Vrije Universiteit Brussel

FACULTY OF SCIENCE
Department of Computer Science
System and Software Engineering Lab

Inconsistency Management in Model-Driven Engineering

An Approach using Description Logics

Ragnhild Van Der Straeten

A dissertation submitted in partial fulfilment of the requirements of the degree of Doctor in Science

September 2005

Advisors: Prof. Dr. Viviane Jonckers, Prof. Dr. Tom Mens

Vrije Universiteit Brussel

FACULTEIT VAN DE WETENSCHAPPEN
Vakgroep Informatica
Laboratorium voor Systeem en Software Engineering

Inconsistentiebeheer in modelgebaseerde ontwikkeling

Een benadering gebruikmakend van Description Logics

Ragnhild Van Der Straeten

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen

September 2005

Promotoren: Prof. Dr. Viviane Jonckers, Prof. Dr. Tom Mens

Abstract

Model-driven engineering (MDE) is an approach to software development where the primary focus is on models, as opposed to source code. Models are built representing different views on a software system. Models can be refined, evolved into a new version, and can be used to generate executable code. The ultimate goal is to raise the level of abstraction, and to develop and evolve complex software systems by manipulating models only. The manipulation of models is achieved by means of *model transformation*. Because model-driven engineering is still in its infancy, there is a need for sophisticated formalisms, techniques and associated tools supporting model development, evolution and transformation.

The main concern of the research presented in this dissertation, is the *definition, detection and resolution of model inconsistencies*. We focus on two aspects of these activities: defining and resolving inconsistencies, and a feasibility study of Description Logics (DLs) as a formalism for supporting these activities.

Since the Unified Modeling Language (UML) is the generally accepted object-oriented modelling language, it ought to play an essential role in MDE. A software design is typically specified as a collection of different UML diagrams. Because different aspects of the software system are covered by different UML diagrams, there is an inherent risk that the overall specification of the system is inconsistent. Also model transformations, such as (arbitrary) model evolutions, can transform a model into an inconsistent state. Unfortunately, current-day UML CASE tools provide poor support for managing inconsistencies between (evolving) UML models.

Inconsistency management is a complex process consisting of different activities. It is a well-studied process that is also well-known within software engineering. However, in the UML context, little research has been done taking into account a wide range of inconsistencies over different kinds of UML diagrams. Inconsistency management in the UML context, is quite complicated due to several reasons. The most obvious reasons are the missing formal semantics for the UML and the UML being a general purpose language that can be applied to several application domains and in several software development processes.

To be able to define the occurrence of UML model inconsistencies precisely and unambiguously, there is first of all a need to formalise the UML's abstract syntax and concepts. A first contribution is to *formalise an important fragment of the abstract syntax*. The studied UML fragment allows the specification of the static structure of an application and the specification of behaviour of individual objects and the possible interactions between objects. As a second contribution, we propose a *classification of inconsistencies*. The definitions of these inconsistencies are based on our formalisation. Inconsistency management, as a process, also includes the activity of *resolving inconsistencies*. Different resolution strategies are known in literature. The resolution of inconsistencies gives rise to some particular challenges. We give an overview of these challenges in the context of our classified inconsistencies.

From the formalisation of a fragment of the UML abstract syntax, from our classification of inconsistencies, from the definition of different inconsistencies and from the different resolution strategies, we distil a set of *key criteria*. The requirements for each of these

criteria can be used to evaluate a formalism and tool support supporting the detection and resolution of inconsistencies.

Through the second focus of this dissertation, we discovered that Description Logics (DLs) and DL systems are suited (or can be made suited) for the detection and resolution of inconsistencies on a fairly high-level of model abstraction.

DL is a two-variable fragment of first-order predicate logic, defining a family of logic languages, offering a classification task based on the subconcept-superconcept relationship. DLs are very suited for reasoning about hierarchies and about the satisfiability of knowledge bases. Different DL systems are developed and can be used to validate this formalism for the purpose of inconsistency detection and resolution.

DLs are validated against our key criteria in three successive steps. First, we investigate to which extent it is possible to *encode the fragment under study of the abstract syntax of the UML*. We also answer the question if *DLs can be used as a semantic domain* for some possible semantics for UML diagrams. Second, we show *how inconsistencies can be detected using this formalism*. Finally, we investigate if it is possible to *resolve inconsistencies using DLs* and the capabilities of state-of-the-art DL systems.

Model transformations are considered to be the heart and soul of MDE. One particular kind of model transformation and evolution is *model refactoring*. Model refactorings restructure models as opposed to source code refactorings, which restructure source code and are well-known and well-studied. Model refactorings preserve behaviour. We show how some of the classified inconsistencies correspond to *behaviour preserving properties* that can be expressed between a UML model and its refactored version. A second idea about model refactorings introduced in this dissertation, is to use inconsistency detection and resolution techniques for supporting a software engineer in executing model refactorings.

Our ideas are illustrated and validated on a simplified, yet complex enough, set of models of an Automatic Teller Machine (ATM) simulation application using a prototype tool, called *RACOoN*. This tool is integrated in a commercial UML CASE tool using the latter's built-in plugin mechanism.

Samenvatting

Modelgebaseerde ontwikkeling is een bepaalde benadering van software ontwikkeling waarbij de nadruk ligt op modellen in plaats van op broncode. Modellen worden gebouwd vanuit een bepaalde kijk op het software systeem. Modellen kunnen verfijnd worden, evolueren naar nieuwe versies en kunnen, bijvoorbeeld, gebruikt worden om broncode te genereren. Het ultieme doel van modelgebaseerde ontwikkeling is om het abstractieniveau te verhogen en om complexe software systemen te ontwikkelen en te laten evolueren enkel door modelmanipulatie. Deze modelmanipulaties worden bekomen door middel van modeltransformaties. Modelgebaseerde ontwikkeling staat echter nog in zijn kinderschoenen waardoor er een nood is aan gesofisticeerde formalismen, technieken en geassocieerde programma-ondersteuning voor het ontwerpen van modellen, de evolutie van modellen en transformatie van modellen.

De hoofdinteresse van het onderzoek dat gepresenteerd wordt in deze thesis, is de *definitie, detectie en resolutie van model inconsistenties*. De aandacht wordt gevestigd op twee aspecten van deze activiteiten, nl., de definitie van inconsistenties en resolutie strategieën, en een onderzoek naar de haalbaarheid van "Description Logics" (DLs) ter ondersteuning van deze activiteiten.

Vermits de "Unified Modeling Language" (UML) de standaard objectgeoriënteerde modelleertaal is, zou het een belangrijke rol moeten spelen in modelgebaseerde ontwikkeling. Een software ontwerp is meestal een collectie van UML diagrammen, waardoor er een inherent risico is dat de globale specificatie van het systeem inconsistent is. Modeltransformaties, zoals bijvoorbeeld een willekeurige evolutie van het model, kunnen inconsistenties veroorzaken. De bestaande UML ondersteunende programma's bieden echter onvoldoende ondersteuning voor het beheren van inconsistenties tussen (evoluerende) UML modellen.

Inconsistentiebeheer is een complex proces bestaande uit verschillende activiteiten. Het is een goed bestudeerd proces dat bekend is binnen software ontwikkeling. In de context van UML is er nog niet zo veel onderzoek gedaan naar verschillende soorten inconsistenties over een aantal overlappende UML diagrammen. En dit omwille van een aantal redenen. De belangrijkste zijn: UML heeft geen formele semantiek, daarbovenop kan één enkel diagram op verschillende manieren geïnterpreteerd worden en UML is een taal voor algemeen gebruik in de zin dat ze kan gebruikt worden voor verschillende applicatiedomeinen en binnen verschillende software ontwikkelingsprocessen.

Om UML model inconsistenties op een precieze manier te kunnen definiëren, hebben we nood aan een formalisatie van de abstracte syntax van UML en een mogelijke semantiek. Een eerste bijdrage van dit werk is de formalisatie van een belangrijk deel van de abstracte syntax van UML. Dit deel van UML laat toe om de statische structuur van een applicatie te specificeren en eveneens de specificatie van het gedrag van een bepaald object en de mogelijke interacties tussen objecten. Een tweede bijdrage van dit werk bestaat uit een classificatie van mogelijke inconsistenties tussen en binnen modellen die beantwoorden aan het geformaliseerde deel van UML. Deze inconsistenties worden gedefinieerd op een precieze manier aan de hand van de vooropgestelde formalisatie. Inconsistentiebeheer omvat niet alleen de definitie maar ook het oplossen van inconsistenties. Deze activiteit brengt een aantal specifieke uitdagingen met zich mee. We geven dan ook een overzicht van deze uitdagingen.

Gebaseerd op onze formalisatie, op de classificatie van een verzameling inconsistenties en op de verschillende uitdagingen gepaard gaande met het oplossen van inconsistenties, kunnen een aantal *sleutelcriteria* gedistilleerd worden. Elk van deze criteria heeft een aantal specifieke vereisten die gebruikt kunnen worden om een formalisme en ondersteuning voor de definitie, detectie en oplossing van inconsistenties, te evalueren.

Het tweede aspect waarop de aandacht gevestigd wordt in deze thesis, is de evaluatie van "Description Logics" als formalisme voor inconsistentie detectie en resolutie en dit aan de hand van de sleutelcriteria. DL is een fragment van eerste-orde predikatenlogica en definieert een familie van logische talen. Deze talen bieden een classificatietaak aan gebaseerd op een subconcept-superconcept relatie. Ze zijn zeer geschikt voor het redeneren over hiërarchieën en over de consistentie van kennisbanken.

In dit werk gaan we niet alleen na in hoeverre DLs geschikt zijn voor het representeren van het door ons geformaliseerde UML fragment, maar ook in hoeverre de gedefinieerde inconsistenties kunnen gedetecteerd worden en in hoeverre het mogelijk is om inconsistenties op te lossen gebruik makende van deze logica's en de bestaande ondersteunende systemen.

Modeltransformaties worden beschouwd als zijnde het hart en de ziel van modelgebaseerde ontwikkeling. Modelherstructureringen zijn één welbepaalde soort van modeltransformaties. Deze transformaties herstructureren modellen. De tegenhanger van modelherstructureringen zijn broncodeherstructureringen. Deze herstructureringen zijn bekend en goed bestudeerd. Herstructureringen bewaren gedragseigenschappen die uitgedrukt kunnen worden tussen het originele model en het geherstructureerde. Wij tonen hoe een aantal van onze gedefinieerde consistenties corresponderen met bepaalde gedragseigenschappen. Een tweede idee is om inconsistentie detectie en resolutie technieken te gebruiken ter ondersteuning van het uitvoeren van modelherstructureringen door een software ontwikkelaar.

Om de ideeën en benadering voorgesteld in deze thesis te illustreren en te valideren wordt gebruik gemaakt van een verzameling modellen die de specificatie van een geldautomaat simulatie voorstellen. Hiervoor hebben we een prototype programma ontwikkeld, genaamd *RACOoN*, dat geïntegreerd is in een bestaand UML ondersteunend programma.

Acknowledgements

First, I would like to thank Prof. Dr. Viviane Jonckers, for granting me the opportunity to obtain a Ph.D., for allowing me to freely choose research topics I believed interesting, for supporting me these past years and for proofreading this dissertation.

I am also greatly indebted to Prof. Dr. Tom Mens. Without his support and collaboration during the last two years, I would probably still have to start writing. Our collaboration also resulted in writing several papers together. He read every draft of this dissertation and always provided me with valuable comments.

I owe my gratitude to my Ph.D. committee members, for taking the time to read this dissertation in detail and for providing me with valuable comments. Apart from my advisors Viviane and Tom, the committee members are Prof. Dr. Theo D'Hondt, Prof. Dr. Dirk Vermeir, Prof. Dr. Geert-Jan Houben, Prof. Dr. Gregor Engels and Prof. Dr. Ralf Möller.

I should not forget to thank all the people that proofread parts of this dissertation. Besides Tom, Viviane and Ralf, Dr. Maja D'Hondt and Prof. Dr. Kim Mens provided me with valuable comments.

Many thanks also go to Jocelyn Simmonds. During her master's thesis, Jocelyn helped shaping some of the ideas of this dissertation. She also started developing the prototype tool presented in this dissertation.

I am grateful to the people of the DL community: to Dr. Ulrike Sattler and Dr. Carsten Lutz for introducing me in this community, to Prof. Dr. Volker Haarslev, Ralf and Michael Wessel for developing the RACER system. Special thanks goes to Ralf and Michael for always answering the enormous amount of e-mails and questions I sent them about DLs and RACER.

I owe my gratitude to my colleague Maja for brainstorming about the rule-based approach presented in this dissertation, and for the nice little chats we had and the way she encouraged me to finish this dissertation. I am also grateful to my colleague Dennis Wagelaar for the discussions on model-driven engineering. Many thanks also go to all my other colleagues of the System and Software Engineering Lab for providing an inspiring working environment: Miro Casanova, María Agustina Cibrán, Bruno De Fraine, Niels Joncheere, Davy Suvee, Dr. Wim Vanderperren and Bart Verheecke. I also want to thank the (former) assistants of the department of Computer Science, with whom I shared some teaching tasks, for the nice collaboration: Dr. Katja Verbeeck, Dr. Tom Lenaerts, Steven Claes, Joke Reumers, Bram Vanschoenwinkel and Dr. Joris Van Looveren.

I would like to thank my boyfriend Brecht Vandermeiren for putting up with me, for always being there and for his moral support.

Last but not least, I would like to thank my parents and my grandparents. Special thanks to my mother for asking me a thousand times when my dissertation would finally be finished. She and my father gave me the opportunity to study and to pursue whatever degree I wanted. I also want to thank them for supporting every decision I made.

"Een open geest hebben is cruciaal. Je moet dingen die alle anderen ervaren als voldongen feiten toch in twijfel durven trekken. Het grondig te bekijken om te weten of het echt wel waar is. Daardoor zie je dingen die anderen niet zien, gewoon omdat ze de vraag niet stellen."

Prof. Dr. Catherine Verfaillie in een interview in *De Morgen*,
zaterdag 22 januari 2005.

Contents

List of Figures

List of Tables

List of RACER Fragments

Chapter 1

Introduction

1.1 Problem Statement

Model-driven engineering (MDE) is an approach to software engineering where the primary focus is on models, as opposed to source code. A model describes certain views of the software system at a certain level of abstraction. Models are composed of submodels. Each submodel represents a different *view* on a software system. For example in a bank application, different views are customer management, transaction management and account management. Each of these views is modelled by a different submodel. In the remainder of this dissertation, everywhere the word "model" is mentioned, it can be substituted by submodel unless specified otherwise.

Models can be refined, evolved into a new version, and can be used to generate executable code. The ultimate goal is to raise the level of abstraction, and to develop and evolve complex software systems by manipulating models only. The manipulation of models is achieved by means of *model transformation*, which is considered to be the heart and soul of model-driven engineering [SK03]. Because model-driven engineering is still in its infancy, there is a need for sophisticated formalisms, techniques and associated tools supporting the different model-driven engineering activities.

The *Unified Modeling Language* (UML) is currently the standard modelling language for object-oriented software development and well on its way to become a standard in MDE. Originally, the UML was conceived for modelling object-oriented software systems. However, the class of systems covered by the UML has grown beyond just software systems. The UML now covers any system for which it is useful to make statements about the data maintained or about the behaviour the system exhibits relative to its environment, e.g., businesses, hardware, and so on. The visual representation of UML consists of a set of different diagram types. Each diagram type is described in a certain language. Examples of such languages are class diagrams, sequence diagrams, communication diagrams and state machine diagrams. The different diagram types describe different *aspects* of a software system under study. A class diagram renders the static structure of the system. Sequence diagrams focus on the interaction of different instances of classes, i.e., objects, in a certain context. Communication diagrams describe how different objects are related to each other. Finally, state machines define how the state of a certain object changes over time. A model consists of different such diagrams.

During the software development life-cycle or in an evolution context, models can be

transformed into other models. Two well-known kinds of transformations are *refinements* and *model refactorings*. *Refinements* are performed during the software development life-cycle and transform a model into another model that is on a more detailed level of abstraction. In each refinement step, models get replaced by more refined ones. Model refinement means that more and more detail will be added to models in each refinement step, e.g., the behaviour specification can be extended in a stepwise fashion.

The inherent complexity of (design) models will continue to grow as these models evolve, according Lehman's second law of software evolution: "*As a program is evolved its complexity increases unless work is done to maintain or reduce it*" [LRW+97]. To counter this growing complexity, we need techniques that restructure the design to improve its quality and reduce its complexity. Such techniques, called *model refactorings*, are the design level equivalent of source code refactorings that have been thoroughly studied in literature ([Opd92, Fow99, MT04]).

Inconsistencies in models can arise due to several reasons. Inconsistencies can arise within a model due to the coexistence in the model of different diagrams that overlap with each other. Inconsistencies can also occur between different submodels within a certain model. The different submodels can overlap due to two reasons. First, the submodels describe different aspects of the software application, e.g., static view versus dynamic view, and secondly, the submodels describe different, but mostly related, views of the software application. A model can get refined during the software development life-cycle. Inconsistencies can arise between the original model and its refinement. Evolution of models can also introduce inconsistencies. Inconsistencies can express different things. They can express lack of certain model quality attributes. An inconsistency can express that the specification of a software system does not meet its requirements because the model specifying the implementation of a software system is inconsistent with (part of) the model specifying the requirements of the system. The occurrence of an inconsistency can also give an indication that further analysis of the model(s) is needed. Inconsistencies can express that certain properties are not guaranteed. Inconsistencies can also be interpreted as positive things. For example, due to the occurrence of inconsistencies, the software developer is forced to think again about some issues of the software system and as such, can come to a greater insight in the software system under study.

Current state-of-the-art UML CASE tools provide little or no adequate support for *inconsistency management* or *consistency maintenance*. *Inconsistency management* is a process composed of several activities. As stated in [SZ01], these activities are: *detection of overlaps*, *detection of inconsistencies*, *diagnosis of inconsistencies*, *handling of inconsistencies*, *tracking of inconsistencies*, and *specification and application of a management policy for inconsistencies*. This dissertation focuses on the activities: *detection of inconsistencies* and *handling of inconsistencies*. While in the process of inconsistency management, the existence of inconsistencies is tolerated, *consistency maintenance* emphasises the explicit avoidance of inconsistencies. The ultimate goal of consistency maintenance is to maintain the consistency in whatever situation. This is however, not realistic in a real-world project on which several developers are working at the same time.

Several authors have recognised that inconsistency management is difficult, and more so in the context of UML models. Several reasons are to be stated: (1) UML lacks formal semantics, as a consequence, a single UML diagram can have different interpretations; (2) the distribution of an inconsistency across different diagram types; (3) there may be many

inconsistencies of which some are related; (4) for some applications, some inconsistencies are more important than others. The handling of inconsistencies is considered critical to inconsistency management, but also extremely difficult [Fin00]. Again different reasons are to be stated: (1) handling inconsistencies involves changes in different models; (2) handling a certain inconsistency can be of varying importance for certain domains; (3) handling an inconsistency is in most cases system or application dependent. Due to these reasons, among others, it is hard to define, detect and handle inconsistencies.

In literature different approaches have been proposed for defining and detecting inconsistencies in the context of the UML. Some motivate a general methodology to deal with inconsistencies [EHK01]. Most publications, however, concentrate on particular inconsistencies and defend a certain approach for the detection of these inconsistencies [ET00, EHHS02, EHKG02, FMP99, Tsi01, SKM01].

Literature dealing with the handling of inconsistencies in the context of UML is less abundant. Some focus on the handling of particular inconsistencies, such as syntactic inconsistencies [HHS02], while others focus on a particular level of abstraction, for example, specifications [KZ04] or requirements [vLLD98].

Inconsistency management plays an important role in the context of MDE due to the following reasons.

- Models are assets in MDE. Different views of the software system are covered by different models. Because of the wide variety of models and the many relationships that can exist between them, managing these models is a very complex task and inconsistencies can arise easily.

- A model is described in a certain modelling language, e.g., the UML. The UML contains several diagram types, each described in a certain language. Each model must be legitimate with respect to the languages in which it is expressed.

- Because transformation of models is another important part of MDE, consistency between, e.g., refined models or between different evolved versions of a model is also an important issue.

- For some companies inconsistencies are more than the specification of general coherence rules between or within models. Models are regarded as inconsistent if they do not comply with specific software engineering practices or standards followed by the company.

1.2 Research Objectives and Approach

1.2.1 Objectives

After studying the multitude of work on inconsistency definition and detection and the work on inconsistency handling, we still observe some shortcomings in the context of the UML. Due to lacking formal semantics for the UML, no precise definitions of inconsistencies are given. Such definitions guarantee an unambiguous interpretation of inconsistencies. Most literature focuses only on checking inconsistencies or only on resolving inconsistencies. No

general framework is provided for the detection and handling of different kinds of inconsistencies in the UML context. Because of the important role of inconsistency management in MDE, it is a worthwhile endeavour to investigate the aforementioned problems.

A global objective of this dissertation is to develop a *coherent inconsistency management framework for the definition, detection and handling of inconsistencies in the context of object-oriented models* with special focus on UML models. This framework will enable a *precise definition and detection of inconsistencies* and the semi-automatic *handling* of inconsistencies.

To obtain our global objective, we propose the objective of using a *declarative formalism for this inconsistency management framework*. The *use of a fragment of first-order logic as declarative formalism* will be evaluated. The reasoning capabilities of this formalism can be used to reason about models. We believe that adequate tool support for model-driven engineering has to be simple and quite powerful and as such profits from having a formal framework as its foundation. This leads us to our research hypothesis.

By using a fragment of first-order logic, i.e., Description Logics, as underlying formalism of a coherent inconsistency management framework, the precise definition of inconsistencies and the automatic detection and semi-automatic handling of inconsistencies between different UML models can be supported by well-defined procedures.

1.2.2 Approach

The approach to achieve our objectives is the following.

We will formalise an important fragment of the UML version 2.0 metamodel. This formalisation must satisfy the following criteria: (1) It will be powerful enough to capture the essence of object-oriented design in the UML. (2) We want a general formalisation, i.e., independent of any specific detection or handling procedure. As a result, different detection or handling approaches can be based on this formalisation. (3) The formalisation must allow the expression of different interpretations of UML diagrams. Some UML diagrams can be interpreted in different ways. Sequence diagrams are such a kind of diagram. It is not our intention to capture all possible interpretations, but basic semantics will be provided on which more elaborate and customised semantics can be built. The interpretations of the different diagrams will be discussed in the context of a concrete formalism for the detection and handling of inconsistencies. As a result, it is possible to show which formalism dependent constructs are needed to encode the different interpretations.

We will present a set of inconsistencies based on the model elements occurring in the fragment of the UML metamodel we restrict ourselves to. Of course, it is impossible to present an exhaustive list of possible inconsistencies in UML models. One of the reasons is that inconsistencies are often domain dependent. We will ignore inconsistencies that are violations of UML well-formedness rules because these rules are considered to be integrated in the modelling language and to be part of the language's abstract syntax.

Description Logics (DLs) that are a fragment of first-order logic, are investigated as a formalism for the definition, detection and handling of inconsistencies. DLs are a family of logic languages that are primarily used for modelling database conceptual schemata and ontology engineering. DLs and DL systems are evaluated in this dissertation for the purpose of inconsistency detection and handling. This evaluation is based on a set of *key criteria*.

These key criteria are distilled from literature studies on inconsistency detection and the different handling and resolution strategies and from our classification of inconsistencies.

In the remainder of this chapter, first, we introduce MDE. Secondly, model refactorings are explained because we will show how some of our ideas can be applied to the domain of model refactorings. Next we introduce inconsistency management, the criteria used to evaluate potential formalisms for inconsistency detection and handling, and finally, we present our contributions. At the end of this chapter, we provide the reader with an outline of this dissertation.

1.3 Model-Driven Engineering

Models are the primary assets in MDE. In this section, we first introduce how models are defined in literature. Next, we give our own definition of a model independent of any modelling language. We also discuss the role of the UML in MDE. Because model transformation is the heart and soul of MDE and because we will show in this dissertation how some of our ideas can be used for the support of model refactorings, we consider a classification of model transformations given in literature.

"Model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing". [MCF03]

MDE aims to make models the primary assets in all aspects of software engineering. The ultimate goal is to have a software development environment at our disposal with off-the-shelf models and mapping functions that transform one model into another. Instead of building and rebuilding systems as the application or technology changes, models are selected, subsets are taken or models are extended, woven together with other models to build the system. It goes without saying that inconsistency management will play a prominent role in the achievement of this goal.

1.3.1 What is a Model?

In [MCF03], a model is defined as:
A model is a coherent set of formal elements describing something (e.g., a system, bank, phone or a train) built for some purpose that is amenable to a particular form of analysis, such as

- *communication of ideas between people and machines,*
- *completeness checking,*
- *race condition analysis,*
- *test case generation,*
- *viability in terms of indicators such as cost and estimation,*
- *standards,*
- *transformation into an implementation.*

Because our objective is to define, detect and handle inconsistencies on models, we propose the following definition of a model:

Definition 1 *A* model *is an accurate (partial) description of a system under study at some level of abstraction.*

The following statements all apply to our notion of a model:

- *A model consists of several submodels describing a certain view of a system under study.*

  A complex problem can better be divided into smaller pieces each describing different parts of the solution from different viewpoints. A large design is traditionally split into a part describing the static structure of the system under study and into a part describing the dynamic structure. To be manageable, both parts are even further subdivided. Removing or hiding detail that is irrelevant for a given viewpoint lets us understand the essence more easily and this is also a means to cope with complexity. Another advantage of decoupling several models is their ability to evolve independently which in turn increases the models' longevity.

  However, working with multiple, interrelated models requires a significant effort to ensure their overall consistency.

- *A model is expressed in some language existing at some level of language abstraction.* Abstraction is a well-known means to cope with complexity. For example, it is easier to write a certain program in a high-level program language such as Java then to encode it in an assembler language. A modelling language enables the expression of different views and aspects of a system under study in (different) models.

- *A model needs not be complete.* In an early phase of the software development life-cycle, models are often used as a way to communicate designs describing important, high-level concerns. These models are not complete. For example, a model can emphasise the collaboration of different objects in a sequence diagram and hide the behaviour specification of a single object as specified in a state machine diagram.

  Remark that incompleteness and a high degree of abstraction do not equate to imprecision.

1.3.2 The Role of the UML in MDE

MDE does not require the UML. Any modelling language can be used. However, the UML has become the de facto modelling language for modelling software systems. Models expressed in the UML, capture knowledge about a system at different abstraction levels, from requirements and analysis models over design models to models used as programming specifications. Originally, the UML was conceived for modelling object-oriented software systems. Nowadays, the UML covers any system for which it is useful to make statements about the data maintained or the behaviour the system exhibits relative to its environment, e.g., businesses, hardware and so on. The different diagram types of the UML are introduced in Chapter 2.

At the heart of the role of the UML in MDE are the different ways in which developers want to use it. The UML has several applications in MDE. It can be used as a general-purpose modelling language. It can also be used as a basis for (domain-specific) extension and reuse.

Martin Fowler argued in his invited talk at the ≪UML2003≫ conference and in [Fow99], that there are three different modes in which developers use the UML: sketch, blueprint, and programming language.

1. *UML as sketch* This usage is used to communicate informally software designs describing important things but that are not detailed. The essence of sketching is selectivity. The models do not give a complete view of the software system. Only the most important aspects are selected for communication. Informal tools such as a whiteboard are used to draw and communicate.

2. *UML as blueprint* The models are used to communicate detailed instructions to programmers. The essence of blueprint is completeness. The models should be sufficiently complete in that all design decisions are laid out. A precise notation and semantics are necessary in this case. Blueprints require more sophisticated modelling tools than sketches do.

3. *UML as programming language* If the UML is used as a programming language, the UML models drive the whole process. UML diagrams are compiled directly to executable code, and the UML becomes the source code. It is obvious that this usage of UML demands sophisticated tools and further research is necessary to obtain this usage.

1.3.3 Transformation of Models

Models are specified at different levels of abstraction and sometimes also in different languages. Transformation from one or multiple source models to one or multiple target models, called *model transformation* is an important issue within MDE. In Kleppe *et al.* [KWB03] the following definition of model transformation is provided:

Definition 2 *A* transformation *is an automatic generation of a target model from a source model, according to a transformation definition. A* transformation definition *is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A* transformation rule *is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.*

In [MCV05], a taxonomy of model transformations is discussed. Two orthogonal dimensions are defined, *horizontal versus vertical* and *rephrasing versus translation.*

- *Horizontal transformation* indicates transformation between different models at the same level of abstraction. *Model refactoring* is an example of such a transformation because the source model is restructured and the target models are at the same level of abstraction. In the next section, we go into more detail on model refactoring.

- *Vertical transformation* indicates a transformation where the source and target models reside at different levels of abstraction. *Refinement* is an example of such a transformation. The original model and its refined version are at different levels of abstraction.

- *Rephrasing* indicates a transformation where the models are expressed in the same modelling language. This kind of transformation is also called an *endogenous* transformation. Examples of rephrasing are *optimisation*, which aims at improving certain operational properties while preserving the semantics of the software, and *refactoring* which aims at improving certain software quality characteristics while preserving the software's behaviour.

- *Translation* indicates a transformation where the source and target models are expressed in different languages. This kind of transformation is also called an *exogenous* transformation. Examples of translation are *reverse engineering* which extracts a higher-level specification from a lower-level one, and *migration* which translates a program written in one language to another, while keeping the same level of abstraction.

1.4 Model Refactoring

Software restructuring is "the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behaviour (functionality and semantics). [...] While restructuring creates new versions that implement or propose change to the subject system, it does not normally involve modifications because of new requirements. However, it may lead to better observations of the subject system that suggest changes that would improve aspects of the system. Restructuring is often used as a form of preventive maintenance to improve the physical state of the subject system with respect to some preferred standard" [CC90]. In an object-oriented context, the term *refactoring* is used instead of *restructuring* [Opd92, Fow99].

The inherent complexity of (design) models will continue to grow as these models evolve, following Lehman's second law of software evolution: "As a program is evolved its complexity increases unless work is done to maintain or reduce it" [LRW+97]. To counter this growing complexity, we need techniques that restructure the design to improve its quality and reduce its complexity. Such techniques, called *model refactorings*, are the design level equivalent of source code refactorings that have been thoroughly studied in literature ([Opd92, Fow99, MT04]).

We define a model refactoring as follows:

Definition 3 *A* model refactoring *is a transformation of a model into a different model expressed in the same modelling language to improve quality attributes of the models of the system under study without changing the modelled external behaviour of the system under study.*

Although source code can be regarded as a model and it meets Definition 1, we assume that in the above definition, models do not refer to source code. As such, this definition is the design level equivalent of the one of source code refactoring [Fow99].

Following the aforementioned Lehman's second law of software evolution, model refactorings play an important role in the context of MDE. Refactorings can be situated on different levels within MDE, e.g., on requirement documents, design models or source code. For example, in the context of the Model-driven Architecture (MDA), model refactorings can be applied on two different levels. MDA distinguishes between Platform Independent

Models and Platform Specific Models. Platform Independent Models as well as Platform Specific Models can be refactored. Platform Independent Model refactorings boil down to a restructuring of the Platform Independent Models independent of any platform. These restructurings improve simplicity of the models. In case of Platform Specific Models, dependent on the chosen platform, different platform-specific refactorings can be executed on Platform Specific Models.

Like the process of source code refactoring [MT04], the process of model refactoring consists of some distinct activities:

1. Identification of where the design should be refactored.
2. Determine which model refactoring(s) should be applied to the identified places.
3. Apply the model refactoring(s).
4. Guarantee that the applied model refactoring preserves behaviour.
5. Assess the effect of the refactoring on quality characteristics of the design (such as complexity, understandability, maintainability).
6. Maintain consistency between the refactored design and other software artifacts (such as requirements, program code, etc.).

Topic 6 is treated by the process of inconsistency management, which is, as explained in Section 1.5 a research area in its own right.

In this dissertation, we will show how the definition of some consistencies correspond to behaviour preservation properties (topic 4) and how our ideas for inconsistency resolution can be used to support the application of model refactorings (topic 3). In the remainder of this section, we will go into more detail on the topic of behaviour preservation.

A model refactoring is not supposed to change the behaviour specified by the models in question. Model refactoring is a rather recent research issue and as such definitions of behaviour preservation properties are not yet given. Moreover, in the context of the UML such definitions do not exist because there is no consensus on a formal definition of behaviour. Also for source code refactorings, definitions of behaviour preservation are rarely provided. Opdyke [Opd92] suggests the following definition of behaviour preservation: "for the same set of input values, the resulting set of output values should be the same before and after the refactoring". To ensure this kind of behaviour preservation, *refactoring preconditions* and *postconditions* need to be specified [Rob97]. However, as explained by [MT04], this kind of behaviour preservation is sometimes insufficient since many other aspects of the behaviour may be relevant as well. This implies the need for a wide range of definitions of behaviour preservation depending on domain-specific or user-specific or company-specific concerns.

Behaviour preservation can also be dealt with in a more pragmatic approach. A first approach is by means of rigourous testing. Another pragmatic approach is to specify a weaker notion of behaviour preservation that is not sufficient to guarantee the full program semantics preservation, but focuses on specific issues. Mens *et al.* [MDJ02] adopt this approach and specify different kinds of behaviour preservation.

At design level, specifications of behaviour of the software are provided in a certain modelling language, e.g., the UML. These specifications need not be complete, as the models

containing these specifications are also not necessarily complete. In the context of model refactorings, questions such as which kinds of behaviour are important for which model refactorings and how behaviour preservation can be checked, preserved or proved, arise. We will focus on these questions in Chapter 9.

1.5 Inconsistency Management

1.5.1 Inconsistency Management Process

Inconsistency management has been defined in Finkelstein *et al.* [FST96] as "the process by which inconsistencies between software models are handled so as to support the goals of the stakeholders concerned".

Finkelstein *et al.* [FST96] and Nuseibeh *et al.* [NER00] propose general frameworks describing the activities of this process. Both approaches agree that the process of inconsistency management includes activities for detecting, diagnosing, and handling them. These activities are extended by Spanoudakis and Zisman [SZ01]: detection of overlaps, detection of inconsistencies, diagnosis of inconsistencies, handling of inconsistencies, tracking, specification and application of an inconsistency management policy. Additionally, Spanoudakis and Zisman present a survey of techniques and methods supporting the management of inconsistencies.

The *definition of particular inconsistencies* is not included in the process of managing inconsistencies. In our opinion this must be included in the process of inconsistency management too. As recognised by Spanoudakis and Zisman, different kinds of inconsistencies can be distinguished. The definition of a particular inconsistency must be unambiguous. This makes one single, clear interpretation of each inconsistency by the software developer possible.

Detection of inconsistencies is the activity of checking for inconsistencies in software models. Different approaches to the detection of inconsistencies are possible. In Spanoudakis and Zisman, four broad categories of approaches are listed. In the *logic-based approach*, models are expressed in some logic language. The inconsistency procedures are well-defined an have sound semantics. The logic-based approach is applicable to arbitrary consistency rules. As disadvantages, Spanoudakis and Zisman report on the semi-decidability of first-order logic and the inefficiency of theorem proving. The *model checking* approach translates models in a particular state-oriented language. The results of the translation are used by a model checker, applying specialised model checking algorithms. These algorithms are used to detect certain inconsistencies. In a model checking approach the inconsistency detection procedures are well-defined and have sound semantics, but due to state explosion the approach is not always efficient and only specific kinds of consistency rules can be checked. A third approach is to use *specialised forms of automated analysis*. Models are expressed or translated into a specific language and specific kinds of consistency rules can be checked. A fourth approach is the usage of *human-based collaborative exploration*. Models are expressed in informal modelling languages. Stakeholders are expected to inspect the models on inconsistencies. This approach is very labour intensive and difficult to use with large models. In our work, we prefer the logic-based approach and restrict our approach to a decidable fragment of first-order logic. We will motivate this approach in Chapter 5.

Diagnosis of inconsistencies is "concerned with the identification of the source, the cause and the impact of an inconsistency" [SZ01]. The source and the cause of an inconsistency can play an important role in the activity of *inconsistency handling*.

Inconsistency handling is concerned with the following activities according to [SZ01]:

1. the identification of the possible actions for dealing with an inconsistency,
2. the evaluation of the cost and the benefits that would arise from the application of each of these actions,
3. the evaluation of the risks that would arise from not resolving the inconsistency, and
4. the selection of one of the actions to execute.

A last activity that must be added to the above list, is the execution of the selected action. There are different techniques for handling an inconsistency. The model can just be marked as being inconsistent and the modelling elements involved in the inconsistency can be annotated. It is up to the software designer to take action or to leave the inconsistency in the model. In some cases, the model can be corrected manually, in other cases, it can be corrected automatically or semi-automatically. If the model is corrected, the *inconsistency* is *resolved* through so-called *resolution actions*. Our work focuses on the identification of the possible resolution actions and the execution of these actions.

1.5.2 Dimensions of Consistencies

The model of a system under study is typically expressed as a (large) collection of interdependent and partially overlapping submodels. These models describe the system from different viewpoints and at different levels of abstraction and granularity. These models may also be expressed using different notations, and different software developers can be involved in the software modeling process. All these issues are bound to lead to inconsistencies among models. An *inconsistency* is described in [SZ01] as "a state in which two or more overlapping elements of different software models make assertions about aspects of the system they describe which are not jointly satisfiable".

Overlaps are defined as "relations between interpretations ascribed to software models by specific agents" [SZ01]. An agent may be a person or a computational process. Models overlap when they incorporate elements referring to common concerns of the system under study.

Nuseibeh *et al.* [NER00] define an inconsistency as "*any situation in which a set of descriptions does not obey some relationship that should hold between them. The relationship between descriptions can be expressed as a consistency rule against which the descriptions can be checked*".

In literature, two orthogonal dimensions of consistencies are described. The first dimension concerns *horizontal* versus *vertical* versus *evolution* consistency. The second dimension concerns *syntactic* versus *semantic* consistency.

As illustrated in Figure 1.1 and described in the literature, a first dimension distinguishes between three different types of consistencies.

1. *Horizontal consistency* [KRSH02], also named intra-consistency or intra-model consistency, indicates consistency within a model or between different models at the same

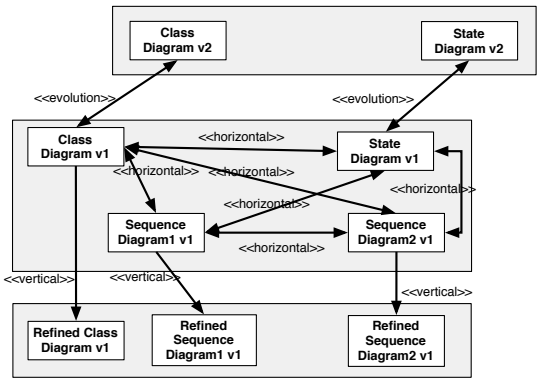

Figure 1.1: Three types of consistency between UML models.

level of abstraction, and within the same version. For example, in any UML model, the class diagrams and associated sequence diagrams and state diagrams should be mutually consistent.

2. *Evolution consistency* [EKHG02] indicates the consistency between different versions of the same model. For example, when a class diagram evolves, it is possible that its associated state diagrams and sequence diagrams become partially inconsistent.

3. *Vertical consistency* [KRSH02], also named inter-consistency or inter-model consistency, indicates the consistency between models at different levels of abstraction. "Vertical" refers to the process of refining models and requires the refined model to be consistent with the one it refines [EKHG01].

A second dimension, which is orthogonal to the first, distinguishes between syntactic and semantic consistencies.

1. *Syntactic consistency* ensures that a specification conforms to the abstract syntax of the modelling language, specified by, for example, a metamodel. This guarantees that the overall model be well-formed.

2. *Semantic consistency* requires models to be semantically compatible. Semantic compatibility addresses the compatibility of the meaning of the different models. To be able to define this notion, the meaning of a model must be well-defined. In general, in a horizontal consistency setting, semantic consistency requires *the different submodels of a model* to be semantically compatible. In an evolution context, *the refactored model* needs to be semantically compatible with the *one it refactors*. With respect to vertical consistency, this requires *a refined model* to be semantically compatible with *the one it refines*.

1.6 Key Criteria

From the literature on the different approaches and formalisms used to support the different activities of inconsistency management (especially inconsistency definition, detection and

handling) and from the research we did on those topics, we distilled a set of *key criteria*. Each criterion consists of some requirements. These requirements must be (partially) met by approaches or formalisms supporting definition, detection and handling of inconsistencies. We will briefly introduce each criterion here, but detail them in Chapter 4.

- *Syntax and semantics representation.* It must be possible to express the abstract syntax of the modelling language in the formalism, this enables well-formedness checking of the user-defined models. The representation of the semantics of the modelling language must be supported. In some cases, this might require the integration of different formal languages defining the semantics of the different sublanguages involved in the modelling language. In other cases, it might be sufficient to have a formalism that is powerful enough to define the semantics of (most of) these sublanguages.

- *Precise definitions of inconsistencies and inconsistency detection.* The formalism must enable the precise definition of inconsistencies and provide a mechanism to detect the inconsistencies. Different kinds of inconsistencies must be expressible in the formalism. In terms of the dimensions mentioned in the previous section, it must be possible to define syntactic and semantic inconsistencies. The abstract syntax of the language as well as the semantics of the language can be used to define (in)consistencies.

  Furthermore, it would be desirable to have a detection mechanism that has some formal properties such as soundness, completeness and decidability.

- *Precise definitions and management of interactive inconsistency resolutions.* A formalism for inconsistency resolution must enable the precise definition of *resolutions for inconsistencies*. Resolutions applied on a certain model resolve particular inconsistencies in this model.

  A mechanism for executing these resolutions is demanded. This mechanism must be highly interactive. Several resolutions can be applied to resolve a certain inconsistency. In some cases, the software developer has the responsibility to decide which resolution will be executed and which not. As a result the inconsistency resolutions are interactively managed.

1.7 Contributions

The contributions of this dissertation are:

- We present a lightweight formalisation of an important fragment of UML 2.0. This formalisation serves as a basis for the definition, detection and handling of inconsistencies. It enables an unambiguous specification of the different mentioned activities. In a more general context, it can also be seen as a basis for the formalisation of the complete UML language.

- We provide a two-dimensional classification of domain-independent inconsistencies. Each of these inconsistencies is defined using the aforementioned formalisation. We do not claim that this is an exhaustive list of inconsistencies, but we show that this is a relevant list. This classification results in a set of possible, unambiguous inconsistencies. Moreover, new inconsistencies can be added to the classification easily, if desired.

- We show to which extent DLs can be used for the formalisation and encoding of the abstract syntax of the UML fragment we formalised. This straightforwardly enables the well-formedness of the UML models.

- We also show how DLs can be used as a semantic domain for the different UML diagram types considered. Consequently, the possible meanings of these diagram types are determined and well-defined.

- Using the two previous contributions, we define the detection of inconsistencies in DLs. These definitions have, among other things, led to the development of a sophisticated query language for RACER, one of the state-of-the-art DL systems. This query language has been developed by the RACER authors based on, among other things, our concrete input.

- We define an approach for the handling of the classified inconsistencies. This approach is a rule-based system using DLs. This system allows for the automatic detection of inconsistencies and it proposes several resolution actions to the user.

- Based on our formalisation of the UML fragment, we show the correspondence between consistencies and behaviour preservation properties. This allows to check and to prove that certain properties hold for model refactorings using exact the same approach as for the detection of certain inconsistencies.

- We introduce the idea to use the rule-based approach for handling inconsistencies to support a software developer in the execution of some model refactorings. We show how our rule-based system that is in the first place meant to resolve inconsistencies, can be used by support for the application of some model refactorings.

- While the previous contributions are scientific contributions, as a more practical contribution we developed a prototype tool capturing our approach. The prototype tool is integrated into *Poseidon* [gen05], a state-of-the-art UML CASE tool.

These contributions are partially presented in different research papers. The general and preliminary ideas of using DLs for inconsistency management are published in [Van02b]. Our classification of inconsistencies is presented in [VMSJ03] and [SVJM04]. These publications also elaborate the idea of formalising a fragment of the UML metamodel and using a DL system for querying user-defined models. This approach has also been presented in an evolution context as a chapter in the book "Software Evolution with UML and XML" [MVS05]. In [VSM03], we showed the need for a powerful query mechanism for state-of-the-art DL reasoning systems. This was one of the motivations for the development of a query mechanism for RACER, a state-of-the-art DL system, by the RACER authors. This query language is introduced in [HMSW04]. We contributed to this publication by using the query language for the detection of inconsistencies. The initial ideas for using DLs as a semantic domain for sequence diagrams and protocol state machine diagrams are introduced in [Van04]. The elaboration of this idea is also presented in [VMJ06]. The correspondence between inconsistencies and behaviour preservation properties is introduced in [VJM04] and elaborated in [VMJ06].

1.8 Outline

Chapter 2: Lightweight Formalisation of UML Fragment The chapter starts by introducing the requirements of a case study. This case study will be used throughout the whole dissertation.

In this dissertation UML 2.0 is used. The high-level specifications of this UML version are introduced. We based ourselves upon the final adopted draft specifications of version 2.0. The reader familiar with the UML can skip this part. Next, we motivate why that particular fragment of the UML metamodel is chosen. Finally, the different selected parts of the UML metamodel are described and a formalisation of the abstract syntax and possible semantics of these parts is given in terms of a mathematical model. This model is kept clean in the sense that not the full expressiveness of the UML is considered. The elements described by the selected fragment of the UML metamodel are elements used by class diagrams, sequence, communication diagrams and protocol state machine diagrams. Each type of diagrams is described in a distinct section. The sections are organised as follows: first the UML metamodel part is described, and next, a formalisition of this abstract syntax and possible semantics is given. The formalisation introduced, enables the precise definition of inconsistencies in and between UML models.

Chapter 3: Conceptual Classification of Inconsistencies This chapter starts with an outline on how the different consistency dimensions presented in the introduction are related to the UML. A second step is to classify different inconsistencies along two dimensions. The classification is motivated based on the nature of the UML and on a literature study. The set of presented inconsistencies is not exhaustive, but we argue that this is a relevant set. New inconsistencies can be added to the classification easily.

Each inconsistency is described in terms of the involved UML metamodel elements. Each inconsistency is defined based on our formalisation of the UML fragment and exemplified by models from the previously introduced case study.

Chapter 4: Inconsistency Handling This chapter focuses on the activity of inconsistency handling in the inconsistency management process. First, we agree upon the used terminology. Our approach will focus on resolution actions. These actions modify models in order to resolve inconsistencies. An overview of the different challenges related to resolution actions is given and exemplified by examples of occurrences of our classified inconsistencies. We catalogue different possible resolution actions and relate this catalogue to the classified inconsistencies presented in the previous chapter.

Next, constructions rules and their possible applications are shown on some of our classified inconsistencies. Construction rules specify how a model can be modified preserving some consistencies. These rules are used in the process of consistency maintenance rather than in the process of inconsistency management.

Finally, the three key criteria introduced in the introduction are elaborated. These key criteria will be used to evaluate Description Logics as a formalism for inconsistency definition, detection and resolution.

Chapter 5: Introducing Description Logics Description Logics are introduced in this chapter based on "The Description Logic Handbook" [BCM$^+$03], on the Ph.D. thesis

of Carlos Areces [Are00] and on overview papers by Sattler *et al.* [Sat03, BHS05]. Readers familiar with DLs can skip this part of the chapter. The basic building blocks of these logics and their reasoning abilities are explained. The different expressive means leading to different kinds of DLs are also described. These expressive means will play a prominent role in the translation of UML model elements to a certain DL.

In the next part of this chapter, DL systems are analysed. The analysed systems are CLASSIC, LOOM, FACT and RACER. The reader familiar with DL systems can skip this part. Based on this analysis, we motivate the selection of the RACER system. This system will be used in the next chapters and by our tool support.

Chapter 6: Encoding of UML Model Elements The goal of this chapter is to evaluate the first criterion, *abstract syntax and semantics representation* of the UML in DL. First, the translation of the UML metamodel fragment into a DL is defined and discussed. Next, different interpretations of UML class diagrams, UML sequence and communication diagrams and state machine diagrams are briefly discussed. Translations in DLs for class diagrams, protocol state machines, sequence and communication diagrams are considered consecutively. We also compare DLs and the Object Constraint Language (OCL) and show which OCL constraints can be expressed by DLs. Finally, we present a DL representation framework representing UML models which contain different kinds of UML diagrams and represent different interpretations of these diagrams.

Chapter 7: A DL Inconsistency Detection Approach The goal of this chapter is to evaluate the second criterion, *enable definitions of inconsistencies and inconsistency detection.* The definitions of the inconsistencies are based on the encoding of the UML elements presented in the previous chapter. Our inconsistency classification introduced in Chapter 3 is revisited from the viewpoint of our DL representation framework. Depending on the representation of the models in DL and the inconsistencies to be checked, different detection approaches are defined. A first approach is to detect inconsistencies by sophisticated queries on DL knowledge bases that represent the UML metamodel and the user-defined models as instances of this metamodel. Another approach is to use the standard DL reasoning tasks on the DL knowledge bases representing an interpretation of state machine diagrams, sequence diagrams, class diagrams or a combination of those diagrams. We also discuss the formal properties guaranteed by both approaches. This chapter is concluded by a discussion on the advantages and limitations of the usage of DLs as an inconsistency definition and detection approach.

Chapter 8 A Rule-based DL Inconsistency Resolution Approach The goal of this chapter is to evaluate the third criterion, *precise definitions and management of interactive inconsistency resolutions.* First, we define resolution actions in DLs. The identified problems and challenges in the context of inconsistency resolution are exactly those that are addressed by rule-based systems. We identify two types of rules in our DL representation framework. A set of requirements for a DL rule-based system can be distilled. The current support for rules by DL systems is compared to these requirements. Finally, the proposed resolution approach is evaluated against the third key criterion.

Chapter 9: Supporting Model Refactorings This chapter can be divided into two parts. In a first part, behaviour preservation properties of model refactorings are discussed. It is shown that some of our defined consistencies correspond to behaviour preservation properties in a refactoring context. Using this knowledge, properties about the behaviour of refactored super- or subclasses can be proved.

In a second part, we argue that our rule-based inconsistency approach can be used for supporting some model refactorings. In order to execute some refactorings, a chain of user input and inconsistency detection and resolution steps is performed. We argue by an example of a specific model refactoring, that manually determining inconsistency resolution scenarios that correspond to all possible situations is an unmanageable and error-prone task. The identified problems are exactly those that are addressed by our rule-based inconsistency resolution approach. Hence we use the inconsistency resolution rules defined in Chapter 8, to support model refactorings. We present an overview of the relation between a larger set of model refactorings and our classified inconsistencies.

Chapter 10: Proof-of-concept Tool Support In this chapter proof-of-concept tool support is presented implementing different inconsistency checks and resolutions. The presented prototype *RACOoN* is developed as a plug-in of the state-of-the-art UML CASE tool Poseidon and uses the state-of-the-art DL reasoning engine RACER. First, we show how the detection of our classified inconsistencies can be implemented through the implementation of our DL representation framework. Next, some resolution actions are implemented and used in rules which can be used not only in an inconsistency resolution approach but also in the context of supporting model refactorings. The execution of one model refactoring using a set of representative resolution rules is demonstrated. This chapter provides small yet representative implementations of our approach.

Chapter 11: Conclusions This chapter concludes this dissertation. An overview of our ideas is given and future work is discussed.

Chapter 2

Lightweight Formalisation of UML 2.0 Fragment

This chapter starts with the introduction of the requirements of a case study (Section 2.1). This case study models an *ATM* application and is used as running example throughout this dissertation.

Next, the definition of the UML [Obj04a] is discussed (Section 2.2). The language architecture of the UML is based on a metamodelling approach. Recently, the UML version 2.0 has been adopted. The specification of this language consists of several documents each specifying different parts of the language definition. The UML 2.0 notation consists of different standard diagram types which can be used throughout the entire software development cycle. In this dissertation, we will deliberately confine ourselves to three types of diagrams. We motivate this research restriction and the choice of the diagrams (Section 2.3). We introduce the three types of diagrams. First, *class diagrams* expressing the static structure of the software are introduced (Section 2.4). Next, *sequence diagrams*, emphasising on the sequences of message sends and *communication diagrams*, emphasising on the links between the different objects, are introduced (Section 2.5). Finally, *state machine diagrams* (Section 2.6) that specify the different states of a particular object and the possible state transitions, are introduced. The model elements occurring in these 3 different types of diagrams, and their relationships are formalised in the respective sections. Based on these definitions, a definition of the concept *UML model* is given (Section 2.7).

2.1 Case Study Requirements

In this section, the case study used as a running example throughout the next chapters, is introduced. Our case study is a design of an automated teller machine (ATM) simulation and is an adapted version of the ATM simulation example by Russell C. Bjork [Bjo04]. We now specify the requirements for the ATM example.

The simulated ATM possesses a magnetic card reader for reading an ATM card, a customer console (keyboard and display) for interaction with the customer, a slot for depositing envelopes, a dispenser for cash, a printer for printing customer receipts, and a key-operated switch to allow an operator to start or stop the machine.

This key-operated switch allows an operator to start and stop the servicing of customers.

After turning the switch to the "on" position, the operator will be required to verify and enter the total cash on hand. The machine can only be turned off when it is not servicing a customer. When the switch is moved to the "off" position, the machine will shut down, so that the operator may remove deposited envelopes and reload the machine with cash, blank receipts, etc.

The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN). Both of which will be sent to the bank for validation as part of each transaction. The customer will then be able to perform one or more transactions.

The ATM must be able to provide the following services to the customer:

- A customer must be able to make a cash withdrawal from any suitable account linked to the card. Approval must be obtained from the bank before cash is dispensed.

- A customer must be able to make a deposit to any account linked to the card, consisting of cash and/or checks in an envelope. The customer will enter the amount of the deposit into the ATM, subject to manual verification when the envelope is removed from the machine by an operator. Approval must be obtained from the bank before physically accepting the envelope.

- A customer must be able to make a transfer of money between any two accounts linked to the card.

- A customer must be able to make a balance inquiry of any account linked to the card.

- A customer must be able to abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine.

The ATM will communicate each transaction to the bank and obtain verification that it is allowed by the bank. A transaction will be considered complete by the bank once it has been approved. In the case of a deposit, a second message will be sent to the bank indicating that the customer has deposited the envelope. If the customer fails to deposit the envelope by pressing cancel instead, no second message will be sent to the bank and the deposit will not be credited to the customer.

The ATM will provide the customer with a printed receipt for each successful transaction, showing the date, time, machine location, type of transaction, account(s), amount, and ending and available balance(s) of the affected account ("to" account for transfers).

Possible extensions to this example are:

- If the bank determines that the customer's PIN is invalid, the customer will be required to re-enter the PIN before a transaction can proceed. If the customer is unable to successfully enter the PIN after three tries, the card will be permanently retained by the machine.

- If a transaction fails for any reason other than an invalid PIN, the ATM will display an explanation of the problem, and will then ask the customer whether he/she wants to do another transaction.

- The ATM will also maintain an internal log of transactions to facilitate resolving ambiguities arising from a hardware failure in the middle of a transaction. Entries will be made in the log when the ATM is started up and shut down, for each message sent to the Bank (along with the response back, if one is expected), for the dispensing of cash, and for the receiving of an envelope. Log entries may contain card numbers and amounts, but for security will never contain a PIN.

2.2 UML 2.0 Specification

UML originates from the unification of different object-oriented graphical modelling languages flourishing in the early 1990s. In 1997, UML version 1.1 was adopted by the Object Management Group (OMG) as an official OMG standard. Due to these standardisation efforts UML became the state-of-the-art modelling language it is now. As a consequence, UML is well-known and there is a lot of tool support available.

2.2.1 A Metamodelling Approach

The UML specification is defined using a metamodelling approach. A *metamodel* specifies what can be expressed in the valid models of a certain modelling language.

A language definition normally consists of an abstract syntax, a concrete syntax and semantics. The UML metamodel includes all the concrete graphical notation, abstract syntax and, however informal, the semantics for UML. The UML abstract syntax consists of UML class diagrams. The concrete syntax is informally specified UML notation. Well-formedness rules are constraints on the abstract syntax and they specify when an instance of a particular language construct is meaningful. These rules are described partially in the *Object Constraint Language* (OCL) [Obj04d] and in English. The semantics of UML defines a model's meaning and is described in natural language (English) and OCL.

The UML metamodel has been architected following a four-layer architectural pattern (see Figure 2.1).

2.2.2 The Four-Layer Metamodel Hierarchy

The *meta-metamodelling layer* forms the foundation of the metamodelling hierarchy. This layer, often referred to as M3, is a specification of the language used to express metamodels. OMG has defined metamodels for languages other than UML. The OMG Meta-Object Facility (MOF) is an example of a meta-metamodel. It defines an abstract language and framework for specifying and constructing metamodels. There is no need to have additional meta-layers above MOF because MOF is reflective. There are some advantages using this meta-metamodelling approach. It can be used to compare different modelling languages or to shift from one modelling language to another as long as they are expressed in the same meta-metamodel. It also makes adding new features to a modelling language easy.

A *metamodel* is an instance of a meta-metamodel, which means that every element of the metamodel is an instance of an element in the meta-metamodel. The responsibility of this layer, often referred to as M2, is to define a language for specifying models. UML and the Common Warehouse Metamodel (CWM), which is a specification describing metadata interchange among data warehousing, business intelligence and knowledge management, are

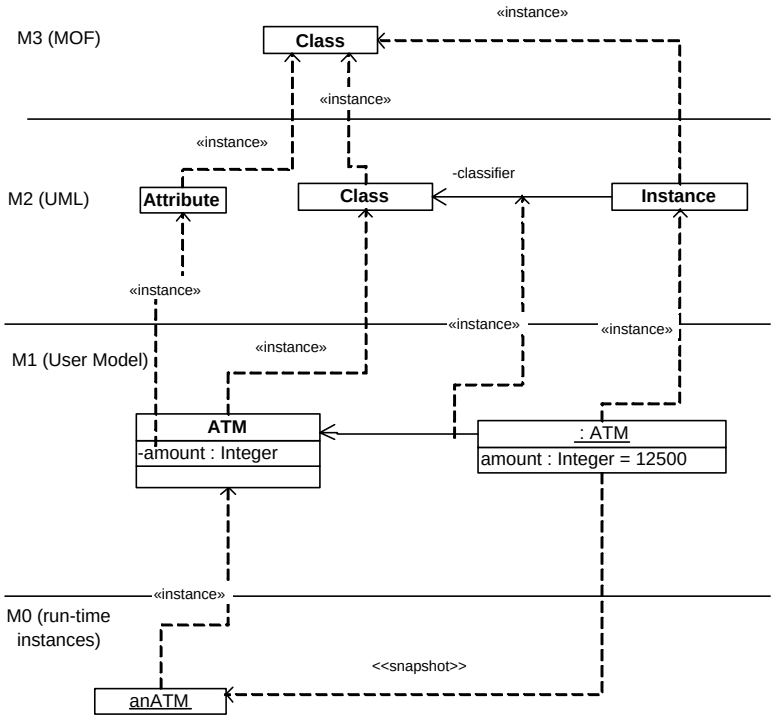

Figure 2.1: Example of the four-layer metamodel hierarchy.

examples of metamodels which are instances of a meta-metamodel. Every UML metamodel element is an instance of exactly one model element in MOF. As shown in Figure 2.1, the concepts *Class*, *Attribute* and *Instance* in the UML metamodel are instances of the concept *Class* in MOF.

The *model layer*, referred to as M1, has the primary responsibility to define languages that describe certain domains. A user model is an instance of the UML metamodel. User models contain model elements and sets of instances of these model elements. For example in Figure 2.1 the class *ATM* is an instance of the *Class* concept in the UML metamodel. The object *:ATM* is contained in M1 and is an instance of the *Instance* concept in the UML metamodel.

The lowest meta-layer is called M0 and contains the run-time instances of model elements defining a specific domain. Objects defined in the user models are snapshots of the run-time instances contained in M0.

2.2.3 Specifications

The UML 2.0 specification consists of several documents:

- UML 2.0 Infrastructure [Obj04c]
- UML 2.0 Superstructure [Obj04e]
- UML 2.0 Object Constraint Language (OCL) [Obj04d]

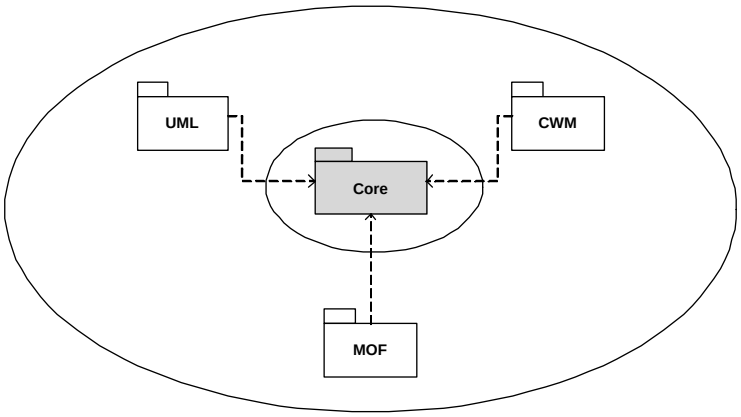

Figure 2.2: The role of Core in the context of other metamodels.

- UML 2.0 Diagram Interchange [Obj04b]

UML 2.0 Object Constraint Language contains the definition of OCL version 2.0 which is aligned with UML 2.0. OCL is used to express well-formedness rules and parts of the semantics. However, OCL has some limitations. OCL is a constraint language which is also used to navigate over UML class diagrams. Three types of constraints can be expressed in OCL, i.e., invariants, pre- and postconditions. An OCL constraint must always be specified in a certain context. For pre- and postconditions this context is a certain operation defined in a class diagram. For invariants, the context is a *Classifier*. This boils down to classes, associations, association classes, interactions and state machines. However, only navigation over static structure elements such as associations is permitted. As a result, OCL almost only addresses static UML diagrams. It is also not possible to formalise UML diagrams using OCL and to reason about them in the same way as a logic formalism can. And last but not least, the tool support available for OCL emphasises on support for writing OCL expressions and generating code for these expressions [Uni04a, HDF00]. These limitations point out that we need something more powerful in the realm of expressing constraints on behavioural UML diagram elements and in the realm of reasoning on UML diagrams.

The goal of the *UML 2.0 Diagram Interchange* specification is *to enable a smooth and seamless exchange of documents compliant to the UML standard between different software tools.* [Obj04b]. This document provides a metamodel (MOF-compliant) for UML diagram information (including notational information).

The *UML 2.0 Infrastructure* defines the foundational language constructs required for UML 2.0. It is complemented by *UML 2.0 Superstructure* defining the user level constructs required for UML 2.0. These two specifications constitute a complete specification for the UML version 2.0 language.

The Infrastructure of UML 2.0 is defined by the *InfrastructureLibrary* package. This package consists of the packages *Core* and *Profiles*. *Core* contains core concepts used when metamodelling, while *Profiles* defines mechanisms used to customise metamodels.

The *Core* package is a complete metamodel. Other metamodels at the same metalevel import or specialise the specified metaclasses. Figure 2.2 shows how UML, CWM, and MOF each depend on a common *Core*.

One of the primary uses of the UML 2.0 Infrastructure specification is that *it should be reused when creating other metamodels* [Obj04c]. The UML metamodel reuses the *InfrastructureLibrary* in two different ways:

1. The UML metamodel is instantiated from meta-metaclasses that are defined in the *InfrastructureLibrary*.

2. The UML metamodel imports and specialises all metaclasses in the *InfrastructureLibrary*.

The *InfrastructureLibrary* is used both at the M2 and M3 metalevels, since it is being reused by UML and MOF.

The UML Superstructure metamodel is structured into three parts: the first part, *Structure*, defines the static, structural constructs used in structural diagrams, such as class diagrams, component diagrams and deployment diagrams. The InfrastructureLibrary is primarily reused in this part that adds more capabilities to the modelling constructs which are not necessary to include for purposes of reuse or alignment with MOF. The second part, *Behaviour*, specifies the dynamic constructs used in behavioural diagrams, such as activity diagrams, sequence diagrams and state machine diagrams. The third part, *Supplement*, defines auxiliary constructs and profiles to customise UML for various domains, platforms and methods.

2.2.4 UML 2.0 Diagrams

The UML 2.0 specification describes 13 diagram types. Those diagrams can be classified into structure and behaviour diagrams. Table 2.1 based on [Fow03], summarises the purpose of the different diagram types.

Diagram	**Purpose**
Structure	
Class	Types of objects and static relationships among them
Component	Structure and connections of components
Composite Structure	Run-time decomposition of a class
Deployment	Deployment of artefacts to nodes
Package	Group elements together into higher-level constructs
Behaviour	
Interaction	
Communication	Possible interaction between objects; emphasis on links
Sequence	Possible interaction between objects; emphasis on sequence
Timing	Possible interaction between objects; emphasis on timing
Interaction overview	Mix of sequence and activity diagram
Activity	Procedural logic, business process and work flow
Object	Possible configurations of instances
State machine	States of an object and state changes over its life
Use case	How users interact with a system

Table 2.1: UML 2.0 diagram types.

The types are not rigid in the sense that it is legal to use elements from one diagram type on another diagram.

A *class diagram* describes the types of objects and the various kinds of static relations existing among them. *Component diagrams* on the contrary, specify the static structure and connections between components and are subject to a small range of modelling elements. *Composite structure diagrams* show how complex classes can be decomposed, much of its notation is used by component diagrams. *Deployment diagrams* show a system's physical layout, displaying which (pieces of) software run on what (pieces of) hardware. These diagrams only contain two modelling elements: a node, which can be a device or execution environment and an artefact. Finally, *package diagrams* describe packages, which contain classes and packages, and some possible relations between them.

State machines are a quite well-known technique to describe the behaviour of a system and more in particular, a state machine defines the possible states a certain object can possess and the different state transitions existing between states. In contrast to state machines, *activity diagrams* are used to describe work flow or procedural logic. In both cases multiple objects can be involved. Next to the description of the behaviour of a single object, we also need to be able to model possible interactions between various objects and to indicate how these objects are linked at run-time. For this purpose, *interaction diagrams* are used in UML. These diagrams come in different flavours. An interaction implies two major aspects: first of all the *sequence* in which messages are sent between different objects and secondly, the *communication paths* between the different interacting objects. Both aspects are described by interaction diagrams, however, *sequence diagrams* emphasise on the sequence aspect, while *communication diagrams* emphasise the communication aspect. Other interaction diagrams are *timing diagrams* which show timing constraints between message sends or state changes on different objects and *interaction overview* diagrams which combine sequence diagrams and activity diagrams. A last type of behaviour diagrams is a *use case diagram*. Use cases are a technique for capturing functional requirements of a system and as such are used in the requirements phase of the development cycle. Use cases are also well known, however, nothing in UML describes how the content of a use case should be captured.

2.3 Scope of UML 2.0 Fragment

We will only consider one kind of structure diagram, class diagrams, in our work for the following reasons.

- It is the most important structure diagram in the sense that it describes the types of objects and the various kinds of static relations existing among them. These concepts are fundamental to object-oriented development.
- A class diagram is subject to the greatest range of modelling elements.
- Elements defined by class diagrams are used in the behavioural diagrams. As a consequence there is an overlap between class diagrams and behavioural diagrams.
- Class diagrams are also used for the description of the abstract syntax of UML.

Section 2.4 provides a detailed description of the different modelling elements occurring in class diagrams. This description is quite detailed on purpose, as the UML abstract syntax is described using class diagrams, a good understanding of class diagrams is necessary to understand UML.

As behaviour diagrams, we will only consider sequence, communication and state machine diagrams in our work for the following reasons.

- The research reported upon in this dissertation is in line with earlier research efforts conducted at the System and Software Engineering Lab.

  In the dissertation of Kurt Verschaeve [Ver01], bridges are investigated between object-oriented analysis and design formalisms (OMT/UML style) and formal specification languages (SDL-92 style). This line of research started with the Esprit III project INSYDE (Integrated Methods for Evolving System Design) and is continued in the context of the AIA (Advanced Internet Access) project where an environment for design, implementation and documentation of components for multimedia services is investigated. Iteration between system design in the UML, especially UML state machines, and detailed design in SDL-96 is an important generic result of this line of research and this iteration guarantees round-trip engineering.

  In Bart Wydaeghe's dissertation [Wyd01], concerning component-based development, the tool suite PaCoSuite is presented. This tool suite supports the visual composition of components. Message Sequence Charts (MSC's) on which UML sequence diagrams are partially based, are used to describe the interaction between a set of roles. As such the wiring of components is lifted to a full protocol and automatic compatibility checks and glue-code generation can be performed.

- Sequence diagrams and state machine diagrams are complementary in the sense that the former diagrams describe objects' interactions, while the latter ones describe the behaviour of a single object. Both diagrams types are subject to a great range of modelling elements. There is a significant overlap between these behaviour diagrams and class diagrams.

- Communication diagrams and sequence diagrams are equivalent. They only focus on different aspects of object interactions.

- UML state machine diagrams and sequence, communication diagrams are most described in literature (e.g., [ET00], [EHHS02], [vdB01], [LL99], [LCM+03] and many others.).

- Activity diagrams also contain a large amount of modelling elements. However, due to time restrictions it is not possible to investigate this type of diagram.

- Other behaviour diagrams are very specific, e.g., timing diagrams. Use case diagrams are well-known too but are very loosely described in UML and are realised by sequence diagrams which we will describe.

Sequence and communication diagrams are considered in Section 2.5, while state machine diagrams are considered in Section 2.6.

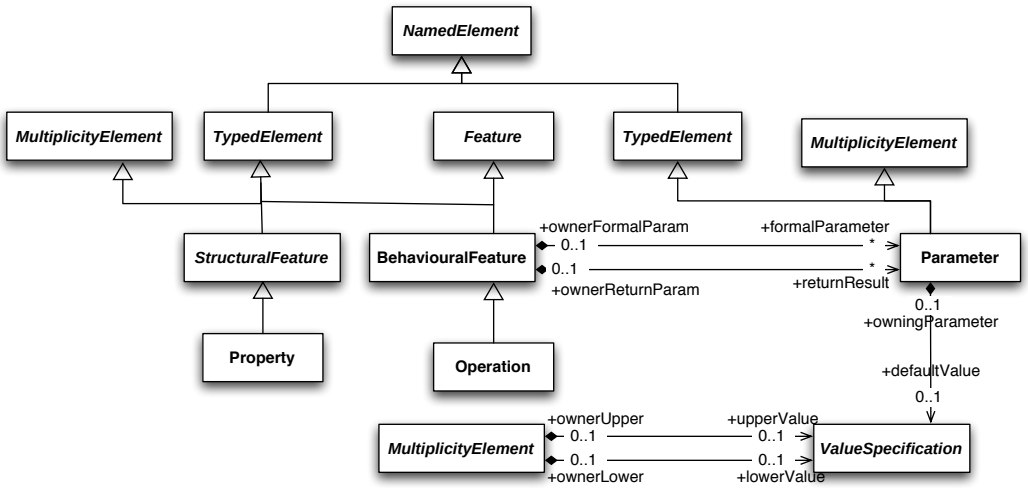

Figure 2.3: Metamodel snapshot concerning *Feature*.

