7 is to perform final checks. By combining these two steps, the final ORM schema in figure 105 is obtained.

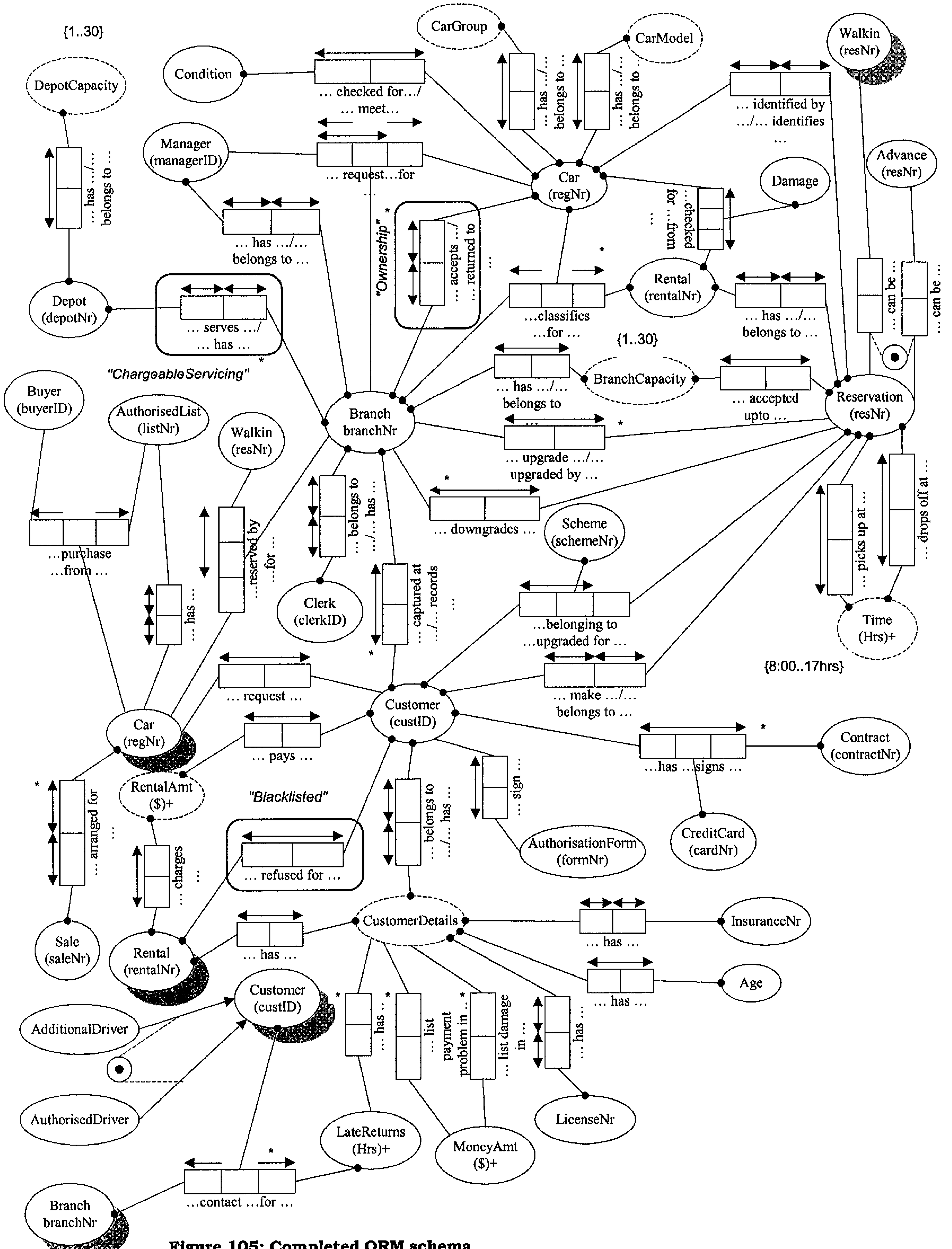


Figure 105: Completed ORM schema

Appendix D

Appendix D consists of the class diagram, which is derived from the ORM schema. This ORM-class diagram is drawn using the ORM-UML mapping process provided by Halpin [2001] in section 4.8.