For the different diagrams studied in this dissertation, structured definitions are given for the basic concepts constituting those diagrams, in view of defining several (in)consistencies. We will not and cannot use the full scale of model elements that a team of software engineers would use on a large real-world project in order to keep our formalisation clean enough for the purpose of inconsistency checking and management. The full UML standard is crowded with a lot of notations on which we will have to compromise. Rigorous definitions of UML model elements and UML elements are necessary to provide precise definitions of inconsistencies between and within UML (sub)models. Our ideas and results can then be extended to more detailed language concepts, closer to real-world modelling, and can be generalised to concepts of UML diagrams which are not detailed in this dissertation.

2.4 UML 2.0 Class Diagram

The first type of UML diagram described here, is *class diagrams*. A class diagram specifies the static structure of the software system under study. This static structure is described in terms of the different *classes* and different kinds of static relationships between them. Class diagrams show the *properties* and *operations* of a class. UML 2.0 uses the term *feature* to indicate properties and operations of a class.

Classes A class represents a set of objects with common *features*. As specified in Figure 2.3, that represents a portion of the UML 2.0 metamodel, a *Feature* can be a *StructuralFeature* or a *BehaviouralFeature*. An example of a *StructuralFeature* is a *Property* and an example of *BehaviouralFeature* is an *Operation*. Properties in UML 2.0 have two distinct notations: *attributes* and *association ends*. If a property is owned by a class, it represents an attribute. If a property is owned by an *Association*, it represents an end of the association.

Graphically, a class is represented by a rectangle divided into three parts. The first part contains the name of the class, this name must be unique within the whole class

diagram. The second part contains the attributes of the class and the third part contains the operations of the class. An example of a class diagram is shown in Figure 2.4.

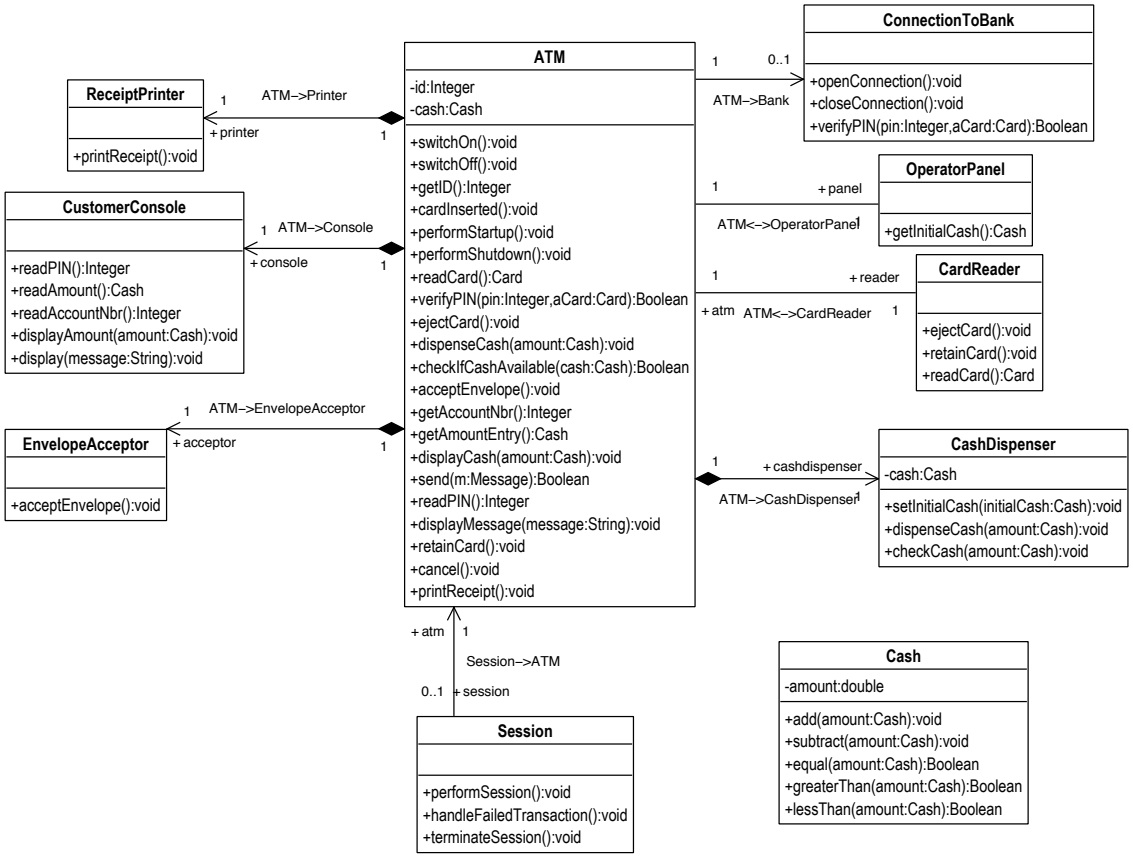

Figure 2.4: A first class diagram from our case study.

In the next paragraphs, the notions of attributes, operations and associations are explained.

Attributes Attributes are represented in the UML 2.0 metamodel by the metaclass *Property*. As shown in Figure 2.5, a *Property* is a subclass of *StructuralFeature* which is a *TypedElement* (see Figure 2.3). A *StructuralFeature* is also a *MultiplicityElement*. A *MultiplicityElement* has an *upperValue* and a *lowervalue* indicating the maximum and minimum cardinality, respectively, for an instantiation of this *MultiplicityElement*. This means that an attribute is denoted by a name, possibly followed by a multiplicity and the associated type of the attribute. The full UML syntax for an attribute is:

```
visibility name: type multiplicity = default
```

- `visibility` indicates whether an attribute is public or private.
- `name` denotes the name of the attribute.

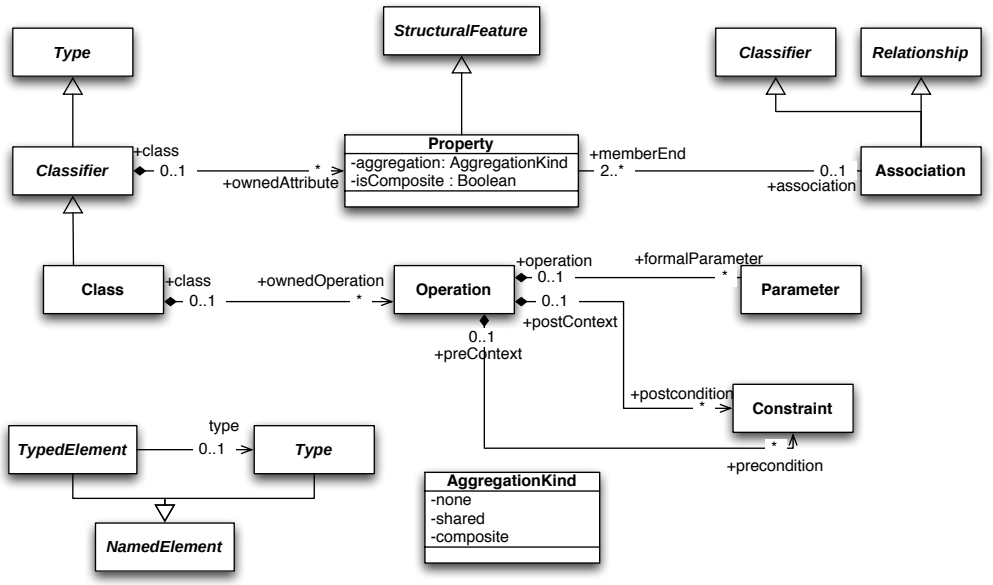

Figure 2.5: Metamodel snapshot concerning *Class*, *Property* and *Operation*.

- The `type` of the attribute specifies which kinds of values or objects an attribute can contain.
- The `multiplicity` of an attribute indicates how many objects may fill the attribute.
- The `default` value is the value for a newly created object.

Only the `name` is mandatory.

Concerning visibility, UML 2.0 provides only two visibility markers: private and public. The intention is that users will define their own visibility markers, from which visibility semantics can be constructed.

Example 1 *The class* ATM *in Figure 2.4, has two attributes:* -id : Integer *and* -cash : Cash. *The first attribute's name is* id *and its type is* Integer. *The second attribute's name is* cash *and its type is* Cash.

Attributes in the *Core* of the *InfrastructureLibrary* and in the UML metamodel, are of a type defined in the metamodel or of a *primitive type*. In *Core* and the UML metamodel, *PrimitiveTypes* are predefined and available to the *Core* and UML extensions. It is intended that every metamodel based on *Core* will reuse those primitive types. These may also be reused in user models, however, CASE tools usually provide their own library of data types to be used when modeling with UML. The available primitive types are *Boolean*, *Integer*, *String* and *UnlimitedNatural*. To be able to define operations in the next paragraph, which do not have a return result, we include *Void* to the set of primitive types.

First, a set of notations for different elements is introduced. Then, these elements are successively defined.

Notation 1 *The set of all available classes is denoted by* $\mathbf{C}$
The set of primitive types is denoted by $\mathbf{PT}$.
The set of types is denoted by $\mathbf{T}$ *and* $\mathbf{T} = \mathbf{C} \cup \mathbf{PT}$.
For naming UML elements, we assume a non-empty set of names $\mathbf{N}$.
The power set of a set $\mathbf{S}$ *is denoted by* $\mathcal{P}(\mathbf{S})$.
The set of all attributes is denoted by $\mathbf{Att}$.
The set of all operations is denoted by $\mathbf{Op}$.
The set of all associations is denoted by $\mathbf{Assoc}$.
The set of all association ends is denoted by $\mathbf{AssocEnd}$.
The set of all properties is denoted by $\mathbf{Prop}$ *and* $\mathbf{Prop} = \mathbf{Att} \cup \mathbf{AssocEnd}$.

Attributes can now be defined as follows:

Definition 4 *An* **attribute att** *is defined as a tuple,* $\mathbf{att} = (n, c)$, *where* $n \in \mathbf{N}$ *and* $c \in \mathbf{C}$.
The **set of all attributes of a set of classes** $R \subseteq \mathbf{C}$ *is defined as* $\mathbf{Att}_R \in \mathcal{P}(\mathbf{N} \times R)$.

The type and multiplicity of an attribute **att** are defined implicitly via the following (total) functions:

Definition 5 *The total function* $\mathbf{type}$*:* $\mathbf{Att} \rightarrow \mathbf{T}$.
The total function $\mathbf{multiplicity}$*:* $\mathbf{Att} \rightarrow \mathcal{P}(\mathbb{N}^+) \setminus \{0\}$.

Because attributes are defined as elements of the extension of the relation $\mathbf{N} \times \mathbf{C}$, each attribute has a unique name in the context of the class owning the attribute.

Example 2 *The attributes of the class* ATM *as modelled in Figure 2.4 can now be specified as:* $\{\mathbf{att}_{id} = (\text{“}id\text{”}, ATM), \mathbf{att}_{cash} = (\text{“}cash\text{”}, ATM)\}$ *and* $\mathbf{type}(\mathbf{att}_{id}) = Integer$, $\mathbf{type}(\mathbf{att}_{cash}) = Cash$ *and* $\mathbf{multiplicity}(\mathbf{att}_{id}) = \{1\}$, $\mathbf{multiplicity}(\mathbf{att}_{cash}) = \{1\}$.

Operations The *Operation* metaclass (see Figure 2.3) is a subclass of *BehaviouralFeature* and possesses some formal *Parameters* and some return results which are also *Parameters*.

The full UML syntax for operations is:

```
visibility name (parameter-list) : return-type-expression
```

- The `visibility` marker is the same as for an attribute.
- The `name` denotes the name of the operation and is a string.
- The `parameter-list` is the list of parameters of the operation. A parameter is a *TypedElement*.
- The `return-type-expression` is a comma-separated list of return types. UML allows for multiple return types.

Remark, in class diagrams, only the signature of operations is defined. This includes the formal parameters, the return types and the type owning the operation.

Example 3 *As an example, consider the operations defined for the class* ATM *in Figure 2.4. One of these operations is* checkIfCashAvailable(cash : Cash) : Boolean. *The name of the operation is “*checkIfCashAvailable*” and the operation has one formal parameter of type* Cash. *The return type of the operation is the primitive type* Boolean.

Definition 6 *An* **operation op** *is defined as a tuple,* $\mathbf{op} = (n, c, (p_1, \ldots, p_k), (r_1, \ldots, r_m))$*, where* $n \in \mathbf{N}$*,* $c \in \mathbf{C}$ *and* $p_1, \ldots, p_k, r_1, \ldots, r_m \in \mathbf{T}$*.* $p_1, \ldots, p_k$ *denote the types of the formal parameters of the operation and* $r_1, \ldots, r_m$ *denote the types of the return values.*
The **set of all operations of set of classes** $R \in \mathbf{C}$ *is defined as* $\mathbf{Op}_R = \{\mathbf{op} | \mathbf{owner}(\mathbf{op}) = c \in R\}$*.*
The total function $\mathbf{owner} : \mathbf{Op} \rightarrow \mathbf{C}$*:* $(n, c, (p_1, \ldots, p_k), (r_1, \ldots, r_m)) \mapsto c$*.*

Overloading of operations is allowed in UML. This happens when operations have the same name, but a different signature, where the signature of an operation consists of the types of the formal parameters and the types of the return values. *Overriding* of operations normally takes place when two operations, one defined in a superclass and another in a subclass, have the same name and signature but behave in different ways. However, it is not possible to specify in a UML class diagram the fact that both operations behave in different ways. In UML 2.0 it is possible to *redefine* some model elements in the context of a *generalisation*. An operation may be redefined in a specialisation of its defining classifier. This redefinition may specialise the types of the formal parameters or return results, add new preconditions or postconditions. The above formalisation of operations allows one to have operations with the same name and a different signature, or with the same name and the same signature defined in two different classes, and it also allows the redefinition of operations.

Example 4 *We can now formally define the operation* $\mathbf{op}_{checkIfCashAvailaible}$ *as modelled in Figure 2.4.* $\mathbf{op}_{checkIfCashAvailable} : (\text{“}checkIfCashAvailable\text{”}, \text{ATM}, (\text{Cash}), (\text{Boolean}))$*.*

As shown in Figure 2.5, constraints can be specified on an operation. *Preconditions* are *an optional set of constraints that must be true when an operation is invoked* [Obj04e]. *Postconditions* are *an optional set of constraints that must be true when the operation is completed* [Obj04e].

Notation 2 *The* **set of all preconditions of an operation op***, i.e., the set of conditions that must be true when the operation* **op** *is invoked, is denoted by* $\mathbf{Pre_{op}}$*. The* **set of all preconditions of all operations** *is denoted by* $\mathbf{Pre} = \bigcup_{\forall \mathbf{op} \in \mathbf{Op}} \mathbf{Pre_{op}}$*.*
The **set of all postconditions of an operation op***, i.e., the set of conditions that must be true when the operation* **op** *is completed, is denoted by* $\mathbf{Post_{op}}$*. The* **set of all postconditions of all operations***, is denoted by* $\mathbf{Post} = \bigcup_{\forall \mathbf{op} \in \mathbf{Op}} \mathbf{Post_{op}}$*.*

Just as UML, we do not prescribe a constraint language used for the pre- and postconditions of operations or any other kinds of constraints that will be defined further in this dissertation. But we assume that these constraints (such as $\mathbf{Pre_{op}}$ and $\mathbf{Post_{op}}$) are sets of predicates, i.e., Boolean expressions.

On a class diagram it is possible to indicate whether an operation is *abstract* or not. An *abstract operation* has no implementation and is pure declaration. An implementation of an operation can be specified in UML by, e.g., a sequence diagram or state diagram(s). The most common way to indicate an abstract class or operation in UML is to italicise the name or to use the label {`abstract`}. An *abstract class* is a class that cannot be instantiated directly, because it has some abstract operations. In the next definition, we define the relation **isAbstract**.

Definition 7 *The unary relation* $\mathbf{isAbstract} : \mathbf{C} \cup \mathbf{Op_C}$*.*

Associations An *association* is a relation between instances of two or more classes. As specified by the UML metamodel (see Figure 2.5), each association has at least two *association ends*. Those association ends are *Properties* and as such can be explicitly named.

An association end also has *multiplicity* indicating how many objects participate in the given association. In general, the multiplicity indicates lower and upper bounds for the participating objects. The lower bound may be any positive number or zero; the upper bound may be any positive number or $*$ for unlimited.

Conceptually there is no difference between an attribute and an association end. An attribute has another kind of notation than an association end. Attributes are usually single-valued. The difference occurs at the implementation level. Attributes imply navigability from the class to the attribute only.

Navigability arrows can be specified on an association end as, for example, on the association end between *Type* and *TypedElement* in Figure 2.5. This indicates that a *TypedElement* is responsible for knowing its *Type*, but a *Type* does not know its associated *TypedElements*. If a navigability exists in only one direction, the association is called *unidirectional*. A *bidirectional* association contains navigability in both directions.

Definition 8 *An* **n-ary association assoc** *is defined as an (n + 1)-ary tuple* $\mathbf{assoc} = (name, c_1, c_2, \ldots, c_n)$*, where* $c_i \in \mathbf{C}, name \in \mathbf{N} \cup \{\omega\}$.

The classes $c_1, \ldots, c_n$ are called the *participating classes of the association*. In the UML, the name of an association is optional. To support this, we introduce ω denoting a dummy name. Remark, that it is not possible to have two associations with the same set of participating classes and the same names, or all dummy names.

An n-ary association is in our model, a tuple belonging to the extension of a relation $R_n : \mathbf{N} \times \underbrace{\mathbf{C} \times \ldots \times \mathbf{C}}_{n}$. Each set of associations of a certain arity defines a certain relation. For each relation R_n, selectors that are total functions and that, given a tuple representing an association, return a certain element of that tuple, can be defined and are used in the definition of association ends.

Definition 9 $\forall i \in \{1, \ldots, n\}$: *the total functions*
$\mathbf{assocType}_{n,i} : \{\mathbf{assoc} | \mathbf{assoc} \in \mathbf{Assoc} \wedge \mathbf{assoc} = (name, c_1, \ldots, c_n)\} \rightarrow \mathbf{C} : \mathbf{assoc} = (name, c_1, \ldots, c_n) \mapsto c_i$
$\mathbf{Selectors}_n = \bigcup_{\forall i \in \{1,\ldots,n\}} \mathbf{assocType}_{n,i}$

$\mathbf{Selectors}_n$ denotes the set of selectors of associations of arity n.

To be able to retrieve the set of associations in which a class participates, we define a (partial) function **participates**.

Definition 10 *The partial function* **participates***:* $\mathbf{C} \rightarrow \mathcal{P}(\mathbf{Assoc})$:
$c \mapsto \{\mathbf{assoc} | \exists\ i \in \{1, \ldots, n\} : \mathbf{assocType}_{n,i}(\mathbf{assoc}) = c\}$.

We can now proceed by defining association ends.

Definition 11 *An* **association end assocend** *is defined as a tuple,* **assocend** = $(\mathbf{assoc}, \mathbf{assocType}_{n,i})$ *where* $\mathbf{assoc} \in \mathbf{Assoc}$*, an n-ary association and* $\mathbf{assocType}_{n,i} \in \mathbf{Selectors}_n$*.*
The **set of association ends of a set of classes** $R \subseteq \mathbf{C}$ *is defined as* $\mathbf{AssocEnd}_R = \{\mathbf{assocend} = (\mathbf{assoc}, \mathbf{assocType}_{n,i}) | \exists j \in \{1, \ldots, n\} \setminus \{i\} : \mathbf{assocType}_{n,j}(\mathbf{assoc}) \in R\}$*.*

An association end consists of the association owning the end and the selector that applied to the association will return the type of the association end. By defining an association end as a tuple consisting of the owning association and a selector, ends of self-associations are uniquely identified. A self-association is a binary association where both ends of the association are attached to the same class.

Furthermore, we introduce the constraint that for each association there exists as many association ends as the arity of the association.

$$\forall \mathbf{assoc} = (name, c_1, \ldots, c_n) \in \mathbf{Assoc}\ \forall \mathbf{assocType}_{n,i} \in \mathbf{Selectors}_n$$
$$\exists \mathbf{assocend} = (\mathbf{assoc}, \mathbf{assocType}_{n,i}) \in \mathbf{AssocEnd}$$

An association end can have a name and has a multiplicity. The function **multiplicity** defined in Definition 4 can be extended to the domain of association ends, returning the multiplicity of an association end or an attribute. Furthermore, a unary relation is defined indicating whether an association end is navigable.

Definition 12 *The partial function* **name***:* $\mathbf{AssocEnd} \rightarrow \mathbf{N}$*.*
The total function $\mathbf{multiplicity} : \mathbf{Prop} \rightarrow \mathcal{P}(\mathbb{N}^+) \setminus \{0\}$*.*
The unary relation $\mathbf{isNavigable} : \mathbf{AssocEnd}$*.*

Example 5 *Consider the association named* ATM<->CashDispenser *in the class diagram of Figure 2.4. This is a binary association* **ATMtoCashDispenser** = (ATM<->CashDispenser*,* ATM*,* CashDispenser)*. This association has two association ends.*

A first association end can be defined as **atmend** = (**ATMtoCashDispenser**,$\mathbf{assocType}_{2,1}$) *and the second association end can be defined as* **cashdispenserend** = (**ATMtoCashDispenser***,* $\mathbf{assocType}_{2,2}$) *where* $\mathbf{assocType}_{2,2}$ (**ATMtoCashDispenser**) = CashDispenser*,* $\mathbf{assocType}_{2,1}$ (**ATMtoCashDispenser**) = ATM*,* **name** (**cashdispenserend**) = *"cashdispenser",* **multiplicity** (**cashdispenserend**) = $\{1\}$ *and* **isNavigable** (**cashdispenserend**) *and* **multiplicity** (**atmend**) = $\{1\}$*.*

Aggregation and composition Two special kinds of binary associations are *aggregations* and *compositions*. An aggregation denotes a part-of relationship. As specified in the UML Superstructure document, the semantics of aggregation varies by application and modeller. This makes it impossible to exactly specify in general the difference between aggregation and association.

A stronger form of aggregation is *composition* that requires a part instance to be included in at most one composite at a time. If a composite is deleted, all of its parts are deleted with it.

The unary relation **isComposite** expresses whether an association end is composite .

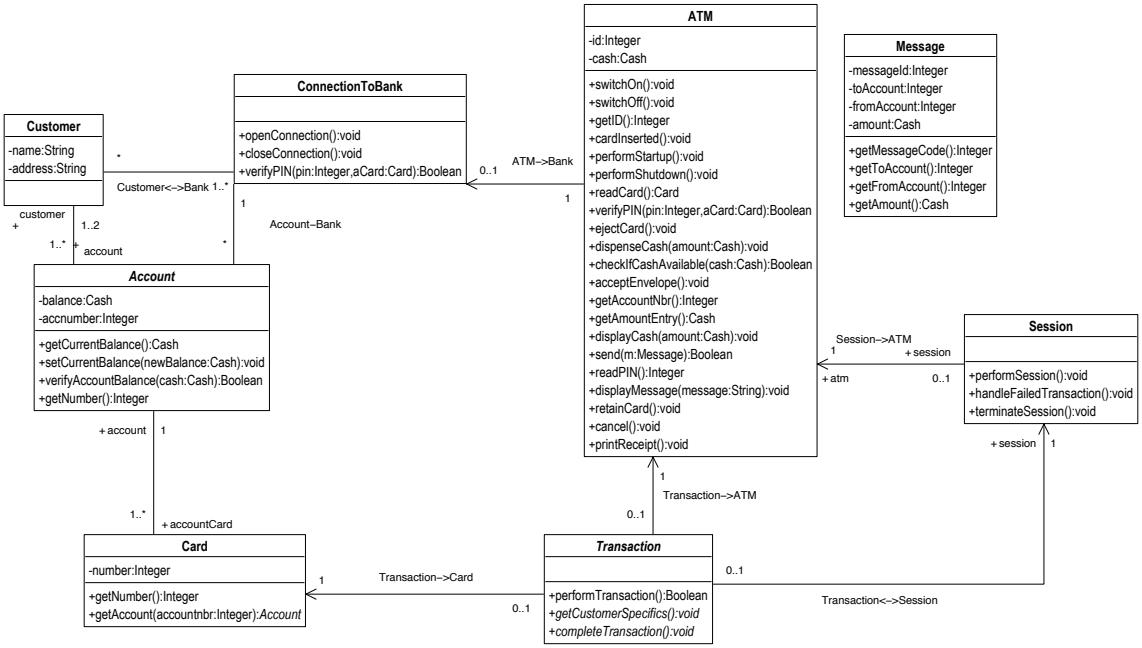

Figure 2.6: A second class diagram from our case study.

Definition 13 *The unary relation* **isComposite** : **AssocEnd**.
$(\mathbf{isComposite}((\mathbf{assoc}, \mathbf{assocType}_{n,i})) \Rightarrow n = 2)$.

Remark that a composite self-association is antisymmetric and transitive.

Example 6 *In Example 5, only* **atmend** *belongs to the relation* **isComposite**, *i.e.,* **isComposite**(**atmend**).

We are now able to define a class formally as:

Definition 14 *A* class $c = (\mathbf{Prop}_c, \mathbf{Op}_c)$ *consists of*

- *a set* $\mathbf{Prop}_c = \mathbf{Att}_{\{c\}} \cup \mathbf{AssocEnd}_{\{c\}}$ *of* properties,
- *a set* $\mathbf{Op}_c = \mathbf{Op}_{\{c\}}$ *of* operations.

Example 7 *The class* ATM *as modelled in Figure 2.4 can be specified using our formal definition as follows:* $\mathbf{Prop}_{\mathrm{ATM}} = \{\mathbf{att}_{cash}\}$ *and* $\mathbf{Op}_{\mathrm{ATM}} = \{\mathbf{op}_{checkIfCashAvailable}\}$ *where* att_{cash} *is defined in Example 1 and* $\mathbf{op}_{checkIfCashAvailable}$ *is defined as in Example 4.*

Example 8 *Figure 2.4 and Figure 2.6 represent a first view on the static structure of our case study. As shown in Figure 2.4, an* ATM *consists of, for example, a* CustomerConsole *indicated by the composition relationship between* ATM *and* CustomerConsole. *The multiplicity restrictions on this composition indicate that an* ATM *only possesses one* CustomerConsole *and a* CustomerConsole *belongs to only one* ATM. *Remark that in the class diagrams of the next sections and chapters, the names of the associations are not always*

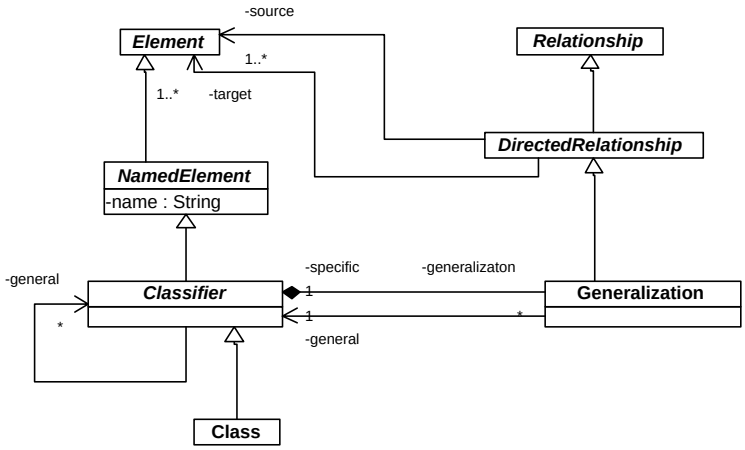

Figure 2.7: Elements of the *Generalisation* package in the UML Infrastructure.

explicitly represented. Showing all the association names would clutter the diagrams and make them unreadable.

All the classes in both figures exhibit quite a lot of behaviour through the definition of different operations.

The class Account *has one attribute* balance *of type* Cash *representing the amount of money of the account. Figure 2.6 shows that the* Account *class is an abstract class. This implies that there are concrete classes, subclasses of the the abstract class, implementing the abstract class' behaviour. This brings us to the* generalisation *relationship in the UML.*

Generalisation and Hierarchies *Generalisation* indicates a taxonomic relationship in UML. A generalisation relationship between a *parent* and a *child class* specifies that each instance of the child class is also an instance of the parent class. This implies that the child class inherits all the features of the parent class. But the child class can add features that do not hold for the parent class.

In the UML metamodel, *Generalization* is a metaclass which has a link to its general *Classifier* and its specialising *Classifier*. As specified in Figure 2.7, a *Classifier* also keeps track of all its immediate ancestors through the directed *general* association. A *Classifier* is a classification of instances. It is an abstract metaclass. Concrete subclasses of *Classifier* are *Class*, *Association*, *PrimitiveType*, but also *Behaviour* explained in Section 2.5.

Using generalisation, several classes can be grouped together to form class hierarchies. On the class diagram, the generalisation arrowhead can be labelled with the name of the generalisation set. Generalisation sets are by default *disjoint*: any instance of the parent class may be an instance of only one of the child classes within that set (see the Restriction 2.1). Remark that UML allows multiple inheritance.

Definition 15 *The binary non-reflexive, transitive relation* **generalisationOf** : $\mathbf{C} \times \mathbf{C}$.

If **generalisationOf**(c, c') where $c, c' \in \mathbf{C}$ then the class c' is called a *child class* of c and c is called a *parent class* of c'.

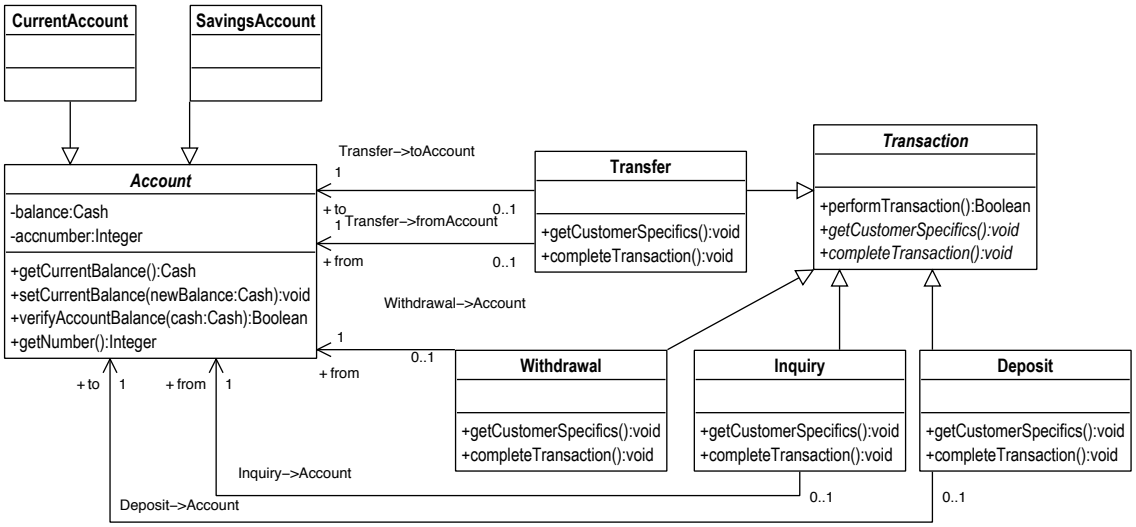

Figure 2.8: A third class diagram from our case study.

Further restrictions can be applied to this **generalisationOf** relationship. For example, the following relation is a restricted **generalisationOf** relationship.

Definition 16 *Let* $c = (\mathbf{Prop}_c, \mathbf{Op}_c) \in \mathbf{C}$ *and* $c' = (\mathbf{Prop}_{c'}, \mathbf{Op}_{c'}) \in \mathbf{C}$*, then* $\mathbf{inheritorOf}(c, c')$ *if and only if*

- $\mathbf{generalisationOf}(c, c') \wedge$
- $\mathbf{Prop}_c \subseteq \mathbf{Prop}_{c'} \wedge$
- $\mathbf{Op}_c \subseteq \mathbf{Op}_{c'}$.

In UML 2.0 properties and operations inherited by a child class can be redefined in the context of this class. How these redefined properties and operations are related to the original properties and operations, must be defined by the user.

Example 9 *Figure 2.8 represents another piece of the static structure of our case study. This class diagram represents the transaction and account hierarchy. An* Account *can be specialised into a* CurrentAccount *or a* SavingsAccount.

Four possible transactions are defined by the following classes: Deposit, Inquiry, Transfer *and* Withdrawal *corresponding to the transaction a user can execute on an ATM.*

The set of properties defined in a class together with its inherited properties is called a *full descriptor* in the UML. The notion of full descriptor can be formalised as follows:

Definition 17 *The* **full descriptor** *of a class* $c \in \mathbf{C}$ *is a tuple* $\mathbf{FD}_c = (\mathbf{Prop}_c^*, \mathbf{Op}_c^*)$ *where*

- $\mathbf{Prop}_c^* = \mathbf{Prop}_c \cup \bigcup_{\forall c' : \mathbf{generalisationOf}(c', c)} \mathbf{Prop}_{c'}^*$
- $\mathbf{Op}_c^* = \mathbf{Op}_c \cup \bigcup_{\forall c' : \mathbf{generalisationOf}(c', c)} \mathbf{Op}_{c'}^*$

2.5 UML 2.0 Sequence and Communication Diagram

A variety of behaviour specification mechanisms are provided by UML, such as *Statemachine*, *Activity*, *Usecase* and *Interaction*. As argued in Section 2.3, we confine ourselves to *Statemachine* and *Interaction* as specifications of behaviour. We will start by basic definitions of *Interaction*.

As described in chapter 14 of [Obj04e], " *Interactions are used in a number of different situations. They are used to get a better grip of an interaction situation for an individual designer or for a group that need to achieve a common understanding of the situation. Interactions are also used during the more detailed design phase where the precise inter-process communication must be set up according to formal protocols. When testing is performed, the traces of the system can be described as interactions and compared with those of the earlier phases.*"

The semantics of an *Interaction* is given by a pair of sets of traces [Obj04e] representing valid traces and invalid traces, respectively. Only the valid traces are described in [Obj04e].

Interactions do not only specify the different possible traces but also the communication paths used for the message passing. Depending on its purpose, an *Interaction* can be displayed with different types of diagrams. We will consider sequence and communication diagrams. As already pointed out in the section on the different UML diagram types, sequence and communication diagrams combine two different run-time aspects: the sequence of messages sent between different objects and the links through which different objects communicate. We will call the former aspect the *interaction view* and the latter aspect the *communication view* of sequence and communication diagrams.

2.5.1 Communication View

We start with a discussion on the fragment of the UML 2.0 metamodel, shown in Figure 2.9, representing the *connections* between lifelines. Based on this metamodel fragment and on previous definitions the main concepts inherent to the communication view are formalised.

A *Lifeline* represents an individual participant in the *Interaction* which is a *ConnectableElement*. Within UML 2.0 there are four types of *ConnectableElements*, i.e., *Parameter*, *Property*, *Variable* and *Port*. We will not consider *Port*, because this is a concept inherent to composite structure diagrams. The metamodel element *Parameter* is discussed in Section 2.4. The *Variable* metaclass is defined in the *Action Semantics* part of the UML metamodel - this part was introduced in UML version 1.5 for MDA purposes and is not considered in this dissertation - as *an element for passing data between actions indirectly* and it stores values. The concept in the metamodel identifying a value or values in a model, is the abstract metaclass *ValueSpecification*. However, in the current UML 2.0 metamodel there is no connection between this metaclass and the metaclass *Variable*. Because, in our opinion, it is important in some cases to know a variable's value, we adapted the UML 2.0 metamodel by introducing a meta-association between *Variable* and *ValueSpecification*. This association, surrounded in Figure 2.9 by a rectangle, expresses that a variable can contain certain values. A *ValueSpecification* can be an *Expression*, a *LiteralSpecification* or an *InstanceValue*. An *Expression* is a tree of *symbol*s denoting a set of values when evaluated in a certain context. A *LiteralSpecification* is an abstract metaclass identifying a literal constant to be modeled as expressed by the different concrete specialisations of this class. An *InstanceValue* identifies an instance and specifies the value modelled by an

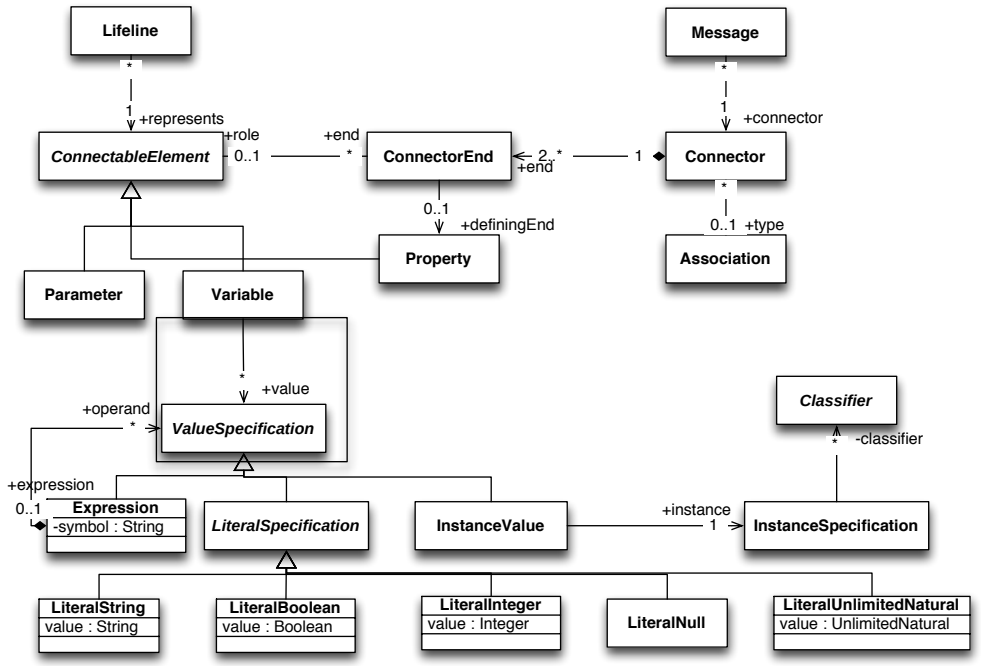

Figure 2.9: UML metamodel fragment of *Connections*.

InstanceSpecification. In the context of interactions, we focus on instance values and their specifications. An instance, i.e., an object in a modelled system at a certain point in time, is represented by the metaclass *InstanceSpecification*. This instance can be classified by one or several classes.

We define objects as instances of classes, i.e., identifiable entities that are referred to by a unique object identifier (oid). Each object is uniquely determined by its identifier and vice versa.

Notation 3 *The set of all object identifiers is denoted by* $\mathbf{O} = \{o_1, o_2, o_3, \ldots\}$. *The set of all object identifiers of a set of classes* R *is denoted by* $\mathbf{O}_R$.

As modelled in Figure 2.9, an *InstanceSpecification* can be classified by several classes. We define a the (partial) function **instanceOf** associating an object identifier to a set of classes of which the object is an instance.

Definition 18 *The partial function* $\mathbf{instanceOf} : \mathbf{O} \to \mathcal{P}(\mathbf{C})$ *and* $\forall o \in \mathbf{O}_R \subseteq \mathbf{O}\ \exists\ C\ \subseteq\ \mathbf{C} : (R\ \subseteq\ C\ \wedge\ \mathbf{instanceOf}(o, C))$.

A **connectable element** abstracts away from a single object. It represents a set of objects that are not necessarily instances of the same set of classes. We define a connectable element as a set of objects.

Definition 19 *A* **connectable element ConnEl** *is defined as* $\mathbf{ConnEl} \subseteq \mathbf{O}$.

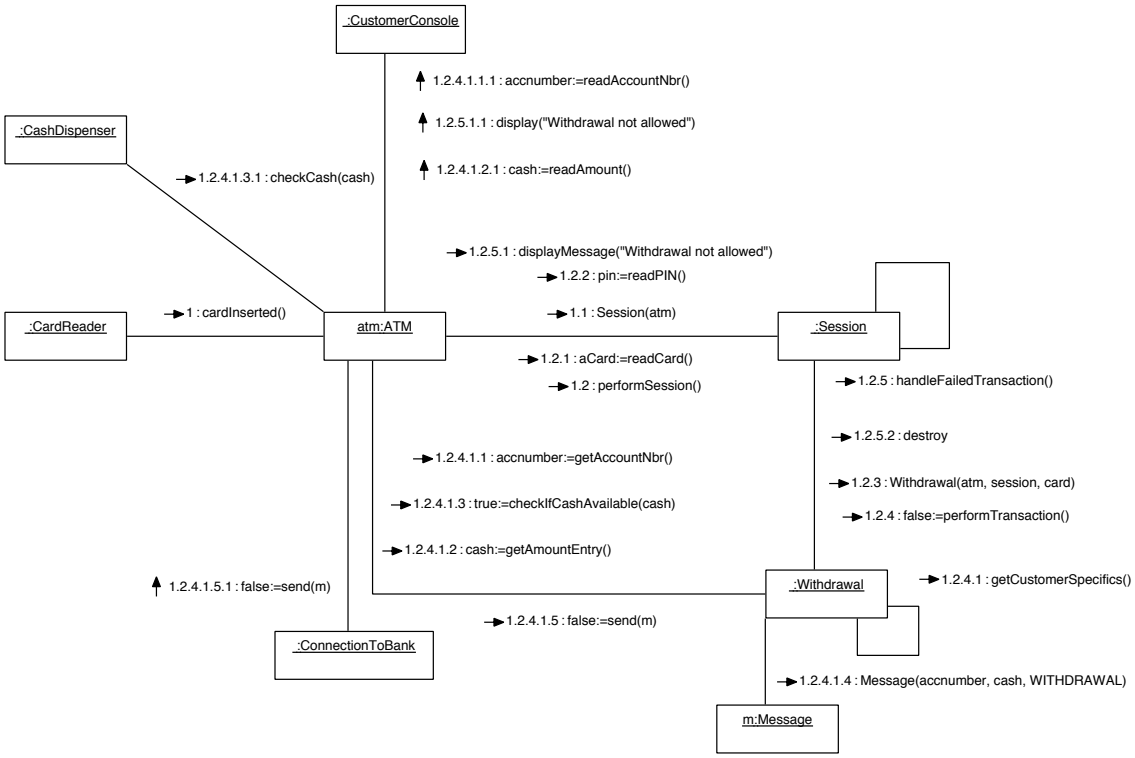

Figure 2.10: UML communication diagram for user session and withdrawal transaction.

Example 10 *The* Interaction *described by the communication diagram in Figure 2.10 shows a scenario where a user starts a session and does a withdrawal from his account. The central concepts are the objects and the different connections between them. The different lifelines are connected to objects. The objects specified in this diagram are 8 anonymous objects and the object named* atm *and the object named* m*. For example, the set of objects of the class ATM is a singleton containing only object identifier* oid_1*,* $\mathbf{O}_{\{ATM\}} = \{oid_1\}$.

Lifelines in a sequence or communication diagram are connected to each other by *Connectors*. The metamodel element *Connector* specifies links enabling the communication between two or more instances. A *Connector* consists of two or more *connector ends*. Each of these *connector ends* represents the participation of instances of classes typing the *connectable elements* attached to this end. Indeed, a *ConnectorEnd* is an endpoint of a *Connector*, attaching the *Connector* to a *ConnectableElement*. Each *ConnectorEnd* is part of exactly one *Connector*. A *ConnectorEnd* is typed by a *Property*. This *Property* represents the corresponding *AssociationEnd* on the *Association* which types the *Connector* owing this *ConnectorEnd*.

Notation 4 *The set of all connectors is denoted by* **Connectors**.

Definition 20 *A* **connector Connector** *is defined as* $\mathbf{Connector} \in \mathcal{P}(\mathbf{O}_{C_1} \times \ldots \times \mathbf{O}_{C_n})$, *where* $\forall i \in \{1, \ldots, n\} : C_i \subseteq \mathbf{C}$.
The partial function $\mathbf{connectorType} : \mathbf{Connectors} \rightarrow \mathbf{Assoc} : \mathbf{Connector} \mapsto \mathbf{assoc}$. *The* **set of connectors typed by an association assoc** *is defined as,* $\mathbf{Connectors_{assoc}} = \{\mathbf{Connector} | \mathbf{connectorType}(\mathbf{Connector}) = \mathbf{assoc}\}$.
A **link l** *is defined as* $\mathbf{l} = \{(o_1, \ldots, o_n)\} \in \mathbf{Connector} \in \mathcal{P}(\mathbf{O}_{C_1} \times \ldots \times \mathbf{O}_{C_n})$.

Example 11 *Between the different lifelines in Figure 2.10 connections are specified that act as communication paths. Consider the connection between the lifeline representing an instance of* Withdrawal *and the instance* atm *of* ATM*. The typing association of this connection is the association specified between the class* ATM *and the abstract class* Transaction *in the class diagram of Figure 2.6. This association is only navigable from* Transaction *to* ATM*. This restriction has an implication on the sending of messages over connections typed by this association. Only messages may be sent from a* Transaction *class to the* ATM *class.*

We can now define the disjointness restriction on a set of classes. Let $R = \{c_1, \ldots, c_n\}$,

$$\forall\ c_i \in R\ \forall\ o \in \mathbf{O}_{\{c_i\}}\ \forall\ c_j \in R \setminus \{c_i\} : o \notin \mathbf{O}_{\{c_j\}} \tag{2.1}$$

2.5.2 Interaction View

To clarify the definitions of the different concepts related to the interaction view, the relevant parts of the UML metamodel are introduced first.

Figure 2.11 shows the relevant fragments of the UML metamodel dealing with interactions. An *Interaction* consists of some *Lifeline*s which are covered by *EventOccurrence*s. *EventOccurrence*s are *MessageEnd*s representing either the receiving event of a *Message* or the sending event of a *Message*. A *Message* is a *NamedElement* that defines one specific kind

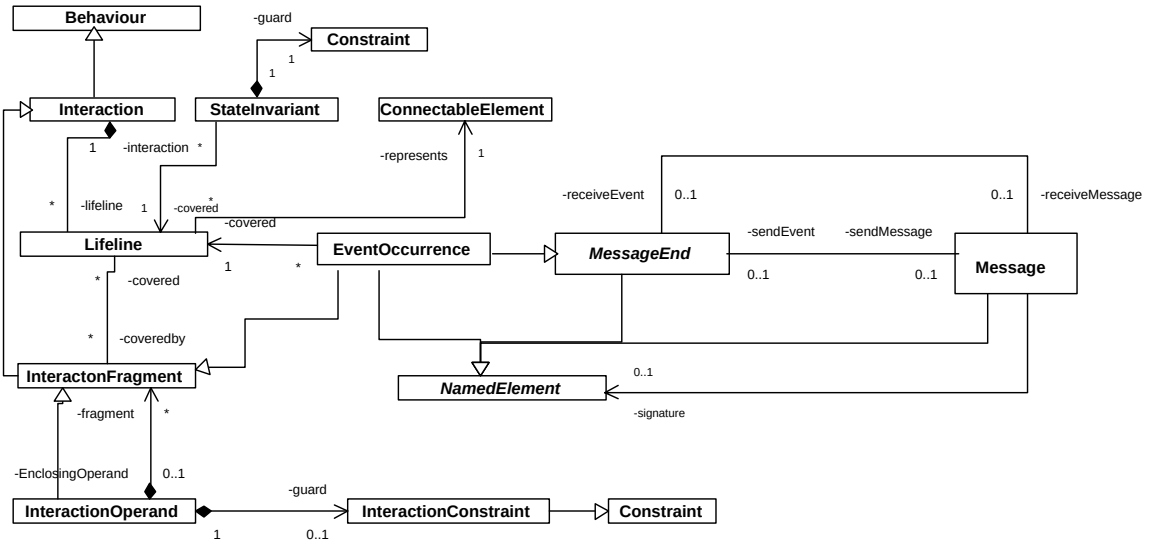

Figure 2.11: UML metamodel fragment for *Interaction*s.

of communication, represented by another *NamedElement*, e.g., an *Operation* in the case of an operation invocation. A *Lifeline* represents a *ConnectableElement*. A *ConnectableElement* represents a set of instances owned by a containing classifier instance. On a *Lifeline*, *StateInvariant*s can be specified. An *InteractionFragment* is a piece of an interaction, which is an interaction in its own right. An *InteractionOperand* is an *InteractionFragment* with an optional guard expression. Only *InteractionOperand*s with a guard that evaluates to true at this point in the interaction will be considered for the production of the traces of the enclosing *Interaction*.

The guards are *InteractionConstraints*, which are *Constraints*. A *Constraint* in UML is a condition or restriction specified in natural language or in a machine readable language. It specifies some additional semantics to certain elements. For example, in the UML metamodel, OCL is used to add well-formedness rules and some semantical restrictions to metamodel elements. In the section on class diagrams we already introduced a notation for pre- and postconditions.

Notation 5 *The set of all possible constraints is denoted by* **Constraints**, *where* **Pre** $\subseteq$ **Constraints** *and* **Post** $\subseteq$ **Constraints**.

Because the user can define constraints in whatever language, it is not possible to generally state how constraints relate to each other. In the UML metamodel, *Constraint* is related to *Element* through a meta-association. This meta-association indicates the elements that are referenced by a particular constraint. We will define a function that applied to a constraint returns the elements referenced. Because we still have to define some model elements the definition of this function is given in Section 2.7.

We formally define a *sequence diagram (SD) trace* for a particular instance of a certain class. To be able to define such a trace, we first need to define the concepts event occurrence and message. We define messages as identifiable entities that are referred to by a unique

message identifier (mid). Each message is uniquely determined by its identifier and vice versa. A message can invoke an operation.

Notation 6 *The set of all message identifiers is denoted by* $\mathbf{M} = \{m_1, m_2, \ldots\}$.
The set of all event occurrences is denoted by $\mathbf{E}$.
The set of all SD traces is denoted by Υ.
The set of all sequence diagrams is denoted by Δ.

Definition 21 *The partial function* **invoked** *:* $\mathbf{M} \rightarrow \mathbf{Op}$.

We can now define an event occurrence and related functions. We also define the equality relation between two events. We use the well-known equal sign subscript e, $=_e$, to denote this relation.

Definition 22 *An* **event occurrence e** *is defined as a triple,* $\mathbf{e} = (\mathbf{m}, \mathbf{Cons}, \mathbf{direction})$ *where* $\mathbf{m} \in \mathbf{M}$, $\mathbf{Cons} \subseteq \mathbf{Constraints}$, $\mathbf{direction} \in \{$ *"send", "receive"* $\}$.
The total function $\mathbf{connected} : \mathbf{E} \rightarrow \mathcal{P}(\mathbf{O}) : \mathbf{e} \mapsto \mathbf{ConnEl} \subseteq \mathbf{O}$.
A binary relation $=_e : \mathbf{E} \times \mathbf{E} : =_e(\mathbf{e} = (\mathbf{m}, \mathbf{Cons}, \mathbf{direction}), \mathbf{e}' = (\mathbf{m}', \mathbf{Cons}', \mathbf{direction}'))$ *if and only if*
$\exists\ \mathbf{invoked}(\mathbf{m}) : \mathbf{invoked}(\mathbf{m}') = \mathbf{invoked}(\mathbf{m}) \wedge \mathbf{Cons} = \mathbf{Cons}' \wedge \mathbf{direction} = \mathbf{direction}'$.

The set of constraints associated to an event occurrence are the constraints specified on the lifeline of the connectable element **ConnEl** before the occurrence of the event occurrence on that lifeline. In UML metamodel terminology, the elements of **Cons** are *StateInvariants* and *InteractionConstraints*.

Definition 23 *A* **SD trace** υ *is defined as a n-tuple of* event occurrences*, denoted* $< e_1, \ldots, e_n >$.
A binary relation $=_\upsilon : \Upsilon \times \Upsilon : =_\upsilon(\upsilon =< e_1, \ldots, e_n >, \upsilon' =< e'_1, \ldots, e'_m >)$ *if and only if* $m = n \wedge \forall i \in \{1, \ldots, n\} : =_e(e_i, e'_i)$.

Remark that we adopt a different notation for an n-tuple in case of a trace (and also in case of a call sequence, see Section 2.6). We use here the notations as introduced in the Superstructure UML 2.0 document [Obj04e]. By using a different notation, we differentiate between the elements defined as part of the abstract syntax of UML and the elements that are not defined in this abstract syntax and that are rather concepts defining a certain semantics.

Different subsequences, obeying a certain predicate, of a SD trace can be defined. First, we define what a subsequence obeying a predicate means.

Definition 24 *A* **subsequence** *of a given SD trace* $\upsilon =< e_1, \ldots, e_n > \in \Upsilon$ *obeying a predicate* P, $\mathbf{sub}_P(\upsilon) =< e'_1, \ldots, e'_m >$ *is defined as:*

- $n \geq m \wedge$
- $\forall\ e'_i \in \mathbf{sub}_P(\upsilon) : P(e'_i) \wedge$
- $\forall\ e'_i \in \mathbf{sub}_P(\upsilon)\ \exists e_j \in \upsilon : (=_e(e'_i, e_j) \wedge j \geq i) \wedge$

- $\forall\ e'_i\ \in\ \mathbf{sub}_P(\upsilon) : (=_e(e'_i, e_j) \wedge =_e(e'_{i+1}, e_k) \wedge j, k \in \{1, \ldots, n\}) \Rightarrow (k \geq j \wedge \nexists\ e \in < e_j, \ldots, e_k >: (=_e(e, e') \wedge e'\ \in\ < e'_1, \ldots, e'_m >))$.

Examples of subsequences of SD traces that will be useful for defining inconsistencies are:

Definition 25 *A* **SD trace** $\upsilon_{\mathbf{O}_R} = \mathbf{sub}_P(\upsilon)$*, where* $R \subseteq \mathbf{C}$ *and* $P(e) = (\mathbf{connected}(e) = \mathbf{O}_R)$.
A **SD trace** $\upsilon_{\mathbf{Op}_R} = \mathbf{sub}_P(\upsilon)$*, where* $R \subseteq \mathbf{C}$ *and* $P(e = (\mathbf{m}, \mathbf{Cons}, \mathbf{direction})) = \mathbf{invoked}(\mathbf{m}_i) \in \mathbf{Op}_R$.

A sequence diagram, as well as a communication diagram typically consists of several traces, as defined below:

Definition 26 *A* **sequence diagram** $\delta \subseteq \Upsilon$ *is a set of SD traces.*

In the remainder of this dissertation, sequence diagram is used to denote sequence diagram as well as communication diagram unless specified otherwise.

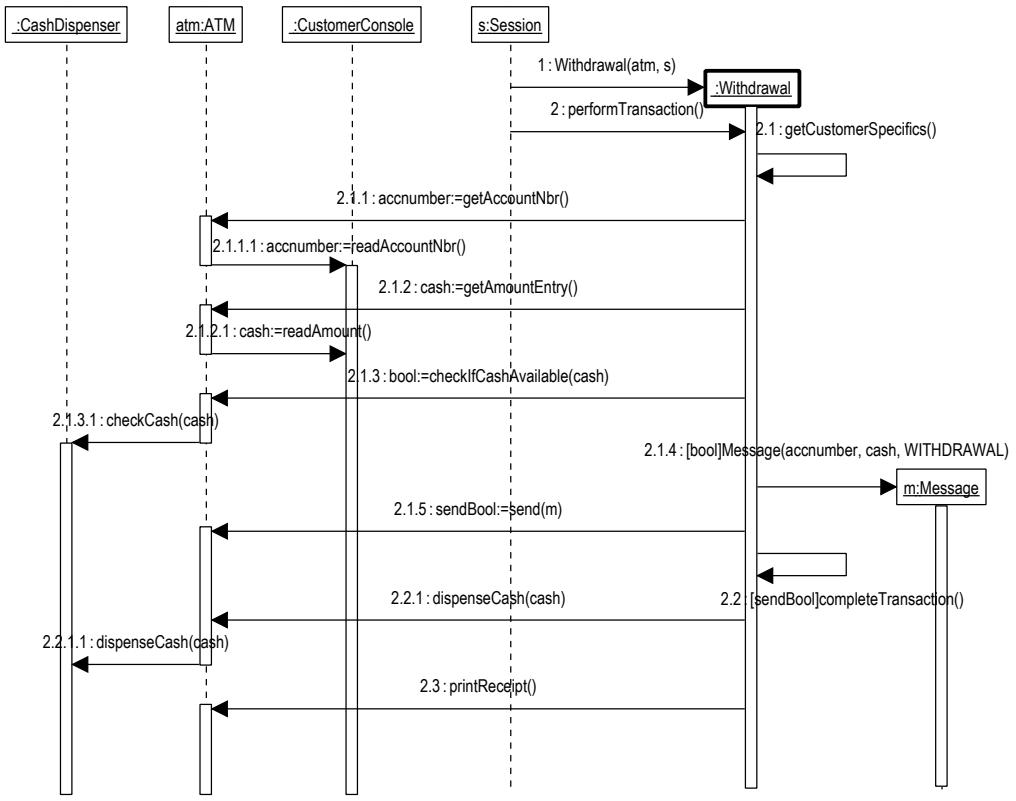

Figure 2.12: UML sequence diagram for the withdrawal transaction.

A sequence diagram does not contain necessarily the whole universe of SD traces. It typically consists of several valid SD traces for instances of several classes. A sequence diagram contains several traces of event occurrences belonging to a certain instance or a set

of instances. The partial function **contained** returns for a certain sequence diagram and a set of classes, a set of connectable elements.

Definition 27 *The partial function* **contained**: $\Delta \times \mathcal{P}(\mathbf{C}) \rightarrow \mathcal{P}(\mathbf{O}) : (\delta, R) \mapsto \{O \subseteq \mathbf{O}_R | \exists\ \upsilon_O \in \delta\}$.

For defining certain kinds of consistency problems, we will only be interested in the order of invocations of an object's operations. As such, the traces of event occurrences representing the receipt of a message are considered. Therefore, we define a *receiving SD trace* as follows:

Definition 28 *A* **receiving SD trace** $\upsilon/^{rec} = \mathbf{sub}_P(\upsilon)$*, where* $P(e) = (e = (\mathbf{m}, \mathbf{Cons}$, *"receive"*$) \wedge \exists\ \mathbf{invoked}(\mathbf{m}))$.

Example 12 *A receiving SD trace of the instance* atm *of class* ATM *in the sequence diagram* δ *of Figure 2.12 is* $< e_1, e_2, e_3, e_4, e_5 >$*, where* e_1 *represents the receipt (by* atm*) of the message* getAccountNbr*,* e_2 *represents the receipt of the message* getAmountEntry*,* e_3 *represents the receipt of the message* checkIfCashAvailable*,* e_4 *represents the receipt of the message* send *and* e_5 *represents the receipt of the message* dispenseCash.

Notation 7 *The* **set of all event occurrences denoting the receipt of a message** *for a set of instances of a set of classes* R *and belonging to the sequence diagram* δ*,* $\mathbf{E}_{\delta,R}$*, is defined as* $\mathbf{E}_{\delta,R} = \{\mathbf{e} | \mathbf{e} = (\mathbf{m}, \mathbf{Cons}$, *"receive"*$) \wedge \mathbf{connected}(\mathbf{e}) \subseteq \mathbf{O}_R \wedge \exists\ \upsilon\ \in\ \delta : \mathbf{e}\ \in\ \upsilon\}$.

Example 13 *Let* δ *and* e_i *be defined as in* Example 12*. Then* $E_{\delta,\{\text{ATM}\}} = \{e_1, e_2, e_3, e_4, e_5\}$

2.6 UML 2.0 State Machine Diagram

UML 2.0 differentiates between two kinds of state machines, *behavioural state machines* and *protocol state machines*. *Behavioural state machines* are used to specify the behaviour of various model elements. *Protocol state machines* (shortly, PSM) are always defined in the context of a single classifier and are used to express usage protocols. A classifier can have several protocol state machines. Protocol state machines express the legal transitions that can be triggered by a classifier. As such they are a convenient way to define a *lifecycle of an object* or an *order of the invocation of its operations*. Because, in the context of the consistency problems we will discuss, the order of invocation of operations is the most important, only protocol state machines are considered here.

Again we will first discuss the relevant part of the UML metamodel shown in Figure 2.13 and used in our work.

A *ProtocolStateMachine* is a piece of *Behaviour* that owns one or more *Region*s. *Region*s own *Vertices* and *ProtocolTransition*s. A *ProtocolStateMachine* is always defined in the context of a *Class*. It specifies which operations of the class can be called in which state and under which condition. A *ProtocolTransition* specifies a legal transition for an *Operation*. Transitions of protocol state machines have next to their *Trigger*, which is an operation invocation, a *pre-* and a *postcondition*. A *Vertex* is an abstraction of a node in a statemachine. It can be the source or destination of any number of transitions. Concrete

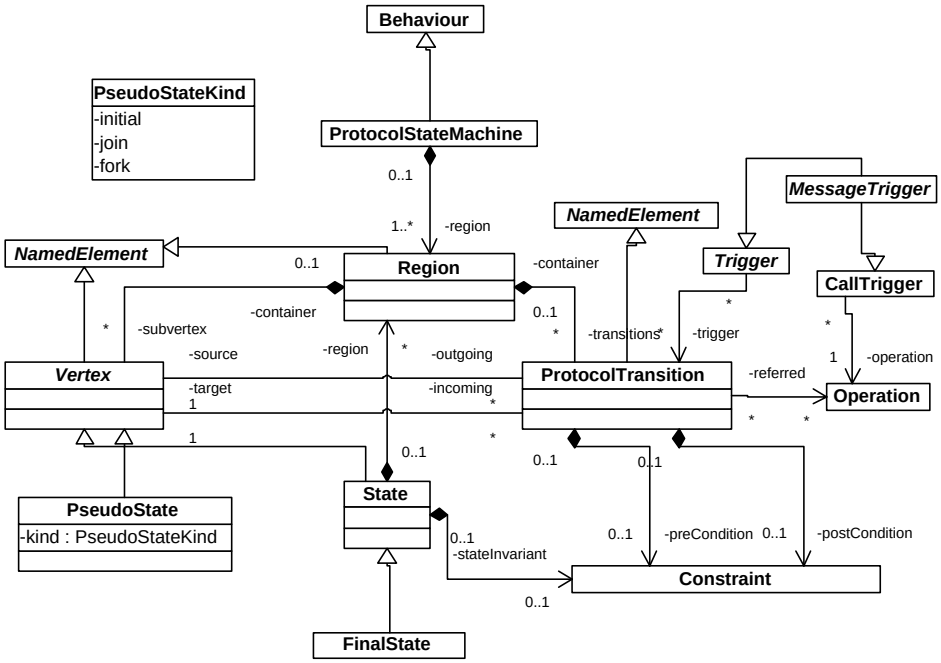

Figure 2.13: The UML 2.0 metamodel fragment used for Protocol State Machines.

subclasses of *Vertex* are: *State*s and *Pseudostate*. A *State* models a situation during which some (usually implicit) variant condition holds. A state invariant is a *Constraint* and specifies conditions that are always true in a certain *State* when this *State* is the current *State*.

Example 14 *The protocol state machine shown in Figure 2.14 specifies the possible operation calls in case the customer chooses to withdraw money. First of all the PIN code is read and verified. If the customer is not able to specify a valid PIN after 3 tries, the card is retained by the ATM. If the customer specifies a valid PIN a transaction is chosen by the customer. In this case, a withdrawal transaction is executed. First the specific data for this transaction must be specified. It is possible for the customer to cancel the transaction in progress. If not, the cash is dispensed and a receipt is printed.*

Figure 2.15 shows part of the protocol state machine of Figure 2.14 that has changed. This part specifies the possible operation calls in case a customer withdraws money and charges his/her card at the same time. Remark that the same account is used to withdraw money and to charge the card. Also the same amount is withdrawn and charged on the card.

Figure 2.16 shows the same part of the protocol state machine as shown in Figure 2.15, but now the behaviour specification is changed such that the different amounts can be withdrawn and charged from different accounts by the user.

The different kinds of *States* distinguished by the UML metamodel, are *simple state*, *composite state* and *submachine state* and *final state*. A simple states does not contain any other states, i.e., substates. A composite state is either a simple composite state, containing one *Region* or, it is a orthogonal composite state, which means that it is decomposed two or more *Region*s. A *Region* contains *States* and *ProtocolTransitions*. Regions define

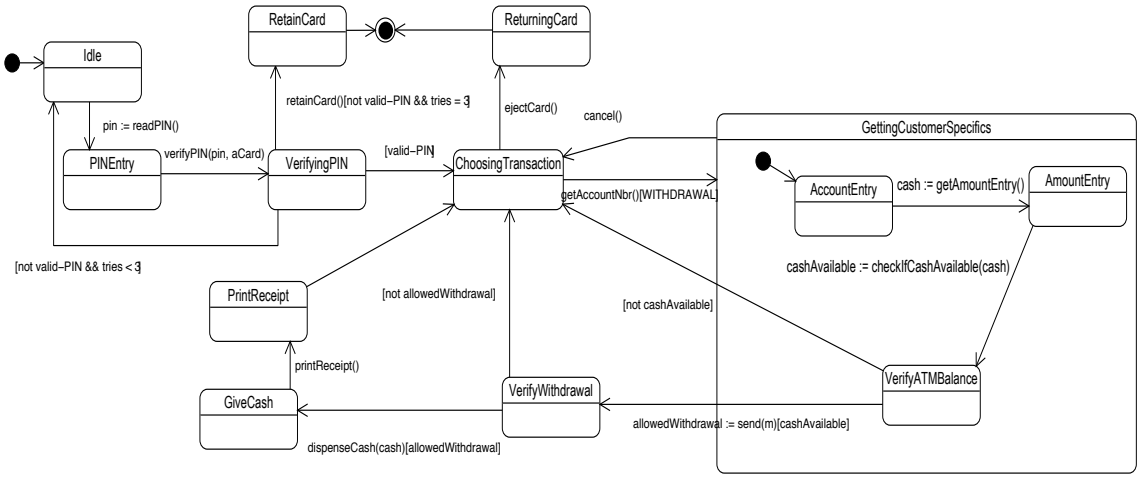

Figure 2.14: Protocol state machine diagram for a withdrawal transaction.

orthogonal parts of a state machine or of a composite state. Each *Region* has *"a set of mutually exclusive disjoint subvertices and a set of transitions"* [Obj04e]. A *submachine state* specifies *"the insertion of the specification of a submachine state machine"* [Obj04e]. Because a submachine state is semantically equivalent to a composite state, we do not explain this kind of state into detail.

The metaclass *FinalState* specialises the metaclass *State*. A *FinalState* specifies that the enclosing region is completed.

Example 15 *The protocol state machine in Figure 2.14 contains a simple composite state, i.e., the state named* GettingCustomerSpecifics*. This composite state contains three different substates and one initial state (denoted by a solid filled circle) which is a kind of* PseudoState *(see next paragraph).*

The part of the protocol state machine in Figure 2.15 contains a simple composite state named GettingCustomerSpecifics*. This state contains an orthogonal composite state, named* VerifyingTransaction*. This orthogonal composite state contains two orthogonal regions. The first region contains four simple states* (InitWithdrawal, VerifyATMBalance, VerifyWithdrawal, GiveCash)*, an initial state and a final state. This final state indicates that this first region is completed and is denoted by a circle surrounding a small solid filled circle. The second region contains three simple states* (InitCharging, VerifyCharging, CardCharging)*, an initial state and a final state.*

The metaclass *PseudoState* is an abstraction subsuming different types of vertices. The types of *PseudoStates* are *initial*, *deepHistory*, *shallowHistory*, *join*, *fork*, *junction*, *choice*, *entryPoint*, *exitPoint* and *terminate*. An *initial* pseudostate represents a default *Vertex* that is the source for a single transition to the default state in, for example, a composite state. A *join pseudostate* merges several transitions emanating from source vertices in different orthogonal regions. A *fork pseudostate* splits an incoming transition into two or more transitions terminating on orthogonal target vertices, i.e., vertices in different regions of an orthogonal composite state. Following the UML 2.0 specifications, *deepHistory* and

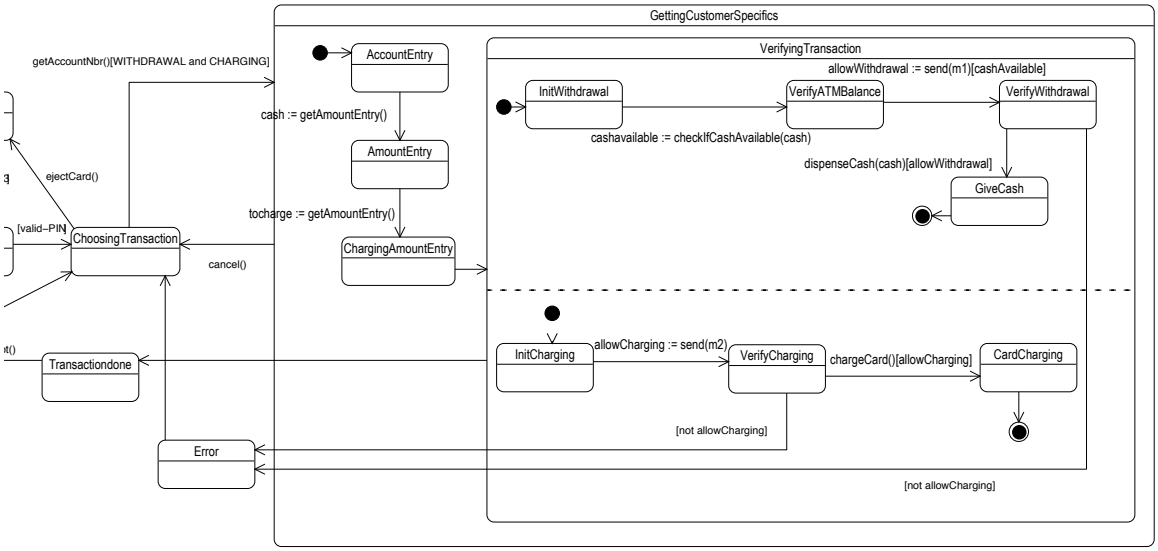

Figure 2.15: Part of protocol state machine diagram for withdrawal and charging transaction.

shallowHistory states can not occur in protocol state machine diagrams. *EntryPoint* and *exitPoint* are defined in the context of submachine states. We will not treat them in our work, but because *submachine* states are semantically equivalent to composite states, the formalisation presented below can also cope with these concepts. We also do not consider *junction* and *choice* vertices. Junction vertices are, as specified by the UML Superstructure document, semantic-free vertices and are used to chain together multiple transitions. In the remainder of this section, we define concepts that chain together different transitions. Choice vertices realise a dynamic conditional branch. Because, we focus on static checking, this kind of vertex is not considered in this work.