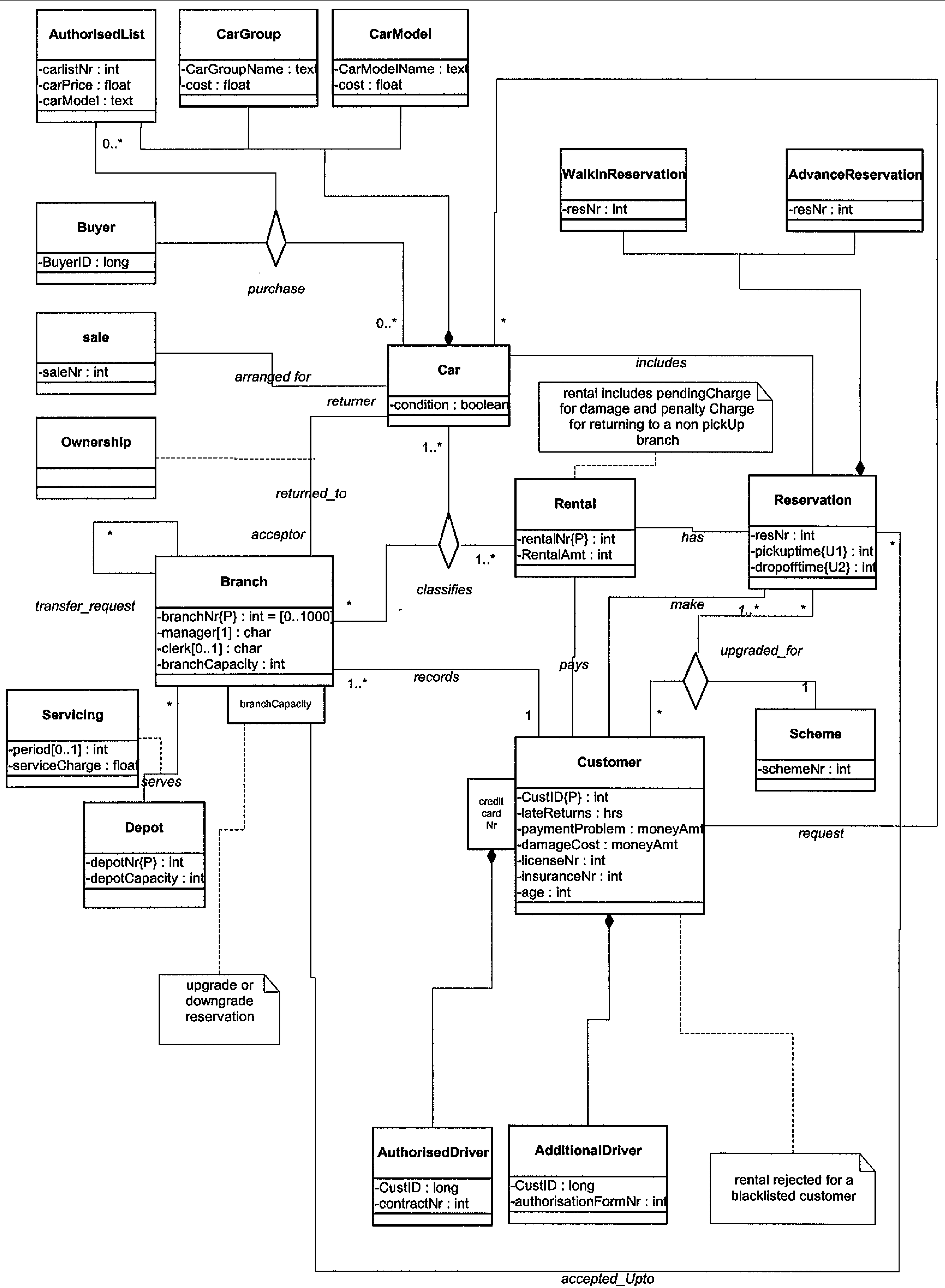


Figure 106: ORM-Class Diagram

Appendix E

This appendix contains the questionnaire that was used to test the user satisfaction of the two conceptual class diagrams (ORM-class diagram and UML domain model class diagram). The results of the questionnaire are summarised in a table at the end of the appendix.

Questionnaire

This questionnaire is prepared in order to test the conceptual class diagrams on the user community. The set of principles used as criteria to test the conceptual class diagrams can be found in section 3.4. Based on the testable principles, certain questions are put forward to the users and the user is expected to give the most appropriate rating to each question. The users are also provided with the set of requirements that are the relevant aspects of the system, which need to be captured by the conceptual class diagrams.

On a scale of 1 (very poor), 2 (poor), 3 (acceptable), 4 (good), 5 (exceptionally good), how would you rate the following?

1. Does the UML domain model class diagram capture all the object structures that are relevant in the problem statement?

   Rating

1	2	3	4	5

2. Does the UML domain model class diagram capture all the constraints that are relevant in the problem statement?

   Rating

1	2	3	4	5

3. How complete is the UML domain model class diagram in representing all the business rules that are mentioned in the problem statement?