Example 16 *The previously introduced (part of) protocol state machines include different initial states. One initial state points to the initial default state of the protocol state machine. Other initial states are defined in a simple composite state or in the regions of an orthogonal composite state indicating the default initial state of this particular state or region.*

In the part of the protocol state machine of Figure 2.16, a fork and junction state are modelled. The fork state splits the incoming transition into two transitions. A first transition has the state WithdrawalAccountEntry *as target, while the second transition has the state* ChargingAccountEntry *as target. These target states are in different regions of the orthogonal composite state* GettingCustomerSpecifics*. The junction state merges two transitions starting from the states* GiveCash *and* CardCharging*, both in different regions of the orthogonal composite state* GettingCustomerSpecifics*, into a single transition terminating on the state* Transactiondone*.*

Notation 8 *The set of all protocol state machines is denoted by* Π*.*
The set of all states is denoted by $\mathbf{S}$*.*

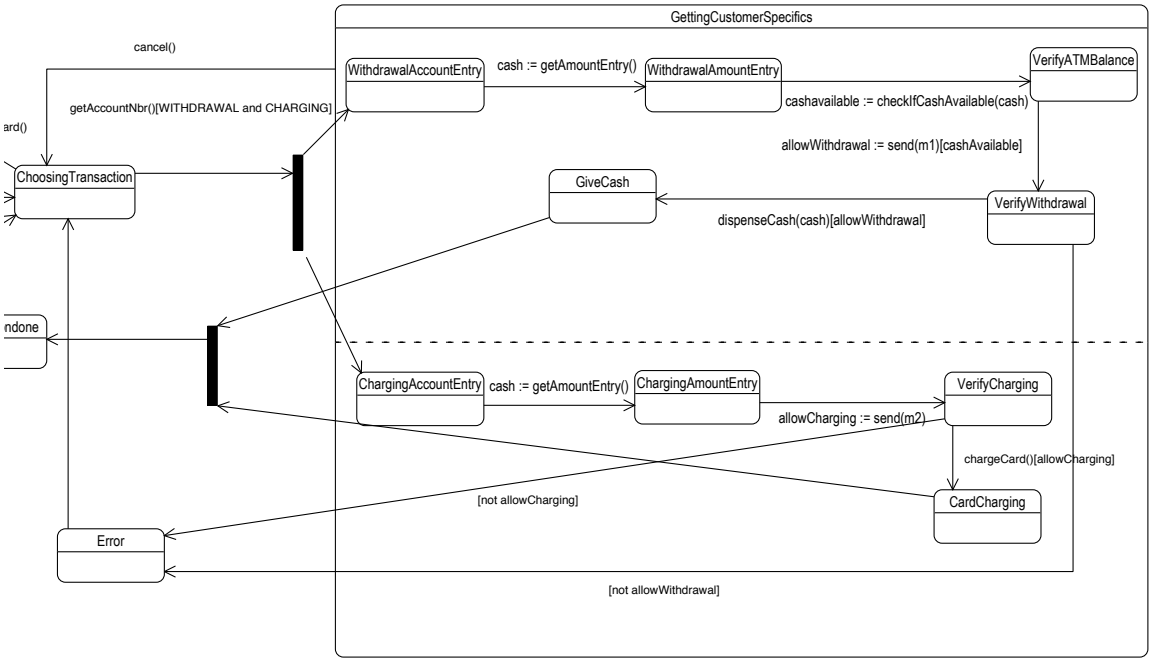

Figure 2.16: Part of changed protocol state machine diagram for a withdrawal and charging transaction.

The set of all labels is denoted by $\mathbf{L}$.

A protocol state machine (PSM) can be defined as follows (based on the definition in [ST03] and in [SS00]):

Definition 29 *A* **protocol state machine** $\pi_c = (S_c, T_c, L_c, \rho_c, \Lambda_c) \in \Pi$ *for a class* c*, consists of a set of states* $S_c \subseteq \mathbf{S}$ *and a labelled transition set* $T_c \subseteq \mathcal{P}(S_c) \times L_c \times \mathcal{P}(S_c)$ *containing labelled relations* (S, τ, S') *such that* $\tau \in L_c$ *where* $L_c \subseteq \mathbf{L}$ *is a set of labels.*

A label τ *is defined as a triple* $(\mathbf{op}, g, h)$ *where* **op** *is an operation,* $g \subseteq \mathbf{Pre_{op}}$ *specifies the precondition of the transition (which is evaluated as part of the precondition of the operation op), and* $h \subseteq \mathbf{Post_{op}}$ *specifies the postcondition of the transition (which is part of the postcondition of the operation* **op***), or, as a triple* $\tau = (\epsilon, g, \{\})$*, where* ϵ *corresponds to a dummy operation and* $g \in$ **Constraints** *specifies the guard.*

ρ_c *denotes the top-most initial state, and* $\nexists S \subseteq S_c, \nexists S' \subseteq S_c$*:* $\rho_c \in S' \ \wedge \ (S, \tau, S') \in T_c$.

Λ_c *denotes the set of final states of the state machine, for which* $\nexists S \subseteq \Lambda_c, \nexists S' \subseteq S_c$*:* $(S, \tau, S') \in T_c$.

Note that in UML PSMs there can be transitions without operation calls and without any guard, as well as transitions that only have a guard specified but no operation call. To support both kinds of transitions in the above definition, we use labels of the form $l = (\epsilon, g, \{\})$.

Due to the fact that a transition is specified as a relation between *sets of states*, simple, composite and orthogonal composite states are also supported. Our definition also supports

high-level, compound and completion protocol transitions. We first define the notion of *(active) life cycle state configuration*, and then we explain the different kinds of transitions and show how these transitions and the different kinds of states are supported by Definition 29.

Definition 30 *A* **(life cycle) state configuration** Σ_c *in a PSM* $\pi_c = (S_c, T_c, L_c, \rho_c, \Lambda_c) \in \Pi$ *is a tree of states belonging to* S_c*.*

A **(life cycle) state configuration** is a *tree* of states, because of the existence of composite and orthogonal composite states.

Example 17 *Consider as an example, the part of the protocol state machine shown in Figure 2.15. A possible life cycle state configuration is the tree (we represent a tree by nested sets)* { GettingCustomerSpecifics, {AmountEntry}}*. Not only the simple state* AmountEntry *is considered as part of the life cycle state configuration but also all the directly or transitively composite states to which the simple state belongs. Another example of a life cycle state configuration is the tree* {GettingCustomerSpecifics, {VerifyingTransaction, {VerifyWithdrawal, CardCharging}}}*.*

The *active state of an object* at a given point in time is defined by the set of states the object is in. This set of states is referred to as the *active (life cycle) state configuration* of the object. In the example above, this is the set containing the leaf states *VerifyWithdrawal* and *CardCharging*.

Definition 31 *An* **active (life cycle) state configuration** σ_c *in a PSM* $\pi_c = (S_c, T_c, L_c, \rho_c, \Lambda_c) \in \Pi$ *is defined as* $\sigma_c \subseteq S_c$ *and corresponds to the set of leafs of a (life cycle) state configuration* Σ_c*.*

We now show how Definition 29 supports the concepts of composite, orthogonal composite states and high-level, compound and completion protocol transitions by transforming UML protocol state machines containing these concepts into a canonical form such that only transitions between simple states are considered maintaining the semantics of the different modeled kinds of states and transitions.

- A **composite state** that is not orthogonal, is a hierarchical state of which exactly one of its substates can be active.
- If a **composite orthogonal state** is active, all of its regions are active, one substate in each region. Transitions can be called on an active composite orthogonal state, which is supported by our definition of state machines. After all, a labelled transition is defined between *sets* of states.
- A **high-level transition** is a transition that originates from a composite. The transition with operation call *cancel()* originating from the state *GettingCustomerSpecifics* in Figure 2.14 is an example of a high-level transition. Such a transition is inherited by all substates of the composite state and gives rise to the generation of as much labelled transitions as there are substates in the composite state in our definition. In the example, the following transitions are generated:
  $(\{AccountEntry\}, \tau, \{ChoosingTransaction\})$,

($\{AmountEntry\}, \tau, \{ChoosingTransaction\}$) and
($\{VerifyATMBalance\}, \tau, \{ChoosingTransaction\}$).
If the composite state is orthogonal, a labelled transition is generated for each possible set of active state configurations of the orthogonal state, having this set of active state configurations as source state.

- A **compound transition** is "*an acyclical unbroken chain of transitions joined via join or fork pseudostates that define a path from a set of source states (possibly a singleton) to a set of destination states, (possibly a singleton). ... A (simple) transition connecting two states is therefore a special common case of a compound transition*" [Obj04e]. The state machine of Figure 2.16 contains a fork and a join state giving rise to compound transitions. The incoming and outgoing transitions on a fork state are translated into one labelled transition (S, τ, S') where S denotes the set of source states of the incoming transition of the fork state. S' is the set of the target states of the outgoing transitions of the fork state. In the example, the following transition is generated ($\{ChoosingTransaction\}, \tau, \{WithdrawalAccountEntry, ChargingAccountEntry\}$)

  In case of a join state, a similar translation happens. Again, the incoming and outgoing transitions on a fork state are translated into one labelled transition (S, τ, S'). S denotes the source states of the incoming transitions of the join state. S' denotes the target states of the outgoing transition of the join state. In the example, the following transition is generated ($\{GiveCash, CardCharging\}, \tau, \{Transactiondone\}$).

- A **completion transition** is a transition where the source is a composite state and without an explicit operation call but a precondition can be specified. A completion transition is represented by one labelled transition (S, τ, S'), where S is the final state of a composite or set of final states of an orthogonal composite state. S' is the set of target states of the completion transition. The protocol state machine in Figure 2.15 contains such a completion transition. The transition starts on the orthogonal composite state named *VerifyingTransaction* and terminates on the simple state *Transactiondone*. It does not specify any operation invocation nor any pre- or postconditions. This transition gives rise to the following labelled transition: ($\{GiveCash, CardCharging\}, \tau, \{Transactiondone\}$).

The firing of a transition enables the change of active state configuration.

Definition 32 *A* **PSM trace** γ_c *in a PSM* $\pi_c = (S_c, T_c, L_c, \rho_c, \Lambda_c) \in \Pi$ *is a n-tuple of active (life cycle) state configurations, denoted* $< \sigma_{c,1}, \ldots, \sigma_{c,n} >$*, such that*
$\sigma_{c,1} = \{\rho_c\} \wedge \forall i \in \{1 \ldots n-1\}\ \exists \tau \in L_c : ((\sigma_{c,i}, \tau, \sigma_{c,i}) \in T_c \vee (\sigma_{c,i}, \tau, \sigma_{c,i+1}) \in T_c)$.

Definition 33 *A* **call sequence** μ_c *in a PSM* $\pi_c = (S_c, T_c, L_c, \rho_c, \Lambda_c) \in \Pi$ *is a n-tuple of labels, denoted* $< \tau_1, \ldots, \tau_n >$ $(n \geq 1)$*, such that* $\forall\ i\ \in\ \{1, \ldots, n\} : \tau_i \in L_c$.

Definition 34 *Given a call sequence* $\mu_c =< \tau_k, \ldots, \tau_n >$ *and an active state configuration* $\sigma_{c,k}$ *and a PSM* $\pi = (S, T, L, \rho, \Lambda)$*:*
valid *is a ternary relation such that* **valid**$(\mu_c, \sigma_{c,k}, \pi)$ *if* $\exists \gamma_c\ =\ < \sigma_{c,1} \ldots \sigma_{c,k} \ldots \sigma_{c,n+1} >$*, where* γ_c *is a PSM trace in* π *and* $\forall i \in \{k, \ldots, n\}$*:* $(\sigma_{c,i}, \tau_i, \sigma_{c,i+1}) \in T$.

Example 18 $< dispenseCash, printReceipt, ejectCard >$ *is a valid call sequence on the active state configuration* $\{\text{VerifyWithdrawal}, \text{CardCharging}\}$ *of Figure 2.16.*

There is of course a relation between sequence (or communication) diagrams and state machine diagrams. A state machine diagram gives the complete behaviour of a single object, whereas a sequence (or communication) diagram gives a single behaviour (trace) of a set of objects. Each type of diagram contains complementary notions revealing different dynamic aspects of the software system and they can be used to support each other's specification.

In order to relate transitions and messages to each other, we need an auxiliary definition. The restriction of a sequence to a certain set is the sequence obtained by removing all elements from the sequence that do not belong to the set or are not equal to an element of this set.

Definition 35 *The* **restriction** $\mu_L =< \tau'_1, \ldots, \tau'_m >$ **of a sequence** $\mu =< \tau_1, \ldots, \tau_n >$ **to a set** L $(n \geq m)$ *is defined as:*
$(\forall\ \tau'_i \in \mu_L : \tau'_i \in \mu \wedge (\tau'_i \in L \vee \exists\ \tau \in L : = (\tau'_i, \tau))) \wedge (\forall\ j \in \{1, \ldots, m\} : (= (\tau_i, \tau'_j) \wedge = (\tau'_{j+1}, \tau_k) \wedge i, k \in \{1, \ldots, n\}) \Rightarrow (i \leq k \wedge \nexists \tau \in < \tau_i, \ldots, \tau_k >: (= (\tau, \tau') \wedge \tau' \in \mu_L))$, *where* $=$ *is* $=_e$ *in case* μ *is a SD trace.*

An event occurrence defined in a sequence diagram can now be mapped onto a label belonging to a PSM.

Definition 36 *The function* $\mathbf{label} : \mathbf{E} \rightarrow \mathbf{L} : (\mathbf{m}, \mathbf{Cons}, \text{"receive"}) \mapsto (\mathbf{op}, g, h)$ *maps an event occurrence onto a label as follows:*
$\mathbf{op} = \mathbf{invoked}(\mathbf{m})$
$g = \mathbf{Pre_{op}} \cup \mathbf{Cons}$
$h = \mathbf{Post_{op}}$.

Now we defined the model elements that will be used in the next chapters, we can also define the function **constrainedElements** that applied on a constraint returns the elements referenced by the constraint. First, we introduce some notions used in this definition.

Definition 37 *The partial function* $\mathbf{constrainedElements}: \mathbf{Constraints} \rightarrow \mathbf{Prop} \cup \mathbf{Op} \cup \mathbf{M} \cup \mathbf{S} \cup \mathbf{Connectors} \cup \mathbf{C}$ *returns the set of elements referenced by a constraint.*

2.7 UML 2.0 Models

UML 2.0 defines a model as *a description of a physical system with a certain purpose, such as to describe logical or behavioural aspects of the physical system to a certain category of readers.*

In our view, a UML model is an abstraction of a software system, owning all the necessary elements representing the system according to the purpose of this model. Using the formalisation of the UML subset used in this work, we define a *UML model* as follows:

Definition 38 *A* **UML model** $\mathcal{M}$ *is a tuple*

$$(\mathcal{C}_\mathcal{M}, \mathbf{Connectors}_\mathcal{M}, \mathbf{Constraints}_\mathcal{M}, \mathbf{Pre}_\mathcal{M}, \mathbf{Post}_\mathcal{M}, \mathcal{O}_\mathcal{M}, \Delta_\mathcal{M}, \Pi_\mathcal{M},$$
$$\mathbf{type}, \mathbf{multiplicity}, \mathbf{participates}, \mathbf{isAbstract}, \mathbf{isNavigable}, \mathbf{isComposite},$$

$$\mathbf{generalisationOf}, \mathbf{instanceOf}, \mathbf{connectorType}, \mathbf{connectorEnd}, \\ \mathbf{connectorEndType}, \mathbf{contained}, \mathbf{invoked}, \mathbf{connected}, =_e, \mathbf{valid}) \tag{2.2}$$

where:

- $\mathcal{C}_{\mathcal{M}} \subseteq \mathbf{C}$,
- $\mathbf{Connectors}_{\mathcal{M}} \subseteq \mathbf{Connectors}$ *is a set of connectors,*
- $\mathbf{Constraints}_{\mathcal{M}} \subseteq \mathbf{Constraints}$ *is a set of constraints,*
- $\mathbf{Pre}_{\mathcal{M}} \subseteq \mathbf{Constraints}_{\mathcal{M}}$ *is a set of preconditions,*
- $\mathbf{Post}_{\mathcal{M}} \subseteq \mathbf{Constraints}_{\mathcal{M}}$ *is a set of postconditions,*
- $\mathcal{O}_{\mathcal{M}}$ *is a set of objects,*
- $\Delta_{\mathcal{M}}$ *is a set of which each element is a sequence diagram describing interactions among objects of* $\mathcal{O}_{\mathcal{M}}$,
- $\Pi_{\mathcal{M}}$ *is a set of state diagrams consisting of protocol state machine diagrams* π,
- *the functions* **type**, **multiplicity** *and* **participates** *are defined in Section 2.4,*
- *the binary relations* **isAbstract**, **isNavigable** *and* **isComposite** *are defined in Section 2.4,*
- **generalisationOf** *reflects the generalisation hierarchy of classes and is defined in Secton 2.4,*
- *the relations* **instanceOf**, **connectorEnd**, **invoked** *and* $=_e$ *are defined in Section 2.5,*
- *the functions* **contained**, **connected**, **connectorType** *and* **connectorEndType** *are defined in Section 2.5,*
- *the relation* **valid** *is defined in Section 2.6.*

2.8 Conclusion

This chapter provides an introduction to UML 2.0 and to a case study that will be used as a running example through the rest of this dissertation. The architecture of the UML 2.0 and its different diagram types, allowing the specification of different aspects of a software system, are explained.

The research restriction considered here, is that only a limited set of UML concepts is taken into account. This means that only class diagram concepts representing the specification of the static structure of a software system, and sequence (communication) diagram concepts representing the specification of possible interactions and protocol state machine concepts are taken into account.

A first contribution, presented in this chapter, is the formalisation of that fragment of the UML. Using this formalisation, several inconsistencies in and between UML models can be precisely specified. This is the subject of the next chapter.

Chapter 3

Conceptual Classification of Inconsistencies

In this chapter, we present a conceptual classification of different inconsistencies that can occur in object-oriented models. First, the different consistency dimensions as explained in the introductory chapter are elucidated in the context of UML (Section 3.1). Next, a two-dimensional classification, based on the kind of the model affected and on the behavioural or structural nature of the inconsistencies is introduced (Section 3.2).

The structural inconsistencies are described first (Section 3.3, Section 3.4 and Section 3.5). Before we start with a detailed discussion of the different behavioural inconsistencies (Section 3.7, Section 3.8 and Section 3.9), a general discussion on behaviour and inheritance of behaviour in object-oriented analysis and design is presented (Section 3.6).

The description of the different inconsistencies is followed by a discussion and related work on existing inconsistency definitions (Section 3.10). Finally, the contribution of this chapter is summarised and issues to be tackled in the next chapter are introduced (Section 3.11).

3.1 UML and Consistency

In the introduction of this dissertation, we introduced the consistency dimension distinguishing between horizontal, evolution and vertical consistency. In this section, we will show how these definitions apply to the UML and more specifically, to the part of the UML discussed in the previous chapter. We will also provide examples of violations of the different kinds of consistency using our case study.

According to Spanoudakis [SZ01], an *inconsistency* is a state in which two or more overlapping elements of (different) software models make assertions about the aspects of the system they describe which are not jointly satisfiable. This definition is directly applicable to UML models that are specified as a set of overlapping UML diagrams. Different diagrams can relate to the same model elements and can make conflicting assertions about these elements.

Horizontal consistency problems can occur in UML models due to two reasons. The first reason is inherent to the UML language. The UML incorporates different types of diagrams allowing the modeller to describe different aspects of the application. For example, class di-

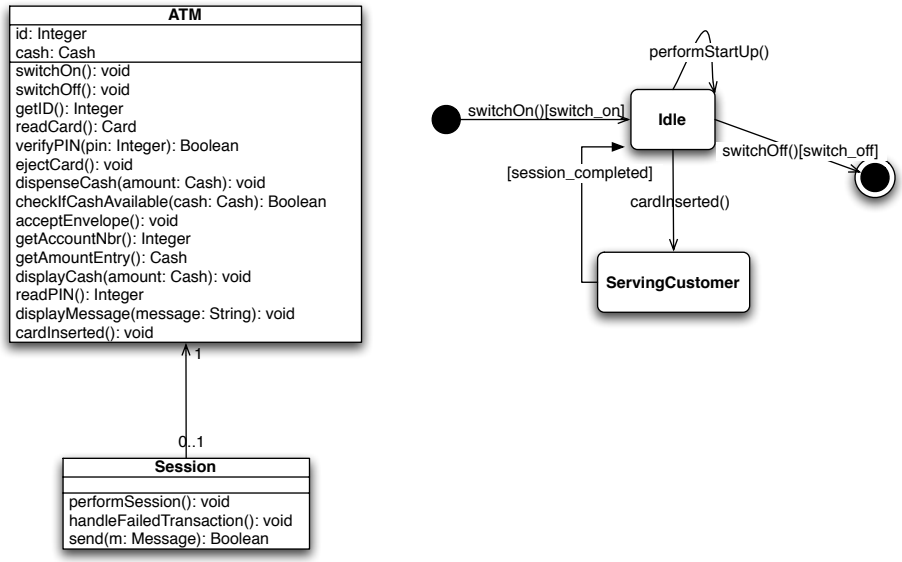

Figure 3.1: Example of a horizontal consistency conflict.

agrams constitute a static view on the application, while sequence, communication and state machine diagrams are used to model a specific view on the behaviour of the application. Structure diagrams specifying the static structure, and behaviour diagrams specifying the behaviour of the application, do overlap and as such can become inconsistent. For example, an operation may be (re)moved in a class diagram while an instance of this class (i.e., an object) in a sequence diagram still relies on this operation to handle a message it receives from another object. Behaviour diagrams can also contain overlapping information. For example, sequence and communication diagrams often represent the same information, but the first one emphasises the interactions and the second one the communication infrastructure.

The second cause of a horizontal consistency problem is due to the fact that a model can consist of different submodels. Each of these submodels focuses on a different part of the application. For example in Figure 2.4, Figure 2.6 and Figure 2.7, class diagrams are specified for different parts of the ATM application. These submodels are not independent, i.e., some parts of the submodels overlap or submodels are complementary. Remark that for horizontal consistency only submodels at the same level of abstraction are considered.

Example 19 *In Figure 3.1, on the left-hand side, a simple class diagram is taken from our ATM case study and on the right hand side, a PSM diagram expressing start-up and shutdown behaviour of our ATM example is shown. The model consisting of those two diagrams contains a horizontal consistency conflict. The operation* performStartup(): void *is used in the state machine diagram but not defined in the class diagram.*

In the introduction, we defined *evolution consistency* between different versions of the same model at the same level of abstraction. Evolution consistency conflicts arise between UML models when a UML model or a submodel is restructured or refactored. They can also occur when model elements are added, modified or deleted from UML models or submodels.

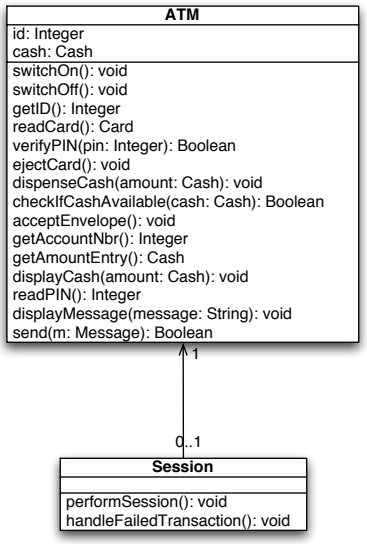

Figure 3.2: *Move Operation* on a class diagram.

Example 20 *As an example, consider the model refactoring* Move Operation *(see also Section 9.4.3). This model refactoring is comparable to the source-code refactoring* Move Method*, but at UML model level, we do not have source code at our disposal. We apply this refactoring to the* send(m: Message): Boolean *method defined in the class* Session *in the class diagram in Figure 3.1. This operation is moved to the class* ATM *as shown in the class diagram in Figure 3.2. This model refactoring may impact sequence diagrams specifying interactions concerning* ATM *objects or* ATM *or* Session *PSM diagrams.*

Vertical consistency exists between models or submodels at different levels of abstraction. Vertical consistency conflicts can result from refining a model or submodel or by adding or modifying models or submodels.

Example 21 *In Figure 3.3, a sequence diagram modelling the object interactions as a consequence of the start-up of an ATM is shown. Together with the state diagram shown on the right-hand side of Figure 3.1, there is a possibility to have a vertical consistency problem. We assume here that the state machine diagram is on a higher level of abstraction than the sequence diagram. The trace in the sequence diagram received by an* ATM *object must conform to a call sequence in the state machine diagram. One kind of conformance specifies that the trace is included in a call sequence of the state machine diagram. In this case, the sequence* switchOn(), performStartup() *is also included in the state machine diagram.*

Syntactic consistency guarantees conformance of models to the abstract syntax of the modelling language. In case of the UML, this means that the user-defined UML models must conform to its abstract syntax. The abstract syntax of UML is a set of class diagrams together with so-called well-formedness rules. These well-formedness rules are expressed in the OCL. Current UML CASE tools have incorporated ad-hoc support for compliance with

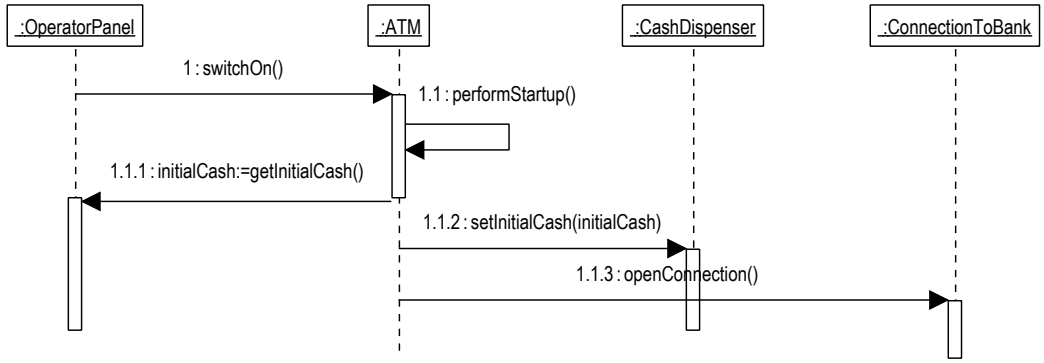

Figure 3.3: Sequence diagram modelling ATM start-up behaviour.

UML well-formedness rules. Due to this definition of syntactic consistency, our classification only addresses *semantic consistency* violations.

UML lacks formal semantics. Its semantics is described by OCL constraints and some informal statements in natural language, more specifically, in English. UML is a general purpose modelling language, as a consequence, even the informal specified semantics can change depending, for example, on the modelling process used. UML specifications do not address *semantic consistency* issues at all.

Based on the formalisation defined in the previous chapter, we introduce a basic set of semantic inconsistencies. Depending on the application or the modelling process, this set can be extended by domain-specific or process-specific inconsistencies.

In the remainder of this dissertation, we mix the terms *consistency* and *inconsistency*. An inconsistency is in this work the negation of a consistency and vice versa .

3.2 Conceptual Classification Explained

A two-dimensional classification of different inconsistencies is introduced in this section. The different dimensions are explained and a motivation for the different inconsistencies is given.

3.2.1 Overview

Two dimensions are distinguished: *structural versus behavioural* and *specification versus instances*.

Structural versus Behavioural

The first dimension indicates whether *structural* or *behavioural* aspects of the models are affected. Structural inconsistencies arise when the specification of the application's structure is inconsistent or when this specification is inconsistent with the specification of behaviour.

Behavioural inconsistencies arise when the specification of the application's behaviour is inconsistent.

In addition to classifying the inconsistencies by the affected aspects of the models, they are also classified by the level of the affected model.

Specification versus Instance

The second dimension concerns the level of the affected model. We differentiate between two levels, the *Specification* level and the *Instance* level. The specification level contains model elements that represent specifications for instances, such as classes, associations and messages. Model elements specifying instances, such as objects, links are at the instance level. In terms of UML diagrams, this would naturally imply that structure diagrams, such as class diagrams belong to the specification level and behaviour diagrams, such as sequence and state machine diagrams belong to the instance level. However, sequence diagrams can also belong to the specification level.

Although sequence diagrams are widely used, the interpretation of sequence diagrams is rather vague. In the previous chapter, we focused on the interaction and communication view. These views can be seen as *semantic dimensions*. Sequence diagrams can exhibit different semantic alternatives, i.e., multiple interpretations of the same syntactic symbol is possible. A more in depth discussion of the different semantic dimensions of sequence diagrams and the corresponding numerous interpretations found in literature is opened in chapter 6, that introduces DLs as semantic domain for UML model (elements).

One of the semantic dimensions that sequence diagrams can have is explained here because of its importance for understanding the classification and the different categories of inconsistencies. The UML draws a distinction between *interactions between objects* and *interactions between roles*. The latter is the specification of an interaction between *roles* objects can play, and a set of messages between these roles. In this form, sequence diagrams abstract away from particular objects and focus more on properties typical to a particular connectable element. Sequence diagrams are used to describe interactions between roles when their intention is to describe prototypical interactions of design patterns. In the former case, i.e., when sequence diagrams represent *object interactions*, they contain a set of operation calls specified between different, *particular* objects. This usage is employed, e.g., in the case of program testing.

In previous versions of the UML (versions 1.x), the syntactic distinction between those two possible interpretations is made by underlining or otherwise, the names of the participants in the interaction. Both representations map to different UML version 1.x metamodel elements. In UML 2.0, both interpretations are still possible but the syntactic difference is blurred and both interpretations map to the same metamodel elements.

In our classification of inconsistencies, we will also make a distinction between sequence diagrams representing object interactions and the ones representing role interactions. The former ones belong to the instance level, the latter ones to the specification level.

In Table 3.1, the different categories of inconsistencies are listed. Before explaining the different inconsistencies in the next sections, we motivate why we listed and classified this particular set of inconsistencies.

	Behavioural	Structural
Specification	invocation interaction inconsistency observation interaction inconsistency	dangling type reference inherited cyclic composition connector specification missing
Specification-Instance	specification incompatibility specification behaviour incompatibility invocation behaviour inconsistency observation behaviour inconsistency	instance specification missing
Instance	invocation inheritance inconsistency observation inheritance inconsistency instance behaviour incompatibility	disconnected model

Table 3.1: Two-dimensional inconsistency table.

3.2.2 Motivation

The set of inconsistencies in Table 3.1 is based on the model elements occurring in the fragment of the UML metamodel we restricted ourselves to. The addition, deletion and modification of those model elements can lead to inconsistent models. But also the specification of behaviour and especially the inheritance of behaviour (see Section 3.6), can introduce specific inconsistencies between or within UML models. Remark that we ignore inconsistencies that are violations of UML well-formedness rules because these rules are considered to be integrated in the modelling language and to be part of the language's syntax. This classification is based on an extensive literature study ([Men99], [ET00], [EE95], [KRSH02], [EKHG01]).

We do not claim that the list of inconsistencies presented in Table 3.1, is exhaustive. Most of the listed inconsistencies can be found in literature. We have good reasons to assume that this list contains the most important inconsistencies that can be found in industrial cases. When we published our inconsistency classification for the first time [VMSJ03], a set of inconsistencies was also presented in [LCM+03] and later in [Lan03] based on six large-scale industrial case studies. A subset of our observed set of structural inconsistencies, are also recognised by this work together with some additional model incompleteness issues. Incompleteness occurs when there is an element in the overlapping part of diagrams in the one diagram without matching counterpart in the other diagram [Lan03]. As argued in Chapter 1, a model can be incomplete and henceforth, we only consider model inconsistencies. Behavioural inconsistencies are not studied by Lange *et al.* [Lan03].

We also presented our classification and inconsistency detection mechanism in [SVJM04] and as a chapter in the book "Software Evolution with UML and XML." [MVS05]. Although the inconsistencies are tailored towards the UML, they can be applied to models represented in whatever modelling language as long as the abstract syntax and semantics of (part of) that modelling language corresponds to our previously introduced definitions.

3.2.3 Inconsistency Template

In the next sections every category defined in Tabel 3.1 is studied in detail. We describe the inconsistencies using the following template.

UML Model Elements A list of UML metaclasses involved in the inconsistency.

Definition Each inconsistency is defined using the formalisation of UML models and modelling elements introduced in the previous chapter. The notation $\mathcal{M}$ is used to indicate the UML model under consideration (shortly, MUC) and the tuple (2.2) defined in Definition 38 is used to define and denote the different elements of a model.

Example The inconsistency is exemplified by simple, yet representative UML diagrams.

This template lends a consistent format to the description of the classified inconsistencies.

3.3 Structural Specification Inconsistencies

At the specification level, we identify three kinds of structural inconsistencies, *inherited cyclic composition inconsistency*, *dangling type reference* and *connector specification missing*.

3.3.1 Inherited Cyclic Composition Inconsistency

An *inherited cyclic composition inconsistency* arises when a composition relationship (the multiplicity constraints of this composition relationship are important) and inheritance relationships between classes of a model form a cycle that produces an infinite containment of the instances of the affected classes.

Involved UML Model Elements: *Class*, *Association* which is a composition, *Property*, *MultiplicityElement* and *Generalization*.

Definition 39 $\mathcal{M}$ *suffers from an* **inherited cyclic composition inconsistency** *if and only if* $\exists$ **c, c'** $\in \mathcal{C}_{\mathcal{M}}$: (**generalisationOf**(**c**, **c'**) $\wedge$ $\exists$ $\mathbf{end}_1 = (\mathbf{assoc}, \mathbf{assocType_{2,1}})$ $\exists$ $\mathbf{end}_2 = (\mathbf{assoc}, \mathbf{assocType_{2,2}})$: ($\mathbf{assocType_{2,1}}(\mathbf{assoc}) = \mathbf{c} \wedge \mathbf{assocType_{2,2}}(\mathbf{assoc}) = \mathbf{c}' \wedge \mathbf{isComposite}(\mathbf{end}_1) \wedge 0 \notin \mathbf{multiplicity}(\mathbf{end}_2)$)).

Example 22 *The UML model consisting of the class diagram shown in Figure 3.4, contains this inconsistency. The class* ATM *which is in this case the root of a class hierarchy (which is not mandatory) is composed of at least 1 instance of the class* WithdrawPrintingATM. *This class inherits from the classes* WithDrawATM *and* PrintingATM *which are subclasses of the class* ATM. *The asymmetric property of the composition relationship and the fact that this relation is specified between a superclass and a subclass, where the superclass is composed of at least one subclass, and that the composition instances form a forest, causes an infinite chain of instances.*

This inconsistency can have many causes. The generalisation relationship can be superfluous, or perhaps the composition restriction on the association is redundant or the multiplicity of the composition relationship is too strong.

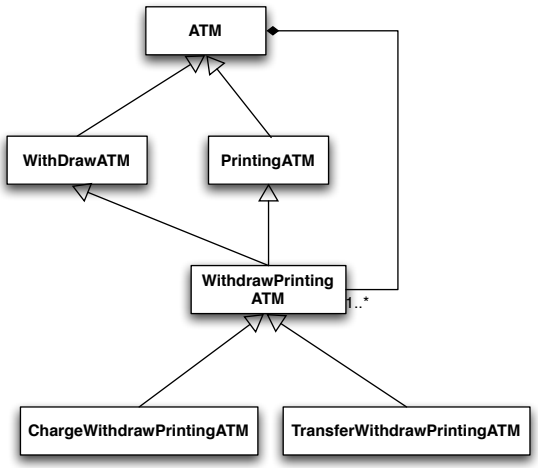

Figure 3.4: Example of an *Inherited Cyclic Composition Inconsistency*.

3.3.2 Dangling Type Reference

Dangling type reference occurs when a parameter's or attribute's type refers to a class that does not exist in the model.

The reasons for this inconsistency to occur are twofold. The type of the attribute or the parameter involved, may have been removed or the type is not yet included in the model. To remove this type of inconsistency from the UML model, the type can be added to the UML model involved.

Involved UML Model Elements:*Property*, *Operation*, *Parameter*, *Model* and *Type*.

Definition 40 *$\mathcal{M}$ suffers from a* **dangling type reference** *if and only if one of the following conditions hold:*

1. $\exists$ **att** $\in$ $\mathbf{Att}_{\mathcal{C}_\mathcal{M}}$: **type**(**att**) $\notin$ $(\mathcal{C}_\mathcal{M} \cup \mathbf{PT})$.
2. $\exists$ **op** $= (n, c, (p_1, \ldots, p_k), (r_1, \ldots, r_m)) \in \mathbf{Op}_{\mathcal{C}_\mathcal{M}} : (\exists i \in \{1, \ldots k\} : p_i \notin (\mathcal{C}_\mathcal{M} \cup \mathbf{PT})$ $\vee\ \exists j \in \{1, \ldots m\} : r_j \notin (\mathcal{C}_\mathcal{M} \cup \mathbf{PT}))$.

Example 23 *Consider the UML model only consisting of the class diagram in Figure 2.6. Several occurrences of this type of inconsistency can be found in the model. Due to the fact that the class* Cash *is not included in the model, the type of the attribute* cash *is not known, also the type of the parameter* amount *of the* dispenseCash *operation of the class* ATM *is not known to this model.*

3.3.3 Connector Specification Missing

Connector specification missing represents a category of inconsistencies. Three kinds of inconsistencies can be differentiated.

- *classless connectable element*: there is a connectable element in a sequence diagram whose base class does not exist in the model.

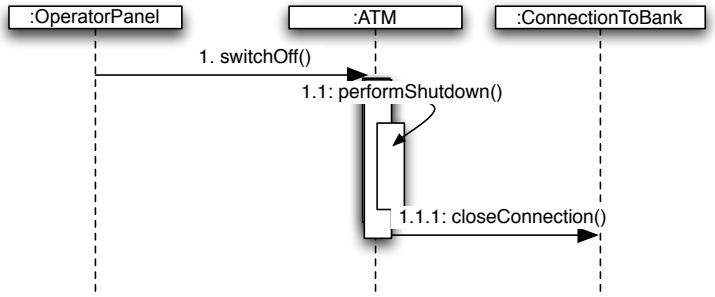

Figure 3.5: Sequence diagram at specification level.

- *dangling connectable feature reference*: a message references a non-existing operation in the corresponding class(es) (and its ancestors).
- *dangling connectable association reference*: a connector is not related to an association or it is related to an association that does not exist between the base classes of the corresponding connectable elements.

Remark that similar inconsistencies also occur on the specification/instance level within or between sequence diagrams at instance level.

Classless Connectable Element

A connectable element can represent a set of non-existing classes due to the fact that these classes are not yet included in the model or due to the fact that the class is deleted, e.g., from a previous version of the model. This inconsistency also occurs if a connectable element is not connected to any class.

Involved UML Model Elements:*ConnectableElement* and *Class*.

Definition 41 *$\mathcal{M}$ suffers from a* **classless connectable element inconsistency** *if and only if* $\exists$ **ConnEl** $\subseteq \mathcal{O}_{\mathcal{M}}$ $\exists$ $o \in$ **ConnEl** $\nexists$ $C \subseteq \mathcal{C}_{\mathcal{M}}$: **instanceOf**(o, C).

Example 24 *In the UML model consisting of the class diagram in Figure 2.6 and the sequence diagram in Figure 3.5, this inconsistency occurs due to the fact that one of the lifelines in the sequence diagram of Figure 3.5 references the class* OperatorPanel *that does not belong to the UML model.*

Dangling Connectable Feature Reference

A connector can represent a non-existing operation due to the fact that this operation is not yet included in the model. In an evolution or refinement context, the operation can be deleted from a previous version of the model under consideration. Another possibility is that the class owning this operation is deleted from the set of classes represented by the corresponding connectable element. An operation can also have moved from one class to another causing this inconsistency.

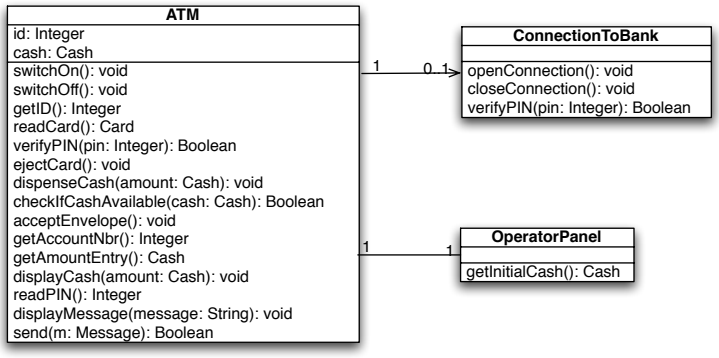

Figure 3.6: Class diagram constituting, together with the sequence diagram of Figure 3.5, a model.

Involved UML Model Elements:*Message*, *Operation*, *Generalization*, *Lifeline*, *EventOccurrence*, *ConnectableElement* and *Class*.

Definition 42 *$\mathcal{M}$ suffers from a* **dangling connectable feature reference inconsistency** *if and only if* $\exists\ C\ \subseteq \mathcal{C}_{\mathcal{M}}\ \exists\ \mathbf{e} = (\mathbf{m}, \mathbf{Cons},$ *"receive"*$) \in \upsilon_{\mathbf{O}_C} \in \delta \in \Delta_{\mathcal{M}}$ *($\upsilon_{\mathbf{O}_C}$ is an SD trace and δ is a sequence diagram)* $\forall c \in C : \mathbf{invoked}(\mathbf{m}) \notin \mathbf{Op}_c^*$.

Example 25 *Consider as an example a UML model consisting of the classes and relationships as shown in the class diagram of Figure 3.6 and a sequence diagram of Figure 3.5. The operation* performShutdown() *sent to objects of type* ATM *is not known to these objects which causes an inconsistency.*

Dangling Connectable Association Reference

The *dangling connectable association reference* occurs if there is no association typing the connector, or if the association typing the connector is defined between at least one class not contained in the MUC, or if a connector is typed by an association that does not exist between the classes of the corresponding objects (nor between any of the ancestors of these classes).

Following the UML metamodel a connector may be typed by an *Association*. There are common modelling situations where a connector is not an instance of an association. This is the case if the connector represents the ability of the connected instances to communicate because their identities are known by virtue of being passed as parameters, their identities are held in variables, or because the communicating instances are the same object. Genova *et al.* [GLF03] proposes to follow the rule that *every connector is typed by an association*. It is however, not necessary that every association appears in a class diagram. It is sufficient that the association is represented in the underlying model.

Involved UML Model Elements:*Association*, *Connector*, *Class* and *Model*.

Definition 43 *$\mathcal{M}$ suffers from a* **dangling connectable association reference inconsistency** *if and only if one of the following conditions hold:*

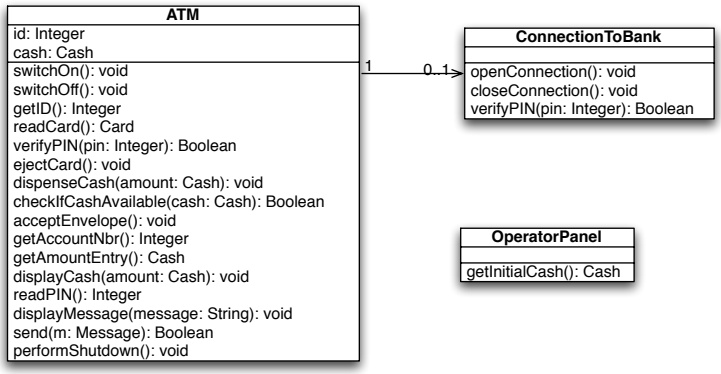

Figure 3.7: Class diagram constituting, together with the sequence diagram of Figure 3.5, a model.

1. $\exists$ **Connector** $\in$ **Connectors**$_{\mathcal{M}}$ $\nexists$ **assoc** : **connectorType**(**Connector**) = **assoc**
2. $\exists$ **Connector** $\in$ **Connectors**$_{\mathcal{M}}$ $\exists$ **assoc** = **connectorType**(**Connector**) $\exists\, c \notin \mathcal{C}_{\mathcal{M}}$: **assoc** $\in$ **participates**(c)
3. $\exists$ **Connector** $\in \mathcal{P}(\mathbf{O}_{C_1} \times \ldots \times \mathbf{O}_{C_n}) \in$ **Connectors**$_{\mathcal{M}}$ $\exists$ **assoc** = **connectorType**(**Connector**) $\forall\ c \in C_i$: (**assoc** $\notin$ **participates**(c) $\wedge$ $\forall$**generalisationOf**(c', c) : **assoc** $\notin$ **participates**(c')), *with* $i \in \{1, \ldots, n\}$.

Example 26 *The UML model consisting of the class diagram of Figure 3.7 and of the sequence diagram of Figure 3.5 suffers from this inconsistency. There is no explicit association specified between the class* ATM *and the class* OperationPanel *and we also assume that there is no association implicitly present in the UML model. This leads to the specified inconsistency, because in the sequence diagram there is a connector connecting instances of the class* ATM *and instances of the class* OperatorPanel.

3.4 Structural Specification/Instance Inconsistencies

Between specifications and instances, we can have the problem of *instance specification missing*. This means that a model element specification does not exist in a model, as it has either been removed from a diagram or not included yet.

3.4.1 Instance Specification Missing

Instance specification missing inconsistencies occur between UML model elements contained in class diagrams and model elements contained in sequence diagrams at instance level and in state diagrams. We observed 4 different occurrences of this inconsistency category. Three occurrences are very similar to the inconsistencies defined by the *Connector specification missing inconsistency* category.

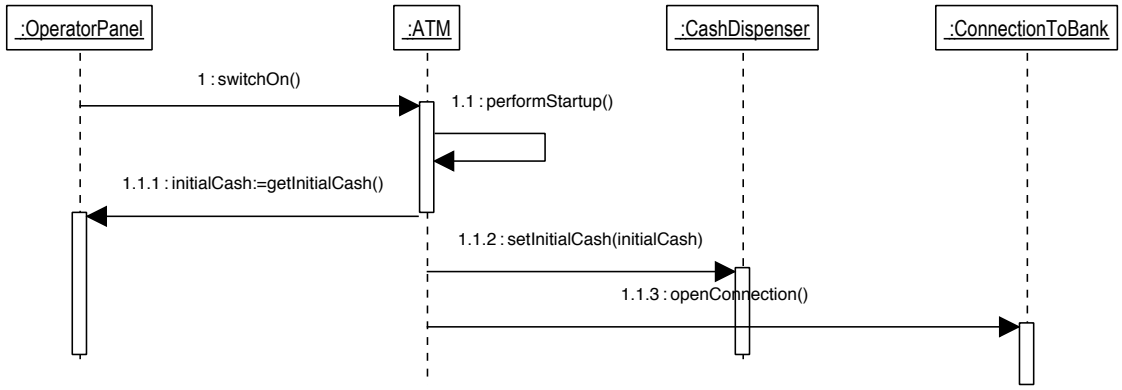

Figure 3.8: UML sequence diagram for the startup of an ATM.

Classless Instance

A *classless instance* inconsistency occurs if an object in a sequence diagram, specified on the instance level, is the instance of a class that does not exist in the UML MUC or the object's class is not specified.

This inconsistency can be caused due to the fact that the original class is not yet included in a model. Another cause can be the deletion of the class from the model in a refactoring or refinement step or the change of type of the object referred to in the sequence diagram.

Involved UML Model Elements: *InstanceSpecification* and *Class*.

Definition 44 *$\mathcal{M}$ suffers from a* **classless instance inconsistency** *if and only if* $\exists o \in \mathcal{O}_{\mathcal{M}} \nexists\, C \subseteq \mathcal{C}_{\mathcal{M}}$*:* $\mathbf{instanceOf}(o, C)$.

Example 27 *Consider the sequence diagram shown in Figure 3.8. This sequence diagram shows a scenario for the startup of an ATM. An operator switches on the ATM, causing the initialisation of the ATM. This initialisation includes setting the initial cash and opening the connection to the bank. Suppose this sequence diagram is adapted to the requirement that a log entry must be made when the ATM is started up. This results in the sequence diagram of Figure 3.9. The UML model consisting of the class diagrams in Figure 2.6 and 2.8, and the sequence diagram of Figure 3.9 contains different occurrences of inconsistencies of the category* instance specification missing.

The fact that the class Log *is not included yet in this model results in a* classless instance *inconsistency.*

Classless Protocol State Machine

This inconsistency arises when a protocol state machine is associated to a class not present in the UML MUC.

Involved UML Model Elements: *ProtocolStateMachine*, *Class* and *Model*.

Definition 45 *$\mathcal{M}$ suffers from a* **classless protocol state machine inconsistency** *if and only if* $\exists \pi_c \in \Pi_{\mathcal{M}} : c \notin \mathcal{C}_{\mathcal{M}}$

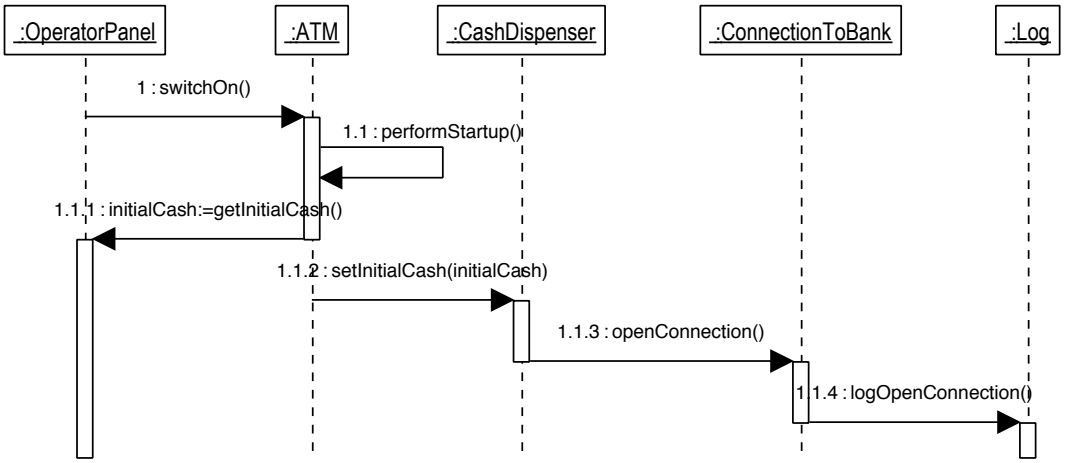

Figure 3.9: Sequence diagram for the startup of an ATM with logging.

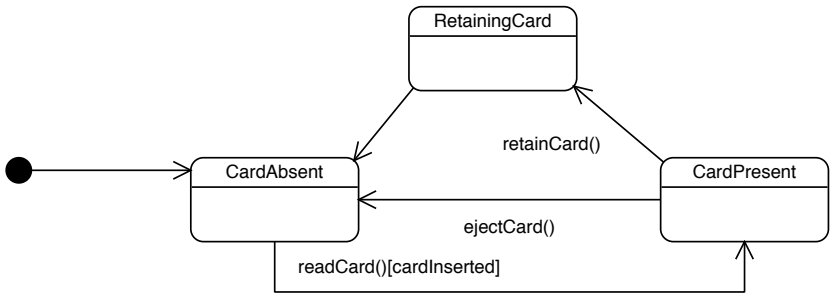

Figure 3.10: Protocol state machine for the class *CardReader*.

Example 28 *Consider again the model consisting of the diagrams shown in Figure 2.6 and 2.8, and 3.9. In a next step the model is more detailed by adding a protocol state machine for the class* CardReader *as shown in Figure 3.10. However, the class* CardReader *is not part of the model. Remark that in most UML CASE tools it is not possible to show the relation between the PSM and its class in the diagram but in another interface component of the CASE tool.*

Dangling Feature Reference

This inconsistency arises when a message in a sequence diagram or a protocol transition in a PSM references an operation that does not exist in the corresponding class nor in any of its ancestors. The inconsistency can also arise when a pre- or postcondition specified on a transition in a PSM diagram references a property or operation that does not exist in the corresponding class nor in one of its ancestors.

These types of inconsistencies occur as horizontal inconsistencies when a feature is referenced but not yet included in the MUC. These types of inconsistencies can also be caused

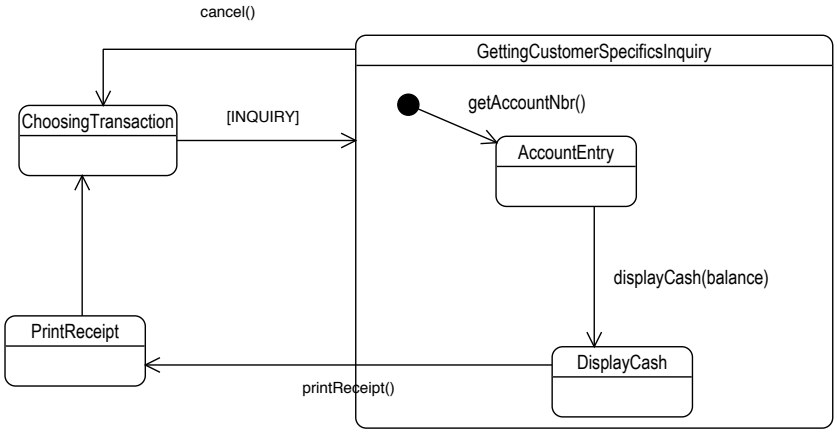

Figure 3.11: Specific states and transitions for an inquiry.

by a refinement or refactoring step by deleting the feature or moving the feature around.

Involved UML Model Elements: *Message*, *Operation*, *ProtocolTransition*, *ProtocolStateMachine*, *Lifeline*, *InstanceSpecification*, *Model*, *Precondition*, *Postcondition*, *Feature*, *Class* and *Generalization*.

Definition 46 $\mathcal{M}$ *suffers from a* **dangling feature reference inconsistency** *if and only if one of the four following conditions applies:*

1. $\exists\ \pi_c = (S_c, T_c, L_c, \rho_c, \Lambda_c) \in \Pi_{\mathcal{M}}\ \exists \tau = (op, g, h) \in L_c : op \notin \mathbf{Op}_c^*$.
2. $\exists\ \delta\ \in \Delta_{\mathcal{M}}\ \exists\ C\ \in \mathcal{C}_{\mathcal{M}}\ \exists\ \{o\}\ \in \mathbf{contained}(\delta, C)\ \exists\ \mathbf{e} = (\mathbf{m}, \mathbf{Cons},\ \textit{"receive"}) \in \upsilon_{\{o\}}$ $\forall c \in C : \mathbf{invoked}(\mathbf{m}) \notin \mathbf{Op}_c^*$.
3. $\exists\ pre \in \mathbf{Pre}_{\mathcal{M}}\ \exists\ el \in \mathbf{constrainedElements}(pre)\ \forall c \in \mathcal{C}_{\mathcal{M}} : el \notin (\mathbf{Prop}_c^* \cup \mathbf{Op}_c^*)$.
4. $\exists\ post \in \mathbf{Post}_{\mathcal{M}}\ \exists\ el \in \mathbf{constrainedElements}(post)\ \forall c \in \mathcal{C}_{\mathcal{M}} : el \notin (\mathbf{Prop}_c^* \cup \mathbf{Op}_c^*)$.

Example 29 *An occurrence of this inconsistency is found in the model consisting of the elements represented by the diagrams in Figure 2.6, Figure 2.4, Figure 2.8, and the protocol state machines defined in the Figure 2.14, Figure 3.11, Figure 3.12 and Figure 3.13. Figure 3.11 shows the states and transitions starting from the* ChoosingTransaction *state in the case of an inquiry. Figure 3.12 and Figure 3.13 show the states and transitions again starting from the* ChoosingTransaction *state in the case of a transfer and a deposit, respectively.*

The operation cancel() *called on transitions on all four protocol state machine diagrams is not defined in the class* ATM.

Another occurrence of this inconsistency occurs in the model consisting of the elements included in the class diagram in Figure 2.4 and the protocol state machine diagram in Figure 3.10. In the protocol state machine the precondition cardInserted *is used on the operation* readCard(). *However, the element* cardInserted *is not defined in the model.*

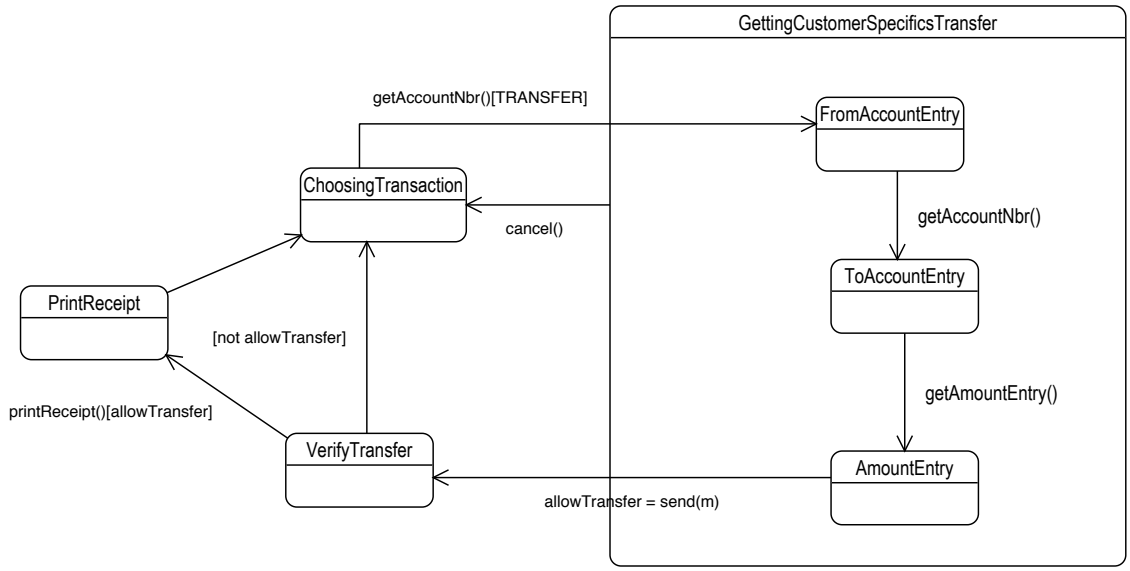

Figure 3.12: Specific states and transitions for a transfer.

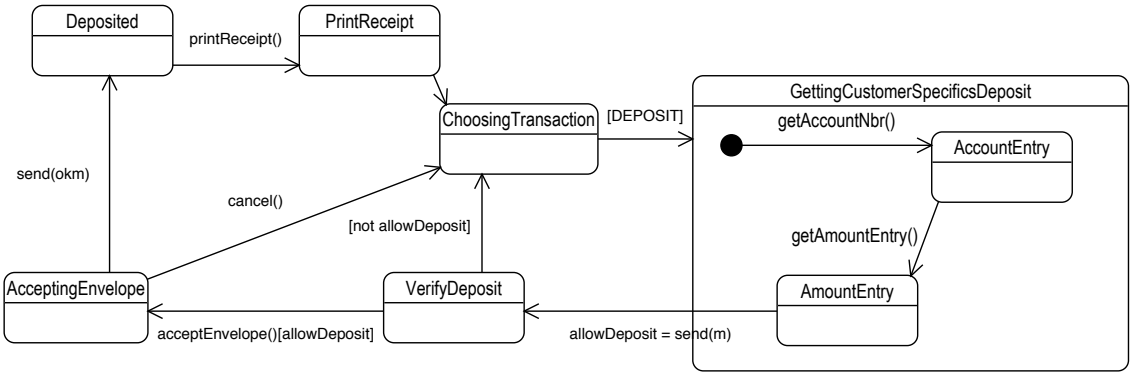

Figure 3.13: Specific states and transitions for a deposit.

Dangling Association Reference

A *dangling association reference* inconsistency occurs if a link is not typed by any association or if a link is typed by an association between classes not belonging to the model. This inconsistency can also occur if a connector in a sequence diagram (at instance level) is related to an association that does not exist between the classes of the corresponding objects (nor between any of the ancestors of these classes).

Involved UML Model Elements: *Association*, *Connector*, *InstanceSpecfication* and *Class*.

Definition 47 *$\mathcal{M}$ suffers from a* **dangling association reference inconsistency** *if and only if one of the following conditions hold:*

1. $\exists\ \mathbf{l} = \{(o_1, \ldots, o_n)\} \in \mathbf{Connectors}_{\mathcal{M}} \nexists\ \mathbf{assoc} = \mathbf{connectorType}(\mathbf{l})$,
2. $\exists\ \mathbf{l} = \{(o_1, \ldots, o_n)\} \in \mathbf{Connectors}_{\mathcal{M}}\ \exists\ \mathbf{assoc} = \mathbf{connectorType}(\mathbf{l})\ \exists c \notin \mathcal{C}_{\mathcal{M}} : \mathbf{assoc} \in \mathbf{participates}(c)$,
3. $\exists\ \mathbf{l} = \{(o_1, \ldots, o_n)\} \in \mathcal{P}(\mathbf{O}_{\{c_1\}} \times \ldots \times \mathbf{O}_{\{c_n\}}) \cap \mathbf{Connectors}_{\mathcal{M}}\ \exists\ \mathbf{assoc} = \mathbf{connectorType}(\mathbf{l})\ \exists\ c_i \in \{c_1, \ldots, c_n\} : (\mathbf{assoc} \notin \mathbf{participates}(c_i) \wedge \forall \mathbf{generalisationOf}(c', c_i) : \mathbf{assoc} \notin \mathbf{participates}(c'))$.

Example 30 *The model composed of the elements represented in the class diagram of Figure 3.14 and in the sequence diagram of Figure 3.15 contains a dangling association reference. Figure 3.15 shows a possible scenario for a transfer transaction.*

In this scenario, there is a message sent from an object of type Transfer *to an object of type* Card*. We assume that there are no implicit associations in the model, i.e., all the associations existing in the model are shown in the class diagram in Figure 3.14. There is no corresponding association explicit between the classes* Transfer *and* Card *and there is also no association between* Transaction *and* Card*.*

3.5 Structural Instance Inconsistencies

At the instance level, we identified the potential problem of *disconnected models*. This problem is related to the topography of the diagrams, specifically, when it contains parts that are disconnected from each other. For example, a state or transition may have been deleted or omitted in a state diagram, resulting in a set of states that are not reachable from the initial state. As another example, an object or connector may have been deleted or omitted in a sequence diagram, resulting in a set of objects that are unreachable.

3.5.1 Disconnected Model

In the case of a PSM this inconsistency arises when a particular state is not reachable. However, there are different forms of reachability of a state. The simplest form implies that a state can be reached starting from the start state by following the transitions. In a more complex form reachability consists also of verifying if the several operations specified on the different transitions exist. This last condition is part of the above discussed *dangling*

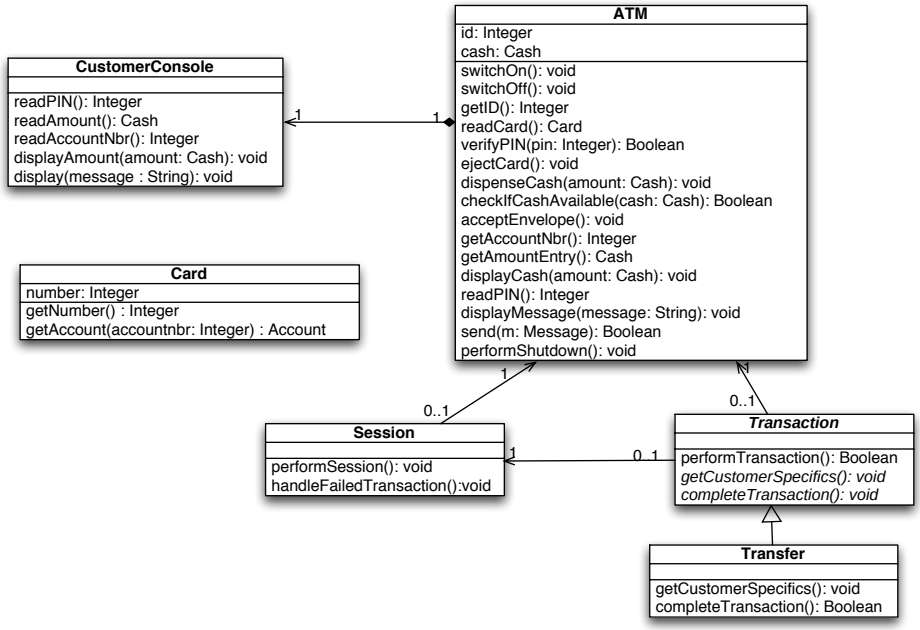

Figure 3.14: Class diagram constituting, together with the sequence diagram of Figure 3.15, a model.

feature reference. The most complex form of reachability is to verify if the operations can be called by the objects in a certain configuration. To do this, the information found in PSM and class diagrams is not enough. We need to know how the application is initialised and which instances are alive. This is beyond the scope of the inconsistency we want to describe here.

This inconsistency can also occur in sequence diagrams. Sequence diagrams suffer from this inconsistency when there is an object that does not receive any message. A possible exception on this is the object that sends the first message.

Again this inconsistency can occur as a horizontal inconsistency which implies that the designer has created a disconnected model, e.g., by forgetting to include a transition in a PSM diagram or by the omission of a connector or the sending of a message in a sequence diagram.

Involved UML Model Elements: in case of a PSM: *ProtocolStateMachine*, *State* and *ProtocolTransition*. For sequence diagrams, these model elements include *Lifeline*, *EventOccurrence* and *Message*.

Definition 48 $\mathcal{M}$ *suffers from a* **disconnected model inconsistency** *if and only if*

- $\exists \pi_c = (S_c, T_c, L_c, \rho_c, \Lambda_c)\ \exists s \subset S_c$ *(s is a state configuration)* $\nexists\ \mu = < \tau_1, \ldots, \tau_n >$: $(\sigma_1 = \{\rho_c\} \wedge \forall i \in \{1, \ldots, n-1\} : ((\sigma_i, \tau_i, \sigma_{i+1}) \wedge \sigma_i \subset S_c \wedge s = \sigma_{n+1}))$.

- $\exists C \subseteq \mathcal{C}_\mathcal{M}\ \exists \delta \in \Delta_\mathcal{M} : (\mathbf{contained}(\delta, C) \neq \emptyset \wedge \mathbf{E}_{\delta,C} = \emptyset)$.

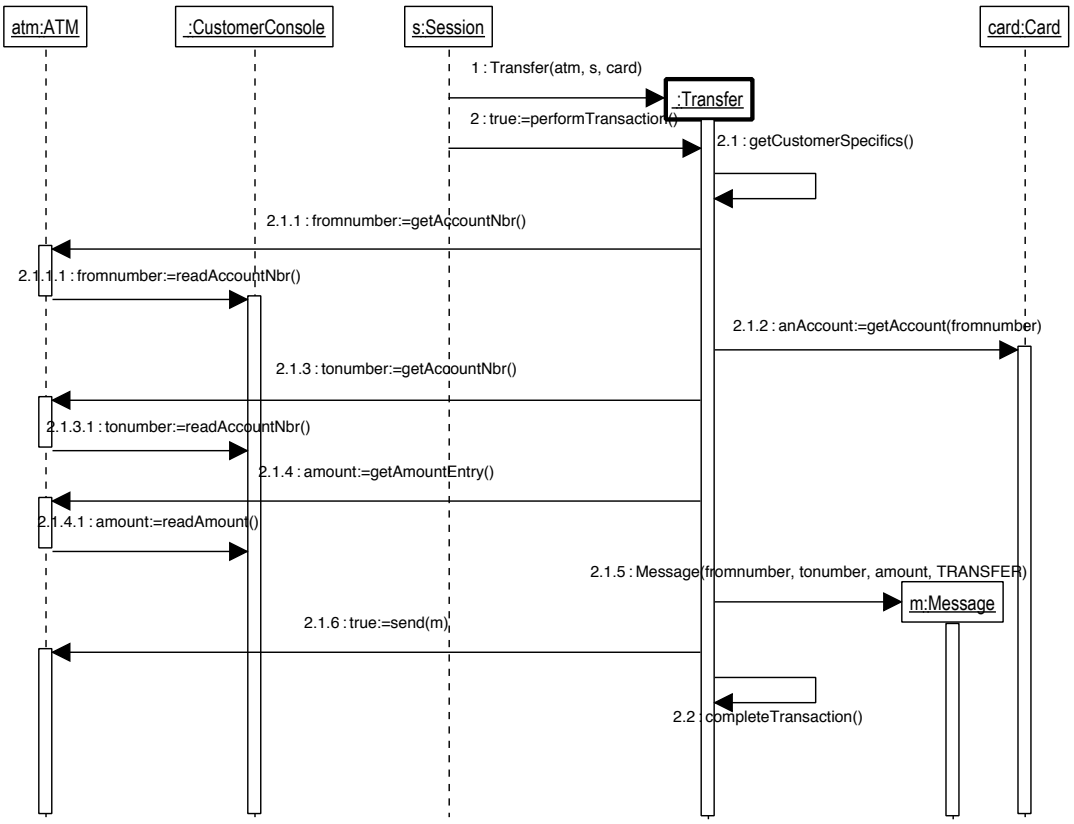

Figure 3.15: UML sequence diagram for a transfer transaction.

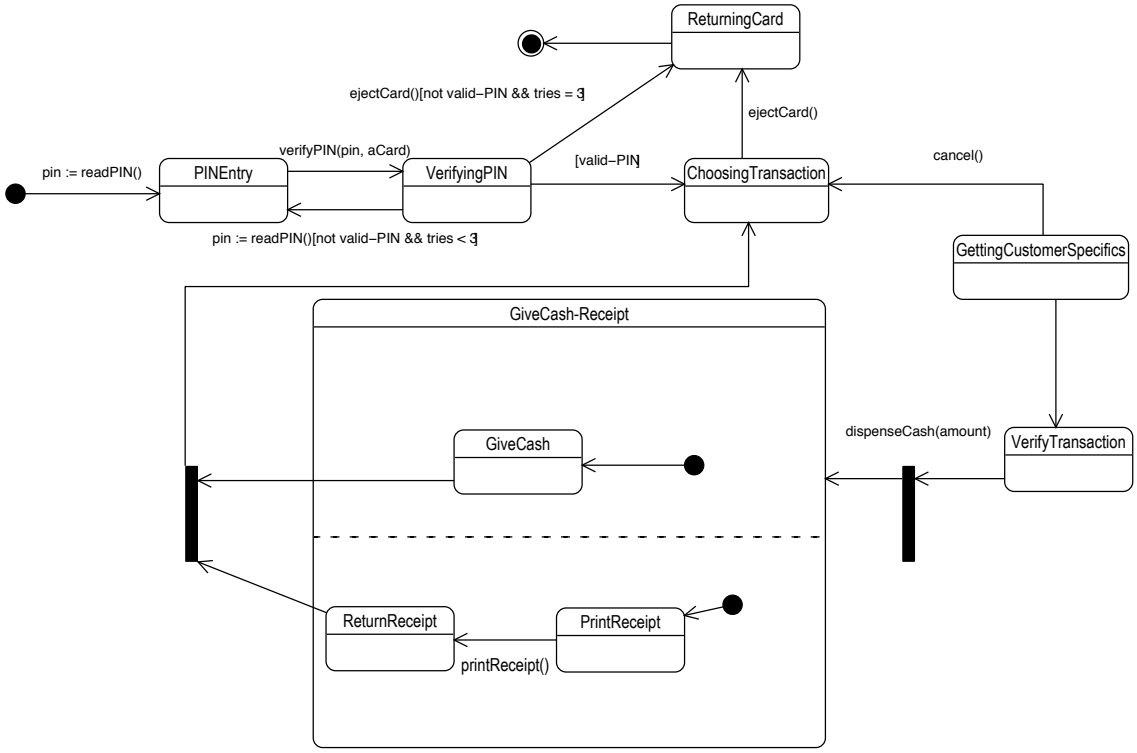

Figure 3.16: Protocol state machine in which the state *GettingCustomerSpecifics* is not reachable.

Example 31 *In the PSM shown in Figure 3.16, there is no call sequence that leads to the state* GettingCustomerSpecifics*. In this case, also the states* VerifyTransaction *and* GiveCash-Receipt *are not reachable.*

3.6 Behaviour and Behaviour Inheritance

Before studying the different categories of behavioural inconsistencies, the notions of behaviour and behaviour inheritance in object-oriented analysis and design and especially in the context of UML, are considered.

In object-oriented systems, inheritance is a central issue. Classes are organised in hierarchies in which subclasses inherit and specialise the behaviour of the superclasses and this mainly to enable reuse. However, these hierarchies can also be used to specify *how* the behaviour of a certain class must be specialised. The questions arising in this context are: What does it mean to specialise behaviour? How can this specialisation be specified and how to enforce restrictions on it?

Following the spirit of the previous chapter, we assume that the *specification of the behaviour of a class* is defined as a combination of its protocol state machine and all sequence diagrams in which instances of the considered class are involved.

In contrast to programming languages, object-oriented design languages provide a high-

level view on the behaviour of an application. The behaviour modelled consists of the specification of how objects interact and on how objects evolve over their life time which is often referred to as *object life cycle*. Different languages exist to specify behaviour at an analysis and design level. The UML incorporates state machines, sequence diagrams, communication diagrams, etc.. Other examples of languages are Message Sequence Charts (MSCs), Petri Nets, Harel statecharts, etc..

To restrict the way the behaviour of a subclass should specialise the behaviour of a superclass in a class hierarchy, so-called *behaviour inheritance consistencies* can be used. Different notions for specialising the behaviour of a class are proposed in literature (for example, [EE95], [SS00], [SS02], [HK99], [van02a]).

Engels *et al.* [EE95] were the first to distinguish - what they call - *observable and invocable consistency* using homomorphisms on state diagrams. *Observable consistency* means that *each sequence of calls which is observable with respect to a subclass must result (under projection of the methods known) in an observable sequence of its corresponding superclass. If a subclass reacts to the invocation of an operation op, where op is also known to the superclass, this reaction must also be reflected in the superclass behaviour specification.* In terms of UML PSMs, this can be rephrased as "after hiding all new operation invocations, each sequence of the subclass state diagram should be contained in the set of sequences of the superclass state diagram".

Invocable consistency on the other hand, means that *any sequence of operations invocable on the superclass can also be invoked on the subclass.* This notion of behaviour inheritance consistency is based on the substitutability principle requiring that an object of subclass B of class A can be used where an object of class A is expected. In terms of UML PSMs, each call sequence of the superclass state diagram should be contained in the set of call sequences of the state diagram for the subclass.

Similar criteria to the ones found in [EE95], for inheritance of object life cycles based on Petri nets are discussed in [SS00], [SS02] and [van02a].

In UML 2.0 interactions and state machines can be generalised. The UML 2.0 specifications define how an interaction and state machine can be specialised. Only an *extension* policy is provided, i.e., features can only be added. Specialising an interaction simply means to add more traces to those of the original. We call the specialised interaction, the *child interaction* and the interaction that is being specialised, the *parent interaction.*

A protocol state machine can also be generalised. A specialised state machine is an extension of the general state machine. This means that regions, vertices and transitions may be added. *A simple state can be extended to a composite state, by adding one or more regions. A composite state can be extended by either extending inherited regions or by adding regions. A region is extended by adding states and transitions and by extending states and transitions* [Obj04e].

In the next sections, we express different kinds of *behaviour inheritance consistencies* in the context of UML 2.0 elements represented in protocol state machine diagrams and sequence diagrams at instance as well as specification level. Rather than inventing our own definitions of behaviour inheritance consistencies, we will rely on two variants, observable and invocable consistency, although it is very well possible that other useful definitions of inheritance consistency exist.

In these sections *consistency* definitions are given (except for Section 3.8.3) as opposed to the definitions of *inconsistencies* in the previous sections. The definitions of the behavioural

consistencies are more readable than the corresponding inconsistencies. A behavioural inconsistency is the negation of a behavioural consistency.

3.7 Behavioural Specification Inconsistencies

At this level *interaction inconsistencies* are identified. Interaction inconsistencies comprise *invocation interaction inconsistency* and *observation interaction inconsistency*. These inconsistencies occur between sequence diagrams at specification level.

3.7.1 Invocation/Observation Interaction Inconsistencies

Interaction *consistencies* specify inheritance restrictions between two interactions. These restrictions are specified on a set of traces received by a set of connectable elements.

An *invocation interaction inconsistency* arises when the set of receiving traces of a connectable element in the parent interaction is not a subset of the set of receiving traces of the corresponding connectable element in the child interaction. An *observation interaction inconsistency* arises when, after hiding the new messages associated to connectable elements belonging to a set of classes, the set of receiving traces of these connectable elements belonging to the child interaction is not a subset of the set of receiving traces of the corresponding connectable elements in the parent interaction.

Involved UML Model Elements: *Operation*, *Constraint*, *Message* and *EventOccurrence*.

Definition 49 *Given* $C, C' \subseteq \mathcal{C}_{\mathcal{M}}$, *where* $\forall c' \in C' \; \exists c \in C : c' = c \vee$ **generalisationOf**(c, c'), *and given the sequence diagrams* δ *and* δ'*:*

- *A SD* δ' *is* **invocation interaction consistent** *with a SD* δ *if and only if,* $\forall O \in$ **contained**$(\delta, C) \; \forall \upsilon_O \in \delta : (\exists O' \in$ **contained**$(\delta', C') \Rightarrow \exists \; \upsilon'_{O'} \in \delta' : =_{\upsilon}(\upsilon_O/^{rec}, \upsilon'_{O'}/^{rec}))$.
- *A SD* δ' *is* **observation interaction consistent** *with a SD* δ *if and only if,* $\forall O' \in$ **contained**$(\delta', C') \; \forall \; \upsilon' = \upsilon_{O'} \in \delta' : (\exists O \in$ **contained**$(\delta, C) \Rightarrow \exists \upsilon''_O \in \delta : =_{\upsilon}(\upsilon''_O/^{rec}, \upsilon'_{E_{\delta,C}}/^{rec}))$.

Example 32 *Consider the sequence diagrams shown in Figure 3.17 and in Figure 3.18. The first sequence diagram models a generic interaction. It is generic because it is an interaction on conceptual level, i.e., the types that are included in the sequence diagrams are roles that will be played by different concrete classes and also the operations called will be refined by different concrete classes. The class* Device *is rather a concept than a class that can represent an* ATM *class in an ATM simulation application or a* Phone *class in a simulation of a Phone banking application.*

The sequence diagram in Figure 3.18 models an interaction at specification level but now for a concrete application. The Phone *class can be regarded as a subclass or subconcept of* Device *and* PhoneSession *is a subclass of* Session *and* PhoneScreen *is a subclass of* Console.

The sequence diagrams are observation consistent but not invocation consistent with $C =$ {Console} *and* $C' =$ {PhoneScreen}. *These diagrams are observation consistent because*

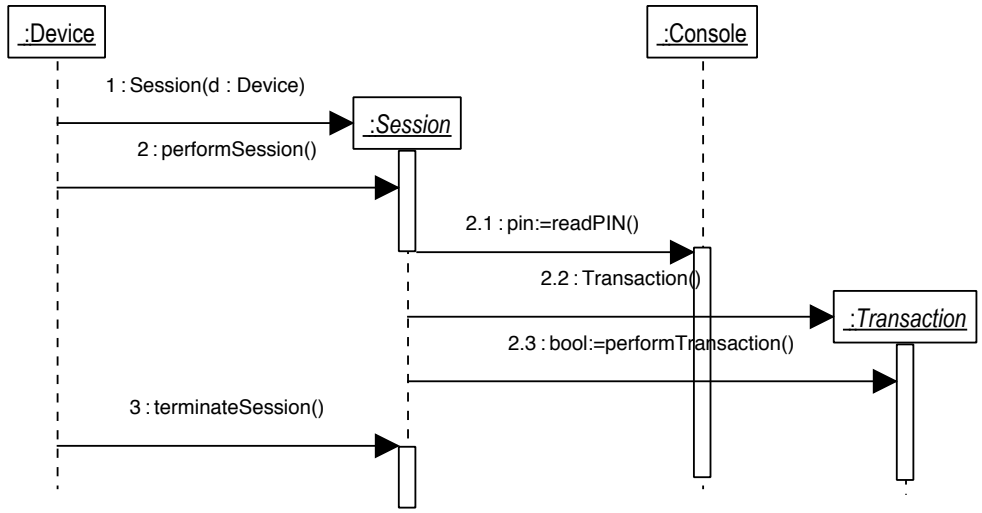

Figure 3.17: Sequence diagram at specification level, generic parent interaction.

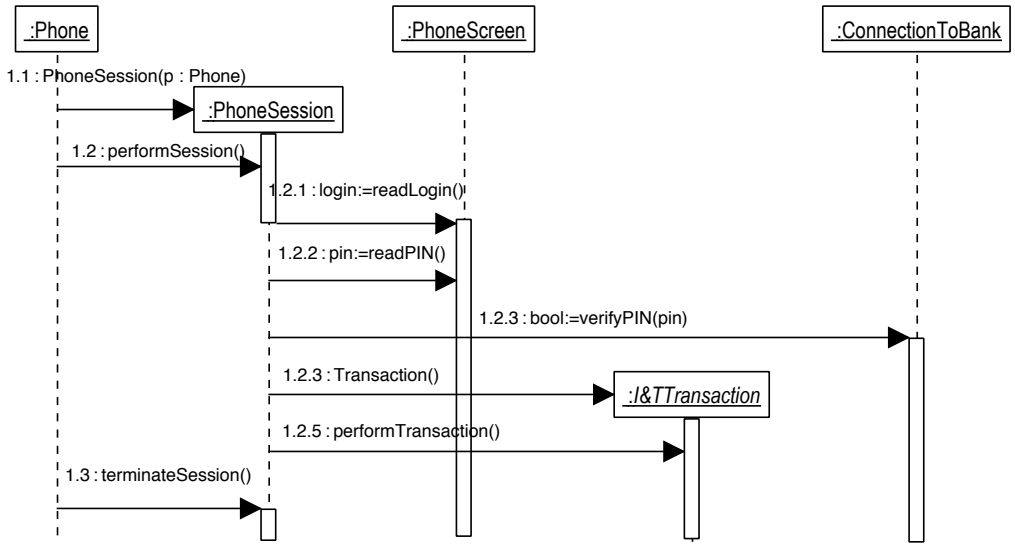

Figure 3.18: Sequence diagram at specification level, concrete child interaction.

after hiding the call to readLogin(), *the trace defined on* Console *and the trace defined on* PhoneScreen *are equal. The trace defined on* Console *cannot be found on* PhoneScreen *resulting in an invocation interaction inconsistency.*

3.8 Behavioural Specification/Instance Inconsistencies

Three kinds of inconsistencies can be differentiated at this level.

- The first kind of inconsistency differentiates between *invocation behaviour inconsistency* and *observation behaviour inconsistency*. These inconsistencies occur between a PSM and sequence diagrams at specification level. These inconsistencies are similar to the ones defined in the previous section.
- Another kind of inconsistency identified at this level is called *specification incompatibility*. This inconsistency arises when instances do not comply with the specifications that characterise them, and it is defined between model elements belonging to class diagrams and model elements belonging to sequence diagrams at instance level.
- A last kind of identified inconsistency is called a *specification behaviour incompatibility*. This inconsistency is similar to the *instance behaviour incompatibility* (see Section 3.9).

Before defining the different consistencies, we need to define an auxiliary property between a SD trace and a PSM call sequence. This property is required if a correspondence between an SD trace, defined on objects of class(es), and a call sequence, defined in the corresponding PSM, is defined. Call sequences can contain labels with an empty operation. Such labels do not correspond to any message in a sequence diagram and should be skipped but the corresponding guard must be taken into account by the next label containing an operation.

Definition 50 *Given a SD trace* $\upsilon =< e_1, \ldots, e_n >$ *and a call sequence* $\mu =< \tau_1, \ldots, \tau_m >$ *defined for a PSM* π*,*
υ *and* μ **are in strict sequence** *if and only if,*
$(\exists\, j \in \{1, \ldots, m\}\ \exists\, i \in \{1, \ldots, n\} : (\tau_j = label(e_i) \wedge \exists\, u \geq 1 : (\tau_{j+u} = label(e_{i+1}) \wedge e_{i+1} = (\mathbf{m}, \mathbf{Cons}, \mathbf{direction})) \Rightarrow \forall\, r \in \{1, \ldots, u-1\} : \tau_r = (\epsilon, g_r, \{\}) \wedge \bigcup_{j \leq r \leq j+u} g_r \subseteq \mathbf{Cons})$.

3.8.1 Invocation/Observation Behaviour Inconsistencies

Involved UML Model Elements: *ProtocolTransition*, *ProtocolStatemachine*, *Operation*, *Constraint*, *State*, *Region*, *Message* and *EventOccurrence*.

Definition 51

- *Given a PSM* $\pi_{c'} = (S', T', L', \rho', \Lambda')$ *and a sequence diagram* δ *and* $C \subseteq \mathcal{C}_{\mathcal{M}}$ *such that* $\exists\ c\ \in\ C$: **generalisationOf**(c, c')*:*

  A PSM $\pi_{c'}$ *is* **invocation behaviour consistent** *with a SD* δ *if and only if,*
  $\forall O \in$ **contained**$(\delta, C)\ \forall\ \upsilon_O{}^{/rec} =< e_1\ \ldots\ e_n >)$ *(with* $\upsilon_O \in \delta)\ \exists\ \mu_{c'} =< \tau_1 \ldots \tau_m >$:
  $(\forall\ \tau_i \in \mu_{c'} : \tau_i \in L' \wedge m \geq n \wedge \exists\ \sigma :$ **valid**$(\mu_{c'}, \sigma, \pi_{c'}) \wedge \forall i \in \{1, \ldots, n\}\ \exists\ j \in \{i, \ldots, m\} : (\tau_j = (op, g, h) = label(e_i) \wedge (\exists \tau_k = label(e_{i+1}) \in \mu'_c \Rightarrow\ k > j) \wedge \mu_{c'}$, $\upsilon_O{}^{/rec}$ **are in strict sequence**)).

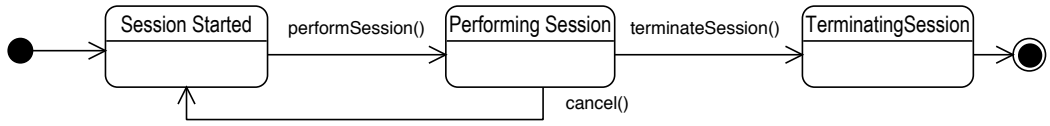

Figure 3.19: State diagram for a *Session* instance.

- *Given a PSM* $\pi_c = (S, T, L, \rho, \Lambda)$ *and a sequence diagram* δ' *and* $C' \subseteq \mathcal{C}_\mathcal{M}$ *such that* $\exists\ c' \in C' :$ **generalisationOf**(c, c')*:*

  A SD δ' *is* **observation behaviour consistent** *with a PSM* π_c *if and only if,* $\forall\ O' \in$ **contained**$(\delta', C')\ \forall\ \upsilon_{O'}/^{rec}_{\mathbf{Op}_c} = < e_1, \ldots, e_n >$ *(with* $\upsilon_{O'} \in \delta')\ \exists\ \mu_c =< \tau_1, \ldots, \tau_m >$: $(\forall\ \tau_i \in \mu_c : \tau_i \in L \wedge m \geq n\ \exists\ \sigma :$ **valid**$(\mu_c, \sigma, \pi_c) \wedge \forall i \in \{1, \ldots, n\}\ \exists\ j \in \{i, \ldots, m\} : \tau_j = (op, g, h) = label(e_i) \wedge (\exists \tau_k = label(e_{i+1}) \in \mu_c \Rightarrow k > j) \wedge \upsilon_{O'}/^{rec}$, μ_c **are in strict sequence**)).

Remark that, in this case, we do not define invocation behaviour consistency between a PSM π_c and a sequence diagram δ' containing instances of a subclass c' of c. Such a definition would imply that all possible scenarios are described by δ', because every trace in the PSM π_c must be also invocable in δ'. This demands completeness of the models which is not always the case, especially in the early phases of the software development cycle. Secondly, we do not define observation behaviour consistency between a PSM $\pi_{c'}$ and an SD δ containing instances of a superclass c of c'. Such a definition would imply that all possible scenarios are described by δ, because every trace in the PSM $\pi_{c'}$ must be observable in δ under projection of the methods known. This demands completeness of the models.

Example 33 *The PSM shown in Figure 3.19 specifies the possible sequences of operations invoked on an instance of the class* Session*. This PSM is observation interaction consistent with the sequence diagram shown in Figure 3.18. This sequence diagram defines one trace of event occurrences received by* PhoneSession *that is a subclass of* Session*. This trace restricted to event occurrences invoking operations of the* Session *class corresponds to a call sequence in the PSM of Figure 3.19.*

3.8.2 Specification Behaviour Incompatibility

The *specification behaviour incompatibility* is defined between a PSM and a sequence diagram. Such an inconsistency occurs when it is not possible to find a call sequence in the protocol state machine of a class c that follows the order established by a receiving SD trace of a set of instances of which at least one object o is an instance of the class c.

The consistency guarantees the compatibility between a PSM for a certain class c and the set of receiving SD traces of instances of the class c.

Involved UML Model Elements: *ProtocolTransition*, *ProtocolStatemachine*, *Operation*, *Constraint*, *State*, *Region*, *Message* and *EventOccurrence*.

Definition 52 *Given* $c \in C \subseteq \mathcal{C}_\mathcal{M}$ *and* $\mathbf{O}_{\{c\}} \subseteq \mathbf{O}_C \subseteq \mathcal{O}_\mathcal{M}$.

A PSM $\pi_c = (S, T, L, \rho, \Lambda)$ *is* **specification behaviour compatible** *with a SD* δ *if and only if,*
$\forall O \in \mathbf{contained}(\delta, C)\ \forall\ v_O/^{rec} =< e_1\ \ldots\ e_n > (with\ v_O \in \delta)\ \exists\ \mu_c =< \tau_1, \ldots, \tau_m >$:
$(m \geq n \wedge \forall\ \tau_i \in \mu_c : \tau_i \in L \wedge \exists\ \sigma : \mathbf{valid}(\mu_c, \sigma, \pi_c) \wedge \forall\ i \in \{1, \ldots, n\}\ \exists\ j \in \{i, \ldots, m\}$:
$(\tau_j = (op, g, h) = label(e_i) \wedge (\exists\ \tau_k = label(e_{i+1}) \in \mu_c \Rightarrow k > j) \wedge v_O/^{rec}$, μ_c **are in strict sequence**)).

Remark that we do not define a specification behaviour incompatibility starting from the call sequences in a protocol state machine because this would imply that all possible specifications of scenario's are described in sequence diagrams.

Example 34 *Consider a UML model consisting of the class diagram in Figure 2.4, and the sequence diagram in Figure 3.17, and the PSM shown in Figure 3.19. The behaviour of the class* Session*, specified by the sequence diagram must be compatible with the behaviour specified by the PSM. This means that every trace contained in the sequence diagram corresponds to a valid call sequence in the PSM. This model is* specification behaviour compatible.

3.8.3 Specification Incompatibility

Three kinds of *specification incompatibility* are recognised, *multiplicity incompatibility*, *navigation incompatibility* and *abstract objects*.

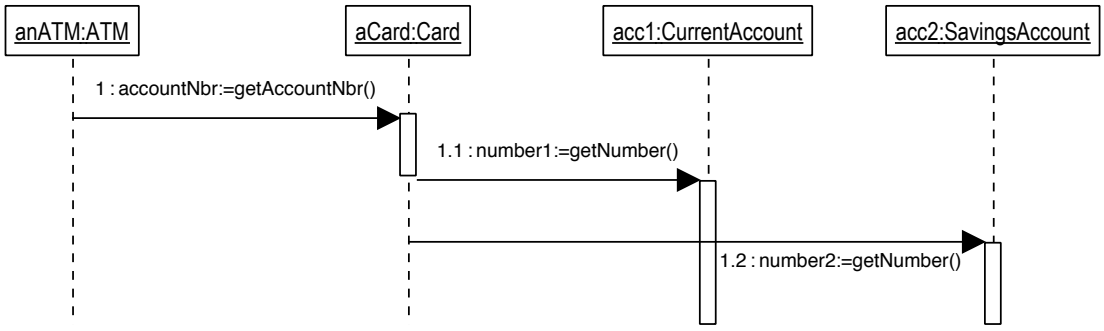

Figure 3.20: Multiplicity error between this diagram and Figure 2.6.

Multiplicity Incompatibility

This kind of specification incompatibility occurs when a connector between connectable elements in a sequence diagram at instance level does not respect the multiplicity restrictions imposed on this connector by the corresponding associations defined (in a class diagram).

Involved UML Model Elements: *Class*, *MultiplicityElement*, *Association*, *Connector*, *Property*, *InstanceSpecification* and *Lifeline*.

Definition 53 $\mathcal{M}$ *suffers from a* **multiplicity incompatibility** *if and only if,*
$\exists o \in \mathbf{O}_{\{c\}} \subseteq \mathcal{O}_{\mathcal{M}}\ \exists\ \mathbf{Connector} \in \mathcal{P}(\{o\} \times \mathbf{O}_{\{c'\}})\ \exists \mathbf{assoc} = \mathbf{connectorType}(\mathbf{Connector})$
$\exists\ \mathbf{end} = (\mathbf{assoc}, \mathbf{assocType}_{2,i})\ :\ \mathbf{assocType}_{2,i}(\mathbf{assoc}) = c' \wedge card(\mathbf{Connector}) \notin \mathbf{multiplicity}(\mathbf{end})$.

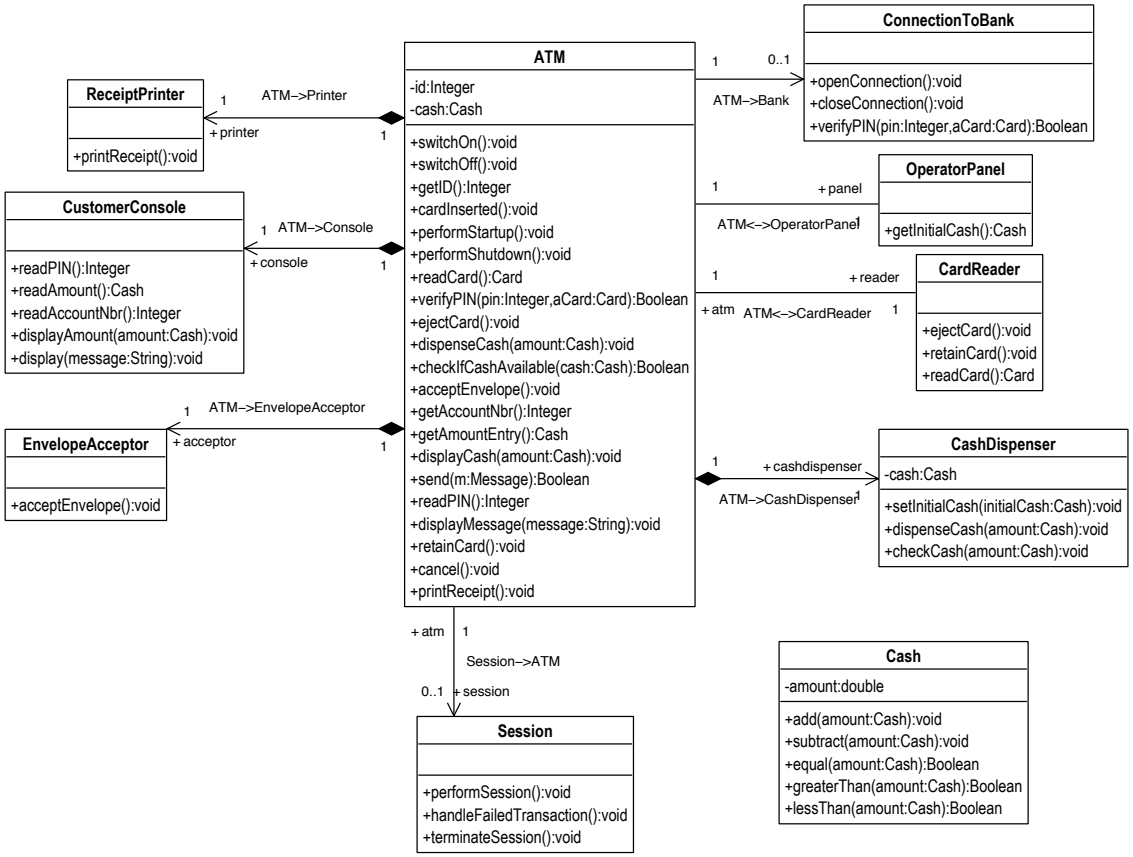

Figure 3.21: Navigation error between this diagram and the sequence diagram in Figure 2.12

card applied on a set returns the cardinality of the set.

Remark that this inconsistency is only defined for binary associations due to the fact that the semantics of multiplicity restrictions on n-ary associations is not defined in UML.

Example 35 *An example of this inconsistency can be found in the model consisting of the sequence diagram shown in Figure 3.20 and the class diagrams shown in Figure 2.6 and Figure 2.8.*

The multiplicity restrictions specified on the association between the class Account *and* Card *indicate that an instance of* Account *must be connected to at least one instance of* Card *and an instance of* Card *can only be connected to one instance of* Account*. In the sequence diagram the operation* getNumber() *is sent over a link, i.e., an instance of the specified association, to two different instances of the* Account *class. As a result an instance of* Card *is connected to two instances of the class* Account *violating the multiplicity defined on the association involved.*

In this example, the multiplicity specified on the association is wrong, a card can be connected to several accounts, e.g., a current-account and some saving-accounts.

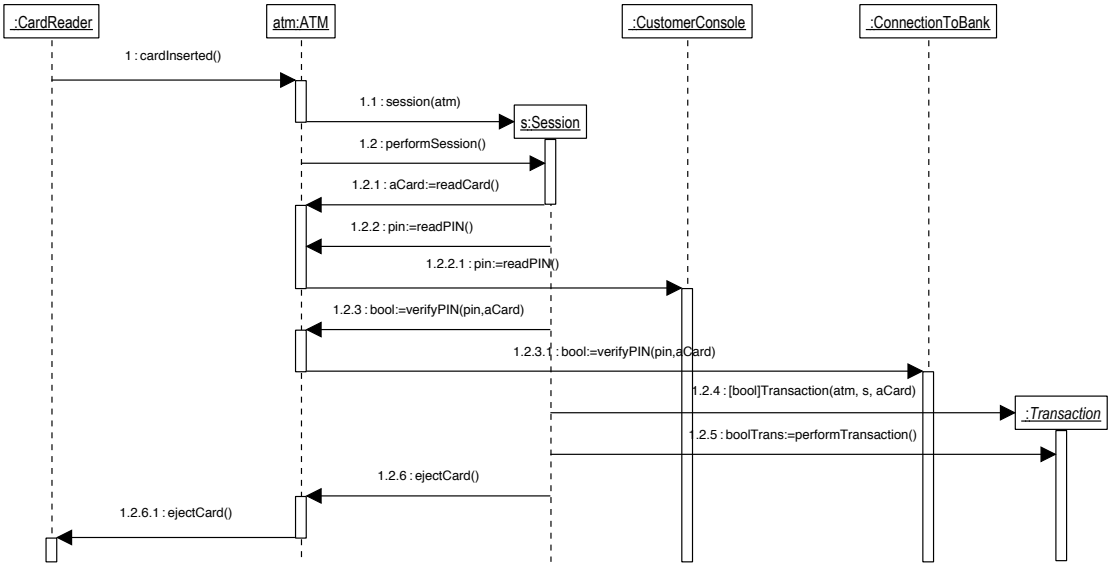

Figure 3.22: UML sequence diagram for a user session.

Navigation Incompatibility

This kind of inconsistency arises when a connector and its connectable element in a sequence diagram does not respect the navigability restrictions imposed on the corresponding class by the specification of the navigability of the relevant association in a class diagram.

Involved UML Model Elements: *Class*, *Association*, *Connector*, *Property*, *InstanceSpecification*, *Eventoccurrence*, *Message* and *Lifeline*.

Definition 54 $\mathcal{M}$ *suffers from a* **navigation incompatibility** *if and only if,*
$\exists$ **Connector** $\in$ **Connectors**$_{\mathcal{M}}$ $\exists$ **assoc** = **connectorType**(**Connector**) $\exists$ **end** = (**assoc**, **assocType**$_{2,i}$) $\exists$ **e** = (**m**, **Cons**, *"receive"*) : **connected**(**e**) $\in$ $\mathbf{O}_{\{\mathbf{assocType}_{2,i}(\mathbf{assoc})\}} \in$ **Connector** $\wedge \neg$ **isNavigable**(**end**).

Example 36 *The model consisting of the sequence diagram shown in Figure 2.12 and the class diagram in Figure 3.8.3 contains an navigation inconsistency. The navigability arrow on the association between the class* ATM *and the class* Session*, indicates that an* ATM *has the responsibility to remember its* Session*s but not vice versa. In the sequence diagram in Figure 2.12 messages are sent from a* Session *to an* ATM*. This is not possible due to the arrow on the association between* ATM *and* Session*.*

Abstract Object

This inconsistency arises when an abstract class defined in the UML model, having no concrete subclasses defined, is instantiated in a sequence diagram. If the abstract class has concrete subclasses, the behaviour specified in the sequence diagram applies to the instances of these subclasses. This corresponds to declaring the abstract class as the static type of

the object, and during execution, dynamically typing it as one of the concrete subclasses of the abstract class.

Involved UML Model Elements: *Class*, *InstanceSpecification*, *Generalization*.

Definition 55 *$\mathcal{M}$ suffers from an* **abstract object inconsistency** *if and only if,* $\exists o \in \mathbf{O}_{\{c\}} \subseteq \mathcal{O}_{\mathcal{M}} \wedge c \in \mathcal{C}_{\mathcal{M}} \wedge \mathbf{isAbstract}(c) \wedge \nexists\, c' \in \mathcal{C}_{\mathcal{M}} : \mathbf{generalisationOf}(c, c')$.

Example 37 *Consider the UML model consisting of the class diagram shown in Figure 2.6 and the sequence diagram shown in Figure 3.22. The sequence diagram shows an interaction creating a* Session *object. This* Session *object controls the reading of the card and the PIN, and the verification of the PIN. If the card can be read and the PIN is correct, then the user can ask for a certain transaction. This results in the creation of a* Transaction *object and the execution of this transaction. If the transaction is finished, the card is ejected. Messages are sent in this sequence diagram to instances of the abstract class* Transaction*. However, there are no concrete subclasses of this class known in the model. This implies an abstract object inconsistency.*

3.9 Behavioural Instance Inconsistencies

At instance level, we identify two different kinds of behavioural inconsistencies: *inheritance inconsistency*, including *invocation* and *observation inheritance inconsistency*, and *instance behaviour incompatibility*.

3.9.1 Invocation Inheritance Inconsistency

Invocation inheritance consistency [EHK01] can be defined between state machines, between sequence diagrams at instance level, and between a state machine and sequence diagrams at instance level. This consistency is violated when any of the following constraints in or between sequence and PSM diagrams is violated: (i) each call sequence of the superclass PSM should be contained in the set of call sequences of the PSM of the subclass; (ii) the ordered collection of messages received by an object of the superclass should be contained in the ordered collection of messages received by an object of the subclass; (iii) the ordered collection of messages received by an object of the superclass in a sequence diagram, should exist as a call sequence of the PSM for the subclass.

Involved UML Model Elements: *ProtocolTransition*, *ProtocolStatemachine*, *Operation*, *Constraint*, *State*, *Region*, *Message* and *EventOccurrence*.

Definition 56 *Given $c, c' \in \mathcal{C}_{\mathcal{M}}$ and* $\mathbf{generalisationOf}(c, c')$*:*

A PSM $\pi_{c'} = (S', T', L', \rho', \Lambda')$ is **invocation inheritance consistent** *with a PSM $\pi_c = (S, T, L, \rho, \Lambda)$ if and only if,*
$\forall\, \mu : (\mathbf{valid}(\mu, \{\rho\}, \pi_c) \Rightarrow \mathbf{valid}(\mu, \{\rho'\}, \pi_{c'}))$ *and for the PSM traces γ corresponding to μ in π_c and γ' corresponding to μ in π'_c, it holds that $\gamma = \gamma'_S$.*

A SD δ' is **invocation inheritance consistent** *with a SD δ, if and only if,*
$\forall O \in \mathbf{contained}(\delta, \{c\})\ \forall\ \upsilon_O \in \delta : (\exists O' \in \mathbf{contained}(\delta', \{c'\}) \Rightarrow \exists\ \upsilon'_{O'} \in \delta' : =_\upsilon(\upsilon_O/^{rec}, \upsilon'_{O'}/^{rec}))$.

A PSM $\pi_{c'} = (S', T', L', \rho', \Lambda')$ is **invocation inheritance consistent** *with a SD δ if and only if,*

$\forall\ O\ \in$ **contained**$(\delta, \{c\})\ \forall\ v_O/^{rec} =< e_1\ \ldots\ e_n >$ (*with* $v_O \in \delta$) $\exists\ \mu_{c'} =< \tau_1 \ldots \tau_m >$: $(\forall\ \tau_i \in \mu_{c'} : \tau_i \in L' \wedge m \geq n \wedge \exists\ \sigma :$ **valid**$(\mu_{c'}, \sigma, \pi_{c'}) \wedge \forall\ i \in \{1, \ldots, n\}\ \exists\ j \in \{i, \ldots, m\} : \tau_j = (op, g, h) = label(e_i) \wedge (\exists \tau_k = label(e_{i+1}) \in \mu_{c'} \Rightarrow k > j) \wedge \mu_{c'}, v_O/^{rec}$ **are in strict sequence**)).

Remark that we do not define invocation consistency between a PSM π_c and a sequence diagram δ' with respect to a subclass c' of c, for the same reasons as in Definition 51.

Example 38 *Consider the PSM in Figure 2.14 which specifies the possible sequences of operations invoked on an instance of the class* ATM. *The PSM in Figure 3.23 specifies the possible sequences of operations invoked on an instance of a subclass of the class* ATM. *This subclass replaces the behaviour of withdrawing money from an* ATM *with the behaviour that allows to withdraw and charge a card with money at the same time.*

The question can now be asked if the behaviour specified by the PSMs is invocation inheritance consistent. This implies that every call sequence specified by the PSM in Figure 2.14 is also specified by the PSM in Figure 3.23. This is not the case, because the behaviour specified by the last PSM does not allow for the withdrawal of money without charging a card. Both should happen. The PSMs would be invocation inheritance consistent if the behaviour specifying the charging of the card is a separate transaction, i.e., if there is only an extension of the existing behaviour as specified by the PSM of Figure 2.14.

3.9.2 Observation Inheritance Inconsistency

Observation inheritance inconsistencies [EHK01] can be defined between state machines, between sequence diagrams at instance level, and between a state machine and sequence diagrams at instance level. Observation inheritance consistencies are violated when any of the following constraints in or between sequence diagrams and state machine diagrams is violated: (i) after hiding all new operations, each call sequence of the subclass state diagram is contained in the set of sequences of the superclass state diagram; (ii) after hiding messages that are associated to newly introduced operations, the ordered collection of messages received by an object of the subclass are contained in the ordered collection of messages received by an object of the superclass; (iii) after hiding messages that are associated to newly introduced operations, the ordered collection of messages received by an object of the subclass in a sequence diagram, exists as a sequence of the state machine diagram for the superclass.

Involved UML Model Elements: *ProtocolTransition*, *ProtocolStatemachine*, *Operation*, *Constraint*, *State*, *Region*, *Message* and *EventOccurrence*.

Definition 57 *Given* $c, c' \in \mathcal{C}_{\mathcal{M}}$ *and* **generalisationOf**(c, c')*:*

A PSM $\pi_{c'} = (S', T', L', \rho', \Lambda')$ *is* **observation inheritance consistent** *with a PSM* $\pi_c = (S, T, L, \rho, \Lambda)$ *if and only if,*
$\forall\ \mu' :$ (**valid**$(\mu', \{\rho'\}, \pi_{c'}) \Rightarrow$ **valid**$(\mu'_L, \{\rho\}, \pi_c)$).

A SD δ' *is* **observation inheritance consistent** *with a SD* δ *if and only if,*
$\forall O' \in$ **contained**$(\delta', \{c'\})\ \forall\ v' = v_{O'} \in \delta'\ (\exists O \in$ **contained**$(\delta, \{c\}) \Rightarrow \exists v''_O \in \delta : =_v (v''_O/^{rec}, v'_{E_{\delta,\{c\}}}/^{rec}))$.

A SD δ' *is* **observation inheritance consistent** *with a PSM* $\pi_c = (S, T, L, \rho, \Lambda)$ *if and only if,*

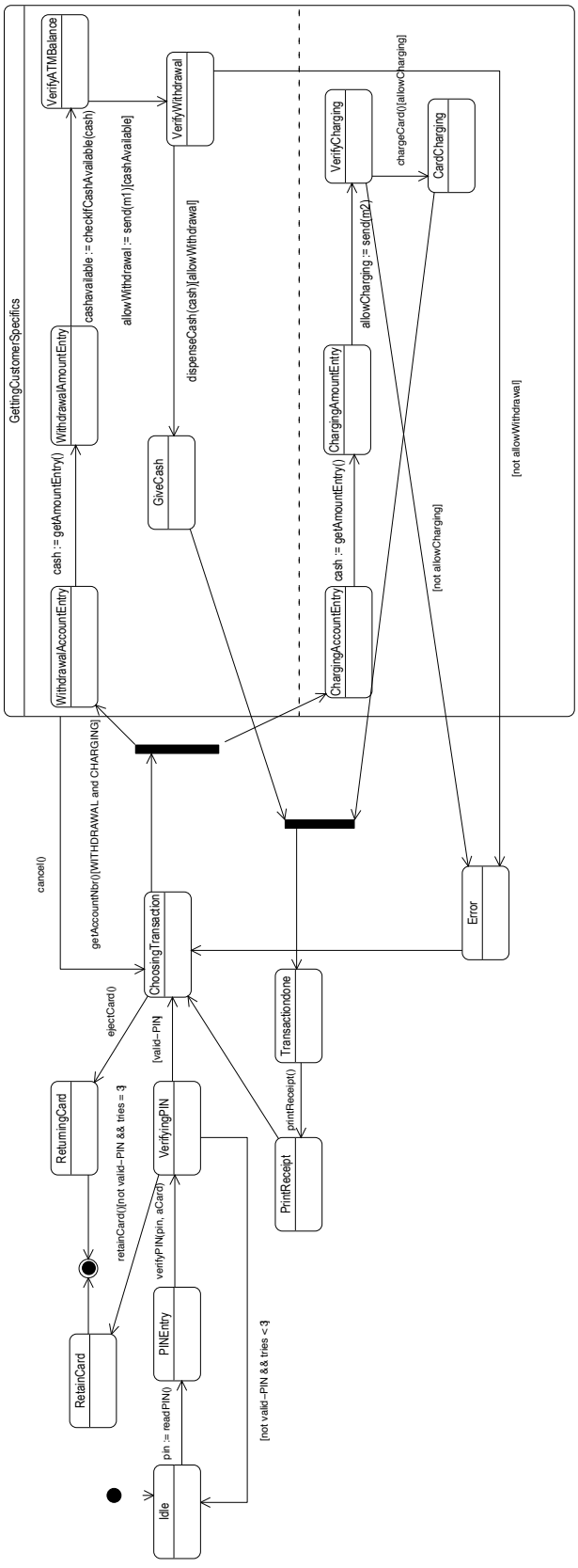

Figure 3.23: PSM for the class *CardChargingATM* subclass of ATM.

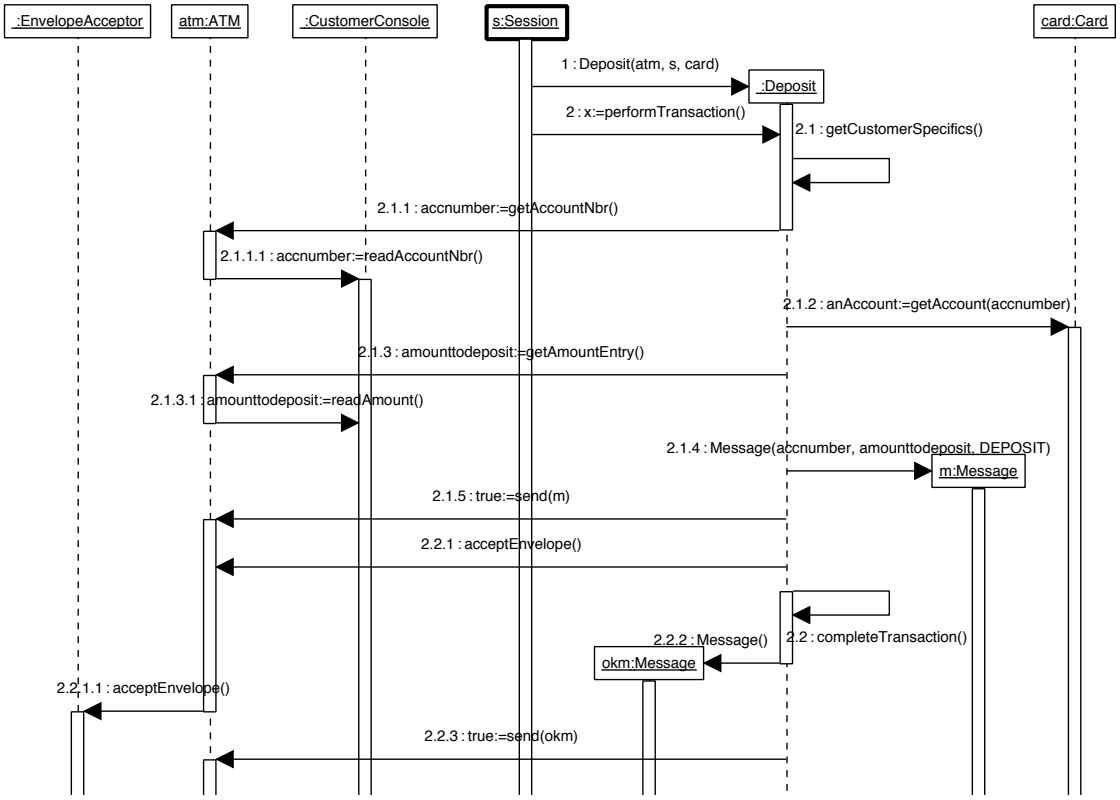

Figure 3.24: UML sequence diagram for a deposit transaction.

$\forall\ O' \in \mathbf{contained}(\delta', \{c'\})\ \forall\ \upsilon_{O'}/^{rec} =< e_1, \ldots, e_n >$ (*with* $\upsilon_{O'} \in \delta'$) $\exists\ \mu_c =< \tau_1, \ldots, \tau_m >$: $(\forall\ \tau_i \in \mu_c : \tau_i \in L \wedge m \geq n\ \exists\ \sigma : \mathbf{valid}(\mu_c, \sigma, \pi_c) \wedge \forall i \in \{1, \ldots, n\}\ \exists\ j \in \{i, \ldots, m\} : \tau_j = (op, g, h) = label(e_i) \wedge (\exists \tau_k = label(e_{i+1}) \in \mu_c \Rightarrow k > j) \wedge \upsilon_{O'}/^{rec}$, μ_c **are in strict sequence**)).

We do not define observation inheritance consistency between a PSM $\pi_{c'}$ and an SD δ with respect to the superclass c, for the same reasons as in Definition 51.

Example 39 *Consider again the PSMs shown in Figure 2.14 and in Figure 3.23. These PSMs are also not observation consistent. After hiding the operations not known to the superclass, which boils down to hiding the call to the operation* chargeCard()*, the call sequences specified by the PSM in Figure 3.23 do not correspond to the call sequences specified by the PSM in Figure 2.14. This is because the concurrent state* GettingCustomerSpecifics *causes a trace where the operation* getAmountEntry() *is called twice. This result corresponds to the intuitive meaning of observation consistency. The behaviour observed by a user of the superclass must be the same as the behaviour observed by a user of the subclass.*

3.9.3 Instance Behaviour Incompatibility

Behaviour compatibility guarantees the compatibility between a PSM for a certain class c and the set of receiving SD traces of instances of a class c. This consistency is defined

between sequence diagrams at instance level and a PSM.

An *instance behaviour incompatibility* occurs when it is not possible to find a call sequence in the protocol state machine of a class c that follows the order established by a receiving SD trace of instances of a class c.

Remark that the specification behaviour incompatibility, defined in Section 3.8.2, is similar to this incompatibility but is defined between a PSM and sequence diagrams at specification level.

Involved UML Model Elements: *ProtocolTransition*, *ProtocolStatemachine*, *Operation*, *Constraint*, *State*, *Region*, *Message* and *EventOccurrence*.

Definition 58 *Given* $c \in \mathcal{C}_{\mathcal{M}}$*:*

A PSM $\pi_c = (S, T, L, \rho, \Lambda)$ *is* **instance behaviour compatible** *with a SD* δ *if and only if,*
$\forall\ O\ \in$ **contained**$(\delta, \{c\})\ \forall\ \upsilon_O/^{rec} =< e_1\ \ldots\ e_n >$ *(with* $\upsilon_O \in \delta$*)* $\exists\ \mu_c =< \tau_1, \ldots, \tau_m >$*:*
$(m \geq n \wedge \forall\ \tau_i \in \mu_c : \tau_i \in L \wedge \exists\ \sigma :$ **valid**$(\mu_c, \sigma, \pi_c) \wedge \forall\ i \in \{1, \ldots, n\}\ \exists\ j \in \{i, \ldots, m\}$*:*
$(\tau_j = (op, g, h) = label(e_i) \wedge (\exists\ \tau_k = label(e_{i+1}) \in \mu_c \Rightarrow k > j) \wedge \upsilon_O/^{rec}$, μ_c **are in strict sequence**$))$.

Example 40 *Consider a UML model consisting of the class diagrams in Figure 2.4, Figure 2.6 and Figure 2.8 and the sequence diagrams in Figure 2.12, Figure 3.15, Figure 3.24 and Figure 3.25 and the PSM that is the result of the PSM in Figure 2.14 together with the different composite states, representing the different transactions of the* ATM*, modelled in Figure 3.11, Figure 3.12 and Figure 3.13.*

Figure 3.24 shows a possible scenario for the Deposit *transaction. In case of a deposit transaction, the user is asked for the amount to deposit and the account number of the account on which to deposit the money. If the transaction is allowed, the envelope is accepted and a message is sent to the bank before the envelope is physically accepted. Figure 3.25 shows a scenario for the* Inquiry *transaction. In case of an inquiry transaction, the account number of the account to inquire is asked and the balance of the account is shown to the user.*

The behaviour of the class ATM*, specified by the different sequence diagrams must compatible with the behaviour specified by the PSM. This means that every trace contained in the different sequence diagrams corresponds to a valid call sequence in the PSM. This model is* instance behaviour compatible*.*

3.10 General Discussion

Different definitions for - what we call - structural inconsistencies can be found in literature.

Ehrig and Tsiolakis [ET00] investigate the consistency between UML class and sequence diagrams. As consistency checks between class and sequence diagrams only existence, visibility and multiplicity checking are considered. Existence checking corresponds to checking the occurrence of a dangling association reference; visibility checking concerns the checking of visibility restrictions defined on classes, attributes and operations according to the visibility restrictions declared in the class diagram. Checking for a dangling feature reference in our work corresponds to a limited version of visibility checking as defined in [ET00]. We

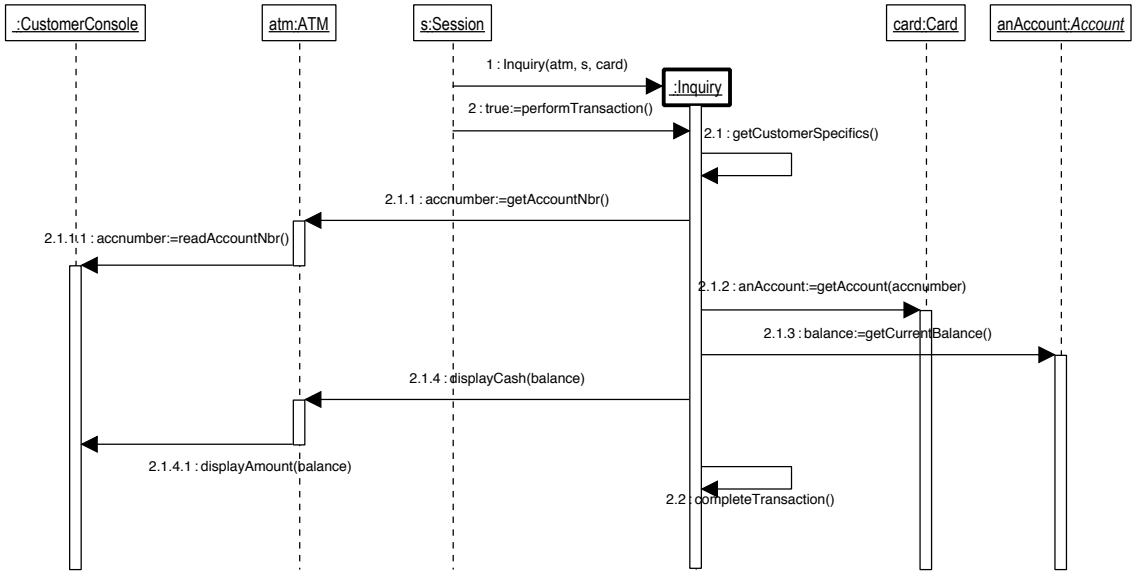

Figure 3.25: UML sequence diagram for an inquiry transaction.

only provided a limited version because the visibility restrictions specified for UML classes, attributes, operations and associations need to be defined by the user in UML 2.0.

Kielland *et al.* [KB01] present a set of inconsistencies comprising syntactical errors and completeness/omissions in UML class, state and sequence diagrams. The set is a subset of our set of structural inconsistencies, their inconsistencies are informally specified and heavily dependent on the XMI-format of Rational Rose.

Lange *et al.* [LCM+03] also present a set of inconsistencies and incompleteness issues. Their set of inconsistencies is a subset of our set of structural inconsistencies. Their inconsistency and incompleteness sets are based on six large-scale industrial case studies and the inconsistencies and incompleteness issues are defined in SQL.

All these approaches only present and check structural inconsistencies and the different inconsistencies are always immediately defined using a particular (formal or informal) inconsistency detection approach.

As already mentioned in Section 3.6 Engels *et al.* [EE95] were the first to define *observable and invocable consistency*. Definitions for restrictions on inheritance of object life cycles based on Petri nets can be found in [SS00], [SS02] and [van02a]. None of these approaches takes a close look on how to define behavioural inconsistencies between UML models consisting of sequence diagrams and PSMs.

Küster [KÖ4] presents in his dissertation a general methodology for consistency management of object-oriented behavioural models. The methodology is illustrated by a sample development process for concurrent object-oriented systems. *Consistency problem types* are defined between one or more submodel types resulting in a so-called consistency problem type candidate matrix. Some consistency types between object-oriented behavioural models (i.e., UML-RT sequence diagrams and state diagrams) are defined using the semantic domain CSP. We define (in this chapter) concrete inconsistencies and we believe that making

the inconsistency management process concrete is an important contribution of our work.

3.11 Conclusion

This chapter presents a second contribution of this dissertation. A classification of distinct inconsistencies is introduced. In the previous chapter, a fragment of the UML metamodel and some semantical concepts were well-defined. These definitions enable well-defined inconsistencies independent of any detection formalism or approach. Each of our classified inconsistencies (or consistencies) are described by specifying the UML metamodel elements involved in the (in)consistency, by a precise definition and by a concrete example using UML class, sequence (communication) and protocol state machine diagrams.

The definitions of our classified inconsistencies specify the conditions for having an inconsistency and can be used by inconsistency detection mechanisms. However, if an inconsistency is detected, what are the possible *actions* for dealing with an inconsistency? Is it possible to *avoid* inconsistencies by specifying co-called *construction rules*? These questions are answered in the *inconsistency handling* activity of the inconsistency management process. In the next chapter, we discuss this activity in the context of the inconsistencies defined in this chapter and the UML fragment defined in the previous chapter.

Chapter 4

Inconsistency Handling

In this chapter, a discussion is opened on how to handle inconsistencies and on how to preserve consistency in and between models. First, we discuss the different inconsistency handling strategies and their terminology (Section 4.1). Next, we focus on *resolution actions* that allow the semi-automatic resolution of inconsistencies (Section 4.2). We show examples of resolution actions for the inconsistencies defined in our classification. A classification of resolution actions is presented. The connection between this classification and the classification of the inconsistencies presented in the previous chapter is clarified.

Consistency maintenance emphasises the preservation of consistency. *Construction rules* can be defined guaranteeing the preservation of certain consistencies when applied on a model (Section 4.3). We also show possible applications of this notion on the inconsistencies defined in our classification.

In conclusion of this chapter, we first summarise our ideas and related work (Section 4.4). A concrete inconsistency handling approach and its implementation will be introduced later on. (Chapter 8 and 10) Next, the different requirements for the evaluation and validation of an inconsistency detection and resolution formalism as first introduced in Section 1.6, are revisited (Section 4.5).

4.1 Terminology

4.1.1 Inconsistency Management

As outlined in the introduction of this dissertation, the process of *inconsistency management* includes activities for detecting, diagnosing and handling inconsistencies. The definitions of our classified inconsistencies specify the conditions for the occurrence of an inconsistency and can be used by inconsistency detection mechanisms. The question remains how to diagnose and handle the inconsistencies?

The activity of *diagnosing* is concerned with the *source*, the *cause* and *impact* of an inconsistency [SZ01]. In our approach, the *source* of an inconsistency is a subset of the UML elements involved in the inconsistency. In Chapter 7, we will explain how the possible sources of an inconsistency are returned by the detection mechanism. In our approach, inconsistency detection returns not only a yes or no answer to the question whether a certain inconsistency occurs, but it also returns the set of elements involved in this inconsistency. The *cause* of an inconsistency can be important in the activity of handling inconsistencies.

This is detailed in Section 4.2.1. Establishing the *impact* of an inconsistency can be necessary for deciding the priority with which the inconsistency has to be handled. Establishing this impact is a task normally deferred to the user. The impact of an inconsistency is application dependent and dependent on the current activity of the software development life-cycle. Therefore, we will not consider the establishment of the impact of an inconsistency in our work.

The activity of *inconsistency handling* is concerned with the questions of *how to deal with inconsistencies*, *what are the impacts and consequences of specific ways of dealing with inconsistencies*, and *when to deal with inconsistencies* [SZ01]. Different handling strategies are known and combinations of the different strategies can occur. In our work, we will focus on the question how to deal with inconsistencies.

The central idea of *inconsistency management* is to tolerate inconsistencies. Nuseibeh *et al.* [NER00] emphasise the importance of tolerating inconsistencies. They propose a framework for inconsistency management which allows inconsistencies to be ignored and deferred. Spanoudakis *et al.* [SZ01] differentiate between *changing* and *non-changing* actions. The latter kind of actions only notify users or perform some analysis that allows safe further reasoning from inconsistent models. The former kind of actions can be subdivided in *partial* and *full resolution actions*. Partial resolution actions ameliorate models with respect to inconsistencies but do not fully resolve the inconsistencies. Full resolution actions modify models in order to resolve inconsistencies. This brings us to *inconsistency resolution*.

Inconsistency resolution is concerned with the *resolution* of inconsistencies. It is considered critical to inconsistency management, but also extremely difficult [Fin00]. The resolution of inconsistencies can be done by user interaction or by automatically changing the models or by a combination of both. Different techniques exist for helping the user to resolve the inconsistency. From now on, we will use the term *resolution actions* to indicate partial and full resolution actions. Full resolution actions are most of the time a combination of different partial resolution actions (see Section 4.2). Resolution actions can consist of a set of guidelines, provided to the user, on how to resolve the inconsistency. The actions can also be a set of pre-defined steps specifying the resolution of a particular inconsistency. The resolution of an inconsistency can be done automatically or semi-automatically by a resolution action. The choice of the resolution action(s) is dependent on the kind of inconsistency and also on the cause of the inconsistency. Section 4.2 explains resolution actions for our defined inconsistencies. Questions to be answered are: *for which inconsistencies in our classification can resolution actions be defined, to which degree is user interaction needed for these actions, and what are the possible consequences of applying resolution actions.*

4.1.2 Consistency Maintenance

As opposed to inconsistency management, *consistency maintenance* guarantees consistency to a certain extent. Two flavours of consistency maintenance are *consistency enforcement* and *consistency preservation*. In the former case, consistency of models is enforced at any time. Only a strict set of operators is allowed for changing a model because consistency must be preserved and temporal inconsistencies during development are not allowed. This is a very restricted way of modelling.

A weaker version of consistency maintenance is to try to *preserve consistency* as much as possible. In this case inconsistencies in the model are allowed. Instead of first detecting

inconsistencies and then handle them, *construction rules* are specified to construct from a consistent model other consistent models. These rules specify the changes that are allowed to a model in order to keep the model consistent. They specify the creation, deletion or modification of abstract syntax elements of the modelling language, e.g., UML metamodel elements. Construction rules can also be used to establish consistency-preserving model-evolution in which case, they specify model transformations. There are different ways to specify these rules, a visual representation can be used, they can be specified in a certain semantic domain, or by other formalisms such as, e.g., homomorphisms. Section 4.3 answers the question if it is possible to specify construction rules for avoiding inconsistencies defined in our classification.

It is not the aim of this thesis to give a complete set of construction rules for every defined consistency. Only some rules will be defined to illustrate or clarify certain issues involved in consistency preservation.

4.2 Resolution Actions

Violations to syntactic inconsistencies in a model, i.e., a model not conforming to the abstract syntax, can in some cases be corrected automatically by pre-defined resolution actions. However, in most cases different solutions are possible and the user must decide which one to choose. In the case of semantic inconsistencies, it is also difficult to correct the inconsistencies in an automatic way. In most cases, the resolution of the inconsistency consists of changes in the model that also change the semantics of the model. A lot of user interaction is needed in that case. We will discuss possible resolution actions for our classified inconsistencies.

4.2.1 Causes of Inconsistencies versus Resolution Actions

Choosing between different possible resolution actions can be dependent on the cause of the inconsistency to be resolved. We will illustrate this by considering the inherited cyclic composition inconsistency defined in Section 3.3.1 and the dangling type reference inconsistency defined in Section 3.3.2.

The inherited cyclic composition inconsistency can be solved in different ways. Examples of solutions are:

- The generalisation relationship can be removed from the model.
- The composition relationship can be replaced by a normal association.
- The composition relationship can be removed.
- The multiplicity can be changed from mandatory to optional, i.e., the multiplicity restriction is weakened.

Which solution to choose depends on the cause of the inconsistency. Suppose the aim of the user is to introduce the well-known *composite* design pattern [GHJV94]. In this case, it is the multiplicity of the composition relationship that is too strong. It can be that the superclass is conceptually not a generalisation of the subclass. In this case, it is the

generalisation relationship that is superfluous. Perhaps the user only wants to specify a part-whole relationship between the two classes. In this case, it is the composition restriction on the association that is redundant. However, without extra knowledge on previous model changes or without extra knowledge from the user, it is not possible to (automatically) detect the causes of this inconsistency. Consequently, in most cases, the user must decide which action must be performed.

However, even with knowledge about the previous model changes, it is sometimes only possible to propose some resolution actions that are more likely to be executed than others depending on the causes of the inconsistency.

Consider the dangling type reference inconsistency. The reasons for this inconsistency to occur are twofold. The type of the attribute or the parameter involved may have been removed from the model or the type is not yet included in the model. Possible resolution actions are:

- The addition of the type to the UML model involved.
- Replacing the type of the attribute or parameter by an existing type in the model involved.
- Removal of the involved attribute or parameter.

In case the inconsistency arises as a consequence of an evolution (or refinement) step that removed the type from the previous version of the model, it is not likely that as a solution to the introduced inconsistency, the type will be added again to the model. However, in some cases, it might be necessary to restore the previous (consistent) version of the elements involved in the model. In general, none of the possible resolution actions may be excluded.

4.2.2 Classification of Resolution Actions

Recall the different definitions of our classified inconsistencies in the previous chapter. For each of the inconsistencies, we listed the involved UML metaclasses. If a particular inconsistency is detected, resolving the inconsistency reduces to changing the involved instances of the metaclasses involved in the definition of the inconsistency.

From the viewpoint of the software developer using our classified inconsistencies, we distinguish three different kinds of resolution actions.

- **Add model element** In this case, a model element is created. An example is the addition of a class to the model. In terms of the UML metamodel, this implies the instantiation of a metaclass.
- **Remove model element** In this case, a model element is removed. For example, an operation is removed from a UML class diagram. This implies the deletion of an instance of a certain metaclass.
- **Change model element** In this case, a model element is changed by changing one of its properties. When an operation is removed on a UML class diagram, the corresponding instance of the metaclass *Operation* is removed. However, this instance, i.e., the operation in question, can still be referenced by a class or by messages. These references are, in terms of the UML metamodel, instances of certain meta-associations.

These meta-associations state the connections between different UML model elements. Changing an element encompasses changing the relationships between this element and another element. For example, messages can be changed by changing the referenced operation. For the software developer this is an atomic action, but on an underlying level, this involves two actions, i.e., *removing and creating* a connection between UML model elements.

The different resolution actions for our classified inconsistencies can be classified using the same dimensions as the dimensions of our inconsistency classification. First, we classify the different UML metamodel elements into the categories: structural/specification, structural/instance, behavioural/specification and behavioural/instance. This classification is shown in Table 4.1. Remark that the set of UML model elements in the behavioural/instance dimension is a subset of the model elements in the behavioural/specification dimension because both interpretations, i.e., instance and specification, map onto the same UML metamodel elements, as explained in the previous chapter.

The classification of resolution actions corresponds to this classification. Three kinds of resolution actions correspond to each metamodel element specified in Table 4.1. For example, the resolution actions corresponding to *class* are *add class*, *remove class* and *change class*. A *change class* can involve the addition or removal of a reference to an attribute, operation, association end and generalisation. The references of the elements that are changed by a *change element* action, are determined by the different meta-associations in which the element participates.

	Behavioural	**Structural**
Specification	eventoccurrence, message, lifeline precondition, postcondition, connector, connectableElement, constraint	class, association, property, multiplicityelement, generalization, parameter, type, operation
Instance	protocol state machine, state,protocol transition, eventoccurrence, message, lifeline, connector, region, constraint, precondition, postcondition	instancespecification

Table 4.1: Two-dimensional resolution actions table.

The classification of resolution actions can be linked to our classification of inconsistencies. For each inconsistency occurring in our inconsistency classification, the UML metamodel elements involved in the inconsistency are described (see previous chapter). Based on these elements, a link can be made between the classification of resolution actions and the classified inconsistencies as shown in Table 4.2. The columns represent the different categories of resolution actions. The rows list the different categories of inconsistencies. For example, consider the category behavioural/specification-instance inconsistency. This category of inconsistencies can be resolved by behavioural/specification, behavioural/instance and structurql/specification resolution actions.

The resolution actions presented here, are fine-grained. In real-world projects, it is sometimes necessary to execute several of these fine-grained resolution actions to cope with a certain inconsistency. For example, the solution of a behaviour incompatibility could be the addition of a sequence of event occurrences. This involves the addition of several event occurrences and the addition of several messages and changing these event occurrences in

Resolution actions / Inconsistencies	Behavioural/ Specification	Behavioural/ Instance	Structural/ Specification	Structural/ Instance
Behavioural/Specification	X		X	
Behavioural/Specification-Instance	X	X	X	
Behavioural/Instance		X	X	
Structural/Specification	X		X	
Structural/Specification-Instance		X	X	X
Structural/Instance		X		X

Table 4.2: Inconsistencies and resolution actions table.

such a way that the correct messages, lifelines and connectors are referenced. The newly added messages must also refer to the correct operations. An inconsistency resolution mechanism must allow the grouping of different fine-grained resolution actions.

In certain application domains, certain resolution actions are more important than others, or the general resolution actions defined, must be fine-tuned. Domain-specific knowledge is necessary, e.g., to order the resolution actions or to exclude some resolution actions as possible solutions for a certain inconsistency. A domain can be the domain being modelled, or the system domain, e.g., real-time systems. In Chapter 9, we show how inconsistency detection and resolution can be applied for the execution of model refactorings. The possible resolution actions for a certain inconsistency can be restricted in the context of a certain model refactoring. This means that the set of possible resolution actions is tailored towards the executed model refactoring.

Fine-tuning possible resolution options by the user is also proposed in the context of conflict resolution during software merging [Men02] [MD94]. In that context, resolution strategies are implemented in a uniform and customisable way. These strategies can be adjusted depending on the application domain.

4.2.3 Dependencies between Resolutions of Inconsistencies

Resolution actions may introduce new inconsistencies. Consider the occurrence of a dangling feature reference inconsistency between the class diagram and communication diagram in the left-hand side model shown in Figure 4.1.[1]. The operation *readPIN* is not known to the class *CardReader*. There are several possible solutions to this inconsistency. In Figure 4.1, the type of the receiving object in the communication diagram is changed from *CardReader* to *CustomerConsole*. This resolution action introduces a new inconsistency: a dangling association reference. This inconsistency occurs because there is no association defined between the class *ATM* and *CustomerConsole*.

To resolve this inconsistency, an association is created between these classes, but the association specifies that navigation is only allowed from the *CustomerConsole* class to the *ATM* class (see right-hand side of Figure 4.2). This resolution action introduces a *navigation conflict*, because in the communication diagram the operation *readPin* is sent from the *ATM* class to the *CustomerConsole* class.

[1]Only the relevant features for the explanation of our ideas are included in the models.

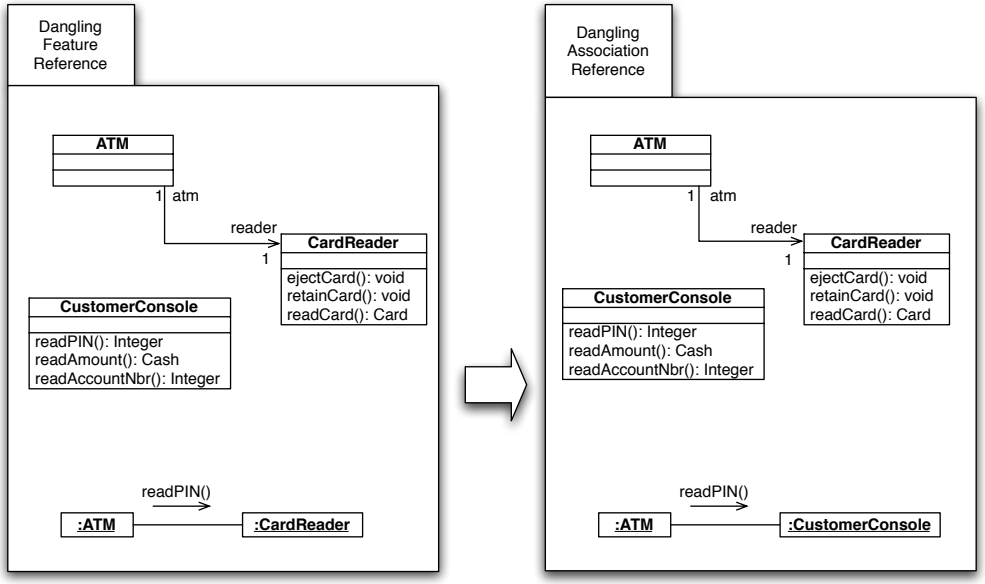

Figure 4.1: Resolution of a *dangling feature* inconsistency introducing a *dangling association*.

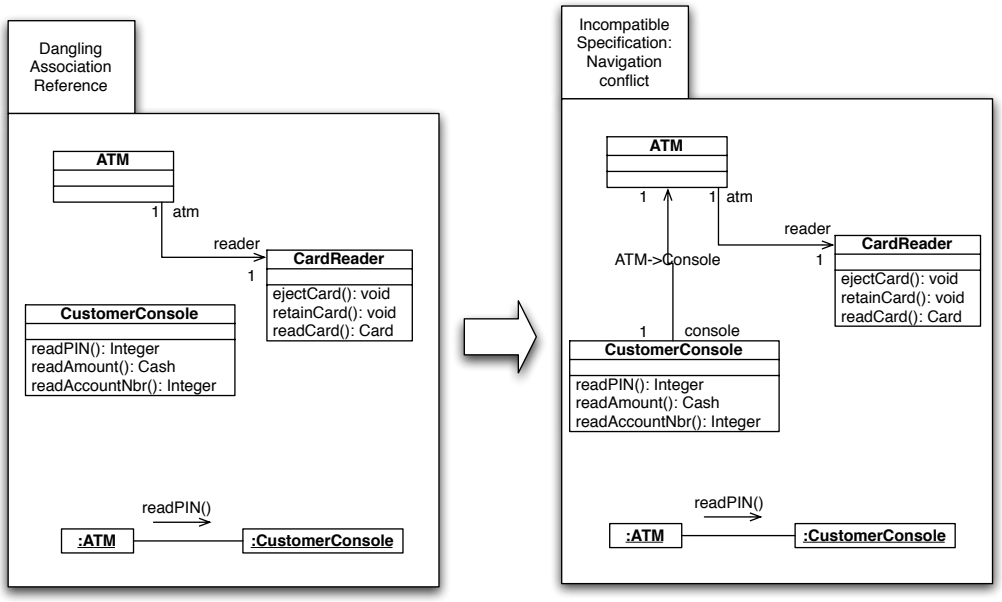

Figure 4.2: Resolution of *dangling association* inconsistency introducing a *navigation incompatibility*.