   Rating

1	2	3	4	5

4. Does each assertion in the UML domain model class diagram reflect the corresponding business rule in the problem statement?

Rating

1	2	3	4	5

5. Can you identify any aspects of the requirements being depicted ambiguously in the UML domain model class diagram?

Rating

1	2	3	4	5

6. How clear is the UML domain model class diagram in representing the requirements?

Rating

1	2	3	4	5

7. Is it possible to verbalise the business rules given in the problem statement from the UML domain model class diagram?

Rating

1	2	3	4	5

8. How simple is the UML domain model class diagram in representing the requirements?

Rating

1	2	3	4	5

9. How would you rate the difficulty encountered in understanding the UML domain model class diagram?

Rating

1	2	3	4	5

10. To what extent does the UML domain model class diagram unnecessarily duplicate any information contained in the requirements?

Rating

1	2	3	4	5

11. How concise is the UML domain model class diagram?

Rating

1	2	3	4	5

12. Considering the Unified Process, how difficult will it be to validate the UML domain model class diagram against the problem statement?

Rating

1	2	3	4	5

13. Does the ORM-class diagram that is derived from the ORM schema capture all the object structures that are relevant in the problem statement?

Rating

1	2	3	4	5

14. Does the ORM-class diagram capture all the constraints that are relevant in the problem statement?

Rating

1	2	3	4	5

15. How complete is the ORM-class diagram in representing all the business rules that are mentioned in the problem statement?

Rating

1	2	3	4	5

16. Does each assertion in the ORM-class diagram reflect the corresponding business rule in the problem statement?

Rating

1	2	3	4	5

17. Can you identify any aspects of the requirements being depicted ambiguously in the ORM-class diagram?

Rating

1	2	3	4	5

18. How clear is the ORM-class diagram in representing the requirements?

Rating

1	2	3	4	5

19. Is it possible to verbalise the business rules given in the problem statement from the ORM-class diagram?

Rating

1	2	3	4	5

20. How simple is the ORM-class diagram in representing the requirements?

Rating

1	2	3	4	5

21. How would you rate the difficulty encountered in understanding the ORM-class diagram?

Rating

1	2	3	4	5

22. To what extent does the ORM-class diagram unnecessarily duplicate any information contained in the requirements?

Rating

1	2	3	4	5

23. How concise is the ORM-class diagram that is derived from the ORM schema?

Rating

1	2	3	4	5

24. Considering the ORM documentation given, how difficult will it be to validate the ORM-class diagram against the problem statement?

Rating

1	2	3	4	5

END OF QUESTIONNAIRE

Results of Questionnaire Survey

Table 15

(1=very poor, 2= poor, 3= acceptable, 4= good, 5= exceptionally good)

Principle	Question	Rating (Benchmark Class Diagram)	Question	Rating (ORM-Class Diagram)
Completeness	1	70% - 4, 30% - 3	13	50% - 3, 50% - 2
	2	60% - 4, 40% - 3	14	50% - 3, 50% - 2
	3	60% - 4, 40% - 3	15	50% - 3, 50% - 2
Accuracy	4	50% - 4, 50% - 3	16	60% - 3, 40% - 2
	5	50% - 1, 50% - 2	17	60% - 1, 40% - 2
Clarity	6	50% - 5, 50% - 4	18	70% - 4, 30% - 3
	7	60% - 2, 30% - 3, 10% - 1	19	70% - 2, 30% - 1
Simplicity	8	40% - 5, 40% - 4, 20% - 3	20	50% - 4, 50% - 3
	9	60% - 3, 20% - 2, 10% - 4	21	40% - 3, 40% - 2, 20% - 1
Uniqueness	10	80% - 1, 20% - 2	22	70% - 1, 30% - 2
	11	70% - 4, 30% - 3	23	80% - 4, 20% - 3
Validation	12	60% - 2, 40% - 1	24	70% - 2, 30% - 1

Summary

Questions 1,2,3,13,14 and 15 refer to the completeness principle, 4,5,16 and 17 refer to the accuracy principle, 6,7,18 and 19 refer to the clarity principle, 8,9,20 and 21 to the simplicity principle, 10,11,22 and 23 to the uniqueness principle and finally 12 and 24 refer to the validation mechanism principle.