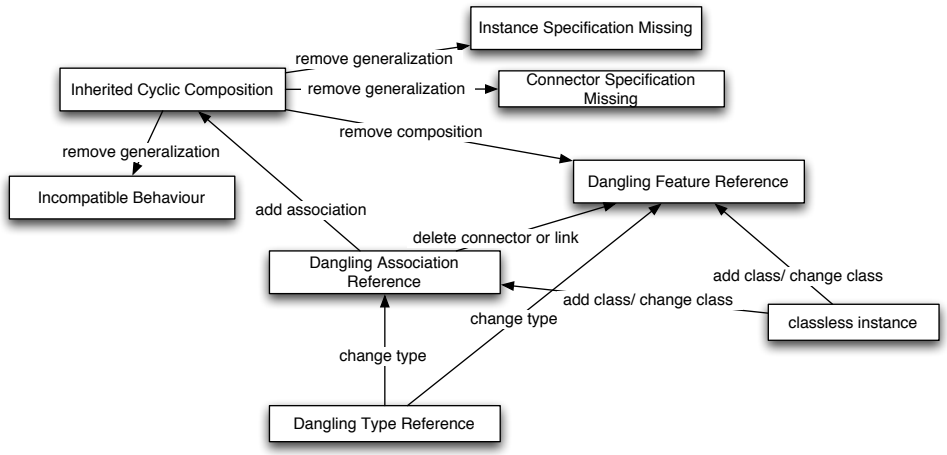

Figure 4.3: Dependencies between resolution of inconsistencies.

The inconsistencies that can be introduced and the inconsistencies that depend on each other, are determined by the resolution actions. A *dependency graph* can be developed, which is a directed graph, expressing the relations between different inconsistencies depending on the resolution actions for the purpose of inconsistency management tools to support resolution. The vertices of the graph are the detected inconsistencies and the edges are labelled with the resolution actions. The graph in Figure 4.3 shows some of the dependencies between inconsistencies. The source vertex shows the detected inconsistency, the target vertex shows the newly introduced inconsistency that can possibly occur after the execution of the resolution action as specified on the edge.

It is obvious that for a certain amount of inconsistencies and resolution actions, this graph becomes quite huge. Constructing such a graph would be an iterative and error-prone process. No matter which approach that will be used to resolve inconsistencies, it should enable the management of these dependencies. Chapter 8 and Chapter 10 will describe how our approach will cope with these dependencies.

Due to the possible introduction of new inconsistencies by resolution actions, in worst case, cycles can occur. These cycles will become visible in the graph. However, it is not because there is a cycle in the dependency graph that this will lead to an infinite chain of resolution actions. This will be the case, for example, if an inconsistency already occurred during a preceding checking and resolution phase for the same set of modelling elements. For example, in case a message or a transition references a non-existing operation, which is an example of a dangling feature reference, a possible solution is to move the operation(s) to the class in question. This resolution action can cause a new occurrence of a *dangling feature reference* but for other model elements. Algorithms can be built to detect such cycles (e.g., in [WGN03]). In case a cycle is detected, the user has to take action to resolve the cycle.

In most cases, a certain amount of user interaction is necessary to resolve an inconsistency. Consider, e.g., the resolution of interaction inconsistencies, inheritance inconsistencies and behaviour incompatibility. If a UML model is behaviour incompatible, the cause

can be in the PSM or in one of the sequence diagrams. In the latter case, the sequence diagrams must be adapted implying adding message sends or deleting message sends. In the former case, the PSM must be adapted by adding transitions and states or deleting transitions and states. User interaction can be necessary to indicate whether the sequence diagrams or the PSMs must be adapted and to indicate the kinds of actions that need to be executed. For these kinds of inconsistencies, one can prefer to give construction rules. Such rules are supported by consistency maintenance. These rules specify how PSMs or sequence diagrams can be constructed that are compatible, or they specify how, given PSMs and/or sequence diagrams specifying the behaviour of a certain class, new invocation/observation consistent PSMs and/or sequence diagrams can be built for a sub- or superclass. In the next section, we will illustrate how construction rules can be defined.

4.3 Construction Rules

Instead of detecting and resolving inconsistencies, an alternative strategy is to avoid inconsistencies by applying *construction rules*. A good usage example of these rules is the specification of how the behaviour of a certain class (or set of classes) can be specialised obeying some behavioural consistencies.

4.3.1 Preservation of Observation/Invocation Consistency

How can we construct, e.g., PSMs and sequence diagrams obeying invocation inheritance consistency? In [EE95], constructive ways are given to build a state transition diagram for a specialised subclass in such a way that the existence of a homomorphism defining invocation or observation consistency, is guaranteed. Constructive rules can be given to enforce invocation or observation inheritance consistency between UML PSMs and SDs. In our case, these rules can be specified using the formalisation of the UML fragment introduced in Chapter 2.

The rules for constructing a subclass' PSM that is invocation inheritance consistent with the superclass' PSM, are:

- States, transitions and regions can be added to the PSM without causing any problem as long as the original state machine is embedded in the new one. After applying this rule, the following constraint must be valid: Given a PSM $\pi_c = (S, T, L, \rho, \Lambda)$, a PSM $\pi_{c'} = (S', T', L', \rho', \Lambda')$, where $\mathbf{generalisationOf}(c, c')$, can be constructed, with $S \subseteq S' \wedge T \subseteq T' \wedge L \subseteq L' \wedge \Lambda \subseteq \Lambda' \wedge \forall \tau : (\mathbf{valid}(\tau, \{\rho\}, \pi_c) \Rightarrow \mathbf{valid}(\tau, \{\rho'\}, \pi_{c'}))$.

- States and regions can be deleted but the transitions must be kept and their mutual order must be maintained. This implies that transitions may not be deleted but their source and target states can change. This rule is equal to the previous rule, without the requirement that $S \subseteq S'$.

If the superclass' behaviour is specified using sequence diagrams, invocation inheritance consistent sequence diagrams exhibiting the interactions between certain objects and instances of a subclass can be constructed by adding new traces to the lifeline of the respective instances but the traces specified for the instances of the superclass may not be altered. It is obvious that deleting a trace is not allowed. The following constraint must be valid after

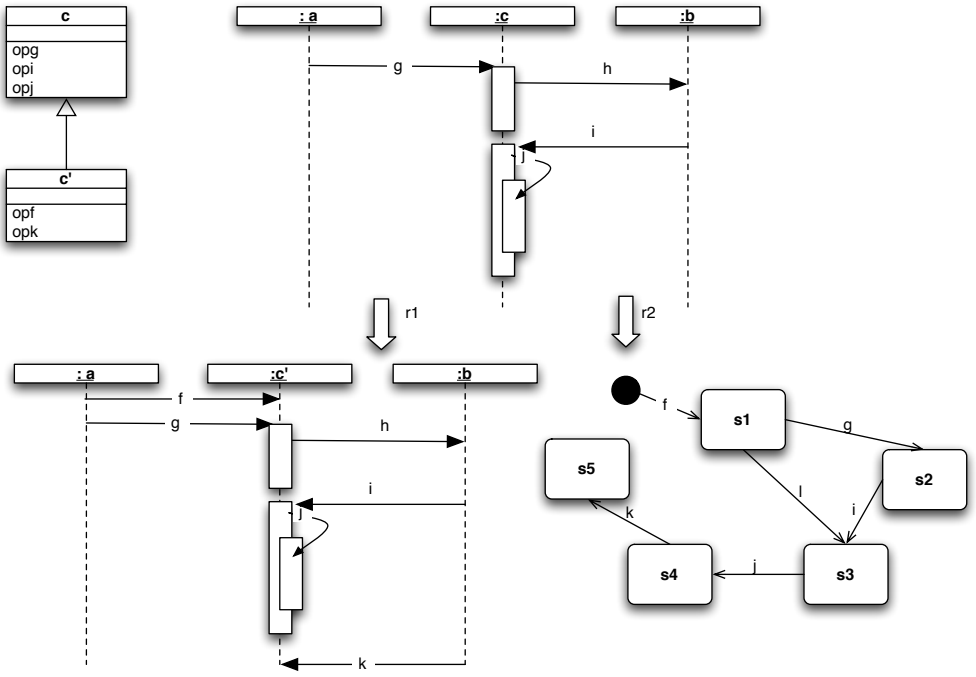

Figure 4.4: Construction rules for invocation consistency.

the execution of the rules: Given a sequence diagram δ specifying interactions between, among others instances of type c, where **generalisationOf**(c, c'), a sequence diagram δ' can be constructed such that every receiving SD trace for instances of class c in δ is also a SD trace in δ' for an instances of class c'. The rule is visualised in the example of Figure 4.4 where it is called $r1$. The SD trace $< g, i, j >$ for the instance of type c is included in the SD trace $< f, g, i, j, k >$ for the instance of type c'. (The correspondence between the messages and the operations in the figure is as follows: a message x corresponds to an operation call opx.)

From sequence diagrams containing different traces in which an instance of a superclass is involved, an invocation inheritance consistent PSM can be built. The only rule that must be valid is that all the traces specified by the sequence diagrams under study must be included into the PSM. There can be more operations called in the PSM and there is also no restriction on the number of states or how they are connected as long as : $\forall \upsilon_{O_{\{c\}}}/^{rec} =< e_1 \ \ldots \ e_n > \in \delta \ \exists \textbf{valid}(\mu' =< \tau_1 \ldots \tau_n >, \sigma, \pi_{c'}) \ \forall \ i \in \{1 \ldots n\} : \tau_i = label(e_i))$. This rule is shown in Figure 4.4 where it is called $r2$. The SD trace $< g, i, j >$ is included in the PSM of the class c'.

If the behaviour of the superclass is specified by a combination of sequence diagrams and PSM, new sequence diagrams and a PSM can be constructed using the rules specified above.

Similar rules can be given to construct from a PSM of a superclass or from sequence diagrams containing interactions involving instances of a superclass, an observation inheritance consistent PSM of a subclass, or sequence diagrams involving a subclass' instances.

In case the superclass' behaviour is specified using a PSM, the following actions can be executed on the PSM in order to construct a subclass' PSM.

- States and transitions can be added to the PSM without causing any problem as long as the operations called by the new transitions are only known to the subclass and not to the superclass.
- Regions can also be added to the PSM as long as the transitions contained in these new regions represent calls of operations only known to the subclass or as long as the traces generated by these transitions are already known to existing regions in the superclass' PSM.
- States, transitions and regions can be deleted without any problem.

If the superclass' behaviour is specified using sequence diagrams, sequence diagrams exhibiting the interactions between certain objects and instances of a subclass can be constructed by adding new traces to the lifeline of the respective instances or by altering existing traces through the inclusion of messages representing calls to operations known only to the subclass. Deleting traces is allowed.

From the PSM of a superclass, sequence diagrams can be built expressing traces in which an instance of the subclass is involved. These traces can be a subset of the traces included in the PSM, or new ones may be added representing calls to operations only known to the subclass or traces included in the PSM may be altered by including calls to new operations. It does not matter where these calls are included in a trace as long as the order is not altered of the existing traces. Remark that calls may be deleted from traces.

If the behaviour of the superclass is specified by a combination of sequence diagrams and PSM, new sequence diagrams and a PSM can be constructed using the rules specified above.

4.3.2 Preservation of Behaviour Compatibility

Construction rules can also be specified for the preservation of behaviour compatibility. Starting from a sequence diagram, a PSM can be generated following the rule that every SD trace for the instances of the specific class must be included in the PSM. Starting from the call sequences included in the PSM of the involved class, different traces occurring in a sequence diagram can be generated.

In both cases, we start from a particular diagram and construct another type of diagram in a behaviour compatible way. However, it is also possible to start from a model containing both types of diagrams and specify rules that preserve behaviour compatibility when changing the model. For example, changing the name of a state in a PSM does not affect behaviour compatibility nor does any modification on class diagrams because class diagrams are not involved in the definition of behaviour compatibility. Only rules considering model elements involved in the definition of an inconsistency must be taken into account.

4.3.3 Preservation of Structural Consistencies

Constructive rules can also be defined for enforcing our structural consistencies. For example, to enforce that a message references an operation known in the model, adding modelling

elements to sequence and class diagrams is allowed. However, deleting operations involved in sequence diagrams is not allowed. For our structural consistencies, it is sometimes easier to specify *negative application rules*, i.e., rules that can lead to the construction of an inconsistent model. This concept is similar to *negative application conditions* [EHHS02] in the context of graph transformations or Dynamic Meta Modeling (DMM) rules. In [EHHS02], consistency conditions for UML dynamic diagrams are checked using DMM rules with *negative application conditions* (NAC). These NACs represent structures that must not be present in the context of a DMM rule application.

4.4 Discussion

4.4.1 Conclusions

Roughly, in inconsistency management, resolution actions are used for handling inconsistencies, while in consistency maintenance models are constructed using constructions rules that guarantee the preservation of the consistencies. Both strategies can be taken and defined in our approach. In the remainder of this dissertation, however, we will only concentrate on resolution actions because the context of this work is inconsistency management as opposed to consistency maintenance.

From the previous discussion, we can draw some important conclusions on inconsistency resolution.

(1) Which resolution actions to choose, can be dependent on the cause of an inconsistency. However, this dependency cannot be captured without extra knowledge provided by the software developer. As a result, most of the time the execution of resolution actions involves a lot of user interaction.

(2) We also introduced a classification of possible resolution actions for our classified inconsistencies. These resolution actions are fine-grained and correspond to (elementary) operations that can be applied on a UML model.

(3) Depending on the granularity of the resolution actions and the application domain involved, it might be necessary to group and combine several resolution actions.

(4) Resolution actions can introduce new inconsistencies. The dependencies between different inconsistencies caused by resolution actions must be taken into account by inconsistency resolution approaches.

The overall conclusion is that tools and an underlying formalism supporting inconsistency resolution are necessary and must be very flexible. Inconsistency resolution often relies on resolving fundamental conflicts or making important design decisions. It remains a very interactive activity. However, support can be offered to the software engineer. How inconsistency resolution can be supported in our approach is clarified in Chapter 8.

4.4.2 Related work

In [HHS02], graph transformations are used to specify resolution actions for the automatic resolution of some syntactic inconsistencies, i.e., UML models violating the UML abstract syntax. Other techniques let the user define and select resolution actions [Eas91], others generate resolution actions [SF97] or provide very specific resolution patterns [vLLD98]. Resolution actions in the context of UML and more specifically, in the context of software

specifications expressed in UML, are defined in [KZ04]. By analysing the inconsistency occurred, resolution actions can be chosen.

Construction rules guarantee the preservation of certain consistencies when applied to a consistent model. These rules restrict the set of possible operations applied to a model emphasizing the set of operations, or these rules specify the consistency conditions. They are very useful in constructing behavioural consistent models.

In [EE95] sufficient rules are given to construct state diagrams related by a homomorphism expressing observation or invocation consistency. In [SS02], necessary and sufficient rules are given to check behavior consistency between object life cycles in the realm of Object Behavior Diagrams. These rules specify consistency conditions rather then operations allowed on the abstract syntax elements. Transformation rules are defined for UML-RT elements in [EHKG02] and [KÖ4]. Such a rule captures a certain model evolution step by describing the modification of the model in terms of UML-RT metamodel elements. Under certain conditions depending on the modification, the new model is consistent.

4.5 Key Criteria

From the discussions on the modelling language fragment in Chapter 2, from our classification of inconsistencies in Chapter 3 and from the discussions on inconsistency resolution in this chapter, a set of criteria can be distilled and detailed. These criteria can be used to evaluate a static inconsistency detection and resolution formalism that serves as the basis for inconsistency management tool support. The criteria set some requirements that, ideally, are met by an inconsistency detection and resolution approach.

Our evaluation criteria are specified in a generic and objective way which makes them applicable to inconsistency detection and consistency resolution formalisms for general object-oriented modelling languages. Of course, we will use them in the context of the introduced UML fragment and related definitions in Chapter 2.

Remark that several key requirements for supporting inconsistency management are recognised in [GHM98]. However, what is called requirements for an inconsistency management environment in [GHM98], are the different activities of the inconsistency management process.

4.5.1 Criterion #1: Abstract Syntax and Semantics Representation

As a first requirement, it must be possible to express the *abstract syntax* in the formalism, this guarantees the well-formedness of the user-defined models. For UML this implies that it must be possible to (1) describe sets of objects, i.e., classes and their attributes; (2) relationships between these classes; (3) generalisations between the classes, i.e., the subset relationship between sets of objects, and (4) some constraints on these concepts. The formalism must provide a sound foundation for this metamodel in order to reason about the metamodel and the user-defined models.

As a second requirement, the *semantics* or at least, part of the semantics of the modelling language must be expressible in the formalism. The UML fragment considered in this dissertation consists of different sublanguages, sequence diagrams, PSMs and class diagrams. A formalism for UML inconsistency detection and resolution must support the

representation of a semantics or part of a semantics of the different languages. For example, in Chapter 2, SD traces and call sequences are defined as semantical concepts. This motivates our focus on the expression of these semantic concepts in the formalism we will study. The representation of a semantics of different sublanguages can be established in different ways. A first way is to integrate different formalisms each defining the semantics of a sublanguage. In this case, the question remains how these different formalisms can be connected, i.e., if it is possible to reason about the different semantics of the different sublanguages. Another way is to require a formalism that is powerful enough to define the semantics of (most of) these sublanguages.

An inconsistency detection and resolution formalism needs to be able to express the abstract syntax and semantics of the modelling language such that well-formedness of the user-defined models can be guaranteed.

4.5.2 Criterion #2: Precise Definitions of Inconsistencies and Inconsistency Detection

The ability of the formalism to express the abstract syntax and semantics may not be sufficient to express the definitions of the inconsistencies. For example, a formalism can be rich enough to express the abstract syntax of UML, but for defining certain inconsistencies, navigation over metaclasses and meta-associations is demanded and the transitive closure of certain meta-associations is required (see definitions of our classified inconsistencies in the previous chapter). A first requirement is the *ability to define inconsistencies (or consistencies) in a precise and unambiguous way.*

A detection formalism can be required to satisfy different *formal properties.* Examples of such properties are *soundness*, *completeness* and *decidability.* A sound formalism guarantees that every statement provable in the formalism is true in all interpretations. No false statements can be deduced. Applied to an inconsistency detection formalism, any inconsistency that can be proved by the formalism is an inconsistency. A formalism is complete if every statement that is true in all interpretations, can be proved. Applied to an inconsistency detection formalism, this means that every inconsistency can be proved by the formalism. A formalism is decidable, if there exists an effective procedure that will determine for every statement, whether or not this statement is provable. Any inconsistency that is expressible in a decidable inconsistency detection formalism can be proven to be valid or not.

For example, some inconsistencies can be detected by navigation over the metamodel, resulting in the execution of queries over the user-defined models. A sound and complete query formalism will guarantee that every inconsistency found is an inconsistency and the complete answer set is returned. Other inconsistencies are defined in the context of semantical concepts of the sublanguages and can be detected by reasoning over and across the semantic domains of these languages. Such inconsistencies defined in our classification make use of SD traces, call sequences and validity of constraints specified on sequence and protocol state machine diagrams. Inconsistencies can be defined as violations of properties of these semantical concepts. Any property is said to be decidable if there exists an effective reasoning procedure that will determine whether or not the property holds.

An inconsistency detection and resolution formalism needs to be able to precisely define inconsistencies and detect them, preferably exhibiting some formal properties.

4.5.3 Criterion #3: Precise Definitions and Management of Interactive Inconsistency Resolutions

As a first requirement, the formalism must *enable the definition of resolution actions*. These resolution actions change the model into a consistent or a less inconsistent model. As observed in this chapter, these actions can depend on the nature of the inconsistency in question or on the causes of the inconsistency.

The activity of resolving an inconsistency by executing resolution actions is a highly interactive process. A formalism for inconsistency resolution must be able *to cope with this interactivity*.

Inconsistency resolution can introduce new inconsistencies. A resolution formalism must *take into account these dependencies*, this is denoted by the word *management* used in the description of this criterion. Suppose certain inconsistencies are detected in a model, the user can decide to resolve one of the inconsistencies. This resolution can introduce new inconsistencies, because the dependencies between different inconsistency resolutions. As a result, different scenario's are possible for the resolution of the different detected inconsistencies.

An inconsistency detection and resolution formalism needs to be able to define resolution actions and enable management of interactive inconsistency resolutions.

Criterion	**Requirements**		
#1	*Abstract syntax representation* e.g., UML: (1) describe classes and their attributes, (2) relationships between these classes, (3) generalisations between the classes, (4) some constraints on these concepts.		*Semantics definition* e.g., UML: call sequences, SD traces.
#2	*Definition of inconsistencies* in a precise way		*Detection of inconsistencies* formal properties: soundness completeness decidability
#3	*Definition of resolution actions*	*Interactivity* to a high degree	*Dependencies* must be supported

Table 4.3: Summary of the requirements for the key criteria.

4.5.4 Tool Support Requirements

Next to the requirements specified by the criteria, tool support requirements can be identified too. These requirements are listed in this section.

Requirements for tool support for inconsistency detection are: it must be easy to create and remove rules for the detection of inconsistencies; the software developer must be able to decide when to check for inconsistencies and which inconsistencies must be checked or whether the detection of certain sets of inconsistencies get priority over the detection of other sets of inconsistencies.

A first tool support requirement for the support of inconsistency resolution calls for the possibility for the software developer to customise the definition of the resolution actions. Consequently, it must be possible to add, remove and modify resolution actions.

Furthermore, the execution process of the resolution actions is demanded to be highly customisable. The software developer has the responsibility to determine which resolution actions will be executed and which not. This indicates the need for an ordering and grouping mechanism for the execution of the resolution actions.

4.6 Conclusion

In this chapter we discussed some strategies for inconsistency handling in inconsistency management and we also discussed on how to preserve consistency in a consistency maintenance context. In this dissertation we focus on resolution actions because the general context of our work is inconsistency management rather than consistency maintenance. A resolution action or a combination of resolution actions resolves a certain inconsistency. Different resolution actions based on the metamodel of the UML fragment are introduced and linked to our classified inconsistencies.

Inconsistency management is crucial in the development of large and complex systems. CASE tools should include support for inconsistency management. Currently this support is limited and the approach to inconsistency detection and resolution is rather ad-hoc. We discussed some key criteria embodying some requirements for an inconsistency detection and resolution formalism. We also introduced some tool support requirements. In this work, focus will rather be on the requirements of our key criteria than on the tool support requirements. Tool support is essential but in this work, it serves as a proof-of-concept of our ideas on inconsistency management and a possible formalism.

With this chapter we finish the part of this dissertation on the formalisation of our UML fragment and the classification of inconsistency definitions and the definitions of resolution actions. We introduced a lightweight formalisation for a relevant fragment of the UML metamodel and we introduced some semantical concepts. As a result, semantics are given to a part of the UML. Next, a classification of different inconsistencies is defined. The inconsistencies contained in this classification are a basic set which can be extended for different applications. Based on the formalisation, the different inconsistencies are well-defined. We discussed the activity of resolving inconsistencies in the context of our classification and distilled three key criteria for inconsistency detection and resolution formalisms.

The question remains which formalism can be used for the detection and resolution of inconsistencies and to which extent does it fulfil the requirements listed in this chapter. This is the subject of the next part of this dissertation.

Chapter 5

Introducing Description Logics

In this chapter, we elucidate the formalism used for inconsistency management in our work. We start this chapter by explaining why a logic formalism is selected (Section 5.1) and more specifically, why Description Logics (DLs) are chosen (Section 5.2). This motivation of the choice of formalism is followed by an introduction to DLs (Section 5.3) and the reasoning tasks they support (Section 5.4). This introduction is based on "The Description Logic Handbook" [BCM+03], on the Ph.D. thesis of Carlos Areces [Are00] and on overview papers by Sattler *et al.* [Sat03, BHS05].

DLs are a family of logic-based knowledge representation formalisms with expressiveness as the characterising factor. We will discuss the main expressive means and show, without going into detail, how they can be used for the representation of UML models (Section 5.5). The higher the expressive power of a DL, the more complex the reasoning tasks become. We discuss the complexity of the logics obtained by combining different expressive means (Section 5.6).

Because a DL is a logic-based representation formalism, it is well-suited to express the static structure of knowledge. However, some DLs have a one-to-one mapping to dynamic logics such as *Modal Logic*. We also report on this mapping because it will enable the reader to understand the different representations of UML models as presented in chapter 6 (Section 5.7).

Several DL systems are implemented. All the systems are designed and implemented taking different positions on the requirements of expressive power, completeness of reasoning, and tractability of reasoning. Some of the early systems, some successor systems and a new optimised generation of very expressive systems are discussed (Section 5.8).

5.1 Why Logic Formalism?

For the representation of the metamodel and the semantics of the different UML sublanguages, a formal and sound foundation is required. For this representation and the detection of UML inconsistencies, we choose a logic-based approach. This choice is based on the following observations:

- The declarative nature of logic is well suited to express the design models that are also of a declarative nature.

- Logics have a declarative semantics, e.g., the Tarskian semantics of predicate logic is the prototype of a declarative semantics. The logic reasoning algorithms are well understood due to the extensively studied, well-defined and sound semantics. First-order logic and theorem proving have been proposed by several authors for expressing software models and the derivation of inconsistencies from these models resp. (e.g., [FGH+93], [NKF94], [NJJ+96], [EFA+99]). Most of these techniques operationalise the consequence relation ($\vdash$) by using theorem proving based on the standard inference rules of classical logic such as resolution, conjunct and negation elimination, instantiation of universally quantified formulas, and other rules for introducing negative information in the models such as the closed-world assumption (CWA). For a survey of the logic-based approaches for detecting inconsistencies we refer to Chapter 7 and [SZ01].

- Some logic reasoning engines can deduce implicitly represented knowledge from the explicit knowledge allowing an adequate treatment of incomplete or subjective or time-dependent knowledge.

5.2 Why Description Logics?

Spanoudakis *et al.* [SZ01] identified two inherent limitations of logic-based approaches in the context of inconsistency management: (i) first-order logic is semi-decidable, hence it is not possible to provide for semantically adequate inference procedures, and (ii) theorem proving is computationally inefficient. Description Logics are an attempt to overcome both problems by restricting the expressive power of the logic language.

Other qualities of DLs that make it suitable for our work, are:

- DLs are designed to represent knowledge. Knowledge representation systems (KRS) are focused on providing high-level descriptions of problem domains, in order to allow the discovery of implicit consequences of the explicitly represented knowledge. These are the so-called "terminological representation systems" that permit the definition of a terminology describing the domain being modelled, using a representation language. Once the domain representation has been established, as concepts and relations between them, it is possible to reason about the individuals of the modelled world. Some of these KRSs use DLs as a representation language. As such, DLs are suited to express knowledge about the static structure of a software application. For example, [CCDL01] translated UML class diagrams to the description logic $\mathcal{DLR}$.

- DLs are based on formal semantics such as *descriptive* semantics [Neb91], which are well-studied and understood.

- A wide range of logics are developed, from very simple ones to very expressive ones combining different expressive means as explained in Section 5.5. As such, we can choose the logic that is most suited for our goals.

  In the last decade, a lot of work investigated DLs w.r.t. their expressive power and computational complexity. This implies that the computational space has been thoroughly mapped out. An overview of the complexity of the different developed DLs is given in Section 5.6.

- The most important feature of a DL is its reasoning ability. This reasoning allows us to infer knowledge that is implicitly present in the knowledge base (shortly, KB). Concepts are classified according to subconcept-superconcept relationships, e.g., `Model` is a `Element`. In this case, `Model` is a subconcept of `Element` and `Element` is the superconcept of `Model`. Section 5.4 provides a detailed overview of the different reasoning tasks of a DL. The inference problems on which the reasoning tasks are based, are decidable.

- Even with all the expressive power of first-order logic, it is not possible to define the transitive closure of a relation in first-order logic. In [BMP+02] this is also recognised as a deficiency of OCL. The well-formedness rules of the UML metamodel which are expressed in OCL make heavy use of additional operations to navigate over the metamodel. These operations are often recursive and this could be avoided if it was possible to express transitive closure in OCL [BMP+02]. Most expressive DLs provide a mechanism to define the transitive closure of a role.

- There is a close relation between description logics and modal logics [BdV01]. For example, there is a one-to-one mapping between a certain DL (called $\mathcal{ALCI}_{reg}$ [Baa91]) and *converse*-PDL [FL79]. This indicates that DLs are also suited to express to a certain extent, behaviour specifications of a software application.

- The DL community is well organised. There have been a series of International Description Logics Workshops (DL) which are associated with major AI conferences. Workshop submissions are thoroughly reviewed by a selected, international program committee. Next to these series of workshops, there is also a yearly workshop on Applications of Description Logics (ADL). Both series of workshops are the ideal place to present our ideas on using DLs and to get feedback from the DL community.

- A wide variety of DL systems have been built. We will discuss them in Section 5.8.

5.3 Concepts, Roles and Knowledge Bases

Description logics (DLs) are a family of logic-based knowledge representation formalisms designed to represent and reason about the knowledge of an application domain in a structured and well-understood way.

The basic notions in DLs are *concepts* and *roles* and a specific DL is mainly characterised by the *constructors* it provides to form complex concepts and roles from atomic ones. The following concept, intuitively, represents a model consisting of classes that have only operations which are abstract.

$$Model \sqcap \exists\, ownedMember.(Class \sqcap \forall\, ownedOperation.(Operation \sqcap \exists\, isAbstract.\top))$$

Different combinations of constructors generate languages with different expressiveness. Table 5.1[Are00] shows a summary of the most common DL constructors, including their semantics. Using these concept and roles constructors, complex concepts and roles can be formed.

Historically, a number of description logics have received a special name. The language $\mathcal{FL}^-$ [BL84] is defined as the description logic allowing universal quantification, conjunction

Constructor	**Syntax**	**Semantics**
concept name	C	$C^{\mathcal{I}}$
top	$\top$	$\Delta^{\mathcal{I}}$
negation ($\mathcal{C}$)	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
conjunction	$C_1 \sqcap C_2$	$C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
disjunction ($\mathcal{U}$)	$C_1 \sqcup C_2$	$C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$
universal quant.	$\forall r.C$	$\{d_1 \mid \forall d_2 \in \Delta^{\mathcal{I}}.((d_1, d_2) \in r^{\mathcal{I}} \rightarrow d_2 \in C^{\mathcal{I}})\}$
existential quant. ($\mathcal{E}$)	$\exists r.C$	$\{d_1 \mid \exists d_2 \in \Delta^{\mathcal{I}}.((d_1, d_2) \in r^{\mathcal{I}} \wedge d_2 \in C^{\mathcal{I}})\}$
unqualified number	$(\geq n\ r)$	$\{d_1 \mid \lvert\{d_2 \mid (d_1, d_2) \in r^{\mathcal{I}}\}\rvert \geq n\}$
restriction ($\mathcal{N}$)	$(\leq n\ r)$	$\{d_1 \mid \lvert\{d_2 \mid (d_1, d_2) \in r^{\mathcal{I}}\}\rvert \leq n\}$
qualified number	$(\geq n\ r.C)$	$\{d_1 \mid \lvert\{d_2 \mid (d_1, d_2) \in r^{\mathcal{I}} \wedge d_2 \in C^{\mathcal{I}}\}\rvert \geq n\}$
restriction ($\mathcal{Q}$)	$(\leq n\ r.C)$	$\{d_1 \mid \lvert\{d_2 \mid (d_1, d_2) \in r^{\mathcal{I}} \wedge d_2 \in C^{\mathcal{I}}\}\rvert \leq n\}$
functional number	$(\leq 1\ r)$	$\{d_1 \mid \lvert\{d_2 \mid (d_1, d_2) \in r^{\mathcal{I}}\}\rvert \leq 1\}$
restriction ($\mathcal{F}$)		
one-of ($\mathcal{O}$)	$\{a_1, \ldots, a_n\}$	$\{a_1, \ldots, a_n\}^{\mathcal{I}} = \{a_1^{\mathcal{I}}, \ldots, a_n^{\mathcal{I}}\}$
role filler ($\mathcal{B}$)	$\exists r.\{a\}$	$\{d \mid (d, a^{\mathcal{I}}) \in r^{\mathcal{I}}\}$
role name	r	$r^{\mathcal{I}}$
role conjunction (∇)	$r_1 \sqcap r_2$	$r_1^{\mathcal{I}} \cap r_2^{\mathcal{I}}$
role hierarchy ($\mathcal{H}$)	$r_1 \sqsubseteq r_2$	$r_1^{\mathcal{I}} \subseteq r_2^{\mathcal{I}}$
inverse roles ($\mathcal{I}$)	r^{-1}	$\{(d_1, d_2) \mid (d_2, d_1) \in r^{\mathcal{I}}\}$

Table 5.1: Common DL operators.

and unqualified existential quantifications of the form $\exists R.\top$. $\mathcal{FL}^-$ was proposed as a formalisation of the core notions of Minsky's frames. Concept conjunction is implicit in the structure of a frame, which requires a set of conditions to be satisfied. Role quantifications allow one to characterise slots: the unqualified existential state the existence of a value for a slot, while the universal quantifier requires that the values of a slot satisfy a certain condition.

The logic $\mathcal{AL}$ (= attributive language) [SSS91] extends $\mathcal{FL}^-$ with negation of atomic concepts. It is customary to define particular $\mathcal{AL}$-languages by postfixing the names of these original systems with the names of the added operators from table 5.1. For example, the logic $\mathcal{ALC}$ is $\mathcal{AL}$ extended with full negation.

It is possible to introduce names for complex concepts. For example, for the concept defined above, we can introduce the name $ModelwithAbstractClasses$. A *Tbox* is used to introduce names, i.e., abbreviations, for complex concepts. More expressive *Tbox* formalisms allow the expression of so-called *general concept inclusion axioms* (GCI's). The following GCI specifies that only classes with abstract operations can be abstract.

$$Class \sqcap \exists\, isAbstract.\top \sqsubseteq \exists\, ownedOperation.(Operation \sqcap \exists\, isAbstract.\top)$$

Definition 59 *[Sat03] Let* $\mathbf{C}$ *and* $\mathbf{R}$ *be disjoint sets of concept and role names. The set of* $\mathcal{ALC}$*-concepts is the smallest set such that each concept name* $A \in \mathbf{C}$ *is an* $\mathcal{ALC}$*-concept and, if* C *and* D *are* $\mathcal{ALC}$*-concepts and* $r \in \mathbf{R}$ *is a role name, then*

$$\neg C, C \sqcap D, C \sqcup D, \exists r.C, \text{ and } \forall r.C \text{ are also } \mathcal{ALC}\text{-concepts.}$$

A general concept inclusion axiom (GCI) *is of the form $C \sqsubseteq D$ for C, D $\mathcal{ALC}$-concepts. C is called a subconcept or child of D and D is called a superconcept or parent of C. The notation $C \doteq D$ is used for $C \sqsubseteq D$ and $D \sqsubseteq C$.*

An interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ *consists of a non-empty set $\Delta^{\mathcal{I}}$, the interpretation* domain *and a mapping $\cdot^{\mathcal{I}}$ which associates, with each concept name A, a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and, with each role name r, a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.*

The interpretation of complex concepts is defined in Table 5.1.

Let $I = \{a, b, c \ldots\}$ be a set of individual names. *An* assertion *is of the form $a : C$, (then a is called an* instance of *C), or $(a, b) : r$ (then b is called an* r-successor *of a) for $a, b \in I$, a role r and a concept C.*

$\top$ is used as an abbreviation for $A \sqcup \neg A$, $\bot$ for $\neg\top$, $C \Rightarrow D$ for $\neg C \sqcup D$, and $C \Leftrightarrow D$ for $(C \Rightarrow D) \sqcap (D \Rightarrow C)$.

DLs split the available knowledge about a particular situation into *terminological knowledge* and *assertional knowledge*.

Definition 60 *A* knowledge base Σ *in a DL is a pair $\Sigma = (\mathcal{T}, \mathcal{A})$ such that*

- *$\mathcal{T}$ is the T(erminological)-Box, a finite, possibly empty set of GCIs.*
- *$\mathcal{A}$ is the A(ssertional)-Box, a finite, possibly empty set of instances and r-successors. Formulas in $\mathcal{A}$ are called* assertions.

A *Tbox* can be divided into the following two, disjoint parts following [Sat03].

Background Knowledge GCIs of the form $C \sqsubseteq D$ for C and D complex concepts can be used to formalise background knowledge of the application domain and to constrain the set of models.

Definitorial Part For each concept relevant in the application domain, we can introduce an atomic concept name A and a concept definition $A \sqsubseteq C$ or $A \doteq C$ for C a complex concept describing necessary or necessary and sufficient conditions for individuals to be an instance of A. A is called a *primitively defined* or a *defined* concept. The axioms in the definitorial part are acyclic.

Tbox was originally thought of as a set of definitions and some restrictions were imposed on the *Tbox*. The two most important restrictions were, *simple terminological axioms* and *acyclic definitions*. A simple terminological axiom is a concept inclusion axiom, $C \sqsubseteq D$, where C is an atomic concept (belonging to $\mathbf{C}$, see Definition 59). Acyclic definitions occur when the graph obtained by assigning a node n_A to each atomic concept A in the *Tbox* $\mathcal{T}$ and drawing an arrow between two nodes n_A and n_B if there is an axiom $C_1 \sqsubseteq C_2$ in $\mathcal{T}$ such that A appears in C_1 and B appears in C_2, does not contain cycles.

Suppose we want to capture the information contained in the following quote:

> *"A protocol transition specifies a legal transition for an operation. Transitions of protocol state machines have the following information: a precondition (guard), on trigger, and a postcondition."*
>
> From UML 2.0 Superstructure document, pg. 482.

The following *Tbox* captures the information contained in the previous paragraph.

$$
\begin{array}{rcl}
\exists\, referred.Operation & \sqsubseteq & ProtocolTransition \\
ProtocolTransition & \sqsubseteq & \exists\, referred.Operation \sqcap \exists\, precondition.Constraint \\
 & & \sqcap \exists\, postcondition.Constraint \sqcap \exists\, trigger.Operation \\
 & & \sqcap (\leq 1 trigger) \\
\exists\, precondition.Constraint & \sqsubseteq & ProtocolTransition \\
\exists\, postcondition.Constraint & \sqsubseteq & ProtocolTransition \\
\exists\, trigger.Operation & \sqsubseteq & ProtocolTransition
\end{array}
$$

In order to specify the different reasoning tasks usually considered in DLs, we specify what it means to satisfy axioms.

Definition 61

- *An interpretation $\mathcal{I}$ satisfies a GCI $C \sqsubseteq D$, (denoted by $\mathcal{I} \models C \sqsubseteq D$) if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.*
- *If $\mathcal{I}$ satisfies all GCIs in $\mathcal{T}$, it is called a* model of $\mathcal{T}$ *($\mathcal{I} \models \mathcal{T}$).*
- *For* Aboxes*, an interpretation maps each individual $a \in I$ to some element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. An interpretation $\mathcal{I}$ satisfies an assertion*
  - *$a : C$ ($\mathcal{I} \models a : C$) iff $a^{\mathcal{I}} \in C^{\mathcal{I}}$ and*
  - *$(a, b) : r$ ($\mathcal{I} \models (a, b) : r$) iff $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$.*
- *An* Abox *$\mathcal{A}$ is consistent with respect to $\mathcal{T}$ iff there exists a model $\mathcal{I}$ of $\mathcal{T}$ that satisfies each assertion in $\mathcal{A}$. If $\mathcal{I}$ satisfies $\mathcal{A}$, then $\mathcal{I}$ is called a model of $\mathcal{A}$ ($\mathcal{I} \models \mathcal{A}$).*

Due to their semantics, DLs can be identified as fragments of first-order predicate logic. Atomic concepts and roles are unary, resp. binary predicates, since an interpretation $\mathcal{I}$ assigns to every atomic concept and role a unary, resp. binary relation over $\Delta^{\mathcal{I}}$. The translation φ from $\mathcal{ALC}$-concepts to first-order logic formulas is specified as follows (Chapter 4 in [BCM+03]):

$$
\begin{array}{rcl}
\varphi^x(A) & = & P_A(x) \\
\varphi^x(\neg C) & = & \neg \varphi^x(C) \\
\varphi^x(C \sqcap D) & = & \varphi^x(C) \wedge \varphi^x(D) \\
\varphi^x(C \sqcup D) & = & \varphi^x(C) \vee \varphi^x(D) \\
\varphi^x(\exists r.C) & = & \exists y.P_r(x, y) \wedge \varphi^y(C) \\
\varphi^x(\forall r.C) & = & \forall y.P_r(x, y) \rightarrow \varphi^y(C)
\end{array}
$$

where φ^x and φ^y are identical but swapping the positions of y and x.

Let C be a concept and $\mathcal{T}$ a (general or acyclic) *Tbox*.

$$
\varphi(C, \mathcal{T}) = \varphi^x(C) \wedge \forall x. \bigwedge_{D \doteq E \in \mathcal{T}} \varphi^x(D) \leftrightarrow \varphi^x(E)
$$

Individual names $a \in I$ appearing in an *Abox* $\mathcal{A}$ map to first-order logic constants c_a. And assertions are translated as follows:

$$\begin{aligned} \varphi^x(a:C) &= \varphi^x(C)[c_a] \\ \varphi^x((a,b):r) &= P_r(c_a, c_b) \\ \varphi^x(\mathcal{A}) &= \bigwedge_{\beta \in \mathcal{A}} \varphi^x(\beta) \end{aligned}$$

5.4 Reasoning Tasks

In DLs, we want to perform inferences given a certain knowledge base. *Tbox* reasoning is used to refer to the ability to perform inferences from a *Tbox*, and similarly, *Abox* reasoning refers to performing inferences from an *Abox*.

The standard reasoning tasks considered in DLs are for a *Tbox*: *subsumption, knowledge base satisfiability, concept satisfiability* and for an *Abox*: *instance checking* and *relation checking*. These standard reasoning tasks can be defined as follows:

Definition 62 *[Are00] Let Σ be a knowledge base, C, $D \in \mathbf{C}$, $r \in \mathbf{R}$ and $a, b \in I$, we define the following* reasoning tasks*:*

- Subsumption, $\Sigma \models C \sqsubseteq D$
  Check whether for all interpretations $\mathcal{I}$ such that $\mathcal{I} \models \Sigma$, we have $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.
- Instance checking,$\Sigma \models a : C$
  Check whether for all interpretations $\mathcal{I}$ such that $\mathcal{I} \models \Sigma$, we have $a^{\mathcal{I}} \in C^{\mathcal{I}}$.
- Relation checking, $\Sigma \models (a, b) : r$
  Check whether for all interpretations $\mathcal{I}$ such that $\mathcal{I} \models \Sigma$, we have $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$.
- Concept satisfiability, $\Sigma \not\models (C \doteq \bot)$
  Check whether for some interpretation $\mathcal{I}$ such that $\mathcal{I} \models \Sigma$, we have $C^{\mathcal{I}} \neq \{\}$. In this case, $\mathcal{I}$ is called a model *of C.*
- Knowledge base satisfiability, $\Sigma \not\models \bot$ *Check whether a knowledge base is satisfied, i.e., if there exists an interpretation $\mathcal{I}$ such that $\mathcal{I} \models \Sigma$. (In this case, $\mathcal{I}$ is called a* model *of Σ).*

The subsumption algorithm determines subconcept-superconcept relationships: a concept C is subsumed by a concept D with respect to a *Tbox* if, in each model of the *Tbox*, each instance of C is also an instance of D. Such an algorithm can be used to compute the *classification* of a *Tbox*. The problem of computing the most specific *concept names* in a *Tbox* that subsume a certain concept is known as computing the *parents of a concept*. The *children* are the most general concept names in a *Tbox* that are subsumed by a certain concept. The term *concept ancestors (concept descendants)* is used to denote the transitive closure of the parents (children) relation between concepts. The computation of all the children and parents of each concept is called *classification*. This inference is needed to build a hierarchy of concepts names.

The basic reasoning task in an *Abox* is instance checking which verifies whether a given individual is an instance of a specified concept. An individual is inconsistent with respect to a *Tbox* if it is classified as an instance of the empty concept $\perp$. Relation checking determines whether two individuals of an *Abox* stand in a given relation.

Checking satisfiability of concepts is a key inference. Subsumption and satisfiability can be mutually reduced to each other: C is satisfiable with respect to a *Tbox* $\mathcal{T}$ if and only if C is not subsumed by $\perp$ with respect to $\mathcal{T}$, and $C \sqsubseteq_{\mathcal{T}} D$ if and only if $C \sqcap \neg D$ is not satisfiable with respect to $\mathcal{T}$.

Based on Definition 62, the notion of *role fillers with respect to an individual* can be defined.

Definition 63 *The* set of fillers of a role, r, with respect to an individual a *is the set* $\{b \mid \Sigma \models (a, b) : r\}$.

Research on description logics has focused mainly on understanding the relations between the reasoning tasks mentioned above, and on establishing their computational complexity. The study of the computational behaviour of DLs has provided a good understanding of the properties of the language constructs and their interaction. Before giving an overview of the computational complexity, we will present an overview of expressive means used in DLs.

5.5 Expressive Means in DLs

In this section, we discuss a variety of expressive means commonly used in DLs. This discussion aims at providing the reader with an understanding on the expressiveness of DLs and preparing the reader on how this formalism is used in our work and what the restrictions of usage are. This discussion is based on [Sat03] and [BCM+03].

5.5.1 *Tboxes*

Tboxes are defined in the previous section. They can be divided in a background knowledge part and a definitorial part. Some DLs only allow for the definitorial part and possibly require this part to be free of cycles. A *Tbox* contains a cycle if there exists an atomic concept that uses itself. Otherwise, a *Tbox* is called acyclic. We say that an atomic concept A uses an atomic concept B if B appears on the right-hand side of the definition of A, and *uses* is defined as the transitive closure of the relation *directly uses*.

Reasoning with respect to acyclic concept definitions can be reduced to pure concept reasoning. However, it turned out that, for a variety of logics, reasoning with respect to acyclic concept definitions is as complex as pure concept reasoning.

5.5.2 Number Restrictions

Number restrictions are a popular expressive means in DLs. Almost all implemented DL systems provide number restrictions. Those restrictions are concepts of the form $(\geq n\ r.C)$ (at least restriction) or $(\leq n\ r.C)$ (at most restriction), for n a positive integer, r a role and C a concept and are called *Qualified Number Restrictions*. In a simpler form, called *Unqualified Number Restrictions*, these number restrictions only allow the concept $\top$ instead

of the concept C above. The interpretation of (un)qualified number restrictions are given in Table 5.1.

A *functional role* is a role for which each individual can only have up to one successor.

As an example, consider two concepts *Bank* and *Account* and a role *belongs_to* between *Account* and *Bank*. The concept $(\leq 1\ belongs\_to.Bank)$ expresses that each individual has at most one successor that is an instance of *Bank* for the role *belongs_to*. The general inclusion axiom $Account \sqsubseteq (\leq\ 1\ belongs\_to.Bank)$ represents in this case the multiplicity restriction that an *Account belongs_to* at most one *Bank*.

5.5.3 Inverse Roles

It is useful and necessary in most applications that roles are bidirectional. For example, we not only want to use the role $hasProperty$ but also the role $isPropertyOf$. To ensure that $< x, y > \in isPropertyOf^{\mathcal{I}}$ if and only if $< y, x > \in hasProperty^{\mathcal{I}}$, some DLs provide the notion of inverse roles. For a role name r, r^- is the inverse role name and this is interpreted as specified in Table 5.1.

Inverse roles are a necessary expressive means in the context of UML. Suppose that an association is represented in a DL by a role. If this association is bidirectional, the role representing this association must be bidirectional too. This constraint requires the existence of the inverse of the role.

5.5.4 Transitive Roles

Transitive roles are special role names that have to be interpreted as transitive relations. If two pairs of individuals i_1 and i_2 and i_2 and i_3 are related via a transitive role r, then i_1 and i_3 are also related via r. Transitive roles are used to model transitive relations such as $isPartOf$.

Another way to extend DLs with transitivity is to allow for the transitive closure operator on roles, i.e., to allow for roles r^*, where $(r^*)^{\mathcal{I}}$ is to interpreted as the transitive closure of $r^{\mathcal{I}}$.

The transitive closure of roles can be used in the UML context for describing the full descriptor of a class.

5.5.5 Role Inclusion Axioms

Role inclusion axioms (RIAs) which are of the form $r \sqsubseteq s$ for r, s roles, are another expressive means on roles. Such axioms are used to introduce subroles and in the case of $r \sqsubseteq s$, r is mapped to a subrelation of s. A *role hierarchy* is a finite set of role inclusion axioms. An interpretation $\mathcal{I}$ satisfies a role hierarchy $\mathcal{R}$ if and only if $r^{\mathcal{I}} \subseteq s^{\mathcal{I}}$ for each $r \sqsubseteq s$ in $\mathcal{R}$. Such an interpretation is called a *model of* $\mathcal{R}$. Remark that if inverse roles are present, these axioms can be used to express symmetric roles using $r \sqsubseteq r^-$ and $r^- \sqsubseteq r$.

Role hierarchies are a weakened form of role intersection, replacing the role expression $r_1 \sqcap r_2$ with a new role name $r_{1,2}$ and adding to the *Tbox* the expressions $r_{1,2} \sqsubseteq r_1$ and $r_{1,2} \sqsubseteq r_2$ which are a weakened form of intersection, since $r_{1,2}^{\mathcal{I}} \subseteq r_1^{\mathcal{I}} \cap r_2^{\mathcal{I}}$.

Role inclusion axioms can also be used to yield a weakened form of transitive closure. Suppose s is a transitive role then $r \sqsubseteq s$ yields a weakened form of transitive closure, because

s is interpreted as *a* transitive role containing r, while r^* is to be interpreted as the *smallest* transitive role containing r.

Role inclusion axioms can for example be used to keep track of ancestors of classes. Consider a transitive role *ancestors* and a role *parent* with $parent \sqsubseteq ancestors$. From the *Abox* assertions $(class_1, class_2) : parent$ and $(class_2, class_3) : parent$, $(class_1, class_3) : ancestors$ is deduced automatically.

5.5.6 General Role Inclusion Axioms

General Role Inclusion Axioms (g-RIAs) are of the form $r_1 \dots r_n \sqsubseteq s_1 \dots s_m$ for r_i, s_j role names and are a generalisation of the above introduced role inclusion axioms. An interpretation $\mathcal{I}$ is a model of such a g-RIA if and only if $r_1^{\mathcal{I}} \circ \dots \circ r_n^{\mathcal{I}} \subseteq s_1^{\mathcal{I}} \circ \dots \circ s_m^{\mathcal{I}}$, where $\circ$ is the composition operator on binary relations. A local form of g-RIAs are *role value maps*, which are concepts of the form $r_1 \dots r_n \Rightarrow s_1 \dots s_m$ with semantics

$$(r_1 \dots r_n \Rightarrow s_1 \dots s_m)^{\mathcal{I}} = \{x \mid \forall y.(x,y) \in r_1^{\mathcal{I}} \circ \dots \circ r_n^{\mathcal{I}} \Rightarrow (x,y) \in s_1^{\mathcal{I}} \circ \dots \circ s_m^{\mathcal{I}}\}.$$

It is shown that subsumption becomes undecidable when adding general role inclusion axioms or role value maps to, even a very weak, description logic [SS89]. However, in many applications it would be useful to be able to express *propagation of properties*, for example we want to express that, if a *PSM* owns a *Region* that has some *State*s as its parts, the PSM also owns these *State*s.

For expressing propagation of properties, only axioms of the form $r \circ s \sqsubseteq s$ or $s \circ r \sqsubseteq s$ are needed. However, extending expressive DLs with these forms of axioms yields an undecidable logic [HS03]. These forms of role inclusion axioms can be further restricted: the role hierarchies are restricted such that they do not contain, what is called *affecting cycles* of length greater than one [Sat03]. *Affecting* is the transitive closure of *direct affecting*. A role name r directly affects a role name s if $r \circ s \sqsubseteq s$, $s \circ r \sqsubseteq s$ or $r \sqsubseteq s$ is contained in the role hierarchy. An *acyclic* role axiom is a role axiom containing no affecting cycles of length greater than one.

5.5.7 Concrete Domains

In many applications it is necessary to refer to concrete domains and predefined predicates on these domains when defining concepts. Consider as an example the primitive types introduced in chapter 2. These are used in the UML metamodel and can also be used in user models. It must be possible to express in a UML class diagram that a certain attribute is of a primitive type and has a certain value. Consider an *Account* class with the property *number* representing the account's number. This number is of type `Integer` and has a certain value.

Next to the definition of a domain, e.g., `Integer`, restrictions on the values of the domain must be allowed. Assume we want to define the concept *Customer* in DL. A person can in some context only be a customer if his age is greater than or equal to 16. To express this in DL, we want to define a new role *has_age* and define $Customer$ by $Person \sqcap \exists has\_age.\geq_{16}$. $\geq_{16}$ stands for the unary predicate $\{n | n \geq 16\}$ of all nonnegative integers greater than or equal to 16.

We now introduce the family of DLs $\mathcal{ALC}(\mathcal{D})$, but first the notion of a concrete domain is defined.

Definition 64 *[BCM+03] A* concrete domain $\mathcal{D}$ *consists of a set* $\Delta^{\mathcal{D}}$*, the domain of* $\mathcal{D}$*, and a set* $pred(\mathcal{D})$*, the predicate names of* $\mathcal{D}$*. Each predicate name* $P \in pred(\mathcal{D})$ *is associated with an arity* n*, and an* n*-ary predicate* $P^{\mathcal{D}} \subseteq (\Delta^{\mathcal{D}})^n$*.*

As an example, consider the concrete domain $\mathcal{N}$ which has as its domain the set of all nonnegative integers $\mathbb{N}$, and $pred(\mathcal{N})$ consists of the binary predicate names $<, \leq, \geq, >$ and the unary predicate names $<_n, \leq_n, \geq_n, >_n$ for $n \in \mathbb{N}$ which are interpreted by predicates on $\mathbb{N}$.

The extended language $\mathcal{ALC}(\mathcal{D})$ should still have decidable reasoning tasks and this requirement adds additional restrictions. The set of predicate names of the concrete domain must be *closed under negation.* This implies that, if P is an n-ary predicate name in $pred(\mathcal{D})$ then there exists a predicate name $Q \in pred(\mathcal{D})$ such that $Q^{\mathcal{D}} = (\Delta^{\mathcal{D}})^n \setminus P^{\mathcal{D}}$. There also must be a unary predicate name that represents $\Delta^{\mathcal{D}}$.

We also have to determine what the *satisfiability problem* for the concrete domain $\mathcal{D}$ is. Consider the conjunction

$$\bigwedge_{i=1}^{k} P_i(\underline{x}^{(i)})$$

where $P_1, \ldots, P_k$ are k predicate names in $pred(\mathcal{D})$ of arities $n_1, \ldots, n_k$ and $\underline{x}^{(i)}$ stands for an n_i-tuple $(x_1^{(i)}, \ldots, x_{n_i}^{(i)})$ of variables. Deciding on the satisfiability of such finite conjunctions is called the *satisfiability problem* for $\mathcal{D}$.

$\mathcal{ALC}(\mathcal{D})$ extends $\mathcal{ALC}$ in two ways. First of all, the set of role names is now partitioned into a set of functional roles and a set of ordinary roles. Both types of roles may occur in universal and existential quantification. There is also a new constructor, called the *existential predicate restriction.* The syntax of this constructor is defined as follows:

Definition 65 *The* existential predicate restriction *which defines a new complex concept, has the following syntax:* $\exists(u_1, \ldots, u_n).P$ *where* P *is an* n*-ary predicate of* $\mathcal{D}$ *and* $u_1, \ldots, u_n$ *are chains of functional roles.*

Concrete predicates $P \in Pred(\mathcal{D})$ give rise to additional *Abox* assertions of the form $P(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are names of concrete individuals. This also implies that in $\mathcal{ALC}(\mathcal{D})$-*ABoxes*, names for abstract and for concrete individuals must be distinguished.

Definition 66 *An* interpretation $\mathcal{I}$ *for* $\mathcal{ALC}(\mathcal{D})$ *consists of a set* $\Delta^{\mathcal{I}}$*, the abstract domain of the interpretation and an interpretation function* $\mathcal{I}$*. The abstract domain of the interpretation and the concrete domain must be disjoint, i.e.,* $\Delta^{\mathcal{D}} \cap \Delta^{\mathcal{I}} = \emptyset$*. Functional roles are now interpreted by partial functions from* $\Delta^{\mathcal{I}}$ *into* $\Delta^{\mathcal{D}} \cup \Delta^{\mathcal{I}}$*.*

The existential predicate restriction is interpreted as follows:

$(\exists(u_1, \ldots, u_n).P)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} | \exists w_1, \ldots, w_n \in \Delta^{\mathcal{D}}$ *such that* $u_1^{\mathcal{I}}(x) = w_1, \ldots, u_n^{\mathcal{I}}(x) = w_n$ *and* $(w_1, \ldots, w_n) \in P^{\mathcal{D}}\}$*, where* $u = f_1 \circ \ldots \circ f_n$ *is a chain of functional roles and* $u^{\mathcal{I}}$ *denotes the composition of the partial functions* $f_1^{\mathcal{I}}, \ldots, f_n^{\mathcal{I}}$*.*

The DL $\mathcal{ALC}(\mathcal{D})$ can be further extended in different directions. One possibility is to use predicate restrictions to define new roles [HLM99]. This means that if P is a predicate of arity $m + n$ and $u_1, \ldots, u_n$ and $v_1, \ldots, v_m$ are chains of functional roles, then $\exists(u_1, \ldots, u_n)(v_1, \ldots, v_m).P$ is a complex role. The semantics of these complex roles is defined as:

$(\exists(u_1,\ldots,u_n)(v_1,\ldots,v_m).P)^{\mathcal{I}} = \{(x,y) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} | \exists w_1,\ldots,w_n,s_1,\ldots,s_m \in \Delta^{\mathcal{D}}$ such that $u_1^{\mathcal{I}}(x) = w_1,\ldots,u_n^{\mathcal{I}}(x) = w_n, v_1^{\mathcal{I}}(y) = s_1,\ldots,v_m^{\mathcal{I}}(y) = s_m$ and $(w_1,\ldots,w_n, s_1,\ldots,s_m) \in P^{\mathcal{D}}\}$.

This extension however has an undecidable satisfiability problem, but in [HLM99] syntactic restrictions are defined on concepts such that the restricted language is closed under negation and has a decidable *Abox* satisfiability problem. This implies that the subsumption and instance problems are also decidable.

5.6 Complexity of Reasoning in DLs

Several reasoning mechanisms are developed for DLs. It is not our aim to give a detailed description of these algorithms, we will only give a very brief overview.

For DLs not allowing negation, so-called *structural subsumption algorithms* can be employed to compute subsumption of concepts. Although these algorithms are of polynomial time, they are only complete for simple DLs.

Tableau-based algorithms are used for expressive DLs to obtain sound and complete satisfiability algorithms and this for a range of DLs extending $\mathcal{ALC}$. For example, in [HS99] such algorithms are used for languages with transitive roles and in [HLM99] for languages with constructors allowing to refer to concrete domains. This approach is also extended to the consistency problem for *Aboxes* [HM00].

There also exist optimal automata-based algorithms to decide for satisfiability of concepts. They allow for an elegant and natural translation of logic and provide ExpTime upper complexity and are optimal for ExpTime-hard logics.

5.6.1 $\mathcal{SHIQ}$

Starting from the DL $\mathcal{ALC}$, we will show how the different expressive means as introduced in the previous section affect the complexity of the different resulting DLs. The accumulation of those expressive means results into the expressive logic $\mathcal{SHIQ}$ used by current state-of-the-art DL systems such as FacT and RACER. A summary of the complexity results for the different DLs can be found in Table 5.2.

Pspace	ExpTime	NExpTime
$\mathcal{ALC}$	$\mathcal{ALC}$	
(w.r.t. empty *Tboxes*)	(w.r.t. general *Tboxes*)	
$\mathcal{ALCN}$		
(w.r.t. acyclic *Tboxes*)		
$\mathcal{S}$		
(w.r.t. empty *Tboxes*)		
$\mathcal{SI}$	$\mathcal{ALCQI}$	
(w.r.t. empty *Tboxes*)	(w.r.t. general *Tboxes*)	
	$\mathcal{SHI}$	
	$\mathcal{SHIQ}$	
		$\mathcal{SHIQ}(\mathcal{D})$

Table 5.2: Overview of the complexity of concept satisfiability.

Satisfiability checking and thus subsumption w.r.t. a general *Tbox* is ExpTime-complete [Sch91] in $\mathcal{ALC}$. Concepts are defined in the context of a *Tbox*. In the context of developing reasoning mechanisms, it is, however, conceptually easier to abstract from the *Tbox* or to assume that it is empty [BCM+03]. The satisfiability problem becomes Pspace-complete in $\mathcal{ALC}$ when considered w.r.t. the empty *Tbox* [SSS91].

The logic obtained by extending $\mathcal{ALC}$ with transitive roles is called $\mathcal{S}$ as shown in Table 5.3. This name is chosen because of the relationship of this logic with the proposition (multi) modal logic $\mathbf{S4}_{(m)}$ (More on the relationship between modal logics and DLs can be found in the next section). The addition of transitive roles to $\mathcal{ALC}$ without *Tboxes* yields a DL which is still Pspace-complete [Sat96]. Remark that adding the transitive closure operator on roles yields an ExpTime-complete logic [FL79]. In current DL systems the cheapest option is chosen: the description logic $\mathcal{ALC}$ is extended with transitive roles.

Constructor	**Syntax**	**Semantics**	
atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$	
universal concept	$\top$	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$	
atomic role	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$	
transitive role	$R \in R_+$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^+$	
conjunction	$C_1 \sqcap C_2$	$C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$	
disjunction ($\mathcal{U}$)	$C_1 \sqcup C_2$	$C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$	$\mathcal{S}$
negation ($\mathcal{C}$)	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$	
value restriction	$\forall R.C$	$\{d_1 \mid \forall d_2 \in \Delta^{\mathcal{I}}.((d_1, d_2) \in R^{\mathcal{I}} \rightarrow d_2 \in C^{\mathcal{I}})\}$	
exists restriction ($\mathcal{E}$)	$\exists R.C$	$\{d_1 \mid \exists d_2 \in \Delta^{\mathcal{I}}.((d_1, d_2) \in R^{\mathcal{I}} \wedge d_2 \in C^{\mathcal{I}})\}$	
role hierarchy	$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$	$\mathcal{H}$
inverse role	R^-	$\{(x, y) \mid (y, x) \in R^{\mathcal{I}}$	$\mathcal{I}$
qualified	$(\geq n\ R.C)$	$\{d_1 \mid \lvert\{d_2 \mid (d_1, d_2) \in R^{\mathcal{I}} \wedge d_2 \in C^{\mathcal{I}}\}\rvert \geq n\}$	$\mathcal{Q}$
number restriction	$(\leq n\ R.C)$	$\{d_1 \mid \lvert\{d_2 \mid (d_1, d_2) \in R^{\mathcal{I}} \wedge d_2 \in C^{\mathcal{I}}\}\rvert \leq n\}$	

Table 5.3: Syntax and semantics of $\mathcal{SHIQ}$.

Further adding inverse roles to $\mathcal{S}$ yields the logic $\mathcal{SI}$. $\mathcal{ALC}$ without *Tboxes* and with transitive roles and inverse roles is of the same complexity as pure $\mathcal{ALC}$, i.e., Pspace-complete [HST99]. A variety of DLs can be extended with inverse roles without affecting their computational complexity [Tob01] [CGLN01].

The logic $\mathcal{SHI}$ is obtained by extending $\mathcal{SI}$ with role inclusion axioms. $\mathcal{ALC}$ extended with transitive roles and role inclusion axioms becomes ExpTime-hard [Sat96].

For a variety of DLs, extending them with number restrictions does not change their complexity. $\mathcal{ALC}$ extended with number restrictions remains in Pspace. The logic $\mathcal{SHIQ}$ adds number restrictions to $\mathcal{SHI}$. Number restrictions are concepts of the form $(\geq n\ s.C)$ and $(\leq n\ s.C)$ for n a nonnegative integer and C a $\mathcal{SHIQ}$-concept and s a *simple* role. A role is *simple* if it is neither a transitive nor has a transitive subrole. This restriction is necessary, because $\mathcal{SHI}$ extended with number restrictions on arbitrary roles yields undecidability [HST99]. Satisfiability of $\mathcal{SHIQ}$-concepts is known to be ExpTime-complete. The algorithm implemented in state-of-the-art DL systems (for example, Fact and Racer) is 2NExpTime [Sat03]. However, several well-known efficient optimisations can be applied so that the algorithms perform much better in practice than their worst-case complexity suggests. For example, the algorithms in Racer are only inspired by tableau calculi for

$\mathcal{SHIQ}$ and different algorithms are used for some subtasks.

$\mathcal{SHIQ}$ can further be extended by more expressive role inclusion axioms as explained in the previous section. However, only the extension of $\mathcal{SHIQ}$ with so-called acyclic role axioms is still decidable and this logic is called $\mathcal{RIQ}$ [HS03].

And, last but not least, adding concrete domains to a DL yields a NExpTime-complete DL as shown in [BH91] and [Lut01]. Remark that worst-case complexity was only investigated for $\mathcal{SHIQ}(\mathcal{D})$.

5.7 On the Relation between DL and Modal Logic

In this section, we will show in a nutshell how DLs are related to modal logic. The discovery of this very close relationship led to the translation of results from modal logic and propositional dynamic logics (PDL) [HKT00] to DLs and as such a range of complexity results were found for DLs. We focus on the relation between modal logics and DLs because a certain modal logic is used to represent sequence diagrams in [Ara98] and PDLs are specifically developed for reasoning about computer programs.

The basic modal logic is called **K** and is defined as follows:

Definition 67 *[BdV01] Given a countable set of* propositional letters, $\mathbf{PROP} = \{p_1, p_2, \ldots\}$. *The* well-formed formulas *of the basic modal logic* **K** *over* **PROP** *are:*

$$\mathbf{FORM} := \top \mid p_i \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \Diamond\varphi$$

where $p_i \in \mathbf{PROP}$ *and* $\varphi, \varphi_1, \varphi_2 \in \mathbf{FORM}$. $\Box\varphi$ *is the abbreviation of* $\neg\Diamond\neg\varphi$.

The semantics of modal formulas is given by a triple $\mathcal{M} =< S, \pi, \mathcal{K} >$ such that S is a non-empty set, π a binary relation on S and $\mathcal{K}$ is a binary relation on S, i.e., $\mathcal{K} : \mathbf{PROP} \rightarrow \mathcal{P}(S)$. $\mathcal{M}$ is a so-called *Kripke structure*, where S is a set of so-called *states* or *worlds*. $\pi(p_i)$ is the set of states where p_i holds and $\mathcal{K}$ is the so-called *accessibility relation*. The semantics is then as follows:

Definition 68 *[BCM+03] Let* φ *be a modal formula and* $s \in S$ *a state, the expression* $\mathcal{M}, s \models \varphi$ *is read as "*φ *holds in* $\mathcal{M}$ *in state* s*",*

$$\begin{aligned}
\mathcal{M}, s \models p_i \quad &\textit{iff} \quad s \in \pi(p_i) \\
\mathcal{M}, s \models \varphi_1 \wedge \varphi_2 \quad &\textit{iff} \quad \mathcal{M}, s \models \varphi_1 \textit{ and } \mathcal{M}, s \models \varphi_2 \\
\mathcal{M}, s \models \neg\varphi \quad &\textit{iff} \quad \mathcal{M}, s \not\models \varphi \\
\mathcal{M}, s \models \Diamond\varphi \quad &\textit{iff} \quad \textit{there exists } s' \in S \textit{ with } (s, s') \in \mathcal{K} \textit{ and } \mathcal{M}, s' \models \varphi \\
\mathcal{M}, s \models \Box\varphi \quad &\textit{iff} \quad \textit{for all } s' \in S, \textit{if } (s, s') \in \mathcal{K}, \textit{then } \mathcal{M}, s' \models \varphi.
\end{aligned}$$

Other kinds of modal logics are established by restricting the Kripke structures. The modal logic **S4** is obtained from **K** by restricting the accessibility relation $\mathcal{K}$ to be reflexive and transitive. Also, the number of accessibility relations may be multiple, such modal logics are called *multi-modal logics*. Each accessibility relation can be thought of as corresponding to one *agent* and is quantified using the multi-modal operators $\Box_i$ and $\Diamond_i$. $\mathbf{K}_m$ stands for the multi-modal logic **K** with m agents.

Schild [Sch91] established the relation between the modal logic $\mathbf{K}_m$ and the DL $\mathcal{ALC}$ by the following translation f from $\mathcal{ALC}$-concepts using role names $r_1, \ldots, r_m$ to $\mathbf{K}_m$:

$$\begin{aligned} f(A) &= A \\ f(C \sqcap D) &= f(C) \wedge f(D) \\ f(\neg(C)) &= \neg f(C) \\ f(\forall r_i.C) &= \Box_i(f(C)) \\ f(\exists r_i.C) &= \Diamond_i(f(C)) \end{aligned}$$

An analogous translation from $\mathbf{K}_m$ formulas into $\mathcal{ALC}$ can be defined. It is now clear that DL interpretations can be viewed as Kripke structures and vice versa; a is an instance of an $\mathcal{ALC}$-concept C in an interpretation $\mathcal{I}$ iff the translation $f(C)$ holds in the state a in the Kripke structure corresponding to $\mathcal{I}$.

5.8 Description Logic Systems

In this section, implemented DL systems are analysed and based on this analysis we choose which system will be by our tool support in order to validate the ideas presented in this work.

Several earlier systems were developed which were not based on DLs but can be seen as predecessors of current DL systems and as such provided important ideas. KL-ONE is a knowledge representation system developed by Brachman [Bra77]. This system as opposed to, e.g., frame-based systems enabled the inference of implicit knowledge from given declarations. The initial versions of KL-ONE are not logic-based.

The KL-ONE language offers so-called concept names. Compound concepts, called concept terms or concept descriptions can be formed using concept-forming operators. A difference is made between classes of individuals and individuals. The latter can be related by roles. These roles can be primitive or composed by role constructors. Concept definitions are assignments of a name to a concept term and cycles are not allowed in those definitions. These concept definitions make the reasoning about the relationships between different concepts important.

As opposed to frame systems, KL-ONE introduced the idea to compute the subsumption hierarchy. Algorithms for two inference problems were developed. A first one implemented the so-called classifier inference that computed the subsumption hierarchy and a second one implemented the so-called realiser inference, computing for each individual mentioned the most specific atomic concepts of which the individual is an instance. The informal specification of the semantics of KL-ONE concept and role constructors led to problems. As a consequence a formal semantics was introduced. Given this semantics it was shown that the algorithms for classification and realisation are incomplete. KL-ONE is also undecidable. Also as a result of the formalisation the notions of *Tbox* and *Abox* were made more clear. KL-ONE systems were implemented in INTERLISP and Smalltalk.

Successors of KL-ONE are KRYPTON, NIKL, PENNI and KL-TWO. These systems are all quite old, not based on a particular DL and nowadays, out-dated. So-called *second generation* DL systems have been used in more real-world application whereas first generation systems rather were used to study knowledge representation problems. From this

second generation systems we will discuss two well-documented systems, CLASSIC [BE89] and LOOM [Mac91].

At the end of the 1990s a new generation of sound and complete DL systems was developed based on theoretical advances of which FACT [Hor99] and RACER [HLM99, HM01] are the most important ones.

5.8.1 Analysis Template for DL Systems

In the next sections we will analyse resp. CLASSIC, LOOM, FACT and RACER. To be able to analyse the different systems consistently, the following template is used. For each of these systems the following aspects are discussed:

Language constructs The language constructs are dependent on the logic supported by the system and on the restrictions imposed on *Tboxes*.

Completeness A description logic system is complete if it is guaranteed to find all the valid inferences. Completeness is dependent on the language constructs provided. Some systems have preferred to restrict expressiveness in order to guarantee tractability and completeness. Having a less expressive concept definition language limits the expressive power of the system, and the situations in which these systems are applicable are reduced. Other systems have decided to include constructs that are known to be intractable or even undecidable. These systems are more expressive, but the classifiers are incomplete. In this case, the users have to be aware that there could be inferences missing.

Open/Closed-World Assumption The reasoning tasks introduced in Section 5.4 can be performed under either the closed or open-world assumption. If closed-world semantics is assumed, the current fillers for all roles are considered to be the only fillers for these roles. In contrast with closed-world semantics, if a relationship is not known to hold in open-world semantics, it is not assumed to be false.

Query language This item discusses the provided query language for retrieving *Abox* individuals, by the DL system.

Additional features This item discusses some additional features offered by the system such as how it can be used by client applications.

5.8.2 CLASSIC

CLASSIC [BE89, RBB+95] was developed in the AI Principals Research Department at AT&T Bell Laboratories. It has been designed for applications where only limited expressive power is necessary, but rapid responses to questions are essential.

Language Constructs

CLASSIC's basic concept definition language supports the logic $\mathcal{ALNF}h^{-1}$. The lowercase letter h indicates that CLASSIC supports role inclusion but not role conjunction, i.e., only single inheritance is supported. Full CLASSIC also supports the concept constructors $\mathcal{O}$ and $\mathcal{B}$, defined in Table 5.1 for referring to individuals in concept definitions. Concept definitions

cannot be cyclic. CLASSIC imposes the unique name assumption which ensures that new atomic concepts are disjoint.

The designers of the system decided that it was essential that the system was able to respond rapidly to questions. The context in which this system was used required that many queries could be given to knowledge bases that are rarely changed.

Completeness

CLASSIC is "almost" complete. However if the semantics for the concept constructors $\mathcal{O}$ and $\mathcal{B}$ is used, where the interpretation function maps individuals in concept terms to disjoint sets of domain objects, the inference algorithms of CLASSIC can be shown to be complete. In CLASSIC a limited kind of disjunction (with concept names for which no definitions exist) can be expressed while retaining polynomial inference algorithms [BCM+03].

Open/Closed-World Assumption

CLASSIC provides support for closed-world reasoning. The user can manually close roles using a system function. Closing a role for an individual means that an appropriate maximum number restriction for the role can be added. CLASSIC will count the number of role fillers and add an *at-most* restriction automatically.

Query Language

CLASSIC is concept-centric and does not provide a general-purpose query language with which *Abox* individuals could be examined. Instead, the user has to use system-defined queries in the form of functions and a procedural interface provided by the underlying implementation language of the system.

Additional Features

Implementation languages for CLASSIC are COMMONLISP and C.

Continual refinements and changes can be made to individuals, but the concept definitions cannot be changed. The invariant is that the knowledge base is *always* consistent. This is achieved by rolling back to the last consistent state of the knowledge base, whenever an inconsistent state is detected. Copies of the individuals that lead to the inconsistent state are saved.

CLASSIC has a rule system. Rules are applied to explicitly named individuals in the *Abox* and this in a forward-chaining way. A rule consists of a precondition, i.e., a concept and a conclusion which is also a concept. If it can be proven that an individual specified in the *Abox* is an instance of the precondition concept, a concept assertion, making the individual an instance of the conclusion concept, is added to the *Abox*. The combination of role closing and rules makes the exact sequence of several closing operations important. This sequence completely determines what holds in the resulting *Abox*.

CLASSIC was one of the first systems offering an explanation facility which can be used to find the cause of knowledge base inconsistencies. This is due to the fact that CLASSIC was one of the first systems designed with respect to users who are not experts in DL theory.

The low expressiveness of the CLASSIC DL made it hard to use the system in many kinds of applications. However, increase in expressiveness has a certain price. This led to the development of expressive DL systems with incomplete algorithms.

5.8.3 LOOM

LOOM [Mac91, Bri93, Uni04b] was developed at University of Southern California's Information Sciences Institute. As a knowledge representation system, LOOM is a very flexible system and offers a wide range of services: reasoning, editing, validation and explanation. It has also been designed so as to incorporate different types of programming paradigms - data-driven, object-oriented and logic programming - on top of a common shared knowledge base.

Language Constructs

LOOM's concept definition language supports the Description logic $\mathcal{ALCQRIFO}$ plus additional constructs for dealing with real numbers. LOOM is based on the KL-ONE language which implies that concept definitions with necessary or with necessary and sufficient conditions play an important role. The first versions were based on DLs, however later versions, called POWERLOOM, have "a classifier[1] that is able to classify descriptions expressed in full first-order predicate calculus" [Uni04c].

Completeness

LOOM implements incomplete algorithms for subsumption and concept satisfiability. The justification given by LOOM's designers for the utility of an incomplete classification algorithm is that complete systems are only of theoretical interest, but are too restrictive to be of use in most applications. Another reason involves the performance requirements for LOOM.

Incomplete algorithms have the problem that a "no"-answer should not be trusted. Another problem with the LOOM approach is that it is difficult to characterise the exact circumstances which will result in incomplete reasoning or intractability. LOOM deals with this by letting the user limit the computational effort expended in seeking "expensive types of inferences". But the limits are still hard to define. The users cannot explicitly control how declarative knowledge is used by the system.

Also many possible inferences are not supported. LOOM cannot reason about cardinality restrictions or role value restrictions. It is also known that reasoning about inverse roles is incomplete. The exact circumstances that lead to incomplete reasoning are not known. What has made it harder to pinpoint situations that lead to incompleteness of the classifier is that, as the system was developed, the concept definition language was made more expressive. As sources of incompleteness were found and solved, new ones arose.

Open/Closed-World Assumption

It is up to the user to decide whether a KB will assume that the world is closed or open, at the moment of the KB's creation. This assumption cannot be changed after expressions have

[1] This denotes the subsumption inference procedure.

been asserted. There are no operators of functions that will switch the world from open to closed or vice-versa. Choosing closed-world means that all the knowledge that is presently in the knowledge base is all that is known about the individuals that inhabit the world. This means that closing a role for a certain individual results in counting the number of known role fillers. In addition to the individuals explicitly mentioned in the *Abox*, existential qualification and minimum number restrictions have to be considered too. This will let LOOM draw additional inferences (as opposed to the default open-world assumption). This in combination with an incomplete inference algorithm might lead to unexpected behaviour as shown in chapter 8 of [BCM+03].

Query Mechanism

LOOM provides an expressive query language for retrieving *Abox* individuals. The query language is a predicate calculus-based language. This language is strictly more expressive than the DLs used in the concept definition language. It is possible to include predicate calculus-based expressions in description logics expressions using special operators. However, the classification algorithm cannot reason about these expressions. The LOOM query language does not support questions returning concepts, only sets of individuals.

Additional Features

LOOM is implemented in COMMONLISP, while POWERLOOM can be obtained for COMMONLISP as well as C and Java platforms.

LOOM also supports rule-based programming. This rule system makes it possible to specify additional necessary conditions for individuals which are explicitly mentioned in the *Abox* but also which are derived to be instances of a certain defined concept. These additional necessary conditions are not taken into account for *Tbox* reasoning. Note also that LOOM rule-based implications are not to be confused with true logical implications. Backward-chaining strategies are used for query answering, however for the rule system forward-chaining techniques are used. The user can control the inference process by combining forward-chaining and backward-chaining inferences by marking concepts, but the user is also responsible for the effects of these declarations.

Inconsistent definitions are not treated as illegal, but are classified under the built-in concept *Incoherent*, that can be easily accessed by the user. Recognition and truth maintenance only takes place when LOOM's matcher is invoked. This causes the re-computation of the types of the modified instances, and the changes are propagated throughout the knowledge base using a forward-chaining algorithm. When a concept is redefined, the necessary reclassifications are made and the knowledge base is checked so as to ensure that knowledge base satisfiability is maintained.

The user can create, manipulate and query knowledge bases interactively and from within applications. There are many pre-defined functions that are designed to facilitate application programming. LOOM has a context mechanism, that allows the definition of various contexts, each with its own *Tbox* and *Abox*, which are organised into hierarchies.

5.8.4 FACT

FACT (Fast Classification of Terminologies) [Hor99] has been developed at the University

of Manchester as a result of research into optimizing tableaux subsumption algorithms.

Language Constructs

At the time of our review of DL systems, the system included two reasoners, one for the logic $\mathcal{SHF}$ ($\mathcal{ALC}$ augmented with inverse roles and qualified number restrictions) and one for the logic $\mathcal{SHIQ}$.

Completeness

In order to be able to provide complete reasoning for an expressive DL like $\mathcal{SHIQ}$, FACT (and also RACER, see next section) does not use structural algorithms as the KL-ONE family does. Instead, it uses sound and complete tableau calculus-based algorithms. This system was the first one to demonstrate the usefulness of expressive DLs for practical applications. Even if the runtime behaviour is exponential in worst case, optimisation techniques can be developed and implemented that prevent DL systems from running into combinatorial explosion. The resulting algorithms are still sound and complete.

Open/Closed-World Assumption

Both FACT reasoners operate under the open-world assumption: what cannot be proven is assumed to be false. There are no operators or functions that allow to close a role. Closed-world assumption requires non-monotonic reasoning that is not supported by state-of-the-art DL systems.

Query Mechanism

At the time of this review, *Aboxes* were not supported by FACT. There is also no *Abox* query mechanism available.

Additional Features

FACT is implemented in COMMONLISP. The source code is available for research purposes. CORBA-FACT is a CORBA based client-server architecture for FACT.

There also exists a graphical interface for developing *Tboxes* in a frame systems based way, called OILED. This interface is described in [BHGS01]. Not only FACT but also RACER can be used as underlying DL system for OILED.

5.8.5 RACER version 1.7

RACER [HLM99, HM01, HM03] was initially developed at the Hamburg University of Technology, Germany. RACER is actively supported and future releases are developed at Concordia University in Montreal, Canada, and at the University of Hamburg-Harburg, Germany. Of all the evaluated KR tools, this system and FACT are the only ones that are actively being developed and maintained.

Language Constructs

RACER supports the DL $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$, where $\mathcal{ALCQHI}_{\mathcal{R}^+}$ corresponds to $\mathcal{SHIQ}$ and $\mathcal{D}^-$ indicates the support for concrete domains and no feature chains. Different sets of predicates are supported for different sets of concrete domains. For natural numbers ($\mathbb{N}$), linear inequations with order constraints and integer coefficients are supported, for integers ($\mathbb{Z}$), interval constraints are supported. For reals ($\mathbb{R}$), linear inequations with order constraints and rational coefficients are supported and for complex numbers ($\mathbb{C}$), nonlinear multivariate inequations with integer coefficients are supported. Finally for Strings, equality and inequality are supported.

Remark that simple roles, which are non-transitive roles that have no transitive subroles, are allowed to have number restrictions. Transitive roles cannot have this type of restriction because of the undecidability that would result in case of this unrestricted syntax.

There have been advances in the development of reasoning algorithms for expressive description logics, but mostly only considering *Tbox* reasoning. In previous systems, with the exception of LOOM, *Abox* reasoning was never implemented. Based on theoretical results, a practical implementation of *Abox* calculi was developed with RACER.

RACER is very flexible w.r.t concept definition, and even allows forward references to concepts that will be introduced later. Roles can be arranged into hierarchies. A role having subroles cannot be cyclic.

Completeness

RACER uses sound and complete algorithms and selects appropriate optimization techniques based on a static analysis of input *Tboxes* and *Aboxes* and inference services asked by a user.

Open/Closed-World Assumption

As FACT, RACER also operates under the open-world assumption.

Query Mechanism

Even though RACER offers *Abox* reasoning, it only provides concept-based instance retrieval. Examples of query functions are *concept-instances* and *related-individuals*. The first query retrieves the set of individuals for which it can be proven that they are instances of a certain concept. The second query retrieves the set of pairs of individuals that are related via a specified role. These sets of individuals can then be manipulated using the underlying implementation language of the DL tool. The reasoning effort used for answering the simple *Abox* queries can be controlled, by choosing either query indexes or exploiting the query subsumption (see RACER manual).

Additional Features

The implementation language of RACER is COMMONLISP. RACER is available as a server, offering various operational modes. It offers a file-based interface, a socket-based TCP stream interface and a HTTP-based stream interface. Both the socket and HTTP interfaces of the RACER server can be used from application programs or graphical interfaces. Various

graphical client interfaces are available for the RACER server, which makes interaction with KBs more user-friendly.

RACER also supports multiple *Aboxes* and *Tboxes*. Assertions can be added to *Aboxes* after inferences are processed and, in addition, assertions can be retracted from *Aboxes*.

5.8.6 Discussion

In this section, we will summarise the different features of the different discussed DL systems and motivate the choice of a particular system.

Standard Inference Services of DL Systems

First, we will summarise the main inference services that are expected to be standard in DL systems. For *Tbox* reasoning these are:

- *concept satifiability*,
- *concept subsumption*,
- *Tbox coherence checks* imply checking the satisfiability of all concept names mentioned in a *Tbox* without computing parent- and child-concepts.
- *classification of a Tbox*.

If a DL system supports *Abox* reasoning, the following inference services are assumed as standard:

- *Abox consistency* with respect to a *Tbox*,
- *instance checking*,
- *direct types of an individual*, i.e., the most specific concept names mentioned in a *Tbox* of which an individual is an instance,
- *retrieval*, i.e., finding all individuals mentioned in an *Abox* that are an instance of a given concept C,
- the *set of fillers of a role* r for an individual i
- the *set of roles between two individuals* i and j.

We decide to use RACER as DL-based reasoning tool for our tool support. RACER, with its highly expressive DL (including transitive roles and support for concrete domains) and its support for *Abox* reasoning is the ideal candidate. However, version 1.7 of RACER lacks an expressive query language enabling expressive reasoning and retrieval capacities over *Aboxes*. We will argue the necessity of having expressive query facilities for *Aboxes* for our application of DLs in chapter 7.

LOOM also has a very expressive concept definition language and a powerful query and retrieval mechanism. But we decided not to choose this system, because it has incomplete algorithms, which implies that it is not a real DL system, and it is not maintained anymore but rather replaced by POWERLOOM, a first-order predicate reasoner.

The main reason for not choosing CLASSIC is the lack of expressiveness of the DL and its restricted reasoning power, e.g., it does not allow for cyclic concept definitions.

FACT has an expressive language but it does support concrete domains and it also has no support for *Aboxes* or *Abox* reasoning.

In Table 5.4, the main features of the different discussed DL systems are recapitulated. The columns entitled **System** and **Language** need no further explanation. The column entitled **CDs** indicates if the system supports concrete domains and the column ***Abox*** indicates wether *Aboxes* are supported. The column entitled **Query Language** specifies if the DL system has an expressive query language. If only the standard *Abox* inference services (as specified in the beginning of this section), are present, a *min* appears in this column. The last column states whether the system is still maintained.

System	**Language**	**CDs**	***Abox***	**Query Language**	**Maintained?**
CLASSIC	$\mathcal{ALNF}h^{-1}$	no	yes	min	no
LOOM	$\mathcal{ALCQRIFO}$	yes	yes	yes	no
FACT	$\mathcal{SHIQ}$	no	no	no	yes
RACER	$\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$	yes	yes	min	yes

Table 5.4: Overview of DL systems.

5.9 Conclusion

In this chapter we argued why a logic-based formalism can be used, and more in particular, why DLs can be used in the context of inconsistency management.

DLs are introduced as a family of logic-based formalisms with decidable inference problems. We explained how different DLs are obtained by combining different expressive means and how these expressive means influence the complexity of these logics.

An overview of four different DL systems is presented, aimed at giving the reader an impression of the available DL systems and their reasoning capabilities and expressivity. This overview is also used to motivate our choice of the DL system used to validate our ideas.

Now that the reader is familiar with the formalism of DLs and the available DL systems, we can start with the evaluation of this formalism against the requirements of our key criteria listed at the end of Chapter 4. The next chapter will study the usability of DLs for the *representation of the syntax and semantics* of our well-defined UML fragment.

Chapter 6

Encoding of UML Model Elements

First, the encoding of the UML metamodel fragment as introduced in Chapter 2 into one of the most expressive DLs, i.e., $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$ or $\mathcal{SHIQ}(\mathcal{D}^-)$ that is supported by RACER, is discussed (Section 6.1).

A UML model element can be represented in a DL knowledge base as an instance of a metamodel element or as a DL concept description representing the element's meaning. This results in multiple representations of the same model element in DL. We observe that this multiple representation is a known issue in literature and we illustrate how the proposed solutions can be applied in our approach (Section 6.2).

The formalisation of our UML fragment specifies different possible interpretations for UML models. We informally recapitulate the interpretations of class diagrams, sequence diagrams and PSMs (Section 6.3). As a result, the encoding of class diagrams is introduced (Section 6.4). Next, the encoding of PSMs is introduced (Section 6.5) and, finally, the encoding of interactions (Section 6.6). Because the mapping of event occurrences on PSM labels also takes into account constraints expressed on event occurrences (cf. Definition 35), the encoding of constraints in DL is also discussed (Section 6.7). The expressiveness of the constraints specified in DLs is compared to OCL constraints. The different presented encodings are summarised and related to each other (Section 6.8).

This chapter is concluded with a discussion on related work and on the different, presented formalisations in DL. Finally, we evaluate the requirements of our first criterion – representation of abstract syntax and semantics – against the different discussed encodings (Section 6.9).

6.1 Encoding of UML Metamodel

We will first discuss the encoding of the abstract syntax of UML into $\mathcal{SHIQ}(\mathcal{D}^-)$ and show how the expressive means of this DL can be used in a straightforward way for the translation of UML metamodel elements.

6.1.1 Encoding

The abstract syntax of the UML metamodel is expressed as a class diagram consisting of classes, associations, generalisations and attributes. The UML metamodel is interpreted by mapping instances of the metaclasses in the metamodel to the syntactic elements of the

UML language. Meta-associations and meta-attributes constrain the allowable structure of, and relationships between, UML model elements interpreted as instances of metaclasses. For the encoding of the different metamodel constructs, we used a similar approach as the one used in [Ber02].

Metaclasses and Primitive Types Metaclasses are translated into concepts of $\mathcal{SHIQ}(\mathcal{D}^-)$. The primitive UML types *Integer* and *String* map into the concrete domains $\mathbb{Z}$ and STRING. The primitive type *UnLimitedNatural* is represented by a concept. How the UML primitive type *Boolean* is represented, is explained in the next item.

Meta-attribute att $= (n, c) \wedge \mathbf{type}(\mathbf{att}) = c'$ A meta-attribute **att** named n, owned by a class c and of type c' is modelled by a role $\mathtt{n}$ and the concept definition: $\mathtt{c} \sqsubseteq \forall \mathtt{n}.\mathtt{c}'$. If the attribute has a multiplicity, $\mathbf{multiplicity}(\mathbf{att}) = [i..j]$, associated, this multiplicity can be naturally captured by the following concept definition: $\mathtt{c} \sqsubseteq (\geq i\ \mathtt{n}\ \mathtt{c}') \sqcap (\leq j\ \mathtt{n}\ \mathtt{c}')$.

Remark that c' can be a DL concept or a concrete domain. Attributes that have the primitive type *Boolean* are represented as roles without a range. The reason for this is obvious. If the value of this attribute is true for a certain instance of the metaclass owning this attribute, then a class in a UML model has the property expressed by the meta-attribute, otherwise it does not have the property. Due to the open-world assumption of DLs, the restriction of not having a certain property must be explicitly stated in the knowledge base.

Meta-associations $\mathbf{assoc} = (assocname, c, c') \wedge \mathbf{assocend}_1 = (\mathbf{assoc}, \mathbf{assocType}_{2,1}) \wedge \mathbf{assocend}_2 = (\mathbf{assoc}, \mathbf{assocType}_{2,2})$ All meta-associations are binary. A bidirectional association **assoc** with association ends $\mathbf{assocend}_1$ and $\mathbf{assocend}_2$, between the classes c and c' is modelled by a role with domain $\mathtt{c}$ and range $\mathtt{c'}$ and its inverse. The role represents one of the association ends of a bidirectional association, while the inverse of the role represents the other association end of the bidirectional association. This captures exactly the meaning of a bidirectional association. Bidirectional associations include the constraint that the two association ends are inverses of each other. In case of a unidirectional association, the inverse of the role is not defined. The domain restriction for the role $\mathtt{assocend}_1$ and its inverse $\mathtt{assocend}_2$ is expressed by the GCI's: $\exists \mathtt{assocend}_1.\top \sqsubseteq \mathtt{c}'$ and $\exists \mathtt{assocend}_1^-.\top \sqsubseteq \mathtt{c}$ and $\mathtt{assocend}_2 \equiv \mathtt{assocend}_1^-$. The range restriction for this role and its inverse is expressed by: $\top \sqsubseteq \forall \mathtt{assocend}_1.\mathtt{c} \sqcap \forall \mathtt{assocend}_1^-.\mathtt{c}'$. Multiplicity constraints, $\mathbf{multiplicity}(\mathbf{assocend}_1) = [n_1..m_1] \wedge \mathbf{multiplicity}(\mathbf{assocend}_2) = [n_2..m_2]$, are captured by $\mathtt{c}' \sqsubseteq (\geq n_1\ \mathtt{assocend}_1.\mathtt{c}) \sqcap (\leq m_1\ \mathtt{assocend}_1.\mathtt{c})$, for $\mathtt{assocend}_1$, and $\mathtt{c} \sqsubseteq (\geq n_2\ \mathtt{assocend}_2^-.\mathtt{c'}) \sqcap (\leq m_2\ \mathtt{assocend}_2^-.\mathtt{c'})$, for $\mathtt{assocend}_2$.

Meta-aggregations are translated in the same way as meta-associations. The distinction between the contained class and the containing class in the aggregation is a consequence of the convention used that the domain of the role is the containing class.

Meta-compositions are translated in the same way as meta-associations. There is however an extra restriction, a part instance can only be included in at most one composite at a time. Suppose classes c_1 and c_2 both having a composition relationship

with class c, meaning that both classes are composed of objects of type c as shown in Figure 6.1. The extra restriction on the associationends **assocend**$_1$ of type c_1 and **assocend**$_2$ of type c_2 and belonging to the class **c**, is encoded by the following GCI : $\exists \texttt{assocend}_1.\texttt{c}_1 \sqcap \exists \texttt{assocend}_2.\texttt{c}_2 \sqsubseteq \bot$.

A transitive relation generalisationOf : C × C Generalisation in the UML metamodel is interpreted as a subclass relationship between the class and its generalisation, augmented with inheritance. A subclass relationship is the inverse of a subsumption relationship, i.e., the *subsumed-by* relationship. This relationship is transitive, that is, if c is subsumed by c' and c' is subsumed by c'' then c is subsumed by c''. These relationships are naturally supported by DLs. **generalisationOf**(c, c') is encoded by the GCI: $\texttt{c'} \sqsubseteq \texttt{c}$.

Inheritance between DL concepts works as inheritance between UML metaclasses. Every tuple in a role having c as domain or range, can have an instance of c' as domain or range. This means that all attributes and association ends of a superclass are inherited by its subclasses.

Disjointness of a set of classes A generalisation arrowhead on a class diagram can be labelled with the name of the generalisation set. Generalisation sets are by default disjoint. This disjointness restriction is specified by the Restriction 2.1. This disjointness restriction among the set of classes $R = \{c_1, \ldots, c_n\}$ can be encoded as: $\texttt{c}_i \sqsubseteq \sqcap_{j=i+1}^{n} \neg \texttt{c}_j$, with $i \in \{1, \ldots, n\}$.

Transitive closure of meta-associations We also exploit the presence of transitive roles in $\mathcal{SHIQ}(\mathcal{D}^-)$ and the subsumption relation between roles to implement the transitive closure of a meta-association. For example, consider the meta-association *general* between the metaclass *Classifier* and itself, as shown in Figure 2.7. This association keeps track of all immediate ancestors of a certain *Classifier*. This association is expressed by a $\mathcal{SHIQ}(\mathcal{D}^-)$ role, say DIRECT-SUPERCLASS and its inverse. However, we also define a transitive superrole, say SUPERCLASS of the role DIRECT-SUPERCLASS. Each instantiation of the role DIRECT-SUPERCLASS will automatically also be an instantiation of the superrole SUPERCLASS. As a consequence, if the tuple (a, b) and (b, c) are fillers of the role DIRECT-SUPERCLASS, automatically the tuple (a, c) will be filler of the role SUPERCLASS. This gives us the ability to retrieve in a very easy way all the ancestors (or descendants) of a certain class.

This facility is also used to keep track of the states related by transitions in a state diagram. Again a role, say DIRECT-SUCCESSOR-STATE, can be defined between the concept STATE, representing the UML metaclass *State*, and itself, and a transitive superrole of this role, say SUCCESSOR-STATE. As in the previous example, this easily allows us to ask all the successor states of a certain state.

Enumeration types The UML metamodel also contains enumerations. An example of such an enumeration is *AggregationKind*. *AggregationKind* specifies the literals for defining the kind of aggregation of a property. These literals are *none*, *shared* and *composite*. Instances of an enumeration type are one of the listed values. The representation of such an enumeration type in DL would require that is allowed for individuals to appear in concepts. In DLs this is not allowed, except for the logics $\mathcal{SHOQ}$ and

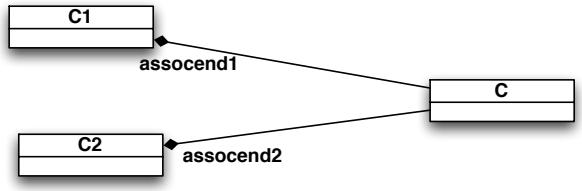

Figure 6.1: Example of a composition relation.

$\mathcal{SHOIN}$. The logic $\mathcal{SHOQ}$ corresponds to the logic $\mathcal{SHQ}$ extended with individuals in concept definitions ($\mathcal{O}$), i.e., the one-of operator as specified in Table 5.1. However, this logic does not provide inverse roles because the complexity of reasoning with inverse roles in combination with individuals is known to be NExpTime [Sat03]. The logic $\mathcal{SHOIN}$ is the logic $\mathcal{SHOQ}$ extended with inverse roles but restricted to unqualified number restrictions. For this logic there is not yet a known practical inference algorithm.

How can enumeration types be represented in, e.g., $\mathcal{SHIQ}(\mathcal{D}^-)$? The different values of the enumeration type are represented as disjoint concepts in the *Tbox*. In the *Abox*, individuals are defined corresponding to the enumerated values. These individuals are instances of a concept with the same name.

6.1.2 Example

In the examples of this chapter and the next ones, we will use the Racer syntax, which is a Lisp-like syntax. Table 6.1 stipulates the Racer syntax of the constructors introduced in Table 5.3. For a detailed description of Racer's syntax, we refer the interested reader to the Racer manual [HMW04].

As an example, consider the UML metamodel snapshot shown in Figure 2.5. The encoding of this snapshot in Racer is shown in RACER Fragment 6.1. The encoding of the complete UML metamodel fragment considered in this dissertation can be found in Appendix A.

Statements (1) until (8) shown in RACER Fragment 6.1, define roles for each of the eight meta-associations modeled in the metamodel. Statements (9) and (10) define roles for each meta-attribute owned by the metaclass *Property*. Statement (11) states the restriction on the composition relations involving the class *Constraint*.

Statements (12) until (19) define GCI's between concepts representing metaclasses involved in a hierarchy. Next the different multiplicity restrictions on meta-attributes and meta-association ends are defined in statements (20)-(24).

Statement (25) defines the encoding of the enumeration type *AggregationKind* and its possible values *none*, *shared* and *composite*. The enumeration type and its values are encoded as concepts (resp. `AggregationKind`, `none`, `shared` and `composite`) and the concept `AggregationKind` is defined as a disjunction of the concepts representing its possible values.

The last two statements state the disjointness of the meta-classes. In DLs concepts are not assumed to be disjoint, a specific restriction is needed to specify this.

```
(1)  (define-primitive-role ownedAttribute :domain class :range Property)
(2)  (define-primitive-role ownedOperation :domain class :range Operation)
(3)  (define-primitive-role association :domain Property :range Association
     :inverse memberEnd)
(4)  (define-primitive-role endType :domain Association :range Type)
(5)  (define-primitive-role formalParameter :domain Operation :range Parameter)
(6)  (define-primitive-role preCondition :domain Operation :range Constraint)
(7)  (define-primitive-role postCondition :domain Operation :range Constraint)
(8)  (define-primitive-role definedtype :domain TypedElement :range Type)
(9)  (define-primitive-role aggregation :domain Property :range AggregationKind)
(10) (define-primitive-role isCompositie :domain Property)
(11) (implies (and (some (inv preCondition) operation) (some (inv postCondition) operation)) bottom)
(12) (implies TypedElement NamedElement)
(13) (implies Type NamedElement)
(14) (implies Class Classifier)
(15) (implies Classifier Type)
(16) (implies Property StructuralFeature)
(17) (implies Property TypedElement)
(18) (implies Association Relationship)
(19) (implies Association Classifier)
(20) (implies Property (at-most 1 isComposite))
(21) (implies Property (at-most 1 aggregation))
(22) (implies TypedElement (at-most 1 type ))
(23) (implies Property (at-most 1 association))
(24) (implies Association (at-least 1 endtype))
(25) (equivalent AggregationKind (or none shared composite))
(26) (disjoint Type TypedElement)
(27) (disjoint Operation Property Parameter Constraint Association AggregationKind
     Class)
```

RACER Fragment 6.1: RACER implementation of the UML metamodel snapshot of Figure 2.5.

Constructor	**DL Syntax**	RACER **Syntax**
concept term subsumes concept name	$CN \sqsubseteq C$	(define-primitive-concept CN C)
concept name equals concept term	$CN \doteq C$	(define-concept CN C)
GCI	$C_1 \sqsubseteq C_2$	(implies C_1 C_2)
Equivalence	$C_1 \equiv C_2$	(equivalent C_1 C_2)
universal concept	$\top$	*top*
atomic role	R	R
transitive role	$R \in R_+$	(define-primitive-role R :transitive)
conjunction	$C_1 \sqcap C_2$	(and C_1 C_2)
disjunction	$C_1 \sqcup C_2$	(or C_1 C_2)
negation	$\neg C$	(not C)
value restriction	$\forall R.C$	(all R C)
exists restriction	$\exists R.C$	(some R C)
role hierarchy	$R \sqsubseteq S$	(define-primitive-role R :parents (S))
inverse role	R^-	(inv R)
qualified number restriction	$(\geq n\ R.C)$	(at-least n R C)
	$(\leq n\ R.C)$	(at-most R C)

Table 6.1: RACER syntax for $\mathcal{SHIQ}$.

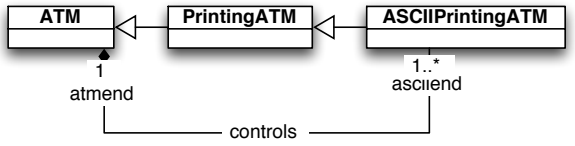

Figure 6.2: User-defined UML class diagram.

The UML metamodel fragment considered in this thesis, can be encoded into a RACER *Tbox* applying the above specified encoding. Next, the question arises how user-defined models can be represented. The UML concepts constituting these models are instances of the different UML metamodel concepts. As such, one way to represent user-defined models is as instances, i.e., individuals in an *Abox*, of the different RACER concepts and roles representing the UML metamodel.

Consider the class diagram in Figure 6.2. The encoding of this user-defined model gives rise to a RACER *Abox* that is shown in RACER Fragment 6.2. This *Abox* contains all the relations between the different UML model elements conforming the UML metamodel. The classes shown in Figure 6.2 are declared as instances of the metaclass *Class* and the two generalisation relationships are declared as instances of the metaclass *Generalization* and connected to the right classes via the meta-associations *generalization* and *general*.

The translation of the user-defined models can be automated (see Chapter 10).

```
(instance none none)
(instance shared shared)
(instance composite composite)
;
(instance atm class)
(instance printingatm class)
(instance asciiprintingatm class)
(instance gen1 generalization)
(instance gen2 generalization)
(instance controls association)
(instance atmend property)
(instance asciiend property)
(related atm gen1 generalization)
(related printingatm gen1 generalization)
(related printingatm gen2 generalization)
(related asciiprintingatm gen2 generalization)
(related atm printingatm direct-superclass)
(related printingatm asciiprintingatm direct-superclass)
(related gen1 atm general)
(related gen2 printingatm general)
(related atm asciiend ownedAttribute)
(related asciiprintingatm atmend ownedAttribute)
(related atmend controls association)
(related asciiend controls association)
(related atmend composite aggregation)
(instance atmend (some isComposite))
(constrained atmend loweratm lower)
(constrained asciiend lowerascii lower)
(constraints (equal lowerascii 1) (equal lowerascii 1))
(instance upperatm LiteralInteger)
(instance upperascii Unlimitednatural)
(related atmend upperatm upper)
(related asciiend upperascii upper)
(constrained upperatm valueatm value)
(constraints (equal valueatm 1))
(instance ourmodel Model)
(related atm ourmodel belongstomodel)
(related printingatm ourmodel belongstomodel)
(related asciiprintingatm ourmodel belongstomodel)
```

RACER Fragment 6.2: RACER *Abox* implementation of the class diagram of Figure 6.2.

6.1.3 Discussion

The proposed encoding defines a semantics for the UML metamodel. The user-defined models are considered as instances of the concepts defined by the UML metamodel. Therefore, the well-formedness of the user-defined models with respect to the metamodel is guaranteed by the reasoning task checking the consistency of the *Abox*.

Because the UML metamodel is described using concepts appearing in UML class diagrams, and DLs can be used as semantic domain for the UML metamodel, DLs can also be used as semantic domain for UML concepts constituting a UML class diagram. However, the class diagrams representing the UML metamodel do not contain all concepts defined in UML class diagrams. We will give the encoding of these concepts in Section 6.4.

The question arises if it is possible to encode the semantics of, e.g., SD traces in DLs. This implies that the same UML model element will have different encodings in a DL. E.g., a UML class is in the context of the UML metamodel an instance of the metaclass *Class*, but a UML class also represents a set of objects. As a consequence it can occur as an instance in a DL *Abox* but also as a concept in a DL *Tbox*. However, in DLs concepts and individuals are strictly kept distinct. The necessity of the different ways of representing one and the same conceptual concept, e.g. a UML class, is already recognised and discussed in literature [WF94]. In the next section, we summarise the ideas expressed in [WF94] and illustrate how these ideas apply to UML model elements.

6.2 Concepts versus Individuals

A DL *Tbox* represents definitions of concepts and roles, i.e., unary and binary predicates, an *Abox* represents instances of these concepts, i.e., constants.

Object-oriented modelling languages and programming languages, provide two basic primitives: classes and instances. A class describes a set of instances and an instance describes an object in the real world. As such, there is a one to one mapping between a *class* and a *concept*, and between an *instance* and an *individual*. The interpretation of representing a real-world object as an instance has always been an implicit assertion that the object exists as an individual.

Welty *et al.* [WF94] argue that the language constructs for instance and class must be kept distinct. DLs make this distinction, because they do not allow individuals to act as descriptions. This is prevented by the syntax. Also the UML syntax makes a difference between classes and instances. A class is denoted by a rectangle containing the name and properties of the class. An instance of a class is denoted by a rectangle containing the name of the instance and the name of the instantiated class. The name of the object and of its type is underlined.

Sometimes an object has two interpretations: as an instance of a certain class and as a class. This multiple interpretation occurs when a model consists of two or more universes of discourse. In one universe of discourse, the object is an instance and in another universe of discourse, the same object is a class. Welty [Wel95] calls these kinds of objects, *spanning objects*.

Examples of spanning objects can be found in meta-facilities of object-oriented languages. In Figure 6.3 M2 denotes the abstract syntax model of the UML. M2 contains the objects defining the UML language, e.g., the object *Class*. One instance of *Class*, is the

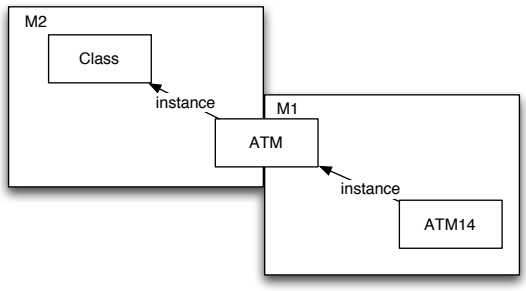

Figure 6.3: Example of a spanning object.

object *ATM*. On the level M1, *ATM* is viewed as the description of the set of ATMs. An instance of this description, is a particular ATM. It is obvious that *ATM* has two interpretations: an instance of *Class* in M2 and a description of the real-world object ATM in M1.

These special objects must be recognised by the modelling language and there must be support for a modelling construct that can be applied to an arbitrary layering of universes by the modeller. This support is lacking in current modelling languages. These special objects can be represented by second-order predicates. However, second-order logic is highly complex and computationally intractable. We will use the notion of spanning objects in our translation of UML concepts into DLs.

Remark that such spanning objects do not always involve meta-objects. Also within a certain domain, second-order objects can exist. The example used in the Artificial Intelligence world is the example of eagles. Harry is a certain instance of eagle and eagle is an instance of species. As such, eagle spans 2 universes of discourse.

Spanning objects are split into two parts. Each part is kept into a separate universe of discourse in which it exists as a normal first-order object. The actual object is referred to as a spanning object because its representation spans two universes of discourse. For example, DLs have three kinds of objects: concepts, roles and individuals. Each spanning object will be composed of an individual in one universe of discourse and a concept or role in another universe of discourse. Each universe, or level is built on objects in the previous universe. Each meta-layer in the UML architecture can be viewed as a universe consisting of spanning objects.

The two parts of such an object are linked by a *spanning function*. This function is defined for each class of second-order objects. For DLs, this function defines how to generate the concept or role part from the individual part.

6.3 Interpretation of UML Models

In Chapter 2 we formally defined the syntax of UML models. We also defined a trace semantics for sequence diagrams and PSMs. In this section, we will quickly recapitulate in an informal way the different possible interpretations of the UML models. This allows for a better understanding of the different translations of UML model elements into a DL. The

different interpretations of UML models will affect the translation of UML model elements into DLs.

6.3.1 Class Diagrams

The interpretation of UML class diagrams is not conventionally fixed. The interpretation might be given by a certain profile for using UML or might be the consequence of local design conventions. For example, in a UML profile for Java, classes are interpreted as Java classes and associations in the class diagram are interpreted as references between instances of the Java classes. A UML profile for business modelling gives a very different interpretation of class diagrams in terms of business entities.

Nevertheless, the UML elements appearing in class diagrams can be interpreted in a generic way. A class describes a set of instances and an instance describes an object in the real world. Instances must conform to the class description. The interpretation of an association consists of a set of links connecting instances. The links must conform to the association. For example, in the case of a bi-directional association, the two association ends are inverses of each other. A generalisation between a parent class and a child class means that every instance of the child class is also an instance of the parent class. The set of instances of the subclass is a subset of the set of instances of the superclass. Inheritance means that the instances of the child class inherit the properties of the parent class.

In the context of the UML metamodel, UML class diagram elements are interpreted as instances of metaclasses and connected through instances of meta-associations (cf. Section 6.1). If the UML model elements, i.e., the instances in the metamodel representation of the UML model, meet the restrictions imposed by metaclasses, meta-associations and meta-attributes, the UML model is well-formed.

6.3.2 Sequence and Communication Diagrams

There are several semantic alternatives for sequence diagrams. In Chapter 2, we differentiate between the *interaction* and *communication* view. We also discussed in Chapter 3 the *specification* and *instance* level.

Numerous interpretations of UML sequence diagrams can be found in literature. An overview of different usage scenarios for sequence diagrams as specified in UML 1.5 and a grouping of syntactic and semantic issues along orthogonal dimensions can be found in [HKS01]. The recognised dimensions are *Function View*, *Ordering*, *Scope*, *Abstraction*, *Composition* and *Time Quantification*. The choices along one dimension represent semantic variation points. Not all of these dimensions and variation points are currently supported by UML version 1.x or version 2.0. The *Composition* dimension which assumes the existence of additional UML language features to compose, refine or structure sequence diagrams is not supported by UML 1.x nor by UML 2.0. Some of these dimensions are covered by some newly introduced diagrams in UML 2.0. The *Time Quantification* dimension which assumes the definition of an underlying time model, is a dimension concerning timing diagrams in UML 2.0.

We briefly explain the *Function View*, *Ordering*, *Scope* and *Abstraction*. We show how the different interpretations we already recognised and defined, fit in those dimensions.

The *Function View* differentiates between interaction and internal activity. This dimension allows for different semantics for an activation. In UML 2.0 an activation is represented

in the abstract syntax by the *ExecutionOccurrence* element. An execution occurrence is represented by two event occurrences. The semantics of an execution occurrence is merely a trace ≪finish, start≫ where finish and start are the respective event occurrences. Due to the fact that we defined a trace semantics for sequence diagrams, this notion is implicitly present in our approach. This does not prevent UML users to define their own semantics or to refine the presented semantics for execution occurrences.

Within the *Ordering* dimension two possible choices are depicted in [HKS01], partial order and total order. The traces belonging to a certain connectable element, and the receiving traces defined in Chapter 2 define two different kinds of partial order. In the first case, the *event occurrences are ordered on a lifeline*, in the second case, the *receiving event occurrences* are ordered. A total order might be required by some applications, however, there is no graphical support by the UML for it.

In the *Scope* dimension, Hausmann *et al.* [HKS01] define two different variation points, *scenario* and *specification*. A scenario captures the expected or possible behaviour, while a specification captures the intended or mandatory behaviour of a system. This dimension allows for a range of interpretations of the specified behaviour by sequence diagrams. From a very loose (scenario) interpretation of the specified behaviour to a very strict (specification) interpretation. Observation and invocation consistent behaviour fit in this dimension. In the definition of *observation consistency*, we assumed that observable behaviour is modelled by (parts of) sequence diagrams and in the definition of *invocation consistency*, we assumed that only invocation behaviour is depicted by (parts of) sequence diagrams.

The *Abstraction* dimension differentiates between sequence diagrams representing object interactions, i.e., sequence diagrams at instance level, and role interactions, i.e., sequence diagrams at specification level. This distinction was already discussed in Chapter 2, Section 3.2. Remark that sequence diagrams representing object interactions can be interpreted as an instantiation of corresponding sequence diagrams representing role interactions. In this case, the instances and messages sent must conform to the connectable elements and messages sent at specification level.

The different variation points in a certain dimension require a different translation in DLs as explained in the next sections.

6.3.3 Protocol State Machines

The UML state machine formalism is a variant of Harel statecharts. Harel statechart semantics extend basic finite-state automata with many features. In UML 2.0 the semantics of state machine diagrams is concerned with event processing based on the run-to-completion assumption, firing priorities, selection of transitions and conflicting transitions.

We focus on PSMs and more in particular on the order of the operation calls, this is a weakened form of the semantics of UML state machines in general. This implies that PSMs are interpreted as a specification of different possible call sequences. The behaviour specified by a PSM is interpreted to be complete with respect to sequence diagrams.

In the definitions of invocation and observation consistency, we again assumed that the behaviour specified by the PSMs is invocation behaviour and observation behaviour, respectively.

6.4 Encoding of UML Class Diagrams

Classes, attributes, generalisations, aggregations and compositions can be translated in the same way as the corresponding meta-concepts. We still have to provide translations for operations and adapt the translation of associations in such a way that n-ary associations are allowed .

Operation op $= (n, c, (p_1, \ldots, p_k), (t_1, \ldots, t_m))$ An operation can be represented in $\mathcal{SHIQ}(\mathcal{D}^-)$ as proposed in [Ber02]. An operation **op** with name n belonging to a class c, is represented by introducing a concept $\mathtt{n}$ and $k + m + 1$ roles $r_1, \ldots, r_{k+m+1}$ where k is the number of input parameters belonging to $p_1, \ldots, p_k$ corresponding to the concepts $\mathtt{P}_1, \ldots, \mathtt{P}_k$, and m the number of return values belonging to $t_1, \ldots, t_m$ corresponding to the concepts $\mathtt{T}_1, \ldots, \mathtt{T}_m$. One role is added to connect the operation with the class owning the operation. The following assertion needs to be added to the respective *Tbox*: $\mathtt{n} \sqsubseteq \forall \mathtt{r}_1.\mathtt{c} \sqcap \exists \mathtt{r}_1 \sqcap (\leq 1\ \mathtt{r}_1) \sqcap \forall \mathtt{r}_2.\mathtt{P}_1 \sqcap \exists \mathtt{r}_2 \sqcap (\leq 1\ \mathtt{r}_2) \sqcap \ldots \sqcap \forall \mathtt{r}_{k+m+1}.\mathtt{T}_m \sqcap \exists \mathtt{r}_{k+m+1} \sqcap (\leq 1\ \mathtt{r}_{k+m+1})$. Remark that types of the parameters can be primitive. In case that the parameters are classes, we also enforce that the operation is the domain of the different roles connecting the operation with the parameter types and also that the parameter types are the respective ranges of these roles.

n**-ary associations assoc** $= (name, c_1, \ldots, c_n)$ In UML user-defined class diagrams n-ary associations are allowed. $\mathcal{SHIQ}(\mathcal{D}^-)$ does not support n-ary predicates. There is one family of DLs that does so, the family of $\mathcal{DLR}$ logics [CGL98]. However, we prefer the logic $\mathcal{SHIQ}(\mathcal{D}^-)$, because there is no tool support for the DLs $\mathcal{DLR}$ and techniques for reifying higher arity predicates are well-known. In $\mathcal{SHIQ}(\mathcal{D}^-)$, an n-ary association between classes $c_1, \ldots, c_n$ is represented by reifying the association. In [Ber02], it is shown that such an implementation is equivalent to an implementation in $\mathcal{DLR}$. The assertion $\mathtt{name} \sqsubseteq (\leq 1\ \mathtt{r}_1) \sqcap \ldots \sqcap (\leq 1\ \mathtt{r}_n) \sqcap \exists \mathtt{r}_1.\mathtt{c}_1 \sqcap \ldots \sqcap \exists \mathtt{r}_n.\mathtt{c}_n$ expresses the fact that the association named *name*, and represented by the concept $\mathtt{name}$ connects instances of n different classes through the roles $\mathtt{r}_1, \ldots, \mathtt{r}_n$ representing the different association ends of the association. Multiplicities can be added to the different association ends of the association. n GCI's of the form $\mathtt{c}_i \sqsubseteq (\geq k_i\ \mathtt{r}_i^-.\mathtt{name}) \sqcap (\leq m_i\ \mathtt{r}_i^-.\mathtt{name})$, for $i \in \{1, \ldots, n\}$, express the different multiplicity restrictions $[k_i..m_i]$.

Using spanning functions, the modelling elements used in UML class diagrams can be generated from the UML metamodel knowledge base. For example, a spanning function for individuals of the concept *Class*, generates a concept description. The function finds the value of the concrete domain attribute *name* which is a property of every *NamedElement* and it uses this name as the name of the concept. The superclasses of this individual are retrieved by using the fillers of the *general* role. The spanning function builds the concept description by creating a list that starts with the word *and* and inserts each of the names of the superclasses of this concept. In the case of the *Abox* shown in the RACER Fragment 6.2, this spanning function applied on the class *ASCIIPrintingATM*, yields the following result: `(implies asciiprintingatm (and printingatm atm))`.

The spanning function for individuals of *Attribute* works as follows. A role is created with name the filler of the concrete domain attribute *name* of the individual of *Attribute*.

The range of this role is determined by the filler of the role *definedtype* for the particular individual. The domain of this role is determined by the filler of the role *class* of the individual.

Similar spanning functions can be defined for associations, compositions, aggregations and operations.

6.5 Encoding of Protocol State Machines

Are DLs also suited as semantic domain for expressing PSMs? In this section, we present a possible translation of PSMs. Because PSMs focus on the possible call sequences on a certain instance of a class, the translation presented here, focuses on the encoding of the different call sequences defined by a PSM.

6.5.1 Call Sequence Encoding

Recall Definition 33, defining a call sequence of a PSM $\pi = (S, T, L, \rho, \Lambda)$. A call sequence $\mu =< \tau_1, \ldots, \tau_n >$ $(n \geq 1)$, where $\tau_i \in L$, can be encoded in $\mathcal{SHIQ}(\mathcal{D}^-)$ in different ways depending on the format of the label τ_i.

Label $\tau_i = (op, g, h)$ A label represents the call of an operation together with possible pre- and postconditions. The call of the operation *op*, encoded by a concept `opname` is encoded by the following concept description: $\forall \texttt{op.opname} \sqcap (= \ 1 \ \texttt{op})$. We introduce a role `op` that will be used to define the different operation calls occurring in call sequences. The preconditions g and the postconditions h are translated into concepts. How pre- and postconditions are translated into $\mathcal{SHIQ}(\mathcal{D}^-)$ and how expressive these constraints can be, is explained in Section 6.7.

Label $\tau_i = (\epsilon, g, \{\})$ In this case only g is translated into a concept.

Sequencing The question remains how to express that the different operations have to be called in sequence. For this purpose, a binary role `r` is used and also the subsumption relation is exploited. Different GCI's of the form $\texttt{g}_1 \sqcap \texttt{call}_1 \sqsubseteq \exists \texttt{r}.(\texttt{call}_2 \sqcap \texttt{h}_1 \sqcap \texttt{g}_2)$ are defined in a *Tbox* representing the different call sequences of a PSM. $\texttt{call}_1$ and $\texttt{call}_2$ represent the *invocation* of a certain operation. $\texttt{g}_1$ denotes the concept representing the preconditions defined on $\texttt{call}_1$, and $\texttt{g}_2$ denotes the concept representing the guard defined on $\texttt{call}_1$ and $\texttt{h}_1$ denotes the postcondition defined on $\texttt{call}_1$.

For example, consider the call sequence $< (readPIN, \emptyset, \emptyset), (verifyPIN, \emptyset, \emptyset), (\epsilon, valid\text{-}PIN, \emptyset), (ejectCard, \emptyset, \emptyset) >$, will be represented by the GCI's:

$$\forall \texttt{op.readPIN} \sqcap (= 1 \ \texttt{op}) \sqsubseteq \exists \texttt{r}.(\forall \texttt{op.verifyPIN} \sqcap (= 1 \ \texttt{op})) \tag{6.1}$$

$$\forall \texttt{op.verifyPIN} \sqcap (= 1 \ \texttt{op}) \sqsubseteq \exists \texttt{r}.((\exists \texttt{validPIN}) \sqcap \forall \texttt{op.ejectCard} \sqcap (= 1 \ \texttt{op})) \tag{6.2}$$

$$\exists \texttt{validPIN} \sqcap \forall \texttt{op.ejectCard} \sqcap (= 1 \ \texttt{op}) \sqsubseteq \forall \texttt{r}.(\exists \texttt{validPIN} \sqcap \forall \texttt{op.ejectCard} \sqcap (= 1 \ \texttt{op})) \tag{6.3}$$

The concepts `readPIN`, `verifyPIN` and `ejectCard` represent the operations $readPIN$, $verifyPIN$ and $ejectCard$, respectively. The concept ($\exists$`validPIN`) represents the precondition $validPIN$ (see Section 6.7). This example shows that if there is a label in the call sequence containing only a guard, this guard is translated as part of the guard of the next label representing an operation call. This choice is made because it will allow the comparison between call sequences and SD traces. GCI (6.3) indicates that the call sequences finishes with a call to $ejectCard$.

If several call sequences are taken into consideration, a label can be followed by several other labels. Consider the PSM shown in Figure 2.14 and the transition calling the operation *printReceipt*. After this transition two transitions are possible, or the card is ejected or the withdrawal transaction is chosen. The GCI ($\forall$`op.printReceipt` $\sqcap$ (= 1 `op`)) $\sqsubseteq$ $\exists$`r`.($\forall$`op.ejectCard` $\sqcap$ (= 1 `op`)) $\sqcup$ ($\forall$`op.getAccountNbr` $\sqcap$ (= 1 `op`) $\sqcap$ ($\exists$ `withdrawal`)) expresses that the calling of the operation *printReceipt* is followed by a call to *ejectCard or* by a call to the *getAccountNbr* operation.

The binary role `r` used in the translation of the different call sequences is similar to the accessibility relation in modal logic (see Section 5.7). The GCI (6.1) specified above, expresses that it is possible to reach the world where `verifyPIN` is triggered, starting from the world where `readPIN` is triggered.

Completeness of call sequences In case a PSM represents the complete set of possible call sequences the different $\exists$`r`.X concepts, where X is a concept variable, are replaced by $\forall$`r`.X. If we also want to express that only one transition is possible starting from a concept, we can add the restriction $\sqcap$ (= 1 `r`) to each $\forall$`r`.X.

Exactly one operation called at a certain moment in time To be able to express this constraint the following GCI's need to be added. First, a GCI is added specifying the disjointness of the different operation calls defined in the call sequence(s). Secondly, a GCI is added expressing that the specified operation calls are the only existing ones.

Example

We exemplify our translation by encoding the PSM shown in Figure 2.14 in a *Tbox*. The call sequences in the PSM shown in Figure 2.14 are represented by the definition of concepts and GCI's in the RACER Fragment 6.3. Statements (1) until (10) express abbreviations for the operation calls. The statements (11) until (18) define the different guards and statement (19) represents a disjointness constraints between different concepts used in the guards.

Statements (20) until (31) specify the different GCI's encoding the different call sequences. Statements (32) until (36) encode the end conditions. Statements (37) and (38) express that exactly one of the specified operation calls can be taken at a certain point in time and that these are the only operation calls that can be taken.

6.5.2 Adding State Information

It is also possible to take into account states, or state invariants. A state in a state machine diagram is an abstraction for a situation during which an invariant condition holds. If this

```
(1)  (define-concept t1 (and (all op readPIN) (exactly 1 op)))
(2)  (define-concept t2 (and (all op verifyPIN) (exactly 1 op)))
(3)  (define-concept t3 (and (all op getAccountNbr) (exactly 1 op)))
(4)  (define-concept t4 (and (all op getAmountEntry) (exactly 1 op)))
(5)  (define-concept t5 (and (all op checkifCashavailable) (exactly 1 op)))
(6)  (define-concept t6 (and (all op send) (exactly 1 op)))
(7)  (define-concept t7 (and (all op dispensecash) (exactly 1 op)))
(8)  (define-concept t8 (and (all op retainCard) (exactly 1 op)))
(9)  (define-concept t9 (and (all op ejectCard) (exactly 1 op)))
(10) (define-concept t10 (and (all op printreceipt) (exactly 1 op)))
(11) (define-concept g1 (and (not (some valid-PIN top)) (< tries 3)))
(12) (define-concept g2 (some valid-PIN top))
(13) (define-concept g3 (and (not (some valid-PIN top)) (= tries 3)))
(14) (define-concept g4 (some withdrawal top))
(15) (define-concept g5 (some checkcashavailable top))
(16) (define-concept g6 (not (some checkcashavailable top)))
(17) (define-concept g7 (some allowedwithdrawal top))
(18) (define-concept g8 (not (some allowedwithdrawal top)))
(19) (disjoint valid-PIN withdrawal checkcashavailable allowedwithdrawal)
(20) (implies t1 (all r t2))
(21) (implies t2 (some r (or (and g1 t1) (and g3 t8) (and g2 t9 (not g4)) (and g2 g4 t3))))
(22) (implies (and t1 g1) (some r t2))
(23) (implies t4 (some r t5))
(24) (implies t5 (some r (or (and g6 t9 (not g4)) (and g6 g4 t3) (and t6 g5))))
(25) (implies (and g2 g4 t3) (some r t4))
(26) (implies (and g5 t6) (some r (or (and t7 g7) (and g8 t9 (not g4)) (and g8 g4 t3))))
(27) (implies (and g7 t7) (some r t10))
(28) (implies t10 (some r (or t9 (and t3 g4))))
(29) (implies (and g8 g4 t3) (some r t4))
(30) (implies (and g4 t3) (some r t4))
(31) (implies (and g6 g4 t3) (some r t4))
(32) (implies (and g3 t8) (some r (and g3 t8)))
(33) (implies (and g2 t9) (some r (and g2 t9)))
(34) (implies (and g6 t9) (some r (and g6 t9)))
(35) (implies (and g8 t9) (some r (and g8 t9)))
(36) (implies t9 (all r t9))
(37) (disjoint t1 t2 t3 t4 t5 t6 t7 t8 t9 t10)
(38) (implies top (or t1 t2 t3 t4 t5 t6 t7 t8 t9 t10))
```

RACER Fragment 6.3: RACER expressions representing call sequences.

invariant condition is modelled explicitly, it can be taken into account in the translation of call sequences.

State $s \in S$ A simple state is translated into an atomic $\mathcal{SHIQ}$ concept.

Composite state A composite state is also explicitly translated into a complex $\mathcal{SHIQ}$ concept consisting of the union of the different substates. The different substates are subsets of the composite state and are disjoint. As an example, consider the composite state *GettingCustomerSpecifics* in Figure 2.14 consisting of three different substates. This composite state is represented by: $\texttt{GettingCustomerSpecifics} \equiv \texttt{AccountEntry} \sqcup \texttt{AmountEntry} \sqcup \texttt{VerifyATMBalance}$ and a disjointness constraint for the three substates.

States in the sequence Consider a call sequence $< (op_1, g_1, h_1), (op_2, g_2, h_2) >$ where the invariant condition s_1 holds before op_1 is called, the invariant condition s_2 holds after the call of op_1 and before the call of operation op_2 and the invariant condition s_3 holds after the call of op_2. This call sequence together with the different invariant conditions is expressed by the GCI's: $\texttt{g}'_1 \sqcap \forall \texttt{op}.\texttt{op}'_1 \sqcap (= 1\ \texttt{op}) \sqcap \texttt{s}'_1 \sqsubseteq \exists\, \texttt{r}.(\texttt{s}'_2 \sqcap \texttt{h}'_1)$, $\texttt{s}'_2 \sqcap \forall \texttt{op}.\texttt{op}'_2 \sqcap (= 1\ \texttt{op}) \sqsubseteq \exists\, r.(\texttt{h}'_1 \sqcap \texttt{s}'_2)$, $\texttt{g}'_2 \sqcap \forall \texttt{op}.\texttt{op}'_2 \sqcap (= 1\ \texttt{op}) \sqcap \texttt{s}'_2 \sqsubseteq \exists\, \texttt{r}.(\texttt{s}'_3 \sqcap \texttt{h}'_2)$. Again $\texttt{op}'_i$ are atomic concepts representing the operation $\texttt{op}_i$, $\texttt{g}'_i$ are concepts representing the preconditions g_i, $\texttt{h}'_i$ are concepts representing the postconditions h_i, and $\texttt{s}'_i$ are concepts representing the state invariants s_i.

Outgoing transitions between a composite state and a simple state can now be translated in a natural way. Due to the semantics of the subsumption relation, the GCI $\texttt{GettingCustomerSpecifics} \sqcap \forall\ \texttt{op.cancel}' \sqcap (= 1\ \texttt{op}) \sqsubseteq \exists\, \texttt{r.ChoosingTransaction}$ expresses that it is possible to go from each substate of *GettingCustomerSpecifics* to the state *ChoosingTransaction* if the operation *cancel* is called.

Completeness and disjointness of states If necessary, completeness of the set of states can also be enforced. Another restriction is the disjointness of the different states of a PSM. If both restrictions are valid, it is guaranteed that two states cannot be active at the same time.

Example

Consider again the PSM of Figure 2.14 and the translation of its call sequences in the RACER Fragment 6.3. The GCI's in the RACER Fragment 6.4 represent the translation of a part of this PSM taking into account the different states. The different $\texttt{t}_i$'s and $\texttt{g}_i$'s used in this *Tbox* fragment are defined in the RACER Fragment 6.3.

The statements (1) until (5) in the RACER Fragment 6.4 encode the composite state *GettingCustomerSpecifics*. The statements (6) until (12) correspond to the statements (20) until (24) of the RACER Fragment 6.3 extended with state information.

```
(1)  (implies accountentry gettingcustomerspecifics)
(2)  (implies amountentry gettingcustomerspecifics)
(3)  (implies verifyATMbalance gettingcustomerspecifics)
(4)  (disjoint readingpin pinentry verifyingpin choosingtransaction accountentry amountentry
     verifyATMbalance verifywithdrawal givecash retainingcard returningcard)
(5)  (equivalent gettingcustomerspecifics (or accountentry amountentry verifyATMbalance))
(6)  (implies (and readingpin t1) (some r pinentry))
(7)  (implies (and pinentry t2) (some r verifyingpin))
(8)  (implies (and verifyingpin g1) (some r readingpin))
(9)  (implies (and verifyingpin g3 t8) (some r retainingcard))
(10) (implies (and verifyingpin g2 (not g4)) (some r choosingtransaction))
(11) (implies (and choosingtransaction g4 t3) (some r accountentry))
(12) (implies (and choosingtransaction t9) (some r returningcard))
```

RACER Fragment 6.4: RACER expressions representing part of PSM of Figure 2.14.

6.5.3 Discussion

The first translation defines how the different transitions are put in sequence without taking into account state information. The second translation extends the first one with state information. The role `r` used in the translations, can be interpreted as an accessibility relation. The translation of the operations called, consists of the role `op` connected to the concepts representing the called operations. As presented in the previous section, operations are translated into concepts in the context of class diagrams. These concepts are used again in the translations presented above.

Again using spanning functions, the representation in RACER of modelling elements used in UML, the translation of call sequences with or without state information can be generated. Spanning functions exist for labelled relations, (orthogonal) composite states and call sequences. Depending on whether extra restrictions on PSM or call sequences must be specified, next to the representations generated by the spanning function, extra GCI's must be specified.

6.6 Encoding of Interactions

In Chapter 2 and in Section 6.3 of this chapter, we distinguished different semantic dimensions of sequence diagrams representing interactions. In case of a sequence diagram at specification level, focus can be on the different interactions defined or on the communication view. The same remark can be made for a sequence diagram at instance level. For each of these dimensions proper semantics using DLs will be provided in the following sections.

6.6.1 At Specification Level

Recall that a sequence diagram at specification level defines message sends between sets of instances of different classes.

Interaction View

The interaction view focuses on the different possible SD traces. The translation of SD traces into the DL $\mathcal{SHIQ}(\mathcal{D}^-)$ is similar to the translation of call sequences. Consider the Definition 28, defining receiving SD traces, containing event

occurrences denoting the receipt of a message. A receiving SD trace $v/^{rec} =< (\mathbf{m}_1, \mathbf{Cons}_1, \text{"}receive\text{"}), (\mathbf{m}_2, \mathbf{Cons}_2, \text{"}receive\text{"}), (\mathbf{m}_3, \mathbf{Cons}_3, \text{"}receive\text{"}) >$, is translated into the different GCI's: $\mathtt{cons}'_1 \sqcap \mathtt{m}'_1 \sqsubseteq \exists\, \mathtt{r}.(\mathtt{m}'_2 \sqcap \mathtt{cons}'_2)$, $\mathtt{cons}'_2 \sqcap \mathtt{m}'_2 \sqsubseteq \exists\, \mathtt{r}.(\mathtt{m}'_3 \sqcap \mathtt{cons}'_3)$, where $\mathtt{m}'_i$ are concepts representing the message m_i, and $\mathtt{cons}'_i$ are concepts representing the set of constraints $cons_i$ which must be valid before the execution of the owning event occurrence. In our formalisation, the different messages m_i only contain the operation invoked. Each message $\mathbf{m}_i$ is defined in DL as $\mathtt{m}'_i \doteq \forall \mathtt{op}.\mathtt{op}'_i \sqcap (= 1\ \mathtt{op})$, where $\mathtt{op}'_i$ represent the operation op_i. This translation of a message is similar to the translation of the call of an operation. This similarity allows straightforward verification of properties between call sequences and SD traces.

The SD trace denoting the receipt of messages by the object `atm` shown in the sequence diagram in Figure 2.12, is represented in $\mathcal{SHIQ}(\mathcal{D}^-)$ by the GCI's in the RACER Fragment 6.5.

```
(define-concept m1 (and (all op getAccountNbr) (exactly 1 op)))
(define-concept m2 (and (all op getAmountEntry) (exactly 1 op)))
(define-concept m3 (and (all op checkIfCashAvailable) (exactly 1 op)))
(define-concept m4 (and (all op send ) (exactly 1 op)))
(define-concept m5 (and (all op dispensecash) (exactly 1 op)))
(define-concept m6 (and (all op printReceipt) (exactly 1 op))))
(implies m1 (some r m2))
(implies m2 (some r m3))
(implies m3 (some r m4))
(implies m4 (some r m5))
(implies m5 (some r m6))
```

RACER Fragment 6.5: RACER expressions representing an SD trace.

Spanning functions can be used to generate the RACER expressions. First, for each instance of the metaclass *Message*, a concept name is defined and the operation called in this message is related to this concept name. For each trace of event occurrences, an `implies` statement is generated combining the messages and constraints of the corresponding event occurrences.

Communication View

The central issue of the communication view is how the different connectable elements are connected. At specification level, these connectable elements abstract away from particular instances and indicate which classes play which roles in a certain interaction. The connectable elements act as placeholders for concrete instances.

The translation of connectable elements and connectors is similar to the translation of classes and associations. This is not surprising because connectable elements are sets of objects and a connector is a subset of an association.

- Connectable elements are translated into concepts. By using the subsumption relationship, it is possible to express that a certain connectable element is a subset of certain classes.
- The assertion $\mathtt{C} \sqsubseteq (\leq 1\ \mathtt{r}_1) \sqcap \ldots \sqcap (\leq 1\ \mathtt{r}_n) \sqcap \exists \mathtt{r}_1.\mathtt{c}_1 \sqcap \ldots \sqcap \exists \mathtt{r}_n.\mathtt{c}_n$ expresses that the connector represented by the concept C connects instances of n different connectable elements.

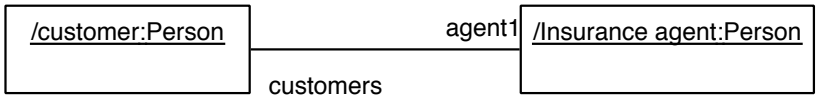

Figure 6.4: Example of a communication diagram.

Example 41 *Consider the communication diagram shown in Figure 6.4 consisting of two connectable elements and one connector. The translation of these connectable elements and the connector into DL is shown in the RACER Fragment 6.6. The first two GCI's express the fact that the connectable elements are sets of objects of type* `Person`*. The last GCI expresses that the set of links represented by the connector* `CustomerAgent` *is a subset of the set of links represented by the association* `PersonAssociation`*. This association is defined between the class* `Person` *and itself.*

```
(implies Customer Person)
(implies InsuranceAgent Person)
(some customers Customer) (some agent1 InsuranceAgent)))
(implies CustomerAgent PersonAssociation)
```

RACER Fragment 6.6: RACER expressions representing the communication diagram of Figure 6.4.

Again spanning functions can be used to generate the different *Tbox* expressions as, for example, shown in the RACER Fragment 6.6. From the different instances of the metaclass *ConnectableElement* and their corresponding base classes, concept subsumption expressions are generated. The same is done for instances of the metaclass *Connector*.

Remark that it is possible to connect messages with connectors. The notion of message must be extended with the connector in question.

6.6.2 At Instance Level

A sequence diagram at instance level describes a particular scenario. If a corresponding sequence diagram at specification level is defined, the sequence diagrams at instance level must conform to this sequence diagram.

Interaction View

The traces of a sequence diagram at instance level can be translated in the same way as the traces of a sequence diagram at specification level. In case the sequence diagram is to be interpreted as an instance of a sequence diagram at specification level, the different sequence diagram traces can also be translated into instances of the corresponding elements of the sequence diagram at specification level. This idea is shown in Figure 6.5.

The message *m1* can be seen in the UML architecture as an instance of the metaclass *Message*. At the same time this message is part of a sequence diagram at specification level and in this universe, the message *m1* is a concept. However, there exists in a sequence

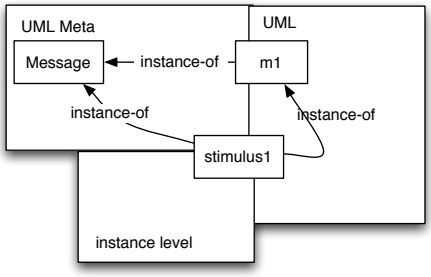

Figure 6.5: Specification versus instance level.

diagram at instance level, a *stimulus1*[1]. This *stimulus1* is an instance of the message *m1* at specification level. At the same time this *stimulus1* is a description of an invocation of an operation, etc. In fact this *stimulus1* is also an instance of the metaclass *Message*. This is why there is an arrow in Figure 6.5 from *stimulus1* to *Message*.

Communication View

The connectable elements in a sequence diagram at instance level, define particular instances of classes. The connectors correspond to links, i.e., instances of associations. As a consequence, the communication view of a sequence diagram at instance level is translated into individuals.