GLOSSARY

Activity Diagram	Activity diagram represents the sequential flow of activities of a business process or a use case.
Actor	Represents a set of roles played by the users of the use cases while interacting with those use cases.
Aggregation	Special type of association, which represents a whole/part relationship where objects or classes are made up of other objects or classes.
Architecture-centric	In the software development context, a process that focuses on the early development of software development.
Artefact	A piece of information produced by a software development process.
Association	A semantic link between classes.
Association class	An association that also has class properties.
Attributes	The data that represent characteristics of interest about an object.
Behavioural modelling	Modelling with emphasis on the dynamics of the system.
Blueprint	A detailed scheme that captures the requirements of the system.
Business rule	A statement that defines or constrains some aspect of the business.
Class	Description of set of objects that share the same attributes, operations, relationships and semantics.
Class diagram	Shows a set of classes, interfaces and collaborations and their relationships. They address the static view of the system.
Collaboration	Defines an interaction and is a society of roles and other elements that work together to provide some co-operative behaviour. Collaboration has structural and behavioural dimension.
Collaboration Diagram	These diagrams are similar to sequence diagrams with the exception that collaboration diagrams do not focus on the sequence of messages.

Component Diagram	These diagrams represent the physical architecture of the system.
Composition	A stronger aggregate association where the individual items become a new object.
Conceptual models	A model of a software system, which focus on the vocabulary, and rules of the system.
Conceptual schema	A visual representation of the ORM conceptual model.
Dependency relationship	Occur between two classes where one class is dependent on the other class.
Deployment Diagram	Deployment diagrams describe the physical architecture of the hardware and software in the system.
Disjunctive mandatory role constraints	Means that the disjunction of two or more roles in the ORM model is mandatory.
Domain experts	Experts who can evaluate a domain model.
Elementary fact	Indicates that a given fact cannot be broken down into smaller facts that can be combined to yield the original fact.
Equality constraint	Can be applied between two or more role sequences played by an object type in ORM where the population of the first role sequence must be equal to the population of the second role sequence.
Fact	Associates many objects that take part in relationships and is a combination of entities, attributes and relationships in ORM.
Frequency constraint	Used to indicate the exact number or range of numbers of occurrences of the instance of the associated role in ORM.
Information hiding	Although objects have the ability to communicate with one another across well-defined interfaces, objects are not aware of how other objects are implemented.
Inheritance	The mechanism by which the more-specific class includes the structure, which is inherited by the less-specific class.
Interaction	Set of messages exchanged among a set of objects.

Mandatory role constraint	Indicates that every entity in the ORM model must have an associated value.
Method	A method is an implementation of an operation.
Model	An abstract representation of something that suppresses unnecessary information.
Multiplicity constraint	Explain how many instances of one object or class, can be associated with one instance of another object or class.
Notation	Comprises a set of symbols together with a set of rules in order to define the use of the symbols.
Object	A thing that encapsulates data and behaviour.
Object diagrams	They model object instances, showing the current values of the instance's attributes.
Object modelling	A technique for identifying objects within the systems environment and the relationships among those objects.
Object orientation	A set of design and development principles based on objects, where an object represents a real-world entity with the ability to interact with itself and with other objects.
Objectified relationship type	Can be considered as an entity having other entities related to it in ORM.
Operation	An implementation of a service that can be requested from any object of the class to affect behaviour. A class may have any number of operations or no operations at all.
ORM	Object Role Modelling is a fact-based technique for constructing software blueprints.
Primary actor	The main actor who will perform the use case.
Private visibility	Indicates that the class cannot be referenced from other classes.

Public visibility	Indicates that the attribute in a class can be referenced outside the class.
Qualifier	An attribute or list of attributes whose values serve to partition the set of objects associated with an object across an association.
Reference type	A relationship type that is used only for primary referencing in ORM.
Requirements	Requirement represents a desired feature of the system.
Secondary actor	Additional actors other than the primary actor who will perform the use case.
Sequence Diagram	These diagrams represent how objects interact with one another via messages in the execution of a use case or operation.
Specialisation	Groups attributes and behaviours that are common to several types of object classes into a class called supertype.
State Diagram	State diagrams are used to model the dynamic behaviour of an object.
Static model	Structural things are the nouns of UML models. These are the static parts of a model, representing elements that are conceptual.
Stereotype	An additional annotation that can be used to enhance the standard UML notation by adding additional semantics to a class.
Structural modelling	Modelling with emphasis on the organisation of the system.
UML	Unified Modelling Language is a standard language for writing software blueprints.
Unified Process	Software development process that is use-case driven, architecture centric, iterative and incremental
Uniqueness constraint	Identifies a unique occurrence of an entity in ORM.
Use-case	An orderly description of the events that happen in a system.

Use-case diagram	Organising and modelling the behaviour of a system using use-cases.
Validate	Confirm the correctness.
Verbalise	Express in words.
Vocabulary and rules	Tell you how to create and read well-formed models.