Example 42 *Consider as an example the communication diagram shown in Figure 2.10. The RACER Fragment 6.7 represents the objects* `atm`*, the object of type* `Session` *and the object of type* `Withdrawal` *and the links between these objects. The first two links are instances of associations that are explicitly modelled in the class diagram shown in Figure 2.6. The last two links are implicit in the class diagram. They represent instances of associations implicitly present in the model due to self sends.*

6.7 Encoding of Constraints

Constraints can be specified on UML class diagrams but also on lifelines in sequence diagrams, as pre- and postconditions on operations and as guards on PSM transitions. Constraints can also be specified on UML class diagram concepts. OCL can be used to specify these constraints. We can now ask the question to which extent OCL constraints can be translated into DL expressions.

The kinds of constraints that can be specified in a DL, are fully determined by its expressive power. For example, the set of constraints that can be expressed in the DL $\mathcal{SHIQ}(\mathcal{D}^-)$ is defined by the constructors which are defined in Table 5.3, together with the predicate names of the supported concrete domains. We will compare the expressiveness of this logic to the expressiveness of OCL constraints.

[1]Description of Stimulus can be found on pg. 31 in [Obj04e]

```
; translation of the objects
(instance atm ATM)
(instance session SESSION)
(instance withdrawal WITHDRAWAL)
;translation of the link between session and atm object
(instance sessiontoatm SESSIONtoATM)
(related sessiontoatm session sessionend1)
(related sessiontoatm atm atmend1)
;translation of the link between withdrawal and atm object
(instance withdrawaltoatm TRANSACTIONtoATM)
(related withdrawaltoatm atm atmend2)
(related withdrawaltoatm withdrawal withdrawalend1)
;translation of the link between session object and itself
(instance selfsession selfSESSION)
(related session selfsession sessionend2)
(related session selfsession sessionend3)
;translation of the link between withdrawal object and itself
(instance selfwithdrawal selfWITHDRAWAL)
(related withdrawal selfwithdrawal withdrawalend2)
(related withdrawal selfwithdrawal withdrawalend3)
```

RACER Fragment 6.7: RACER expressions representing part of the communication diagram of Figure 6.4.

6.7.1 OCL versus DLs

There are some fundamental differences between OCL version 1.x or version 2.0 and DLs in general.

1. OCL is a language based on first-order logic and set theory, it allows for bag semantics, aggregation operations (like *sum* and *size*), etc., while DL is a decidable fragment of first-order logic. Due to this, usually only unary and binary predicates can be captured. Remark that the logic $\mathcal{DLR}$ is an n-ary logic. Techniques for reifying higher arity predicates are well known too.

2. OCL is in the first place designed to specify constraints on elements of class diagrams, they can be expressed on other UML diagram elements only in a limited way. For more detail on this topic, we refer the interested reader to the UML 2.0 OCL Specification document [Obj04d]. The terminological knowledge specified by DL statements, is "generic" or "global", which means that it is true in every model of the situation and for every individual in the situation. In OCL free variables are allowed and quantification can be used as in first-order logic. Using OCL, restrictions can be specified on certain specific individuals.

3. OCL is a query language, it queries for individuals that violate a certain restriction. It specifies "local" information about certain individuals. Such information can only be specified by a DL query language that reasons on individuals defined in a DL *Abox*. The DL system RACER version 1.7 does not offer an advanced query language. In the next chapter we will show why inconsistency detection needs such a query language. Among other things, based on our observations a query language, named *nRQL* [Wes04], was developed and is available from RACER version 1.7.19.

4. DLs have (a form of) the tree model property (except for the DLs including nominals). This means that a DL concept C has a model (i.e., an interpretation) iff C

has a tree-shaped model, i.e., one in which the interpretation of properties defines a tree-shaped directed graph. This requirement severely restricts the way variables and quantifiers can be used. In particular, the quantified variable must occur in a property predicate along with the free variable (DL concepts correspond to predicates with one free variable). One obvious consequence of this restriction is that it is impossible to describe concepts whose instances are related to another anonymous individual via different property paths. For example, it is not possible to define a concept representing classes having attributes and association ends with the same name. Because OCL is based on first-order predicate logic, no restrictions are enforced on quantifiers appearing in the formulae.

6.7.2 OCL Constraints Encoded in $\mathcal{SHIQ}(\mathcal{D}^-)$

In this section, we focus on which constraints, compared to OCL constraints, can be expressed in a $\mathcal{SHIQ}(\mathcal{D}^-)$ *Tbox*.

Due to the fact that OCL constraints must be expressed in the context of a classifier and they are used to navigate over modeled associations, a certain OCL constraint can be specified in several ways. There are also equivalences among OCL operations and between the different logical operators defined in OCL. As a consequence, we start from OCL expressions simplified to a conjunctive normal form that is a conjunction of different disjunctions [CT04]. Conjunction and disjunction of OCL expressions corresponds to conjunction and disjunction of DL complex concepts. We now compare the different literals of a disjunction in the conjunctive normal form (this list is taken from [CT04]) of an OCL constraint, to their corresponding $\mathcal{SHIQ}(\mathcal{D}^-)$ concepts and roles, if possible.

A literal can be one of the following:

boolean attribute This attribute is represented in a DL by a role as explained in Section 6.4.

equality comparison between objects or sets A DL concept is to be interpreted as a set of individuals. In the context of a UML class diagram, a DL concept is interpreted as a set of objects. Equality of sets can be expressed by the equivalence relationship and the subset relation is nothing more than the subsumption relation in DLs. It is not possible to compare individuals at *Tbox* level, only at *Abox* level.

***forAll* iterator over an expression** This can be expressed in a DL statement under the condition that the expression does not state information about a particular individual and that the set over which is iterated can be captured by a complex DL concept. For example, the OCL constraint

```
context Bank inv:
self.customer->forAll(c | c.age >= 18)
```

stating that every customer of a bank must be older than 18 can be translated into a DL statement $\texttt{Bank} \sqsubseteq \forall\, \texttt{customer}.(\texttt{Customer} \sqcap \forall\, \texttt{age}.\, >= (\texttt{age}, 18))$ The classes *Bank* and *Customer* are represented by the DL concepts `Bank` and `Customer`. The association between *Bank* and *Customer* is represented by the DL concept $\texttt{BanktoCustomer} \sqsubseteq (\leq\ 1\ \texttt{customer}) \sqcap (\leq\ 1\ \texttt{bank}) \sqcap \exists\, \texttt{customer}.\texttt{Customer} \sqcap \exists\, \texttt{bank}.\texttt{Bank}$. The attribute *age* is represented by the DL role `age`.

***not* operator** Negation is supported by expressive DLs such as $\mathcal{SHIQ}$. Remark however that this negation is interpreted as the complement. Depending on the context, also the bottom concept, i.e., $\bot \equiv \neg\top$, can be used. To express the OCL constraint, context `Customer` inv: not `trustful`, the GCI $\texttt{Customer} \sqcap \exists \texttt{trustful} \sqsubseteq \bot$ can be used.

***oclIsTypeOf* or *oclIsKindOf* operator over an expression** $oclIsTypeOf(t)$ determines whether t is the same type as the type of the expression on which the operator is applied. $oclIsKindOf(t)$ determines whether t is either the direct type or one of the supertypes of the type of the expression on which this operator is applied. If it is possible to express the OCL expression on which the operators are applied into a DL expression, the subsumption and equivalence relation can be used to represent these operators.

arithmetic comparison Due to the inclusion of the concrete domains integer, reals and complex numbers, it is no problem to represent an arithmetic comparison in a DL concept. However, problems can occur if OCL pre-defined operations are involved in the comparison (see next discussion on OCL pre-defined operations).

We will now discuss how OCL pre-defined operations can be expressed in a DL. Due to the DL semantics, we only investigate operations on the OCL *Collection* type and on the OCL *Set* type. As defined in [Obj04d], there are some equivalences among these pre-defined operations. Using these equivalences, the set of operations can be reduced to (see also [CT04]): *size()*, *count(object)*, *select(expr)*, *forAll(expr)*, *union(set)* and *-(set)*. The *forAll* operation is already discussed above.

- The operation *size()* returns the number of elements of the set on which the operation is applied. If this set can be represented by a complex DL concept of the form $R.C$, i.e. a qualified restriction, qualified number expressions can be used to restrict the size of this concept. If the set cannot be represented by a qualified restriction, it is not possible to express a constraint on its size. The same remark can be made for the *count(object)* operation. This operation returns the number of occurrences of the *object* in the set on which the operation is applied. In current DLs it is not possible to count individuals at the *Tbox* level. However, it would be possible to introduce cardinality restrictions on concepts in a DL. These cardinality restrictions restrict the number of instances of a given concept. Such cardinality restrictions on concepts have been proposed in [BBH93] and they show that the important inference problems stay decidable. However, this has never been implemented in any DL system.

- The *select(expr)* operation specifies a subset of the set on which it is applied and for which *expr* is true. If *expr* can be represented by a DL concept, this DL concept is interpreted as the set for which *expr* is true. The OCL expression `self.select(c | c.age >= 18)` is represented by the DL concept $\texttt{Customer} \sqcap \exists \texttt{age}.{>=}(\texttt{age}, 18)$.

- The operation *union(set)* applied on a set returns the union of this set and *set*. This operation corresponds to the $\sqcup$ constructor. The operation *-(set)* returns the elements of the set on which the operation is applied, which are not in *set*. This operation can be represented in a straightforward way by a combination of the constructors $\neg$ and $\sqcap$.

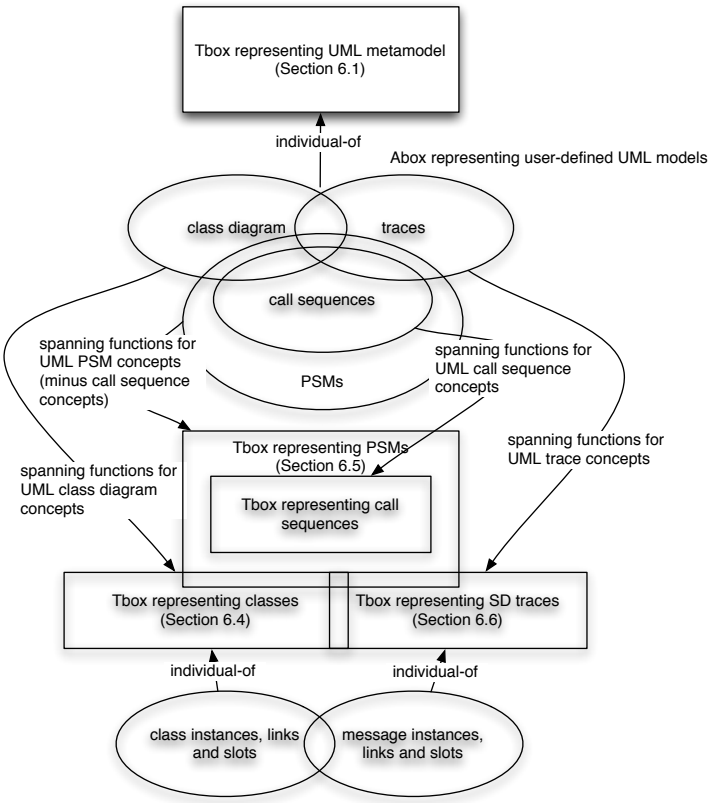

Figure 6.6: General picture on spanning functions.

6.8 A DL Framework Representing UML Models

Using the spanning functions (first defined in [WF94]), the translation of the different interpretations of the distinct UML model elements can be generated from the abstract syntax representation of the user-defined models. The general picture on the use of these spanning functions is illustrated in Figure 6.6.

The upper rectangle represents the *Tbox* containing the encoding of the UML metamodel as explained in Section 6.1. The ovals underneath contain the different user-defined UML models as instances of the concepts representing the UML metamodel. From this knowledge and using the spanning functions, *Tboxes* can be generated representing the semantics of PSMs, call sequences, SD traces and class diagrams. In this framework we assume that constraints on transitions, lifelines and class diagrams are written immediately in DL. The implementation of OCL constraints in DLs and the use of spanning functions for the generation of these constraints from the UML metamodel extended with the OCL metamodel are topics of future work.

Objects and links between objects, for example, can be represented as instances of the concepts representing classes and associations. Sequence diagrams at instance level can be interpreted as instances of sequence diagrams at specification level. The lower two ovals represent the *Aboxes* containing these instances.

In the next chapter, we will show how part of our classified inconsistencies can be detected by queries over the *Aboxes* representing the user-defined UML models as instances of UML metamodel concepts. The other part of our classified inconsistencies can be detected using the standard DL reasoning tasks on the *Tboxes* representing the user-defined models.

6.9 Discussion and Related Work

In the literature, several formalisations for general statecharts and interaction diagrams have been proposed. In this section, we discuss the most recent ones and compare them to our approach. In the definitions of Chapter 2, messages are characterised only by the operation called. Such messages are called symbolic messages, as opposed to parametrised messages containing not only the operations but also the actual parameters of the operation call. As a last topic of this section, symbolic messages/transitions versus parametrised messages/transitions, is explicated and we show to which extent parametrised messages/transitions can be supported in DLs.

6.9.1 Formalising Statecharts

There is an abundant amount of work on model checking for different statechart variants. Recent work concerns the translation of UML state machines into the process algebra CSP [EHK01] and the translation into the model checker Spin [SKM01, LMM99].

Model checking is an automatic, model-based, property-verification approach. It is intended to be used for concurrent, reactive systems. Model checking focuses explicitly on temporal properties and the temporal evolution of systems. It starts with a model described in a description language, and it discovers whether properties described in a specification language, are valid on the model. If they are not valid, it can produce counterexamples, consisting of execution traces. The specification language can be a temporal logic or a labelled transition system.

The specification language used by Spin is PROMELA. Spin can be used as a full LTL (linear-time temporal logic) model checking system, i.e., the properties to be verified are LTL formulae. Temporal logics have a dynamic aspect, since truth of a formula is not fixed in a model, as it is in predicate or propositional logic. The models of temporal logic contain several states and a formula can be true in some states and false in others. Other specification languages are also supported by Spin. In [SKM01] for example, UML state machines are translated into PROMELA, while Büchi automata are used to describe the properties to be verified. In this particular case the properties are UML collaboration diagrams.

In [EHK01] a refinement-based approach is taken. The process algebra CSP is used to describe UML state machines. A process algebra focusses on different ways to compose formulae. CSP defines refinement relations. Whether a certain property fulfils a described state machine depends on the existence of a refinement relation between them.

Some DL reasoning tasks can also be seen as an automatic, proof-based, property-verification approach. In DLs the specification and description languages are the same. The built-in reasoning capabilities are used to verify whether hypotheses asserted by the user are valid on the model. This does not result in counterexamples which consist of execution traces. Model checking starts with a logic model and discovers whether properties asserted

by the user are valid. In a model checking approach the verification relies on an exhaustive search of all states that the system will encounter. This gives rise to the known problem of state explosion. In case of a process algebra approach the verification relies on the available refinement relations. DLs construct models in which the property is valid. Model checking focuses explicitly on temporal properties unlike DLs. However, as explained in Section 5.7, some DLs correspond to certain modal logics of which temporal logics are special cases.

Which properties can be checked by a model checker depends on the expressiveness of the specification language. To conclude the discussion on formalising statecharts, we will list practically relevant properties taken from [HR04] and check whether they can be expressed in a DL. The following properties can be expressed in a DL.

- "It is impossible to get to a state where a property p_1 holds, but p_2 does not hold". This can be expressed by the GCI `(implies (some r (and` p_1 `(not` p_2`))) bottom)` in case the translation of Section 6.5.1 is used.

- "For any state, if a property p_1 holds then a next state can be reached where property p_2 holds." This can be expressed by the GCI `(implies` p_1 `(some r` p_2`))`.

- "For any state, if a property p_1 holds then eventually a state will be reached where property p_2 holds." A restricted form of this property can be expressed by the GCI `(implies top (all s (or (not` p_1`) (some s` p_2`))))`, where `s` is a transitive superrole of `r`.

The following properties (also taken from [HR04]) cannot be expressed by DLs.

- Suppose we model the behaviour of a lift. "*An upwards travelling lift at the second floor does not change its direction when it has passengers wishing to go to the fifth floor.*" This is not expressible in DL. The fact that the lift travels upwards *until* it reaches the fifth floor is not expressible in DL.

- The property "*from any state it is possible to get to a certain state*" is also not possible to express in DLs.

In general, loops as expressed by the until in the behaviour of a lift, cannot be expressed by DLs. Because SD traces describe scenarios, this does not affect the checking of our inconsistencies by DLs.

6.9.2 Formalising Interactions

Several kinds of semantics have been defined for UML interactions in literature. Araújo *et al.* [AM00] as well as Knapp [Kna99] transform UML interactions into a temporal logic. In [AM00] UML sequence diagrams are translated into temporal formula in Object-Z. The temporal logic, which is not included anymore in the Object-Z framework, corresponds to the modal logic **K**. The translation we introduced in this chapter is similar to this translation. As already explained in Chapter 5 roles in DLs can be used for the modal operators. The temporal logic used in [Kna99] is a linear first-order logic which extends **K** with the temporal connective *W*. This connective must be read as "unless".

6.9.3 Symbolic Messages versus Parametrised Messages

Remark that in the definitions of Sections 2.5 and 2.6 and as a consequence, also in the DL formalisation, only symbolic labelled messages and transitions with constraints are considered. In our examples, we used parametrised messages and transitions. Consider the transition from the state `VerifyATMBalance` to the state `VerifyWithdrawal` in the PSM in Figure 2.14. In our translation we only take into account the constraint [`cashAvailable`] and the operation called `send`. The actual value of the parameter `m` and the return parameter `allowedWithdrawal` of the operation are not considered. Nevertheless, we believe that our ideas can be extended and mounted to a more detailed level of messages and transitions. Remark that `allowedWithdrawal` is used in a constraint on the outgoing transitions of the state `VerifyWithdrawal` and translated into a DL concept.

We will discuss some ways dealing with actual parameters of an operation and assignment of properties of a class. The actual parameters of an operation call belong to a message. A message can be defined as a tuple consisting of the operation called and values for each of its parameters. These values, however, are either values of a primitive type such as `Integer` or `String`, or they are specific instances of classes. This means that these values are specified as individuals in a DL *Abox*. However, in case of actual parameters, the goal is to use these parameters in PSMs and sequence diagrams, i.e., at DL *Tbox* level. As explained in Section 6.1, this is not an insurmountable problem.

The same remarks can be made for assignment of values to properties of a class. However, to make things more complicated, the values of properties of a class can change within one SD trace or call sequence. Lets look at some examples.

Example 43 *If an attribute or a parameter has a primitive type, a concept can be defined which contains the assignment of this attribute/parameter to a certain value. Consider an attribute* `has-age` *of type* `Integer` *This is defined on class diagram level in* RACER *as:* `(define-concrete-domain-attribute has-age :type integer)`*. It is possible to define concepts, used on sequence diagram level for example, of the form* `(define-concept age1 (equal has-age 16))` *and* `(define-concept age2 (equal has-age 18))`*. These concepts can be used in more complex concepts and as such it is possible to reason about them. Remark that it is not only possible to assign an integer value to* `has-age` *but also more complex expressions. We refer the interested reader to the* RACER *manual ([HMW04], pg. 47) for the exact definition of these complex expressions.*

Example 44 *Consider the assignment* `m` := `amessage` *where* `m` *and* `amessage` *are both of type* `Message`*.* `m` *denotes an attribute, or an association end. Both* `m` *and* `amessage` *are* Abox *assertions and cannot be used at* Tbox *level in* RACER*. It is also not possible to express in the same* Abox *or* Tbox *that in one case an assertion has a certain value and in another case another value unless the context is explicitly modelled. What do we mean by modelling the context in an explicit way?*

Consider a transition `allowedWithdrawal` := `send`(m_1) *followed by a transition* `allowedCharging` := `send`(m_2)*.* `allowedWithdrawal` *and* `allowedCharging` *indicate the return value of the send operation called with the actual parameter* m_1*,* m_2*, respectively. This call sequence can be expressed in* RACER *as described in the RACER Fragment 6.8. Again in these definitions we included concepts like* `(exactly 1 p1)`*. If* RACER *would allow true reasoning over individuals in* Tboxes*, this would not be necessary, because the reasoner would infer the knowledge that* `(all p1 m1)` $\sqsubseteq$ `(`$\leq$ `1 p1)`*.*

As a last remark, it is quite clear that only static checks can be executed. It is possible to

```
(define-concept t1 (and (all op (and send (all p1 m1) (all r1 allowedWithdrawal) (exactly 1 p1)
(exactly 1 r1))) (exactly 1 op)))
(define-concept t2 (and (all op (and send (all p1 m2) (all r1 allowedCharging) (exactly 1 p1)
(exactly 1 r1))) (exactly 1 op)))
(implies t1 (some r t2))
```

RACER Fragment 6.8: RACER expressions for a parametrised call sequence.

model parametrised messages in DL and to reason about these messages to a certain extent. It is quite clear that only static reasoning is supported.

6.9.4 Evaluation of Criterion #1

In spite of their limited expressiveness, we can conclude that DLs meet the requirement stating the ability to express the abstract syntax of the UML fragment considered in this dissertation. We showed how metaclasses, primitive types, meta-attributes, meta-associations, meta-aggregations, generalisations and enumerations can be encoded in DL. It is also possible to express particular semantics for certain UML model elements as defined in Chapter 2. Through the encoding of class diagrams in DL, well-defined semantics is given to these diagrams. SD traces and call sequences with or without states can be encoded in DL providing well-defined semantics for these concepts. Finally, the communication view of sequence diagrams can be encoded in DLs too, providing a semantics for that particular part of a sequence diagram.

In general, the expressiveness of DLs allows for the verification of high-level UML models.

6.10 Conclusion

The goal of this chapter was to explore to which extent DLs satisfy the requirements of criterion #1, i.e., the representation of syntax and semantics. To evaluate the first part of this requirement, the representation of the abstract syntax, a translation of the UML metamodel to a particular DL, $\mathcal{SHIQ}$ is defined. This translation is based upon the translation of the basic UML class diagram concepts in [Ber02]. The representation of the semantics of the different UML semantical concepts was a next item of study. The same translation of the UML metamodel into $\mathcal{SHIQ}$ can be used to translated class diagrams, completed with some additional class diagram specific features. We also showed how and to which extent PSMs, call sequences and SD traces can be translated into $\mathcal{SHIQ}$ concepts. We also compared the expressiveness of OCL constraints to DL terminologies. This is the first study investigating the use of DLs as semantic domain for UML sublanguages and comparing OCL and DLs.

How the reasoning abilities of DLs together with the translations proposed in this chapter, can be used to detect and resolve inconsistencies, is the subject of the next chapter.

Chapter 7

A DL Inconsistency Detection Approach

This chapter presents DLs (and DL systems) as an inconsistency detection formalism. First, we commence by revisiting our conceptual classification previously defined in Chapter 3 from the point of view of detecting these inconsistencies by DLs and DL query languages (Section 7.1). We state that inconsistency detection in, or between, representations of UML models – resulting from using the encoding of the UML elements in DLs presented in the previous chapter – can be established in two ways.

We continue by introducing the first approach – to detect inconsistencies by querying the user-defined models in terms of metamodel concepts (Section 7.2).

In the previous chapter, we argued that UML model elements have different interpretations in different universes of discourse. These different interpretations are linked to each other via so-called spanning functions. We demonstrate the second way of detecting inconsistencies, i.e., how our DL framework representing UML models, can be used to detect inconsistencies in combination with standard DL reasoning tasks over DL *Tboxes* and *Aboxes* representing the semantics of UML diagrams (Section 7.3).

This chapter is concluded by a discussion on related work and on the advantages and limitations of the DL detection approach and by evaluating this approach based on criterion #2 of Chapter 4 – precise definitions of inconsistencies and inconsistency detection (Section 7.4).

7.1 Conceptual Classification Revisited

In this section, our conceptual classification of inconsistencies as defined in Chapter 3 is revisited from the point of view of the different inconsistency detection approaches. In the next sections, we will introduce two possible inconsistency detection approaches. The first approach is to query the user-defined models interpreted as instances of the UML metamodel. The second approach is to use DLs as semantic domain and to use the reasoning tasks, Tbox *coherence or* Abox *consistency*. An inconsistency belonging to our classification can be detected by one or a combination of both approaches.

Table 7.1 demonstrates the *DL inconsistency detection classification*. This classification is a revision of our conceptual classification. Again, two dimensions are distinguished.

	Metamodel (Section 7.2)	**DL Framework** (Section 7.3)
Specification	dangling type reference inherited cyclic composition connector specification missing	invocation interaction observation interaction
Specification-Instance	multiplicity incompatibility abstract object navigation incompatibility instance specification missing	specification behaviour incompatibility invocation behaviour observation behaviour
Instance	disconnected model	invocation inheritance observation inheritance instance behaviour incompatibility

Table 7.1: A first two-dimensional DL inconsistency detection table.

The first dimension concerns the different approaches to detection. The second dimension concerns the level of the affected model. This latter dimension also occurs in our conceptual classification.

Next to this dimension, our conceptual classification is made with respect to the dimension distinguishing between structural and behavioural inconsistencies. Table 7.2 shows a classification with respect to this dimension. From this table, we draw two conclusions.

	Metamodel (Section 7.2)	**DL Framework** (Section 7.3)
Structural	dangling type reference inherited cyclic composition connector specification missing instance specification missing disconnected model	
Behavioural	multiplicity incompatibility abstract object navigation incompatibility	specification behaviour incompatibility invocation interaction observation interaction invocation behaviour observation behaviour invocation inheritance observation inheritance instance behaviour incompatibility

Table 7.2: A second two-dimensional DL inconsistency detection table.

The inconsistencies classified as structural in Table 3.1 are all detected by using metamodel information. This is not surprising due to the nature of these constraints. In Chapter 3 the structural inconsistencies are specified in terms of UML metamodel concepts.

All behavioural inconsistencies, except for the specification incompatibilities, are detected using DLs as a semantic domain. Again, this is not suprising due to the nature of these constraints. These inconsistencies are defined in Chapter 3 in terms of semantical concepts, except for the specification incompatibilities that are specified in terms of UML metamodel concepts.

In the remainder of this chapter, we introduce both detection approaches and a discussion on the advantages and limitations of these approaches.

7.2 Querying the UML Metamodel

Our structural inconsistencies and behaviour incompatibilities express properties on UML models in terms of UML metamodel elements. To be able to detect these inconsistencies, the user-defined models need to be queried in their capacity of metamodel element instances. The detection of these inconsistencies boils down to the definition and execution of DL queries on DL *Aboxes* representing the user-defined models.

First, we motivate the need for an extensive *Abox* query language for the detection of inconsistencies using DLs. Next, we specify the requirements for this query language based on our experience with the specification of the detection of our inconsistencies with the query language of LOOM. Finally, we demonstrate how the required features are used in the detection of part of our classified inconsistencies.

7.2.1 Motivation for a DL Query Language

Practical DL systems such as RACER offer a functional API for querying a knowledge base, i.e., a tuple of a *Tbox* and an *Abox*. For instance, RACER provides a query function, `(concept-instances C)`, for retrieving all individuals from an *Abox* that are instances of a given concept `C`. This concept `C` can be an atomic or a complex concept. Remark that the function `concept-instances` uses the subsumption reasoning task.

Let us consider the statements `(related atm asciiend ownedAttribute), (related asciiprintingatm atmend ownedAttribute)` defined in the *Abox* of Figure 6.2. If we are interested in finding individuals in the *Abox* for which it can be proven that an attribute exists, the query `(concept-instances (some ownedAttribute Property))` can be used in RACER. However, if we would like to find all tuples of individuals x and y such that a common owner exists of the attribute, it is not possible to express this in sound and complete DL systems such as RACER version 1.7. The reason is two-fold. First, it is not possible to define a concept representing a common owner of an attribute due to the tree model property (cf. Section 6.7). Second, DL systems such as RACER version 1.7, are not equipped with a query language and mechanism that allow variables and individuals as part of the query and return tuples as answer sets. As a consequence, there is no support for queries asking for tuples of related individuals.

In contrast to the state-of-the-art DL systems, the second generation system LOOM has offered an expressive query language right from the beginning. In [Sim03], we translated part of the UML version 1.4 metamodel into LOOM concepts, roles and individuals using the translation introduced in the Section 6.1. We demonstrated how a major part of our defined inconsistencies can be detected using LOOM's query language. For example, the following query can be used to find all parameters of a certain operation whose types are not contained in any model.

```
(do-retrieve (?operation ?parameter ?class1 ?class2)
(:and
  (Operation ?operation) ;?operation is an operation
  (Is-owned-by ?operation ?class1) ;?operation is owned by a class ?class1
  (Has-parameter ?operation ?parameter)
                          ;?operation has a parameter ?parameter
  (Parameter-class ?parameter ?class2)
                          ;type of the ?parameter is a class ?class2
  (In-namespace ?class2 NIL))
                          ;?class2 is not defined in a namespace (i.e., model)
```

```
(format t "Type of the parameter ~S of operation ?S of class ~S, does not
exist in any model" (get-value ?parameter name) (get-value
?operation name) (get-value ?class1 name)))
```

The `do-retrieve` statement defines a query and the variables or individuals are preceded by a question mark. The query atoms are concept names or role names in this particular query. These concepts and roles correspond to UML 1.4 metaclasses, respectively meta-associations. Remark that we decided not to use Loom as DL system in our approach (cf. Chapter 5) because Loom is not maintained anymore and because it has an incomplete algorithm and many possible inferences are not supported.

One could argue that we do not need such a query language in state-of-the-art DL systems and that the function `concept-instances` together with some functions retrieving the fillers of a role, is sufficient. The host programming language can be used to encode the search. For example, in the CommonLisp Racer a similar query to the previous one can be formulated using the loop facility of CommonLisp.

```
(loop for ?operation in (concept-instances ?operation)
    with ?operationname = (first (retrieve-individual-fillers ?operation 'name)) do
   (loop for (and ?class1 in (retrieve-individual-fillers ?operation 'Is-owned-by)
             ?parameter in (retrieve-individual-fillers ?parameter 'Parameter-class)
    with (and ?classname = (first (retrieve-individual-fillers ?class1 'name))
             ?parametername = (first (retrieve-individual-fillers ?parameter 'name)))
    do
     (loop for ?class2 in
                      (retrieve-individual-fillers ?parameter 'Parameter-class)
        when (null (retrieve-individual-fillers ?class2 'In-namespace)) do
   (format t "Type of the parameter ~S of operation ?S of class ~S, does not
exist in any model" ?parametername ?operationname ?classname)
```

`retrieve-individual-fillers?` is a Racer function returning all the individuals that are fillers of a role for a specified individual.

The disadvantages of this approach are quite obvious:

1. The queries written as search programs using Racer API functions result in a bunch of nested loop-statements. The search programs can become quite complicated and unmanageable. The order of the statements in such a search program is important. The developer has to encode *how* the program must search through the *Abox*, while the goal of defining a query is to encode *what* must be searched for.

2. A second disadvantage of writing the queries using, e.g., the loop facility of CommonLisp is that no optimisation of the query can be performed. By having a sophisticated query mechanism, different optimisations can be performed. The amount of data available for the search process can be enhanced. The query mechanism will rewrite the query and add some logical conjuncts without changing the query's semantics. The answer sets of queries can be cached for two reasons. First, the cached results can be used when a query is executed more than once. Second, these results can be used for answering *related* queries. Similar to DL concepts, also DL queries can be classified, this process is called *query classification* [Wes04]. The most specific subsumers and the most general subsumees can be computed.

In Van Der Straeten *et al.* [VSM03], we plead for the integration of such an extensive query language in state-of-the-art DL reasoners. Based on our definitions of the different inconsistencies and on our experience with translating these inconsistencies into the

LOOM query language, a set of requirements can be defined for a DL query language. These requirements must be satisfied by a DL query language to be suitable for our needs.

7.2.2 Requirements for a DL Query Language

We use the following query format:

$$q(\vec{x}) \leftarrow body_1(\vec{x}, \vec{y_1}, \vec{c_1}) \vee \ldots \vee body_n(\vec{x}, \vec{y_n}, \vec{c_n})$$

In the remainder of this section, each part of this query format will be explained together with the requirements for each part.

Query body The body of a query (i.e., the expression following $\leftarrow$) is a disjunction of $body_i(\vec{x}, \vec{y_i}, \vec{c_i})$. $body_i$ is a conjunction of atoms. This expresses our first requirement. *The body of a query can contain disjunctions as well as conjunctions.* $\vec{y_i}$ are all the variables appearing in the body and $\vec{c_i}$ are the constants appearing in the body. The variables included in $\vec{x}$ are the answer set of the query. $\vec{x}$ denotes variables and constants that occur in the body of the query.

The above specified LOOM query has a body containing a conjunction of different atoms. The next requirements specify the possible atoms contained in a query body.

Concept query atom An *atom can have the form* $C(t)$ *where* t *is an individual or variable belonging to* $\vec{x}$, $\vec{y_i}$ *or* $\vec{c_i}$. C is an $\mathcal{SHIQ}(\mathcal{D}^-)$ concept. This concept can be an atomic concept or a complex concept. For example, the atom $\texttt{property}(x)$ denotes all the instances of the atomic concept `property`. These instances will be bound to the variable x. The atom $(\texttt{feature} \sqcap (\neg\texttt{property}))(x)$ is a complex concept query atom.

Role query atom An *atom can have the form* $R(t, t')$ *where* t *and* t' *are individuals or variables belonging to* $\vec{x}$, $\vec{y_i}$ *or* $\vec{c_i}$. R is an $\mathcal{SHIQ}(\mathcal{D}^-)$ role that can be an atomic role, an inverse role or a transitive role. By posing a query containing the atom $\texttt{ownedAttribute}(x, attCash)$, where $attCash$ is a constant, all individuals which are related to the individual $attCash$ through the role `ownedAttribute` will be bound to the variable x.

Concrete domain attributes atom An *atom can have the form* $P(f(t), g(t'))$ *where* t *and* t' *are individuals or variables belonging to* $\vec{x}$, $\vec{y_i}$ *or* $\vec{c_i}$. P *is one of the concrete domain predicates* of $\mathcal{SHIQ}(\mathcal{D}^-)$. $f = f_1 \circ f_2 \circ \ldots f_n$, $g = g_1 \circ g_2 \ldots g_m$ are role chains and f_n and g_m are concrete domain attributes or f and g are individuals from one of the concrete domains supported by $\mathcal{SHIQ}(\mathcal{D}^-)$. An example of such an atom is $\texttt{string=}(\texttt{ownedAttribute} \circ \texttt{name}(x), \texttt{name}(y))$. It expresses that the name of y equals the name of the owned attributes of x.

These concrete domain attribute atoms can be used to compare names of model elements.

Restricting to explicitly presented individuals Reasoning under the open-world assumption can lead to seemingly awkward results for users not familiar with it. Consider the qualified number restriction $(\geq 2\ \texttt{memberEnd property})$, expressing the existence of at least two member ends of type property. If less than two member

ends exist in an *Abox*, this will not result in an inconsistent *Abox* with respect to the qualified number restriction. Due to the open-world assumption, absence of role fillers of *memberEnd* is not interpreted as if there are none, but such role fillers can still be added to the *Abox* later on. In this case we want to query the *Abox* for role fillers of the role `memberEnd`. As a result of this query, we only want to retrieve the individuals for which such a role filler assertion is *explicitly* stated. A requirement for a DL query mechanism is that *variables can only be bound to explicitly present* Abox *individuals in the queried* Abox.

True negation and negation by failure The atom $(\neg\texttt{property})(x)$ in a query will result in the binding of all individuals for which a DL system, such as RACER, can prove that the individual is not a `property`. This is due to the open-world semantics. However, sometimes we want to retrieve all the individuals which are currently not known to be properties. This implies that all the individuals will be retrieved for which a DL system currently cannot prove that those are properties. This is called the *negation by failure semantics*.

Suppose we are looking for operations that do not belong to any class. If the atom $(\neg\texttt{ownedoperation})(x, y)$ is used, this results in the set of pairs (x, y) for which it can be proven that there are no operations. However, if some classes are present in the *Abox* without any operations attached, those will not be returned by a query consisting of the atom $(\neg\texttt{ownedoperation})(x, y)$. Again, due to the open-world semantics, any of the present classes which do not have any explicitly modelled operation in the *Abox*, can still own operations later on.

This negation by failure semantics is also needed for the next requirement.

Presence of explicit role fillers In the context of some of our inconsistencies it becomes particularly relevant to be able to check if there are explicitly specified role fillers in an *Abox*. This is relevant for example, for checking whether a certain model element belongs to the model under study. In this case, we want to check whether there is an explicit ownership relation between the model and the model element.

Retrieve told values Until now, we assumed that the *head of a query* has the form $q(\vec{x})$, where $\vec{x}$ consists of a set of individuals used in the body of the query and a set of variables which will be bound as a result of executing the query. Because $\mathcal{SHIQ}(\mathcal{D}^-)$ and RACER support concrete domains such as STRING, INTEGER, REALS, ..., concrete domain values assigned to so-called concrete domain attributes must be retrievable from an *Abox*. An explicitly asserted concrete domain value is called in RACER, a *told value*.

In the context of inconsistency detection, it is necessary, e.g., to retrieve the multiplicity restrictions specified for a certain association end. These multiplicity restrictions can be INTEGER values. The statement `(attribute-filler end1 2 lowerValue)` can be added to a RACER *Abox*, indicating that the lower value of the multiplicity restriction for the association end `end1` is 2. The query language must enable the retrieval of these told values.

Completeness of answer sets State-of-the-art DL systems are sound and complete reasoning systems. The question arises if the query language must be complete too.

By exploiting *Tbox* information and the standard DL reasoning tasks, the information explicitly stated in an *Abox* can be extended by inferred information. Consider an *Abox* where it is explicitly stated that a certain individual i is an instance of the concept `InstanceSpecification`, and this individual is related to another individual j through the role `is-instance`. This role is defined in the *Tbox* as `(define-primitive-role is-instance :domain InstanceSpecification :range Class)`. If this *Tbox* information is used by a query retrieving all the instances of the concept `Class`, the individual j will be part of the answer set of the query, while j is not explicitly stated an instance of `Class`.

For our purposes, in most cases, we want to search through the explicitly told information from an *Abox*. We will discuss in Chapter 10 what the possible implications of (in)completeness of answer sets are on inconsistency detection and related tool support.

Integrated in a DL system The query language is preferably an integral part of a DL system. This integration eliminates a drastic communication overhead between the DL system and the query processor. It also allows the incorporation of several optimisation techniques in the query mechanism.

7.2.3 *nRQL*

Based on, among others, [Sim03] and [VSM03], a query language for RACER, called the *new* RACER *Query Language*, or *nRQL*, was implemented by Michael Wessel [HMSW04] and included in RACER version 1.7.19. The definition of this query language was first published in [HMSW04]. We contributed to this publication by using this query language in the context of inconsistency detection.

We include the definition of an *nRQL* query and *nRQL* atom. For the complete syntax and semantics of this query language, we refer the interested reader to [HMSW04] and the *nRQL* manual [Wes04]. Some notations are introduced first.

Notation 9 *[HMSW04] Let $\mathcal{I}$ and $\mathcal{V}$ be two disjoint sets of* individual names *and* variable names*, respectively. The set $\mathcal{O} =_{def} \mathcal{V} \cup \mathcal{I}$ is the set of* object names*. We denote variable names (or simply variables) with letters $x, y, \ldots$; individuals are named $i, j, \ldots$; and object names $a, b, \ldots$.*

Definition 69 *[nRQL Query Bodies, Queries & Answer Sets] [HMSW04] A* nRQL Query *has a* head *and a* body. Query bodies *are defined inductively as follows:*

- *Each nRQL atom rqa is a body; and*
- *If $b_1 \ldots b_n$ are bodies, then the following are also bodies:*
  - $b_1 \wedge \cdots \wedge b_n$, $b_1 \vee \cdots \vee b_n$, $\backslash(b_i)$

We use the syntax $body(a_1, \ldots, a_n)$ to indicate that $a_1, \ldots, a_n$ are all the objects ($a_i \in \mathcal{O}$) mentioned in body. A nRQL Query *is then an expression of the form*

$$ans(a_{i_1}, \ldots, a_{i_m}) \leftarrow body(a_1, \ldots, a_n),$$

The expression $ans(a_{i_1}, \ldots, a_{i_m})$ is also called the head*, and $(i_1, \ldots, i_m)$ is an index vector with $i_j \in 1 \ldots n$. A* conjunctive nRQL query *is a query which does not contain any $\vee$ and $\backslash$ operators.*

Query atoms are the basic syntax expressions of *nRQL* and can be defined as follows:

Definition 70 (Query Atoms) *[HMSW04] Let $a, b \in \mathcal{O}$; C be an $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$ concept expression, R a role expression, P one of the concrete domain predicates offered by* RACER*; $f = f_1 \circ \cdots \circ f_n$ and $g = g_1 \circ \cdots \circ g_m$ be feature chains such that f_n and g_m are attributes (whose range is defined to be one of the available* concrete domains *offered by* RACER*, or f, g are individuals from one of the offered concrete domains which means that $m, n = 1$ and f_1, g_1 are 0-ary attributes). Then, the list of* nRQL atoms *is given as follows:*

- *Unary concept query atoms:* $C(a)$
- *Binary role query atoms:* $R(a, b)$
- *Binary constraint query atoms:* $P(f(a), g(b))$
- *Unary bind-individual atoms:* $bind\_individual(i)$
- *Unary has-known-successor atoms:* $has\_known\_successor(a, R)$
- *Negated atoms: If rqa is a nRQL atom, then so is* $\backslash(rqa)$*, a so-called* negation as failure atom *or simply* negated atom.

For the explanation of the different kinds of atoms, we also refer to [HMSW04] and [Wes04]. This query language fulfils all the above specified requirements. Definition 69 is a superset of our first requirement. The *unary concept query atom* corresponds to the requirement for a **concept query atom**. The *binary role query atom* corresponds to the requirement for a **role query atom**. *Binary constraint query atom* corresponds to the requirement for a **concrete domain attributes atom**. The *has-known-successor atom* only retrieves the individuals for which a role filler assertion is explicitly stated. This atom fulfils the requirement that it must be able to check the **presence of explicitly stated role fillers**. Definition 69 and negated atoms fulfil the requirement of having two kinds of negation, i.e., **true negation and negation by failure**.

Definition 69 has been extended to include projection operators in the head of a query. These projection operators allow the retrieval of told values of concrete domain attributes. This exactly corresponds to our requirement **retrieve told values**.

Furthermore, *nRQL* is integrated in the RACER server and different optimisations are applied. It offers different degrees of completeness. The *nRQL* engine only takes into account explicitly presented individuals for inference and search. The engine also allows for incremental, concurrent querying of *Aboxes*.

For the sake of completeness, we also have to mention that other specifications for query languages for DL systems exist. DLs are also used in the context of ontology languages and the Semantic Web. DAML and OWL are two examples of Web ontology languages based on DLs. In the context of the Semantic Web, query language specifications for DAML and OWL (i.e., DQL and OWL-QL) are defined and also protocols for a query answering dialogue. Some DQL implementations already exist, but all of them have some drawbacks compared to *nRQL*. For a comparison between the different approaches, we refer to the Bachelor thesis of Birte Glimm [Gli04].

7.2.4 Inconsistency Detection using *nRQL*

We will elucidate how the features implementing our requirements are used for the detection of some of our classified inconsistencies.

Each of the queries defined in this section, uses the translation (defined in Chapter 6) of the metamodel fragment (defined in Chapter 2). The translation of this metamodel fragment to RACER statements can be found in Appendix A.

It is quite obvious that the first three requirements – concept query atoms, role query atoms and concrete domain attribute atoms – are necessary for inconsistency detection using the encoding of the UML metamodel in DL. It is also quite obvious that we will restrict the queries to explicitly represented individuals. In the remainder of this section, we exemplify the features implementing the remaining requirements by different queries which are necessary for the detection of some of our classified inconsistencies.

Concrete Domain Attributes Atom

In the case of a *cyclic composition inconsistency* we want to retrieve (at least) the classes, related by inheritance and the composition relation introducing an inherited cyclic composition inconsistency. Due to the fact that we would like to have tuples, we have to define a complex query over the *Abox* assertions.

The following *nRQL* query returns the classes and association involved in the inconsistency.

$$
\begin{aligned}
& ans(c_1, c_2, assoc) \\
& \quad \leftarrow \quad general(c_2, c_1) \wedge ownedAttribute(c_2, end_1) \wedge \\
& \qquad\quad aggregation(end_1, aggregKind) \wedge composite(aggregKind) \wedge \\
& \qquad\quad memberassociation(end_1, assoc) \wedge memberassociation(end_2, assoc) \wedge \\
& \qquad\quad ownedAttribute(c_1, end_2) \wedge (\exists(lower).{\geq_1})(end_2)
\end{aligned}
$$

The superclass relationship is represented by the `direct_superclass` role which is a subrole of a transitive `general` role. A class involved in an association end, is linked to this association end by the role `ownedAttribute`. An association end has an aggregation kind which can be none or an aggregation or a composite. This knowledge is represented by the roles `aggregation` and the concept `composite`. An association has two or more association ends, the role `memberassociation` links the association ends to its association. Each association end has a multiplicity. In the UML metamodel, a property is a multiplicity element. A multiplicity has a range which has a lower and upper bound. The lower bound is represented by the concrete domain attribute `lower` which has type `Integer`. To be able to check whether this lower bound is greater than or equal to one, a binary constraint query atom is used.

Negation as Failure

Consider the *dangling feature reference inconsistency*. This inconsistency arises when, e.g., a message references an operation that does not exist in the corresponding class nor in any of its ancestors. To be able to retrieve the tuples of classes and operations of which the

operations are currently not owned by the classes, i.e., such that RACER cannot prove that the operation is owned by the class, the $\backslash(\texttt{ownedoperation}(x, y))$ atom can be used.

$$
\begin{aligned}
&ans(m, op, c)\\
&\quad \leftarrow\ message(m) \wedge signature(m, op) \wedge operation(op) \wedge\\
&\qquad receiveevent(m, mend) \wedge coveredsub(mend, lifeline) \wedge lifeline(lifeline) \wedge\\
&\qquad represents(lifeline, connectableel) \wedge base(connectableel, c) \wedge class(c) \wedge\\
&\qquad \backslash(ownedoperation(c, op))) \wedge \backslash(general(superc, c))) \wedge\\
&\qquad ownedoperation(superc, op))
\end{aligned}
$$

m represents a message, op the operation called by m and c the class represented by a certain lifeline $lifeline$ on which the operation is received through a message end $mend$.

Presence of Explicit Role Fillers

To be able to detect a *dangling type reference*, we need to be able to find out which classes do not belong to any model or to a particular model. This means that we want to retrieve the *Abox* assertions that are not explicitly connected to a model. One way to ask for classes which are not connected to a model is by the query: $\neg(\exists\texttt{member}.\top(x))$. However, suppose a concept `ClassinModel` is defined as $\texttt{ClassinModel} \sqsubseteq \texttt{Class} \sqcap \exists\texttt{member}.\top$. If an instance cl of this concept is defined in an *Abox* without defining a role filler for the role `member`, this instance will not appear in the answer set of the above defined query. This is due to the open-world assumption. The `has_known_successor` atom checks whether there are certain explicitly modelled role successors in an *Abox* without retrieving them. If, in the example, the $\backslash($`has_known_successor`$)$ atom is used, the instance cl will be returned in the answer set. This atom is used in the following query returning the attributes whose type is not known to a model.

$$
\begin{aligned}
&ans(prop, atype)\\
&\quad \leftarrow\ property(prop) \wedge definedType(prop, atype) \wedge\\
&\qquad \backslash(has\_known\_successor(atype, member))
\end{aligned}
$$

Told value

To detect a *multiplicity incompatibility*, the defined multiplicities of association ends must be retrieved from the *Abox*. The following query returns the lower bound of the multiplicity specified for a certain association end *end* owned by a class *cl* and belonging to the association *assoc*. The projection operator $told - value$ present in *nRQL* is used to retrieve this lower bound.

$$
\begin{aligned}
&ans(cl, assoc, (told\text{-}value(lowermultiplicity)))\\
&\quad \leftarrow\ memberEnd(assoc, end) \wedge class(end, cl) \wedge lower(end, multiplicity)
\end{aligned}
$$

Completeness of answer sets

Recall the *dangling type reference* inconsistency. In the previous queries we searched for attributes whose type was not known in any model. What happens if the user specifies the type of an attribute but does not define this type explicitly as a class in the UML model? Consider as an example the *Abox*: $\{\texttt{class}(\texttt{atm}), \texttt{ownedAttribute}(\texttt{cash}, \texttt{atm}), \texttt{property}(\texttt{cash}), \texttt{definedType}(\texttt{cash}, \texttt{Cash})\}$. In this *Abox* it is not explicitly stated that `Cash` is a *Type*, i.e., `complexType(Cash)`. The concept `ComplexType` represents the union of `PrimitiveType` and `Class`. Due to the inference mechanism of DLs that takes into account the *Tbox* knowledge, the assertion `ComplexType`(`Cash`) is inferred. If the following query is submitted to this *Abox*, the answer set will be the empty set.

$$ans(prop, atype)$$
$$\leftarrow \quad property(prop) \wedge definedType(prop, atype) \wedge not(complexType(atype))$$

However, due to the different degrees of completeness of *nRQL*, it is possible to reason in different modes: (1) a mode where only syntactic told information is taken into account; (2) a mode where told information plus *Tbox* information for concept names is taken into account; (3) a mode where all inferences are taken into account. If the previous query is submitted in the second mode, the answer set will be $\{cash\}$. In this case, the ancestors of the concepts `property` and `class` are computed and for each told instance of a certain concept, an assertion is added stating that this instance is also an instance of each ancestor of the concept.

Inconsistencies of our classification that need to be detected using *nRQL* queries are: dangling type reference, inherited cyclic composition, connector specification missing, instance specification missing, specification incompatibility and disconnected model.

7.3 Using our DL Framework Representing UML models

In the previous chapter, we showed how class diagrams, PSMs, call sequences, SD traces and the communication view of sequence and communication diagrams can be translated into terminological knowledge. In this case, DLs are called the semantic domain for these UML concepts, i.e., a suitable formal language. Our behavioural inconsistencies (except for the specification incompatibilities) can be detected by using this knowledge and exploiting the standard DL reasoning tasks. For the translation of these diagrams (or parts of these diagrams) into DLs, the different spanning functions can be used. For example, when translating *receiving* SD traces, only *part* of a mode, in this case the receiving event occurrences of a certain set of instances, are taken into account.

7.3.1 The Use of *Abox* Reasoning Tasks

An example of an inconsistency that can be detected in this way, is the *multiplicity incompatibility*. This inconsistency arises when a connector in a sequence (or communication) diagram does not respect the multiplicity restrictions imposed by the class diagram. Each multiplicity has a lower and upper bound. The multiplicity restrictions that can be checked

are with respect to this lower and upper limit. Due to the open-world assumption the lower bound must be checked using a *nRQL* query as shown in the previous section.

When the class diagram is translated into terminological knowledge using the translation introduced in Section 6.4 and the communication view of the sequence (or communication) diagram is translated as specified in Section 6.6, the reasoning task Abox *consistency* can be used. *Abox* consistency will (among others) check if the upper bounds of the multiplicities specified on the association ends defined in the class diagram are respected by the different specified links in the *Abox*. This means that the number of individuals which are instances of the concepts representing the association ends must conform to the number restrictions specified on the roles representing the association ends. As a result, multiplicity incompatibility is checked for sequence diagrams on instance level.

7.3.2 The Use of *Tbox* Reasoning Tasks

Behaviour compatibility, whether it is on the instance or specification level, guarantees the compatibility between PSMs of certain classes and receiving SD traces of instances of these classes. Tbox *coherence* can be used to check this inconsistency. The PSM(s) are translated to terminological knowledge as specified in Section 6.5.1. The SD traces are translated using the translation of Section 6.6. However, for this inconsistency only the receiving SD traces are relevant. Spanning functions are needed *only taken into account detailed meta-information*. An example of such detailed meta-information are the *receiving* SD traces needed to check the behaviour compatibility. This meta-information can be retrieved from the encoded UML metamodel and the user-defined models through *nRQL* queries. The output of these queries is used to encode the different SD sequences.

One could argue that such complicated manipulation and orchestration of queries and spanning functions is not necessary for checking whether a certain SD trace is included in a PSM. Checking the syntactical inclusion of such a trace in a PSM can be done by *nRQL* queries. However, due to the usage of the subsumption relation, we go beyond a merely syntactical inclusion check. Let us illustrate this by two examples.

Reconsider the protocol state machine diagram of Figure 2.14 expressing the valid call sequences on an *ATM* class allowing a withdrawal transaction. In Figure 7.1, two separate receiving SD traces for an *ATM* object are shown.

First the PSM and the SD traces are translated into terminological knowledge using the translation defined in the previous chapter. The receiving SD traces for a given (set of) object(s) can be retrieved by the following *nRQL* query. The *connectableelement* appearing in the header of this query is a constant representing the objects for which the traces must be searched.

$$
\begin{aligned}
&ans(event_1, op_1, event_2, op_2, connectablelement)\\
&\quad \leftarrow \; eventoccurrence(event_1) \wedge coveredsub(event_1, lifeline) \wedge\\
&\qquad represents(lifeline, connectableelement) \wedge (\exists receivemessage.\top)(event_1) \wedge\\
&\qquad eventoccurrence(event_2) \wedge coveredsub(event_2, lifeline) \wedge\\
&\qquad represents(lifeline, connectableelement) \wedge (\exists receivemessage.\top)(event_2) \wedge\\
&\qquad before(event_1, event_2)
\end{aligned}
$$

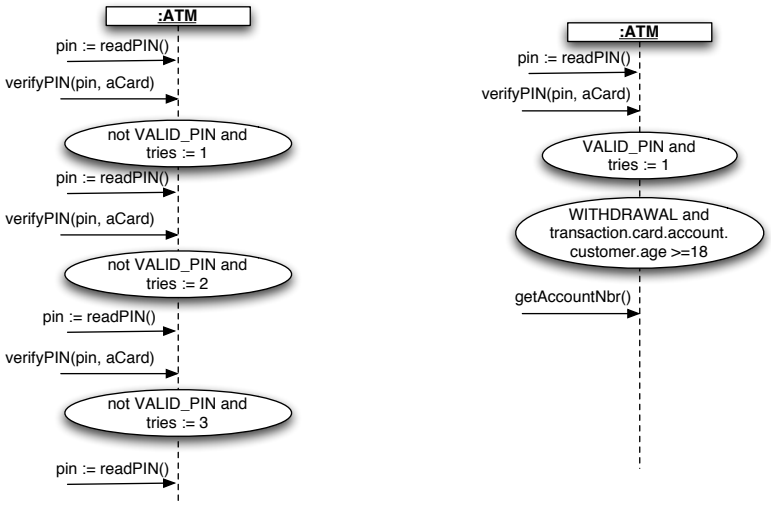

Figure 7.1: Different SD traces.

Using a spanning function, the SD traces can be transformed into terminological knowledge. Finally, the following *Tboxes* are generated. The first one, shown in the RACER Fragment 7.1, contains the encoding of the call sequences of the PSM and the completeness of this set of call sequences (statements (1) until (35)). It also contains the receiving SD trace shown on the left-hand side of Figure 7.1 (statements (36) until (46)). The ovals on the lifelines in Figure 7.1 indicate so-called state invariants. These invariants specify constraints on the state of a lifeline, i.e., they put restrictions on the values of the properties of the objects represented by the lifeline.

The second *Tbox* contains the same encoding of call sequences of the PSM, i.e., statements (1) until (35) of RACER Fragment 7.1, and also the receiving SD trace shown on the right-hand side of Figure 7.1. The encoding of this SD trace is shown in the RACER Fragment 7.2.

The reasoning task Tbox *coherence* can be used on both *Tboxes*. In both cases this reasoning task checks if the PSM and the SD trace are behaviour compatible. In the first case, the *Tbox* is incoherent resulting in a behaviour incompatibility. The reasoning not only takes into account the sequencing of the operation calls but also the constraints specified. The behaviour of the SD trace shown on the left-hand side of Figure 7.1, is incompatible with the PSM call sequences, because the PSM specifies that after 3 times entering a PIN code, the card is retained instead of asking the PIN again. The behaviour of the SD trace shown on the right-hand side of Figure 7.1, is behaviour compatible with the PSM call sequences. The corresponding *Tbox* is coherent resulting in a behaviour compatibility.

For other inconsistencies, for example, *interaction inconsistencies*, UML metamodel information is also needed to be able to translate certain parts of UML models into terminological knowledge. Consider the *invocation interaction consistency* as defined in Section 3.7.1. GCI's can now be constructed for the different *eventoccurrences*. The traces belonging to the child interaction are considered *to be complete*. If in this dissertation, we indicate that a certain UML diagram or set of UML elements is considered *to be complete*, then it

```
(1)  (define-concept t1 (and (all op readPIN) (exactly 1 op)))
(2)  (define-concept t2 (and (all op verifyPIN) (exactly 1 op)))
(3)  (define-concept t3 (and (all op retainCard) (exactly 1 op)))
(4)  (define-concept t4 (and (all op ejectCard) (exactly 1 op)))
(5)  (define-concept t5 (and (all op cancel) (exactly 1 op)))
(6)  (define-concept t6 (and (all op getAccountNbr) (exactly 1 op)))
(7)  (define-concept t7 (and (all op getAmountEntry) (exactly 1 op)))
(8)  (define-concept t8 (and (all op checkIfCashAvailable) (exactly 1 op)))
(9)  (define-concept t9 (and (all op send ) (exactly 1 op)))
(10) (define-concept t10 (and (all op dispensecash) (exactly 1 op)))
(11) (define-concept t11 (and (all op printReceipt) (exactly 1 op)))
(12) (define-concept g1 (and (not (some valid-PIN top)) (¡ tries 3)))
(13) (define-concept g2 (some valid-PIN top))
(14) (define-concept g3 (and (not (some valid-PIN top)) (= tries 3)))
(15) (define-concept g4 (some withdrawal top))
(16) (define-concept g5 (some checkcashavailable top))
(17) (define-concept g6 (not (some checkcashavailable top)))
(18) (define-concept g7 (some allowedwithdrawal top))
(19) (define-concept g8 (not (some allowedwithdrawal top)))
(20) (disjoint valid-PIN withdrawal checkcashavailable allowedwithdrawal)
(21) (implies t1 (all r t2))
(22) (implies t2 (all r (or (and g1 t1) (and g3 t8) (and g2 t9) (and g2 g4 t3))))
(23) (implies (and t1 g1) (all r t2))
(24) (implies t4 (all r t5))
(25) (implies t5 (all r (or (and g6 t9 (not g4)) (and g6 g4 t3) (and t6 g5))))
(26) (implies (and g5 t6) (all r (or (and t10 g7) (and g8 t9 (not g4)) (and g8 g4 t3))))
(27) (implies (and g7 t10) (all r t7))
(28) (implies t7 (all r (or t9 (and t3 g4))))
(29) (implies (or (and g8 g4 t3) (and g4 t3) (and g4 g6 t3) (and g2 g4 t3)) (and (all r t4)))
(30) (implies (and g3 t8) (all r (and t8 g3)))
(31) (implies (and g2 t9) (all r (and g2 t9)))
(32) (implies (and g6 t9) (all r (and g6 t9)))
(33) (implies (and g8 t9) (all r (and g8 t9)))
(34) (implies t9 (all r t9))
(35) (disjoint t1 t2 t3 t4 t5 t6 t7 t8 t9 t10)
(36) (define-concept m1 (and (all op readPIN) (exactly 1 op)))
(37) (define-concept m2 (and (all op verifyPIN) (exactly 1 op)))
(38) (define-concept cons1 (and (not (some valid-PIN top)) (= tries 1)))
(39) (define-concept cons2 (and (not (some valid-PIN top)) (= tries 2)))
(40) (define-concept cons3 (and (not (some valid-PIN top)) (= tries 3)))
(41) (implies m1 (some r m2))
(42) (implies m2 (some r (and m1 cons1)))
(43) (implies (and m1 cons1) (some r m2))
(44) (implies m2 (some r (and m1 cons2)))
(45) (implies (and m1 cons2) (some r m2))
(46) (implies m2 (some r (and m1 cons3)))
```

RACER Fragment 7.1: RACER expressions representing a PSM and SD trace.

```
(define-concept m1 (and (all op readPIN) (exactly 1 op)))
(define-concept m2 (and (all op verifyPIN) (exactly 1 op)))
(define-concept m3 (and (all op getAccountNbr) (exactly 1 op)))
(define-concept cons1 (and (some valid-PIN top) (some trans (some
transcard (some cardacc (some acccust (min age 18)))))))
(implies m1 (some r m2))
(implies m2 (some r (and cons1 m3)))
```

RACER Fragment 7.2: RACER expressions representing an SD trace.

is assumed that this diagram or set contains all relevant elements. The set of SD traces of the parent interaction do not have to be complete, they only have to be included in the set of SD traces of the child interaction. By translating both sets of SD traces into the same *Tbox*, *Tbox* coherence can be used to check whether invocation interaction consistency is guaranteed.

Observation interaction consistency can also be checked using *Tbox* coherence. For this consistency check, only the event occurrences belonging to the child interaction and denoting the receipt of messages invoking operations known to the parent interaction are taken into account. Again a *nRQL* query can be defined to retrieve this information. This information acts as input for the spanning functions.

Other inconsistencies that can be checked in the same way are interaction inconsistencies on specification/instance level, invocation and observation inheritance inconsistencies and instance behaviour incompatibility.

7.4 Discussion and Related Work

7.4.1 Related work

Related work defining different kinds of inconsistencies is discussed in Chapter 3. From this discussion, we conclude that all those works use different ways to detect the inconsistencies ranging from attributed graph grammars, over OCL, over CSP, XMI parsers and SQL to name only a few. The related work presented in this section is divided into three categories. The first two categories are previously defined in the introduction and in Spanoudakis and Zisman [SZ01]. We do not consider the category *human-based collaborative exploration* (described in [SZ01]) because the aim of this work is to provide *formal* definitions and detection of inconsistencies. The model checking approach is already discussed in the previous chapter.

Logic-based Approach

Finkelstein *et al.* [FGH+93] explain that consistency between partial models is neither always possible nor is it always desirable. They present inconsistency handling between *Viewpoints*, locally managed software models. Viewpoints and inter-Viewpoint rules are all translated to first-order predicate logic, and inconsistencies are identified using a classical logic theorem prover with the *Closed World Assumption*. The inconsistencies proved are domain-dependent and are about syntactical and structural information of a certain domain.

Specialised Forms of Analysis Approach

Ehrig and Tsiolakis [ET00] investigate the consistency between UML class and sequence diagrams. UML class diagrams are represented by attributed type graphs with graphical constraints, and UML sequence diagrams by attributed graph grammars. As consistency checks between class and sequence diagrams only existence, visibility and multiplicity checking are considered. In [Tsi01] the information specified in class diagrams and state diagrams is integrated into sequence diagrams. The information is represented as constraints attached to certain locations of the object lifelines in the sequence diagram. The supported constraints are data invariants and multiplicities on class diagrams and state and guard constraints on state diagrams.

Fradet *et al.* [FMP99] use systems of linear inequalities to check consistency for multiple view software architectures. The consistencies are specified for a particular application domain, i.e., a software architecture. A software architecture is defined in their approach as a specification of the global organization of software involving components and connections between them. The views are represented by graphs and consistencies are defined as specific relationships that should be fulfilled between certain domain-dependent nodes and edges. These consistencies can be defined and detected in our approach by DL reasoning tasks and *nRQL* queries.

In [KB01] a consistency checker is implemented using an XML parser and by translating the structure of the XMI documents representing UML models into Prolog. Although there is an UML XMI specification, every tool vendor has created his own XMI. As a result the approach taken in [KB01] is very tool dependent. The checks are limited to syntactical errors and completeness/omissions in UML class, state and sequence diagrams.

In [LCM+03] and [Lan03] inconsistencies and incompleteness issues between UML diagrams are presented and detected using relational databases and SQL. Only our structural inconsistencies and to a certain extent some of our behavioural inconsistencies, can be detected using their approach.

The VIEWINTEGRA approach proposed in [Egy01] checks consistency between different UML diagrams through a direct comparison of the diagrams involved. Diagrams are transformed to 'something alike'. For example, to compare UML state diagrams and sequence diagrams, sequence diagrams are first translated into a state diagram, which is then compared to the original state diagram. Inconsistency rules are comparisons that describe if and how much the diagrams differ. Only syntactical inconsistencies are detected.

7.4.2 Advantages and Limitations of our Approach

The advantages of using *nRQL* queries are quite clear. A *nRQL* query defining an inconsistency and at the same time detecting the inconsistency provides the user with the concrete model elements causing the particular inconsistency. The different modes of completeness are very useful as explained in the first section. Different application domains can use different completeness degrees. A disadvantage of this flexibility is that the results of a query in different degrees of completeness must be correctly interpreted. To be able to correctly interpret these results, one must rather be an advanced user.

Next to the subset of our classified inconsistencies that can be detected by *nRQL* queries, also so-called incompleteness issues can be detected by *nRQL* queries. Such incompleteness issues are described in [KB01] and [Lan03]. An example of such an incompleteness is an

operation that is not called in any scenario. This can be easily checked by the following query.

$$
\begin{aligned}
&ans(op, (told\text{-}value(name\ has\text{-}name)))\\
&\quad \leftarrow\ ownedoperation(cl, op) \wedge base(cl, connectableelement) \wedge\\
&\qquad represents(lifeline, connectableelement) \wedge eventoccurrence(event) \wedge\\
&\qquad receivemessage(event, m) \wedge not(signature(m, op))
\end{aligned}
$$

The current disadvantage of inconsistency detection using *Tbox* coherence or *Abox* consistency checking, is the lack of feedback given to the user. If a *Tbox* is not coherent, the DL reasoning engine returns the set of unsatisfiable concepts. From this information only, we are not able to deduce, e.g., in case of a behaviour incompatibility, which SD traces conform to call sequences in the PSM. To be able to inform the user correctly, i.e., to back-annotate the UML model with information concerning the cause of the inconsistency, two items must be further investigated. First, current DL tools, such as RACER, should give more and proper feedback on the cause of the satisfiability problem in case we check for *Tbox* coherence or *Abox* consistency. Second, the necessary information for reconstructing a UML model from a DL translation must be stored (including lay-out information of the UML diagrams).

By using DLs as semantic domain for UML models, other properties and conformance checks besides our defined inconsistencies, can be checked. An example is a conformance check between sequence or communication diagrams at instance level and sequence or communication diagrams at specification level. The interaction view as well as the communication view of sequence or communication diagrams at instance level is interpreted as an instance of the interaction, resp. communication view of the corresponding diagrams at specification level. *Abox* consistency can be used to check this conformance. Using *Tbox* coherence, the conformance can be checked between the communication view of sequence or communication diagrams at specification level and corresponding class diagrams. All diagrams are translated into terminological statements.

By translating constraints such as invariants and the corresponding diagram, e.g., a class diagram, into a DL *Tbox*, the constraints can be checked for consistency with respect to the class diagram. A constraint is consistent with respect to the class diagram if it can be satisfied without contradicting the conditions imposed by the classes. The set of constraints defined on a certain class can be checked for internal consistency. Furthermore, it would be possible to check constraint equivalence, this boils down to equivalence of logical formulae. Finally, the constraints used in guards on PSMs or on SD traces can be checked for mutual exclusiveness.

7.4.3 Evaluation of Criterion #2

For each of the detection approaches introduced in this chapter, we can conclude that the detection queries or evaluation functions have a precise semantics. The inconsistencies and their detection are expressed in DLs terminological or assertional knowledge or in *nRQL*, all having a precise semantics.

Table 7.3 gives an overview of the formal properties holding for the different detection approaches. All these approaches are *sound* and *decidable*. *Completeness* can vary due to

	Metamodel	DL Framework
Soundness	yes	yes
Completeness	varying degree	varying degree
Decidability	yes	yes

Table 7.3: Formal properties of detection approaches.

the different completeness modes of *nRQL*. Because *nRQL* queries are used in the spanning functions of our DL framework, the second inconsistency detection approach also has a varying degree of completeness. Depending on the completeness mode used by *nRQL*, complete or rather incomplete knowledge will be translated into terminological knowledge.

7.5 Conclusion

In this chapter, we introduced two different ways to detect our classified inconsistencies. A first way for detection is to *query meta-information*. This meta-information cannot be obtained without a sophisticated DL query language. This requirement is motivated by presenting an example of earlier work. We also introduced several requirements for such a query language in the context of inconsistency detection. Based on these requirements, the authors of RACER developed a query language for RACER, named *nRQL*. Several *nRQL* queries are presented. In each of the queries a feature was needed corresponding to the introduced requirements.

DLs as a *semantic domain together with spanning functions* can be used for the detection of behaviourial inconsistencies. The different possible detection approaches of our classified inconsistencies are summarised in a revisited classification of our inconsistencies.

The question remains whether DLs or DL systems can be used for the resolution of inconsistencies. More precisely, is it possible to define resolution actions and to execute these actions in a highly customisable, flexible and interactive way? This is the topic of the next chapter.

Chapter 8

A Rule-Based DL Inconsistency Resolution Approach

In this chapter, an inconsistency resolution approach using DLs is brought forward based on the requirements introduced in Chapter 4. First, we show how resolution actions can be defined as *Tbox* and *Abox* assertions (Section 8.1). Then, we briefly review the different challenges an inconsistency resolution approach has to deal with (Section 8.2). Based on these challenges, a motivation for using a rule-based approach is given (Section 8.3). Before explaining the relation between DLs and rules (Section 8.5), we briefly introduce rule-based systems (Section 8.4).

nRQL is not only a query language and mechanism but also offers some support for the definition and application of rules. This rule part of *nRQL* is still in its infancy. First, we specify the requirements for a DL rule-based system suited for the resolution of inconsistencies and next, we show to which extent *nRQL* satisfies our requirements (Section 8.6).

Finally, related work is discussed and the DL rule-based inconsistency resolution approach is evaluated based on criterion #3 – precise definitions and management of interactive inconsistency resolutions – of Chapter 4 (Section 8.7).

8.1 Definition of Resolution Actions

In this section, resolution actions are defined as assertions, or concept declarations or role declarations.

8.1.1 At *Abox* level

Aboxes can represent user-defined models as instances of the UML metamodel or they can represent user-defined models as instances of other UML user-defined models. For example, the communication view of a sequence diagram at instance level can serve as the interpretation of the communication view of a sequence diagram at specification level. In both cases resolution of inconsistencies detected by queries on these *Aboxes*, boils down to a certain set of *Abox* assertions.

In Chapter 4, we defined three different categories of resolution actions on metamodel level: adding an instantiation of certain metamodel elements, deleting an instantiation of

a certain metamodel element and changing the references of an instantiation of a certain metamodel element.

We show which *Abox* assertions can be used for each category of resolution actions. The RACER syntax is used to introduce the different possible assertions. This will allow us to use these assertions in the next sections and chapters without further explanation.

Add model element

Creating an instantiation of a UML metaclass corresponds to the creation of an instance of a certain DL concept. The syntax for asserting an individual in RACER, is `(instance` In C`)` declaring In as being an individual of the concept C. For example, `(instance session class)` specifies that the individual `session` is an instance of the concept `class`.

Remove model element

Deleting an instantiation of a UML metaclass corresponds to the removal of an instance of a certain DL concept. The syntax for removing an individual in RACER, is `(forget-concept-assertion` In C`)` declaring the removal of In as being an individual of the concept C. For example, `(forget-concept-assertion verifyPIN operation)` specifies that the individual `operation` is being retracted as an instance of the concept `operation`.

Change model element

Changing an instantiation of a UML metamodel element corresponds to changing its references to other instances through certain meta-associations. The syntax for asserting that the individuals $In1$ and $In2$ are related by the role R is `(related` $In1$ $In2$ R`)`. For example, the assertion `(related session getcustomerspecifics ownedoperation)` relates the individual `session` with `getcustomerspecifics` through the role `ownedoperation`. The syntax for retracting a role assertion from an *Abox* is `(forget-role-assertion` $In1$ $In2$ R`)`. For example, the assertion `(forget-role-assertion atm verifyPIN ownedoperation)` retracts the assertion that the individual `atm` and the individual `verifyPIN` are related to each other through the role `ownedoperation`.

Assertions also exist for the addition and retraction of values for concrete domain attributes. Concrete domain assertions can be added to the *Abox* using `(constrained In On An)`. This statement asserts that an individual In is related to a concrete domain object On via an attribute An. Concrete domain predicates can be built and asserted using the statement `(constraints forms)`, where `forms` is a set of concrete domain predicate assertions. For example, the statement `(constrained verifyPIN verifyPINname operationname)` adds the attribute `operationname` to the individual `verifyPIN`. The assertion `(constraints (string= verifyPINname "verifyPIN"))` expresses that the individual `verifyPIN` has as name the string `"verifyPIN"`.

It is sometimes necessary to group several of these primitive resolution actions, for example in the context of a certain application domain. Examples of such groupings will be shown in Section 8.6.1 and in the next chapter in the context of model refactorings.

8.1.2 At *Tbox* level

If DLs are used as semantic domain, additions, changes and deletions of semantical concepts such as SD traces and PSM call sequences can be made on *Tbox* level. In Table 6.1, the RACER syntax is stipulated for the addition of concept and role axioms. Such axioms can be retracted from the *Tbox* by the general `forget` statement. Only explicitly stated information can be forgotten.

However, inconsistencies detected in DL *Tboxes* representing a certain model, can be resolved by adapting the *Abox* containing the model as instances of the UML metamodel and regenerating the corresponding terminological knowledge. Only constraints represented on the different models need to be adapted directly in the corresponding *Tbox*, because we did not take into account a metamodel representation of a certain constraint language. Consider for example, a sequence diagram that is specification behaviour incompatible with a certain PSM because a message is missing in the sequence diagram. A possible resolution is to add the missing message in the sequence diagram. In the corresponding *Abox* instances of the relevant metaclasses can be created (e.g., an instance of *Message*, instances of *Eventoccurrence*) and related to corresponding instances of metaclasses (e.g., the instance of *Message* is related to the correct instance of *Operation* and to the created instances of *Eventoccurrence*).

8.2 Challenges of Inconsistency Resolution

We quickly restate the different challenges, first presented in Section 4.2, that are related to the inconsistency resolution activity.

- Which resolution actions to choose can be dependent on the cause of an inconsistency. However, this dependency cannot always be captured without extra knowledge provided by the software developer. As a result, most of the time the execution of resolution actions involves a certain amount of user interaction.
- Depending on the granularity of the resolution actions and the application domain involved, it might be necessary to group and combine several resolution actions. How these resolution actions are grouped or combined is not always fixed but can be domain-dependent.
- Resolution actions can introduce new inconsistencies. The dependencies between different inconsistencies caused by resolution actions must be taken into account by an inconsistency resolution approach.

8.3 Motivation for a Rule-Based Approach

In the remainder of this dissertation, *an inconsistency resolution* encompasses the detection of a particular inconsistency and one possible way of resolving it.

Rule-based reasoning is used to imitate human thought and problem solving based on the explicit representation of human knowledge as rules. A *rule* has the structure *IF condition THEN conclusion* or a variation thereof. Rule-based systems provide an explicit abstraction for representing rules. A rule can be used to represent a particular inconsistency resolution.

This means that the detection and the resolution actions of an inconsistency resolution are encapsulated.

Each detected inconsistency can be resolved in several different ways. The selection of a particular resolution for an inconsistency can depend on the particular state of the model. For the most part, however, selecting a resolution is a matter of preference of the person who is resolving the inconsistency. A very important characteristic of a rule-based system is that the *definition* of a rule is separated from the *strategy* for selecting the rules to be fired from the applicable ones.

The inconsistency resolution activity becomes even more complicated because resolving a certain inconsistency can introduce other inconsistencies. In Section 4.2.3 a piece of a dependency graph between inconsistencies and resolution actions is shown in Figure 4.3. Note that when implementing such a dependency graph in an imperative programming language using a traditional conditional statement, this results in a complex program which is not robust to changes: similar to the different edges and nodes in the dependency graph, the programmer has to order and nest conditional statements manually. Indeed, every possible situation in the model leads to a potentially different flow of inconsistency resolutions. Moreover, the same inconsistency resolution can occur multiple times in different combinations with other inconsistency resolutions. Rule-based systems dramatically boost reuse of rules, since they only have to be defined once and the appropriate ones for a certain situation are fired. As such, *implicit* flows of rules or, in our case, inconsistency resolutions are constructed.

8.4 Rule-Based Systems

A rule-based system is employed for representing rules explicitly and overcoming the problems mentioned in the previous section. A rule-based system provides a language to define rules which are modular structures for expressing rule-based knowledge. A rule states how to infer new data or manipulate existing data. Although a rule resembles a conditional statement from imperative programming languages, it is not activated in any predetermined order relative to other conditional statements. On the contrary, a *rule engine* is responsible for determining the set of applicable rules and the order in which they are fired.

A *forward-chaining* rule engine considers a rule to be applicable when changed data matches its condition. If this rule is fired, the result is that its conclusion is evaluated for each set of matching data. *Backward-chaining* engines attempt to prove a goal by finding applicable rules that would conclude it. When such an applicable rule is fired, the rule engine similarly tries to prove its condition — the new goal.

Typically, a rule engine employs a particular strategy for selecting and ordering certain rules to be fired from all the applicable rules. This is done to limit the search space and optimise the search process.

8.4.1 Inconsistency Resolution Rules

A generic inconsistency resolution rule has the form: `IF inconsistency X occurs in model M THEN change model M so that X is resolved`. There are typically multiple resolutions for a particular inconsistency and each one is represented by one rule. Hence,

all rules pertaining to a certain inconsistency `X` have the same expression `inconsistency X occurs in model M` in their conditions.

The occurence of an inconsistency in a model is detected by querying the data representing the model, i.e., the model elements. A certain state of the model attests to the presence of a particular inconsistency.

A rule's conclusion states how to resolve the detected inconsistency. It consists of a sequence of statements, where each statement is responsible for either adding data to the model or removing data from the model. As such, the model elements are rearranged so that the inconsistency is resolved. However, in order for a certain inconsistency resolution to be applicable, some model elements typically need to be present or in a particular configuration.

For example, consider a rule for resolving the dangling feature reference inconsistency:

IF the dangling feature reference inconsistency is detected for an operation in a class
AND the operation is defined in another class
THEN add the operation to the first class

The condition of the rule is a conjunction. The first part of the conjunction takes care of detecting the inconsistency. The second part of the conjunction establishes if the model is in a particular situation, more specifically if the missing operation is already defined in another class. If so, the inconsistency resolution of adding the missing operation to the first class is applicable. This resolution, described in the conclusion of the rule, is executed when establishing the condition succeeds. Note that the condition of the rule should in fact be interpreted as *for all* operations and classes for which the dangling feature reference inconsistency is detected. Then, the conclusion is executed for each operation and class pair that fulfils the condition.

In the remainder of this chapter, we will investigate the relation between DLs and rules and how rules can be defined on DL knowledge bases expressing the resolution of our classified inconsistencies.

8.5 Description Logics and Rules

In the DL systems Classic and Loom, in addition to terminological knowledge and assertional knowledge, rules can also be used to express knowledge.

Based on Loom's rule system, it is possible to specify additional necessary conditions for individuals which are explicitly mentioned in the *Abox* and are derived to be instances of a certain defined concept. These necessary conditions are called *implications* in Loom.

In Classic, rules are applied to individuals explicitly named in the *Abox*. Rules are applied in a forward chaining way. Two different types of rules are differentiated, *description* rules and *filler* rules. All Classic rules have as their antecedent a named concept and are fired on an individual when the individual is classified as an instance of the concept. The consequent of a Classic description rule is a classic description, which, when the rule fires on an individual, is merged into the description of the individual. The consequent of a Classic filler rule is the name of a role and a `Lisp` function that will be invoked when the rule fires. The function is passed the individual the rule fired on and the role named in the consequent. It returns a list of new role fillers for that role and individual.

The simplest variant of such rules included in Loom and Classic, is an expression of the form $C \Rightarrow D$, where C, D are concepts. The meaning of such a rule is *if an individual*

is proved to be an instance of C, then derive that it is also an instance of D. These rules are called *trigger rules.*

As defined in [BCM+03], the semantics of a finite set of trigger rules can be described operationally, by a forward reasoning process. Starting with an initial knowledge base, a series of knowledge bases is constructed. Each knowledge base is obtained from a previous knowledge base by adding a new assertion $D(a)$ whenever a rule $C \Rightarrow D$ exists and $C(a)$ is entailed by the latter knowledge base. Note that there is an important difference between the trigger rule $C \Rightarrow D$ and the inclusion axiom $C \sqsubseteq D$. A trigger rule is not equivalent to its contrapositive $\neg D \Rightarrow \neg C$, as opposed to the GCI $C \sqsubseteq D$.

The presented query language *nRQL* is not only a query language but also provides rules as a simple *Abox* augmentation mechanism. In the next section, we will present the requirements for an ideal DL rule-based inconsistency resolution mechanism and we will also investigate to which extent the current rule system of RACER can be employed.

8.6 Rule-Based DL System

Based on the challenges related to inconsistency resolution and by putting these challenges in the context of the different inconsistency detection ways, we discuss the definition of inconsistency resolution rules in detail. Next, we explain how an engine can select and activate the inconsistency rules. From the definition of inconsistency rules, a set of requirements for a rule-based DL system is distilled. Finally, the existing *nRQL* environment is presented and evaluated using the defined requirements.

8.6.1 Rule Definition

An earlier section pointed out the possible format of a generic inconsistency resolution rule: `IF inconsistency X occurs in model M THEN change model M`. The rule's condition corresponds to the detection of an inconsistency, while it's conclusion corresponds to a resolution action or a group of resolution actions. Two kinds of rules can be distinguished, rules on *Abox* assertions and rules on *Tbox* assertions. We will refer to the first kind as *Abox rules* and to the second kind as *Tbox rules.*

Abox rules

In case we are able to detect inconsistencies with queries over an *Abox*, we can use such a query as a rule condition. The conclusion of a rule consists of actions to be undertaken on the *Abox* assertions, ensuring that the model's state is altered in such a way that the inconsistency is resolved. There are a number of important issues to be addressed. We illustrate these issues with an example set of rules for resolving the dangling feature reference inconsistency. Recall that a dangling feature reference occurs when an operation is not known to a class on which it is called in a UML sequence or PSM diagram. Suppose a dangling feature reference is detected on a UML sequence diagram. Two possible resolutions are (1) if the operation is known to another class, move the operation to the class involved in the inconsistency; (2) if the operation is known to another class, change the message such that it sends the operation to the right object.

When successfully inferring a rule condition, its variables – if any – are bound to individuals representing UML metaclasses such as operations, classes, lifelines and so on. For example, the conditions of the rules for resolving the dangling feature reference inconsistency typically consist of the query for detecting this inconsistency: `(check-DFR ?class ?op ?m)`. Note that this query has a name and three variables, which are the identifiers preceded by a question mark. When this query is successfully inferred, the result is a set of bindings for the variables, more specifically a set of class-operation-message tuples.

A rule's conclusion typically consists of actions that are applied to each inferred binding of the variables of the rule's condition. Take, for example, the first resolution possibility for the dangling feature reference inconsistency, i.e., simply adding the operation to the class in question. This is (partly) achieved by the following action in the rule's conclusion: `(related ?op ?class2 ownedoperation)`. When the rule is fired, this expression is evaluated, which leads to a number of concrete assertions being added. As remarked earlier in this chapter, also removing existing assertions can be executed by a rule's conclusion.

Conceptually, an inconsistency detection query would be a sufficient condition for inconsistency resolution rules. However, conditions are usually not only used for checking the situation, but also for retrieving information from the situation. To be able to resolve an inconsistency extra information needs to be retrieved. This information helps in deciding which resolution possibility should be chosen. This becomes apparent when revisiting the previously stated resolution possibilities for the dangling feature reference inconsistencies. Both resolution possibilities start with stating the following sentence: *if the operation is known to another class*. In order to retrieve this information, the rule needs an additional expression in its condition, for example `(?op ?class2 ownedoperation)`. When this expression is successfully evaluated, the variable `?class2` is bound to the correct class(es) and can be used in the conclusion.

We have already indicated that, more often than not, there are several possible resolutions for resolving a particular inconsistency. In the case of the dangling feature reference inconsistency, we suggested two possible solutions, but other solutions exist. In our approach, selecting one of these resolutions is up to the user. As such, our rules' conditions can contain expressions that in fact prompt the user for input. Depending on the answer, typically a "yes" or a "no" or some other user input, inferring that expression succeeds or fails. In our example rule, the expression `(user-option-addOp)` is added to the condition.

After considering all these issues, we get the rule below, which represents our first inconsistency resolution for resolving the dangling feature reference inconsistency. The entire `and` expression is the rule's condition, whereas the last expression is an action of the rule's conclusion. Remark that we use in the rules the concrete syntax of *nRQL* and RACER assertions.

```
(firerule
  (and (check-DFR ?class ?op ?m)
       (?op ?class2 ownedoperation)
       (user-option-addOp)
  )
 ( (related ?op ?class ownedoperation) )
)
```

Rule conclusions can become quite large when different atomic resolution actions are combined. The following two rules represent the second resolution for the dangling feature

reference inconsistency. In particular, the first rule uses an existing lifeline to which the message can be sent, whereas the second rule expresses the creation of a new lifeline which has as base class the class owning the operation. Both rules prompt the user for input in their condition.

```
(firerule
  (and (check-DFR ?class ?op ?m)
           (?m ?mend receiveEvent)
           (user-option-addToLifeline ?lifeline)
          (?mend ?lifelinec coveredsub)
  )
 ( (related ?mend ?lifeline coveredsub)
  (forget-role-assertion ?mend ?lifelinec coveredsub))
)

(firerule
  (and (check-DFR ?class ?op ?m)
           (?class2 ?op ?ownedOperation)
           (?m ?mend receiveEvent)
           (?mend ?lifelinec coveredsub)
          (user-option-addConn ?nameEl)
  )
  ((related ?class2 (new-ind connel ?nameConn) base)
  (related  (new-ind lifeline ?nameConn) (new-ind connel ?nameEl) represents)
  (related ?mend (new-ind lifeline ?nameConn) coveredsub)
  (forget-role-assertion ?mend ?lifelinec represents))
)
```

The last rule contains the operator `new-ind` which creates a new *Abox* individual. After having resolved the dangling feature inconsistency in one of the two latter ways, a dangling association reference inconsistency can occur in case the association over which the message is sent is not known to the new or reused lifeline.

Again different solutions are possible. An existing association can be used or a new one can be created. The first rule below expresses the creation of a new association from the target class to the source class, whereas the second rule uses the existing association if this exists.

```
(firerule
  (and (check-DAR ?assoc ?m)
      (?m ?con connectorr)
      (?m ?mendsend sendEvent)
      (?mendsend ?lifelinesend coveredsub)
      (?lifelinesend ?connectableelsend represents)
      (?connectableelsend ?classsend base)
      (?m ?mendreceive receiveEvent)
      (?mendreceive ?lifelinereceive coveredsub)
      (?lifelinereceive ?connectableelreceive represents)
      (?connectableelreceive ?classreceive base)
      (user-option-addAssoc ?assocname))
(  (related (new-ind assoc ?assocname) ?assocname name)
  (related (new-ind assoc ?assocname)
           (new-ind end ?classsend) memberend)
  (related (new-ind assoc ?assocname)
           (new-ind end ?classreceive) memberend)
  (related ?class (new-ind end ?classsend) ownedattribute)
  (related ?class2 (new-ind end ?classreceive)
                 ownedattribute)
  (related ?con (new-ind assoc ?assocname)
                 associationtype)
  (forget-role-assertion ?con ?assoc associationtype))
 )
```

```
(firerule
  (and (check-DAR ?m ?assoc)
       (?m ?con connectorr)
       (user-option-useAssoc ?assocuser)
  )
  (
  (related ?m (new-ind connector ?assocuser) connectorr)
  (related (new-ind connector ?assocuser) ?assocuser associationtype)
  (forget-role-assertion ?m ?con connectorr)
  )
)
```

Tbox rules

In case inconsistencies are detected by using DLs as a semantic domain, detection is done using the default DL reasoning tasks executed on certain *Tboxes* or *Aboxes*.

Recall the behaviour incompatibility between a sequence diagram and a state machine diagram. This inconsistency occurs when a certain sequence is defined in a sequence diagram but not in the corresponding PSM diagram. To check for incompatible behaviour, coherence of the *Tbox* containing the encoded SD traces and PSM call sequences, is checked. A rule's condition for resolving this inconsistency typically consist of a function `(check-IBehaviour ?trace)`. This function evaluates the *Tbox* for consistency and we assume that it returns the set of SD traces not belonging to the PSM. Other functions can be used to check other inconsistencies based on the *Abox* consistency or *Tbox* coherence reasoning tasks.

Also in this case, the rule's condition can be used for retrieving information from the situation. If an *Abox* is evaluated, queries on this *Abox* can be included in the rules conditions as explained in the previous section. The conclusions of such rules consist of actions to be undertaken on the *Abox* assertions. Actions on *Abox* assertions in the conclusion of a rule are also explained in the previous section.

Resolving inconsistencies detected by standard DL *Tbox* reasoning tasks is far more complicated. Suppose in case of an incompatible behaviour that with respect to a PSM, two message calls are inverted on a sequence in a sequence diagram causing this inconsistency. A possible resolution is to invert the message calls on either the PSM or the sequence diagram. User interaction is crucial in this case to determine where and how to resolve this inconsistency. However, to be able to analyse the cause of this inconsistency, it should be possible – exactly as in case an *Abox* is used – to retrieve extra information from the *Tbox*. This can be achieved by *Tbox* queries such as `atomic-concept-descendants` asking which concept names are descendants of a certain concept or by the so-called *matching* reasoning task [BKBM99]. A relevant matching pattern in this context can be $\exists r.\exists r.X$, where X is a variable. This pattern matching problem will allow us to retrieve the messages called after some other messages. Matching problems in DLs are theoretically well understood. Nevertheless, still no implementation of a general matching algorithm exists.

A rule's conclusion typically consists of actions adding or retracting *Tbox* assertions. A possible rule for the resolution of an incompatible behaviour is stated below. It is assumed that $?trace$ is a set of *Tbox* assertions. The last expression adds this set of assertions to the *Tbox*.

```
(firerule
  (and (check-IBehaviour ?trace)
           (user-option-addTrace)
  )
```

```
 ( (?trace) )
)
```

8.6.2 Rule Engine

The different rules for the resolution of inconsistencies are defined in so-called *rule sets*. For each inconsistency a rule set is defined. This gives the inconsistency detection and resolution environment and the user of this environment the flexibility to select the inconsistencies to be checked. Depending on the application domain, the applicable rules can be fine-tuned by using only subsets of the given rule sets. An example of fine-tuning the applicable rules is shown in the next chapter.

A rule engine typically goes through the *recognise-act* cycle, which consists of the following steps [Jac86]:

1. matching: match conditions of rules against data
2. conflict resolution: if there is more than one rule that could fire, collect them in a *conflict set* and decide which one to execute
3. execution: execute the rule, and go to step 1.

The cycle stops if no more rules become applicable, more specifically if the conflict set is empty. The conflict set consists of *instantiations*, which are pairs of rules and variable bindings derived from pattern matching.

In our approach we employ a forward-chaining rule engine since the engine has to be activated when model elements change. As a result of firing a rule that resolves an inconsistency, the model is changed anew, which again ensures that the rule engine looks for rules that are applicable in the new situation. This cycle continues until there are no applicable rules.

One of the criteria we established earlier in this section is that selecting one of the applicable inconsistency resolutions is up to the developer. Therefore, after the rule engine has determined the applicable rules, the developer is required to select the preferred resolution of an inconsistency. The corresponding rule is subsequently actually fired.

Note that resolving an inconsistency might have as a side effect that another previously detected inconsistency is resolved. After each inconsistency resolution, the applicable inconsistency resolutions are updated by the rule engine.

8.6.3 Requirements for Rule-Based DL System

From the different issues addressed in Section 8.6.1 and Section 8.6.2, the following set of requirements for a rule-based DL system can be distilled.

A rule-based system supporting *Abox* rules must meet the following requirements:

- The *condition of an* Abox *rule* is a conjunction of *Abox* queries and user-input prompts. The *Abox* queries are used to detect the inconsistency and to retrieve extra information for its resolution. The user-input prompts serve two purposes. They let the user decide which resolution to be chosen if several rules fire for the same bindings and they prompt the user for additional input, if necessary.

- The *conclusion of an* Abox *rule* is a sequence of *Abox* assertions. These assertions resolve the detected inconsistency and can only use variables to be bound in the condition of the rule or constants.
- The rules work on explicitly stated knowledge in an *Abox*.

A rule-based system supporting *Tbox* rules must meet the following requirements:

- The *condition of a* Tbox *rule* is a conjunction of *Tbox* evaluation functions, *Tbox* queries, user-input prompts and DL matching problems. The *Tbox* evaluation functions detect a certain inconsistency, *Tbox* queries and DL matching problems are used to retrieve additional information. The *Tbox* queries and DL matching problems only use variables to be bound in the condition of the rules or concepts and roles defined in the *Tbox*. In a *Tbox* rule, the user-input prompts also serve two purposes. These prompts let the user decide which resolution to be chosen and they prompt for additional user input.
- The *conclusion of a* Tbox *rule* is a sequence of *Tbox* assertions. These *Tbox* assertions resolve the detected inconsistency.
- *Tbox* rules work on complete knowledge present in a *Tbox*.

8.6.4 *nRQL* Rules and Rule Engine

nRQL is not only a query language and environment but also provides rules as a simple *Abox* augmentation mechanism.

nRQL rules have an *antecedent* (condition) and a *consequent* (conclusion). The antecedent is a *nRQL* query body. *nRQL* query bodies correspond to *Abox* query bodies as introduced in the previous chapter. It is not possible to include expressions prompting the user for input in the condition of a *nRQL* rule. One possible practical solution is to replace this expression by a query checking whether a certain special individual is present in the *Abox* representing the choice of the user. The first rule introduced in Section 8.6.1 can be rephrased in *nRQL*, using *nRQL*'s concrete syntax, as follows:

```
(firerule
  (and (check-DFR ?class ?op ?m)
       (?op ?class2 ownedoperation)
       (?class2 class)
       (?u user-option-addOp))
 ( (related ?op ?class ownedoperation))
)
```

This rule is very similar to the original one, because we used in the previously defined rules the syntax of *nRQL* as much as possible. The consequent of a *nRQL* rule is a set of RACER *Abox* assertions as *nRQL* rules provide an *Abox* augmentation mechanism. These assertions may reference the variables bound in the antecedent of the rule. These assertions can also use the `new-ind` operator that creates a new *Abox* individual.

It is obvious that *nRQL* rules can be used to express the *Abox* rules (defined in the beginning of this section).

The antecedent of a *nRQL* rule can also contain *Tbox* queries. Evaluation functions or DL matching problems cannot be expressed in the antecedent of a *nRQL* rule. In the expression `(check-IBehaviour ?trace)`, it is assumed that the *Tbox* evaluation function returns the trace not known in the PSM. However, as already observed as a disadvantage of the evaluation functions implementing the standard DL reasoning tasks, no proper feedback is returned by these functions. Currently the feedback given by the DL reasoners is far from sufficient. Because the consequent of a *nRQL* rule is a set of *Abox* assertions, it is not possible to state the addition or retraction of *Tbox* assertions in the consequent. At this moment, it is not possible to express our above defined *Tbox* rules using the *nRQL* mechanism.

Does this mean that we only can support the resolution of inconsistencies that can be detected using *nRQL* queries? In Chapter 10 we will only support the resolution of inconsistencies that can be resolved using *Abox* rules in a semi-automatic way. The inconsistencies that can be resolved by using *Tbox* rules are only detected and must be manually resolved. However, several work-arounds can be used to support *Tbox* rules with the available *nRQL* support. These are presented in Chapter 11 because further investigation is necessary to conclude whether these work-arounds are sufficient for the resolution of the relevant inconsistencies.

The *nRQL* rules are called *simple*, because they must be used in a monotonic way. Applications have to use these rules in a correct way. If a statement is retracted in the *Abox* that is previously used to fire a rule, the statements added by the conclusion of the rule will not be retracted automatically.

When and how to apply a *nRQL* rule to add or remove some new assertions to a factbase is completely under control of the user. Appropriate *nRQL* API functions are provided to support the implementation of user-defined application strategies. In the next chapter, an application strategy is discussed in the context of model refactorings and its concrete implementation is shown in Chapter 10.

8.7 Discussion and Related Work

8.7.1 Related Work

Different techniques have been developed to cope with the problem of resolving inconsistencies in software engineering. Synoptic is a technique developed by Easterbrook [Eas91] in which stakeholders are expected to define and select resolution actions. In [SF97] a *reconciliation method* is developed which uses distance metrics to indicate the type and extent of inconsistencies. Based on these distances, actions are generated and proposed to the users that can partially resolve the different types of model inconsistencies. van Lamsweerde *et al.* [vLLD98] developed the technique called KAOS that uses divergence resolution patterns but only specific kinds of such divergences can be handled. In these approaches, either the stakeholder gets a lot of responsibility by defining the possible resolution actions, or the set of actions is restricted to a specific domain such as, e.g., requirements. These approaches also do not take into account that resolution actions can introduce other inconsistencies.

In [FGH+93], inconsistency handling between *Viewpoints*, locally managed software models, is presented. Viewpoints and inter-Viewpoint rules are all translated to first-order predicate logic, and inconsistencies are identified using the *Closed World Assumption*. A meta-language based on first-order temporal logic, which uses a set of meta-level axioms, is

employed for defining inconsistency resolution rules. The main difference with our approach is that the rules give the user a likely explanation for the occurence of an inconsistency, such as *typographical error* or *conflict between specification*. Kozlenkov *et al.* [KZ04] use abductive reasoning for establishing a user-defined goal consisting of sequences of conditions required for the goal to be achieved. Our approach uses deduction rather than abduction for the detection of inconsistencies. For inconsistency resolution Kozlenkov *et al.* use Prolog rules that are only used as a querying mechanism on assertions as in our approach.

8.7.2 Evaluation of Criterion #3

In Chapter 4 requirements for an inconsistency resolution formalism are defined by criterion #3.

The formalism must allow the definition of resolution actions. Section 8.1 shows that the three categories of resolution actions defined in Chapter 4 can be defined in DLs and DL systems. Due to the usage of rules, the detection part and the different resolution actions of an inconsistency resolution are encapsulated.

The different dependencies between the resolution actions is supported in a natural way in a rule-based approach. A rule engine is responsible for automatically selecting the applicable rules, ordering and chaining them into one flow. An implementation of different inconsistency resolutions can be found in Chapter 10.

Interactivity is also supported by constructs occurring in the rules conditions enabling the selection by the developer of the preferred inconsistency resolution.

Due to the existence of the *nRQL* rule mechanism these requirements, except for the interactivity, are satisfied for inconsistencies that can be defined on *Abox* assertions. In Chapter 10 we show how we deal with this interactivity in our tool support.

For inconsistencies defined on *Tboxes*, it is not always possible to retrieve additional information concerning the inconsistency. At this moment there is no *nRQL* or RACER support for this kind of rules.

8.8 Conclusion

In this chapter, we introduced a rule-based approach to resolve our classified inconsistencies. First, we showed how addition and retraction of *Abox* and *Tbox* axioms can be used for the definition of different resolution actions. We briefly repeated the different challenges of inconsistency resolution. We argued that manually determining inconsistency resolution scenarios that correspond to all possible situations is a daunting and unmanageable task. The identified problems are exactly those that are addressed by rule-based systems. Hence we presented inconsistency resolution rules. Two types of rules are recognised, Abox *rules* and Tbox *rules*. A set of requirements for a DL rule-based system are distilled and compared to the existing *nRQL* rule-based system.

In the next chapter we show how our inconsistency management approach can be used in the domain of model refactorings. Behavioural consistencies, defined in our classification, correspond to behaviour preservation properties in a model refactoring context and the approach presented in this chapter can be used to support the execution of model refactorings.

Chapter 9

Model Refactorings

In this chapter, we show how our ideas about inconsistency detection and resolution can be applied to the domain of model refactorings. Conceptually, the chapter can be subdivided into two parts.

In a first part, we show how some consistencies, defined earlier in a generalisation context, express behaviour preserving properties when transposed to a model refactoring context. First, a motivating example is presented (Section 9.1). Next, behaviour preservation is discussed and behaviour preservation properties are defined (Section 9.2). We show that the notions of consistency and preservation are closely related. Additionaly, we prove some additional properties about the behaviour of classes that already obey certain consistencies and preservation properties (Section 9.3).

In a second part of this chapter, we explore the idea of using the rule-based inconsistency resolution approach, presented in the previous chapter, for supporting the execution of model refactorings. The relationship between inconsistency resolution and refactorings is introduced by a discussion on source code refactoring versus model refactoring and by the in-depth description of one refactoring (Section 9.4). Finally, we present an analysis of a set of model refactorings and open a discussion on different relevant issues in the context of model refactorings through inconsistency resolution (Section 9.5). An implementation of this approach is shown in the next chapter.

To conclude, related work concerning behaviour preservation issues in model refactoring and model refactorings in general is discussed (Section 9.6).

9.1 Motivating Example

As a motivating example, consider the situation depicted in Figure 9.1.

9.1.1 Model Refinement

The class ATM (version 1.0) is *refined* into a subclass $CardChargingATM$ (version 1.1) in a behaviourally consistent way. This means that the behaviour of the $CardChargingATM$ class (expressed by means of a state machine or sequence diagram, for example) should specialise the behaviour of the ATM class under certain conditions as defined by our *behaviour* (see Section 3.8.1), *inheritance* (see Section 3.9.1 and Section 3.9.2) and *interaction* (see Section 3.7.1) consistencies.

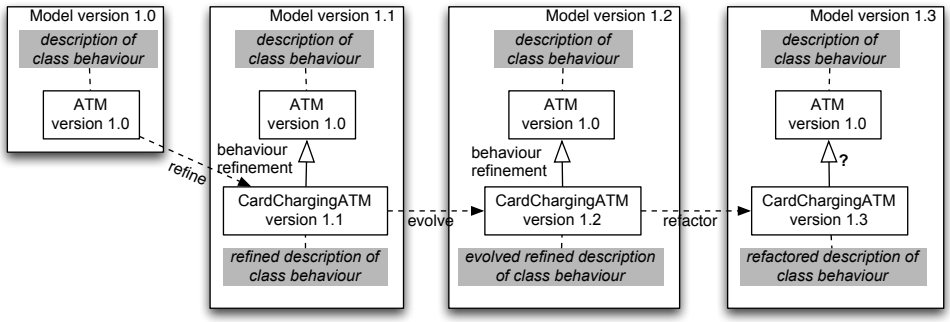

Figure 9.1: Scenario of evolution of our motivating example.

This $CardChargingATM$ denotes a special kind of ATM that allows the customer to withdraw cash money, but also to charge "virtual" money to his/her bank card. The behaviour of the $CardChargingATM$ class, which is a subclass of ATM, is represented by the state machine in Figure 9.2. An orthogonal composite state $VerifyingTransaction$ is added to the existing composite state $GettingCustomerSpecifics$ (cf. Figure 2.14). Its substate $VerifyATMBalance$ is moved one level deeper into the new composite state $VerifyingTransaction$. The customer still has to specify the account number and the amount of cash for withdrawal. The same account number and amount will be used to charge the customer's bank card. Verifying if the customer's account has sufficient funds and if the transactions are allowed by the bank is now done in parallel. Once these checks have been passed, the ATM dispenses the money and at the same time, the $CardChargingATM$ class, unlike its parent, the ATM class, charges the card.

Consider the invocation inheritance consistency relationship between the sequence diagram in Figure 2.12 and the PSM of $CardChargingATM$ as described in Figure 9.2. Remember that this relationship expresses that an instance of $CardChargingATM$ must be usable in each situation where an instance of ATM is required. To guarantee this consistency relationship, *each sequence of the ATM sequence diagram of Figure 2.12 should be contained in the set of sequences of the CardChargingATM state machine diagram of Figure 9.2.* In our case, ATM and $CardChargingATM$ do not obey this consistency, because an instance of $CardChargingATM$ will withdraw money and it will *always* charge a card. It is not possible to skip the charging of the card and immediately choose a new transaction, which is the original behaviour of the ATM class.

9.1.2 Model Evolution

Starting from version 1.1, one can make further changes to the design model. For example, we could continue to *evolve* the behaviour of the $CardChargingATM$ class by introducing a new state ($ChargingAmountEntry$) and transitions in the composite state $GettingCustomersSpecifics$ as shown in the state machine diagram in Figure 9.3. This is an example of an evolution step where we have added new functionality to the $CardChargingATM$ class. The amount to be withdrawn and to be charged on the card can now be different. This is not the case for the previous version of the $CardChargingATM$

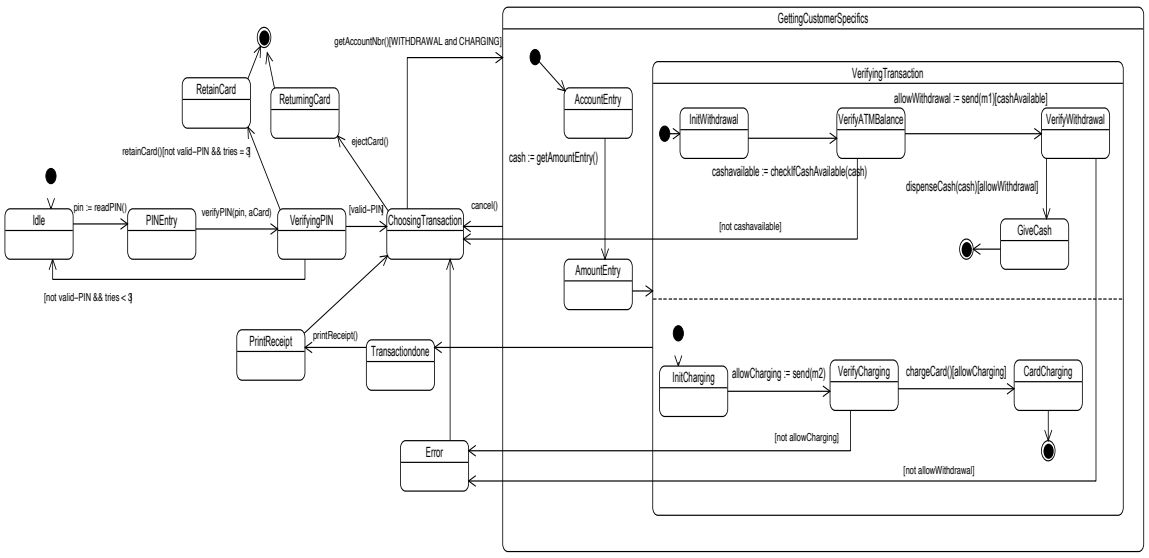

Figure 9.2: UML protocol state machine for *CardChargingATM* class (version 1.1).

(version 1.1 of the design model).

9.1.3 Model Refactoring

We can also consider more restrictive model evolutions that have the purpose of improving some design quality attributes without changing the specification of behaviour. Such model evolutions are called *model refactorings*.

Refactorings are an essential tool for handling software evolution. Model refactorings restructure a model while preserving behavioural properties of this model. Some model refactorings are inspired by source code refactorings [Fow99], others are specifically defined from the model point of view, e.g., state machine refactorings [BSF02].

For model refactoring, the idea of behaviour consistency is very important. If we know that a given design model is behaviour consistent, and we perform a model refactoring, then we expect the evolved design model to be behaviour consistent too. In other words, the refactoring is assumed to preserve certain *behaviour properties*. Being able to verify or guarantee that certain behavioural properties are preserved becomes crucial in this situation.

An example of the result of a complex model refactoring is shown in Figure 3.23 (version 1.3). This protocol state machine represents a refactored version of the one shown in Figure 9.3 (version 1.2). To obtain the new state machine from the original one, a sequence of two refactorings has been applied, *Move states into orthogonal composite state* and *Flatten states*.

The first refactoring, *Move states into orthogonal composite state*, can be seen as the inverse of the *Sequentialize concurrent composite state* refactoring defined in [BSF02]. States are moved into different regions of an orthogonal composite state. In our example the simple state $AmountEntry$ is moved into one region of the $VerifyingTransaction$ state and the state $ChargingAmountEntry$ in the other region. The original state $AccountEntry$

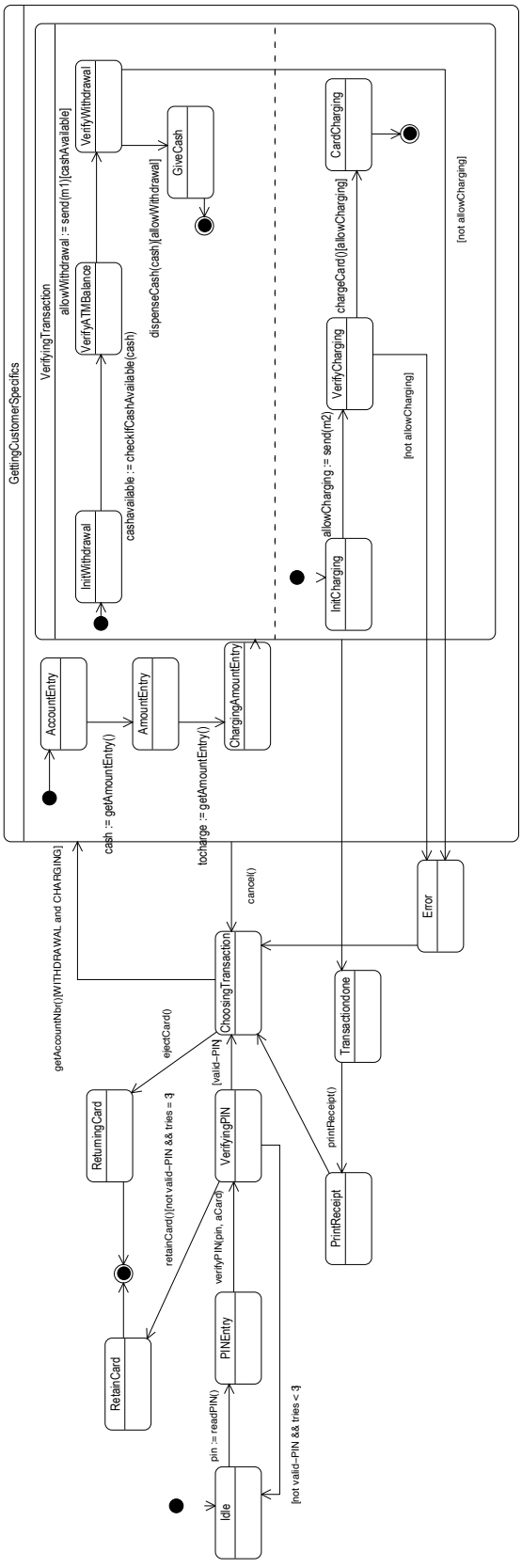

Figure 9.3: State machine for evolved *CardChargingATM* class (version 1.2).

is split into two states ($WithdrawalAccountEntry$ and $ChargingAccountEntry$). These states are the "initial states" of the two different regions. As a result the previous initial states, $InitWithdrawal$ and $InitCharging$ become superfluous and are deleted. Moving a state into a certain region of an orthogonal composite state has some consequences. High-level transitions (i.e., transitions originating from a composite state) originating on the orthogonal composite state, are inherited by the moved states. By moving a state, if the moved state becomes active, other states will be active too, one in each remaining regions. As a result, if the $AmountEntry$ state is active, the $ChargingAmountEntry$ state can be active too.

The second refactoring, *Flatten states*, flattens the states $GettingCustomerSpecifics$ and $VerifyingTransaction$ into a new state also named $GettingCustomerSpecifics$ that is an orthogonal composite state.

After applying both refactorings, it is important to know that original behaviour has been preserved. For example, in this case, we can formally prove that the sequences of operations that can be invoked on the original class $CardChargingATM$ (version 1.2) can also be invoked on the refactored class $CardChargingATM$ (version 1.3).

9.2 Behaviour Preservation

On the source code level, refactorings of an object-oriented program are restructurings that preserve program behaviour. Despite the available tool support for source code refactorings and also model refactorings, it is still an open research question how to define and check behaviour preserving properties for (model) refactorings.

In order to determine whether a given model refactoring preserves behaviour, we need to define precisely what this means. To achieve this, we will take an approach that is very similar to the one taken in Chapter 3: just like the behaviour of a subclass can be consistent with the behaviour of its superclass, the behaviour of a new version of a class can preserve the behaviour of the original version. Even more, the different flavours of observation and invocation consistency that were explored in Chapter 3 also make sense in an evolution context. This will be formalised below.

Before doing this, however, we need to be clear about what it means to be "a new version of a class". We will adopt a very broad view here. It includes changes to the class itself (renaming, adding, removing or modifying operations or attributes), or to its associated behaviour (renaming, adding, removing or modifying PSMs or sequence diagrams). But even more sophisticated changes can be envisioned, such as splitting a class into two or more classes (each of these new classes is then considered to be a new version of the original one), or combining two or more classes into a single merged version. Splitting or merging sequence diagrams or PSMs can also be accommodated in this way.

We will now explain the correspondence between behaviour preservation and observation/invocation inheritance consistencies. Assume that we have a model M_1 and a refactored version M_2 of this model. *Observation call preservation* defines that every call sequence observable with respect to a class in M_2 must result in an observable call sequence of its corresponding class in M_1.

Definition 71 Observation call preservation. *Given $c \in \mathcal{C}_{\mathcal{M}_1}$, $c' \in \mathcal{C}_{\mathcal{M}_2}$, and c' is a refactored version of c:*

A PSM $\pi_{c'} = (S', T', L', \rho', \Lambda')$ *is* **observation call preserving** *with a PSM* $\pi_c = (S, T, L, \rho, \Lambda)$ *if and only if,*
$\forall\ \mu' : (\mathbf{valid}(\mu', \{\rho'\}, \pi_{c'}) \Rightarrow \mathbf{valid}(\mu'_L, \{\rho\}, \pi_c))$.

A SD δ' *is* **observation call preserving** *with a SD* δ *if and only if,*
$\forall O' \in \mathbf{contained}(\delta', \{c'\})\ \forall\ \upsilon' = \upsilon_{O'} \in \delta'\ (\exists O \in \mathbf{contained}(\delta, \{c\}) \Rightarrow \exists \upsilon''_O \in \delta : =_\upsilon (\upsilon''_O{}^{/rec}, \upsilon'_{E_{\delta,\{c\}}}{}^{/rec}))$.

A SD δ' *is* **observation call preserving** *with a PSM* $\pi_c = (S, T, L, \rho, \Lambda)$ *if and only if,*
$\forall\ O' \in \mathbf{contained}(\delta', \{c'\})\ \forall\ \upsilon_{O'}{}^{/rec} =< e_1, \ldots, e_n >$ *(with* $\upsilon_{O'} \in \delta') \exists\ \mu_c =< \tau_1, \ldots, \tau_m >$: $(\forall\ \tau_i \in \mu_c : \tau_i \in L \wedge m \geq n\ \exists\ \sigma : \mathbf{valid}(\mu_c, \sigma, \pi_c) \wedge \forall i \in \{1, \ldots, n\}\ \exists\ j \in \{i, \ldots, m\} : \tau_j = (op, g, h) = label(e_i) \wedge (\exists \tau_k = label(e_{i+1}) \in \mu_c \Rightarrow k > j) \wedge \upsilon_{O'}{}^{/rec}$, μ_c **are in strict sequence**)).

Remark that Definition 71 is almost identical to Definition 57. The main difference is that the words *observation consistent* are replaced by *observation call preserving.* Also, c' does not represent a subclass of c anymore, but a new version of c in the refactored model.

Example 45 *The behaviour of the class CardChargingATM specified by the refactored PSM* $\pi_{1.3}$ *of Figure 3.23 is not observation call preserving with respect to the PSM* $\pi_{1.2}$ *of Figure 9.3. A first model refactoring presented here, moves some simple states into a composite orthogonal state and a second refactoring flattens two states. The amount to be charged on a card must not necessarily be entered after the amount to be dispensed (cf.* $\pi_{1.3}$*). However, this precedence constraint is required by* $\pi_{1.2}$*.*

Invocation call preservation guarantees that each call sequence invocable on the original version of a class, must also be invocable on the corresponding class in the refactored model. The definition of invocation call preserving is almost identical to Definition 56. Also in this case, the main difference is that the words *invocation consistent* are substituted by *invocation call preserving* and the class c' represents a new version of the class c in the refactored model.

Definition 72 Invocation call preservation. *Given* $c \in \mathcal{C}_{\mathcal{M}_1}$, $c' \in \mathcal{C}_{\mathcal{M}_2}$, *and* c' *is a refactored version of* c*:*

A PSM $\pi_{c'} = (S', T', L', \rho', \Lambda')$ *is* **invocation call preserving** *with a PSM* $\pi_c = (S, T, L, \rho, \Lambda)$ *if and only if,*
$\forall\ \mu : (\mathbf{valid}(\mu, \{\rho\}, \pi_c) \Rightarrow \mathbf{valid}(\mu, \{\rho'\}, \pi_{c'}))$ *and for the PSM traces* γ *corresponding to* μ *in* π_c *and* γ' *corresponding to* μ *in* π'_c*, it holds that* $\gamma = \gamma'_S$.

A SD δ' *is* **invocation call preserving** *with a SD* δ*, if and only if,*
$\forall O \in \mathbf{contained}(\delta, \{c\})\ \forall\ \upsilon_O \in \delta : (\exists O' \in \mathbf{contained}(\delta', \{c'\}) \Rightarrow \exists\ \upsilon'_{O'} \in \delta' : =_\upsilon (\upsilon_O{}^{/rec}, \upsilon'_{O'}{}^{/rec}))$.

A PSM $\pi_{c'} = (S', T', L', \rho', \Lambda')$ *is* **invocation call preserving** *with a SD* δ *if and only if,*
$\forall\ O \in \mathbf{contained}(\delta, \{c\})\ \forall\ \upsilon_O{}^{/rec} =< e_1\ \ldots\ e_n >$ *(with* $\upsilon_O \in \delta) \exists\ \mu_{c'} =< \tau_1 \ldots \tau_m >$: $(\forall\ \tau_i \in \mu_{c'} : \tau_i \in L' \wedge m \geq n \wedge \exists\ \sigma : \mathbf{valid}(\mu_{c'}, \sigma, \pi_{c'}) \wedge \forall\ i \in \{1, \ldots, n\}\ \exists\ j \in \{i, \ldots, m\} : \tau_j = (op, g, h) = label(e_i) \wedge (\exists \tau_k = label(e_{i+1}) \in \mu_{c'} \Rightarrow k > j) \wedge \mu_{c'}$, $\upsilon_O{}^{/rec}$ **are in strict sequence**)).

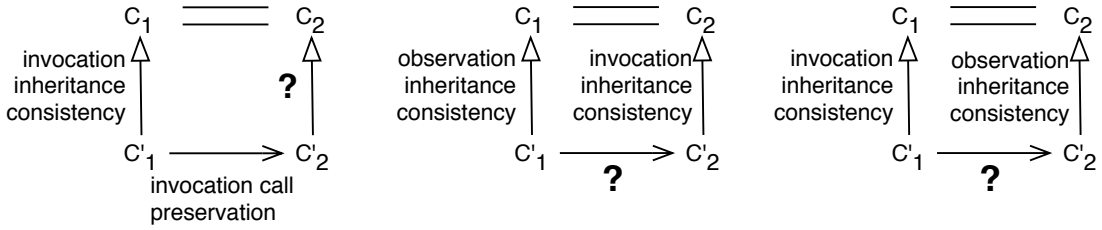

Figure 9.4: Examples illustrating Proposition 1.

Referring to Example 45 above, the behaviour specified by the refactored PSM $\pi_{1.3}$ is invocation call preserving with the PSM $\pi_{1.2}$.

Similar correspondences between observation/invocation behaviour consistencies and behaviour preserving properties can be specified, if behaviour is specified by PSM(s) and sequence diagram(s) at specification level. A correspondence between observation/invocation interaction consistencies and behaviour preserving properties can also be specified. In this case, behaviour is described by sequence diagrams at specification level.

9.3 Behaviour Preservation and Behaviour Inheritance Consistencies

Consider the class ATM version 1.0 and its subclass $CardChargingATM$ version 1.1 in UML model version 1.1 (see Figure 9.1). Consider now the PSM π_1 of the class ATM shown in Figure 2.14. Assume that the PSM $\pi_{1.1}$ is the extension of the PSM π_1 and the PSM shown in Figure 9.2. The PSM $\pi_{1.1}$ expresses the behaviour of the class $CardChargingATM$. The PSMs π_1 and $\pi_{1.1}$ are invocation inheritance consistent. The PSM $\pi_{1.3}$ is observation and invocation call preserving with the PSM $\pi_{1.1}$. The question remains if, based on this information, we can prove that the refactored version of $CardChargingATM$ version 1.3 is still (invocation or observation) consistent with the behaviour of ATM version 1.0.

Until now, we assumed that the PSMs expressed observable call sequences and invocable call sequences, while the sequence diagrams expressed observable traces and invocable traces. In general, however, the set of invocable sequences of operation calls to a class c, denoted by $IS(c)$, is a subset of the set of observable call sequences of operation calls to a class c, denoted by $OS(c)$. This is stated in [EE95]. Abstracting away from SD traces and PSM call sequences, we can define invocation inheritance consistency or invocation call preservation as $IS(c) \subseteq IS(c')$, where c' is subclass of c or where c' is the refactored version of c. Observation inheritance consistency or observation invocation call preservation can be defined by $OS(c' \mid V) \subseteq OS(c)$, where V denotes the set that is the union of the set of labels L_c and the set of event occurrences $\mathbf{E} = \bigcup_{\forall \delta \in \Delta_{\mathcal{M}}} \mathbf{E}_{\delta,\{c\}}$, where c is a superclass of c' or a previous version of c. The **restriction** $U|L$ of a set of sequences U to a set L is defined as $U|L = \{\phi \mid \exists \mu \in U : \phi = \mu_L\}$.

In general, the following properties can be proved:

Proposition 1 *Let c_1 be a class, c_1' a subclass of c_1, c_2 an identical copy of c_1 that is contained in a different model version (e.g., ATM class 1.0 in model version 1.2 and ATM class 1.0 in model version 1.3), and c_2' a subclass of c_2 such that c_2' is a refactored version of c_1'. Let $V_1 = L_{c_1} \cup \mathbf{E} = \bigcup_{\forall \delta \in \Delta_{\mathcal{M}}} \mathbf{E}_{\delta,\{c_1\}}$, and $V_2 = L_{c_2} \cup \mathbf{E} = \bigcup_{\forall \delta \in \Delta_{\mathcal{M}}} \mathbf{E}_{\delta,\{c_2\}}$. (see also Figure 9.4)*

1. *If c_1' and c_1 are* **invocation inheritance consistent** *and c_2' and c_1' are* **invocation call preserving**
   then c_2' and c_2 are **invocation inheritance consistent**.
2. *If c_1' and c_1 are* **observation inheritance consistent** *and c_2' and c_2 are* **invocation inheritance consistent** *and $IS(c_1) = OS(c_1)$*
   then $OS(c_1' \mid V_1) \subseteq OS(c_2')$.
3. *If c_1' and c_1 are* **invocation inheritance consistent** *and c_2' and c_2 are* **observation inheritance consistent** *and $IS(c_1) = OS(c_1)$*
   then $OS(c_2' \mid V_2) \subseteq OS(c_1')$.

Proof 1

1. *c_1' and c_1 are invocation inheritance consistent, hence $IS(c_1) \subseteq IS(c_1')$. c_2' and c_1' are invocation call preserving, hence $IS(c_1') \subseteq IS(c_2')$. Because c_1 and c_2 are identical, we conclude that*
   $IS(c_2) = IS(c_1) \subseteq IS(c_1') \subseteq IS(c_2')$
   This implies that c_2 is invocation inheritance consistent with c_2'.
2. *c_1' and c_1 are observation inheritance consistent, hence $OS(c_1' \mid V_1) \subseteq OS(c_1)$. c_2' and c_2 are invocation inheritance consistent, hence $IS(c_2) \subseteq IS(c_2')$. Because c_1 and c_2 are identical, and given that $IS(c_1) = OS(c_1)$, and, in general, $IS(c_2') \subseteq OS(c_2')$ we can conclude that*
   $OS(c_1' \mid V_1) \subseteq OS(c_1) = IS(c_1) = IS(c_2) \subseteq IS(c_2') \subseteq OS(c_2')$
   This results in: $OS(c_1' \mid V_1) \subseteq OS(c_2')$.
3. *c_1' and c_1 are invocation inheritance consistent, hence $IS(c_1) \subseteq IS(c_1')$. c_2' and c_2 are observation inheritance consistent, hence $OS(c_2' \mid V_2) \subseteq OS(c_2)$. Because c_1 and c_2 are identical, and given that $IS(c_1) = OS(c_1)$, and, in general, $IS(c_1') \subseteq OS(c_1')$, we can conclude that*
   $OS(c_2' \mid V_2) \subseteq OS(c_2) = OS(c_1) = IS(c_1) \subseteq IS(c_1') \subseteq OS(c_1')$
   This results in: $OS(c_2' \mid V_2) \subseteq OS(c_1')$.

From the first item of the proposition, we can conclude that the behaviour of the refactored version 1.3 of class *CardChargingATM* is invocation inheritance consistent with the behaviour of the *ATM* class as specified by the PSM π_1.

The second item in the proposition means that the set of valid call sequences or traces of the class c_1' under the projection of the methods known by c_1, must be included in the set of the call sequences or traces of the class c_2', which is the refactored or evolved version of the class c_1'. Depending on how the behaviour of the different classes is specified, possible conclusions are:

1. Every valid call sequence of the PSM π' of c_1', restricted to the operations known by c_1, is also a valid call sequence of the PSM of c_2';

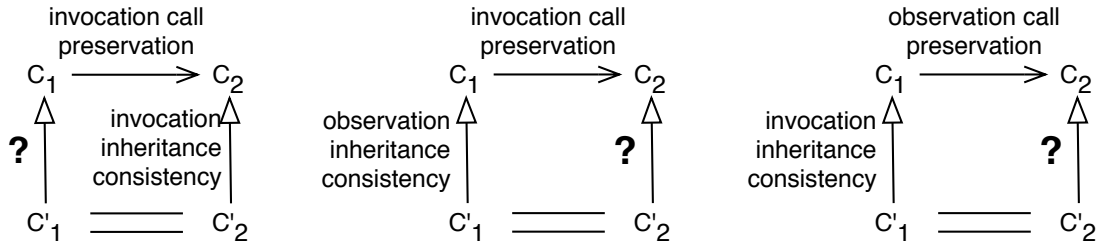

Figure 9.5: Examples illustrating Proposition 2.

2. Assume the existence of sequence diagrams δ, containing instances of c_1 and δ', containing instances of c_1' and δ", containing instances of c_2'. Then for each trace υ' in δ', $\upsilon'_{E_{\delta,\{c_1\}}}$ is an SD trace in δ";

3. Assume the existence of a sequence diagram δ containing instances of c_1' and a PSM $\pi'_{c_2'}$. Then each call sequence $\mu =< \tau_1 \ldots \tau_n >$ such that, for each $i \in \{1, \ldots, n\}$, $\tau_i = label(e_i)$ for receiving traces $< e_1, \ldots, c_n >$ in *delta* on instances of c_1', is also a valid call sequence in $\pi'_{c_2'}$.

The third item in the proposition means that the set of valid call sequences or traces of the class c_2' under the projection of the methods known by c_2, must be included in the set of the call sequences or traces of the class c_1'. As a consequence, the behaviour of the refactored class c_2' is smaller than the behaviour of the original class. Depending on how the behaviour of the different classes is specified, possible conclusions are:

1. Every valid call sequence of a PSM π' of c_2', restricted to the operations known by c_2, is also a valid call sequence of a PSM of c_1';

2. Assume the existence of sequence diagrams δ, containing instances of c_2 and δ', containing instances of c_2'. Then for each trace υ' in δ', $\nu'_{E_{\delta,\{c_2\}}}$ is an SD trace in a sequence diagram δ", containing instances of c_1';

3. Assume the existence of a sequence diagram δ for c_2'. Then each call sequence $\mu =< \tau_1 \ldots \tau_n >$ such that, for each $i \in \{1, \ldots, n\}$, $\tau_i = label(e_i)$ for receiving traces $< e_1, \ldots, e_n >$ in δ on instances of c_2', is also a call sequence in a PSM of c_1'.

In the scenario described in Proposition 1 a certain subclass in a hierarchy is refactored. We can prove similar properties when a superclass in a hierarchy is refactored.

Proposition 2 *Let c_1 be a class, c_1' a subclass of c_1, c_2 a refactored version of c_1, and c_2' an identical copy of c_1' that is contained in a different model version. Let $V_1 = L_{c_1} \cup \mathbf{E} = \bigcup_{\forall \delta \in \Delta_{\mathcal{M}}} \mathbf{E}_{\delta,\{c_1\}}$. (see also Figure 9.5)*

1. *If c_2' and c_2 are* **invocation inheritance consistent** *and c_2 and c_1 are* **invocation call preserving**
   then c_1' and c_1 are **invocation inheritance consistent**.

2. *If* c_1' *and* c_1 *are* **observation inheritance consistent** *and* c_2 *and* c_1 *are* **invocation call preserving** *and* $IS(c_1) = OS(c_1)$
then $OS(c_2' \mid V_1) \subseteq OS(c_2)$.

3. *If* c_1' *and* c_1 *are* **invocation inheritance consistent** *and* c_2 *and* c_1 *are* **observation call preserving** *and* $IS(c_1) = OS(c_1)$
then $OS(c_2 \mid V_1) \subseteq OS(c_2')$.

Proof 2 1. c_2' *and* c_2 *are invocation inheritance consistent, hence* $IS(c_2) \subseteq IS(c_2')$. c_2 *and* c_1 *are invocation call preserving, hence* $IS(c_1) \subseteq IS(c_2)$. *Because* c_1' *and* c_2' *are identical, we conclude that*
$IS(c_1) \subseteq IS(c_2) \subseteq IS(c_2') = IS(c_1')$
This implies that c_1 *is invocation inheritance consistent with* c_2.

2. c_1' *and* c_1 *are observation inheritance consistent, hence* $OS(c_1' \mid V_1) \subseteq OS(c_1)$. c_2 *and* c_1 *are invocation call preserving, hence* $IS(c_1) \subseteq IS(c_2)$. *Because* c_1' *and* c_2' *are identical, and given that* $IS(c_1) = OS(c_1)$, *and, in general,* $IS(c_2) \subseteq OS(c_2)$, *we can conclude that*
$OS(c_2' \mid V_1) = OS(c_1' \mid V_1) \subseteq OS(c_1) = IS(c_1) \subseteq IS(c_2) \subseteq OS(c_2)$
This results in: $OS(c_2' \mid V_1) \subseteq OS(c_2)$.

3. c_1' *and* c_1 *are invocation inheritance consistent, hence* $IS(c_1) \subseteq IS(c_1')$. c_2 *and* c_1 *are observation call preserving, hence* $OS(c_2 \mid V_1) \subseteq OS(c_1)$. *Because* c_1' *and* c_2' *are identical, and given that* $IS(c_1) = OS(c_1)$, *and, in general,* $IS(c_1') \subseteq OS(c_1')$, *we can conclude that*
$OS(c_2 \mid V_1) \subseteq OS(c_1) = IS(c_1) \subseteq IS(c_1') \subseteq OS(c_1') = OS(c_2')$
This results in: $OS(c_2 \mid V_1) \subseteq OS(c_2')$.

The second item in the proposition means that the set of valid observable call sequences or traces of the class c_2' under the projection of the methods known by c_1, must be included in the set of the observable call sequences or traces of the class c_2, which is the refactored or evolved version of the class c_1. As a consequence, the behaviour of the subclass c_2' is smaller than the behaviour of the refactored class viewed from the original class.

The third item in the proposition means that the set of valid observable call sequences or traces of the class c_2 under the projection of the methods known by c_1, must be included in the set of the call sequence or traces of the class c_2'.

9.4 Model Refactoring through Rule-Based Inconsistency Resolution

This section marks the second part of this chapter. In this part, we argue that our rule-based inconsistency approach can be used for the execution of model refactorings.

9.4.1 Source Code Refactoring versus Model Refactoring

At source code level, refactorings are executed to eliminate a bad smell. A bad smell is commonly defined as *(local) structures in the code that suggest the possibility of refactoring*

[MT04]. Source code refactorings are defined on certain source code elements which are chosen by the designer. For example, tool support for source code refactorings in Eclipse [IBM04] lets the user decide on which source code element(s) a certain refactoring must be executed.

Each source code refactoring description in [Fow99] contains a *mechanics* section. That section gives a step-by-step description on how to carry out the refactoring. Each step consists of a certain activity, for example, *copy the body of the method*. In most cases, this step is followed by a description of corrections necessary to make the code compile. These corrections not only affect the newly created elements but also existing elements, and for each application of a refactoring, these corrections can be different. Consequently, although a refactoring is defined on only those elements indicated by the user, it can affect different elements throughout the source code.

The activity of how to carry out a *model refactoring* can also be described step by step. Similar to the execution of source code refactorings, a certain step can affect several elements in the model and introduce *inconsistencies*. The inconsistencies resulting from the execution of a certain refactoring step, need to be resolved and (similar to corrections in source code refactorings) inconsistency resolutions can affect not only the newly created elements but also existing ones. We show that *resolution of inconsistencies can be used for the support of the execution of model refactorings*.

In the remainder of this section, first, we motivate the particular set of refactorings that we (re)designed at model level and next, we will give an in-depth description of the execution of *Move Operation*. We have chosen this particular refactoring because, although it is conceptually a small refactoring, it illustrates very well how inconsistency resolutions can be used for the support of the execution of model refactorings. The description of our remaining redesigned refactorings can be found in Appendix C. An implementation of our inconsistency resolution approach in the context of model refactorings can be found in the next chapter.

9.4.2 Refactorings Considered

Fowler [Fow99] designed a catalogue of source code refactorings. We reconsidered one or two representative refactorings of each category defined by Fowler (see Section 9.5). There are several reasons for considering only refactorings defined in [Fow99]:

- The refactorings defined by Fowler, are well-known and some of these refactorings have already been defined at design level [SPLJ01].
- The corresponding model refactorings are model transformations that transform not only the specification of the static structure as suggested by the UML drawings used in Fowler. These refactorings affect also the specification of the behaviour of the system under study.
- Because the source code refactorings defined in [Fow99] are well-known, it is obvious to consider their corresponding model refactorings in view of exploring the link between these refactorings and (generated) source code in the larger context of MDE.

In the remainder of this section, we show how resolution actions and rules can be used for the support of the execution of *Move Operation*. We will explain the different steps

executing this refactoring on UML models. We will investigate whether inconsistencies can be introduced after each step in the execution and if so, we will focus on which inconsistencies are introduced and how the resolution of these inconsistencies cause a transition to a next step in the execution of the refactoring. Later on we will give an overview of the relation between a larger set of refactorings and inconsistencies.

Figure 9.6 shows a simplified class diagram containing only the information needed in the context of this section, and the impact of the model refactoring on this class diagram.

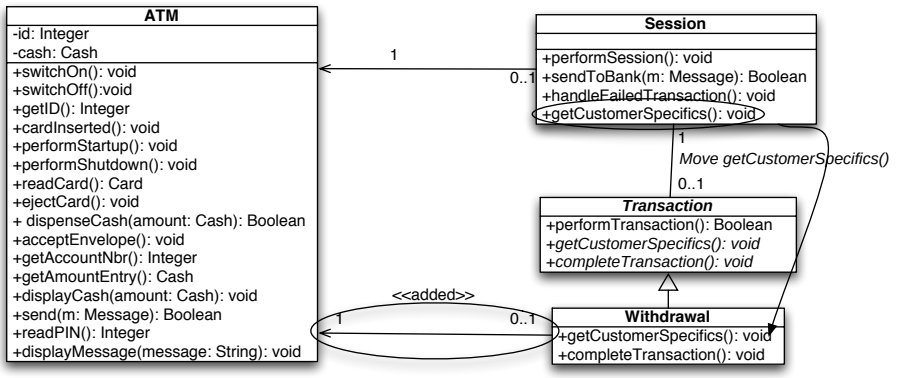

Figure 9.6: Class diagram representing the relevant classes and executed refactoring.

9.4.3 Executing *Move Operation*

Suppose the sequence diagram in Figure 9.7 is a sequence diagram of an early version of the design of our case study. Consider an evolution of this sequence diagram to the sequence diagram shown in Figure 9.8 by moving the operation $getCustomerSpecifics()$ from the source class, i.e., $Session$, to the target class, i.e., $Withdrawal$. This model refactoring is called *Move Operation*. The latter sequence diagram is similar to the previously discussed one, shown in Figure 2.12.

In Figure 9.9, the decision diagram and the different activities of the *Move Operation* refactoring are shown. We use activity diagrams as defined in the UML 2.0 to specify the decision diagram of a model refactoring. The actions representing a refactoring step in the diagram are marked by a *S*. We will now describe step by step the execution of this model refactoring.

Declaring the operation in the target class
In our example, the operation $getCustomerSpecifics()$ is declared in the class $Withdrawal$ (see Figure 9.6). The user has to specify which operation will be declared in which class. No inconsistencies are checked after this step, because the declaration of a new operation in a class does not cause any inconsistencies.

Copy the body of this operation to its new target In this step, answers to the question *how to reference the source object from the target class* and the question *how to reference the referenced objects in the body of the operation from the target class* need to be given. Support for answering these questions can be delivered by an inconsistency

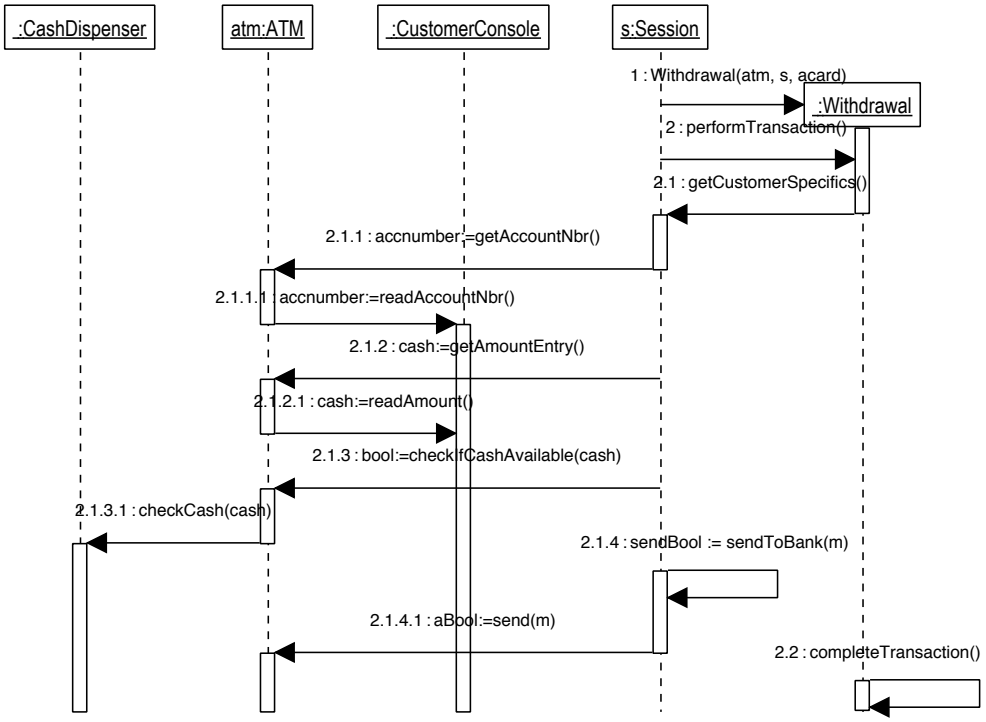

Figure 9.7: Sequence diagram for a withdrawal session scenario on an *ATM*.

resolution approach at model level. If the body of an operation at model level is copied without considering these questions, different inconsistencies can occur.

At UML diagram level, copying the body of an operation implies adding the defined body of the operation to (new) object(s) that are instances of the target class in a (new) sequence diagram. This can be done in interaction with the user. Different possible inconsistencies can occur now, i.e., *instance specification missing*, *incompatible behaviour* and *inheritance inconsistencies*. Two inconsistencies of the category *instance specification missing* are relevant here, *dangling feature reference* and *dangling association reference*. The possible occurrences of these inconsistencies are detailed in the next paragraphs.

A dangling feature reference occurs in this step, when one or more operations of the source class are referenced by the target class. In order to remove this inconsistency, the question *how to reference the source object from the target class* needs to be answered. Possible solutions in the context of this specific model refactoring are: (1) move the operation to the target class as well, in this case the model refactoring *Move Operation* is executed first for the operation in question; (2) create or use a reference, i.e., an association from the target class to the source; (3) pass the source object as a parameter to the operation, in this case the model refactoring *Add Parameter* (similar to the *Add Parameter* source-code refactoring [Fow99]) is executed first. In our example, this inconsistency occurs because, by copying the body of $getCustomerSpecifics$ the operation $sendToBank$ is called on the instance of $Withdrawal$.

The rule representing the first possible solution was explained in the previous chap-

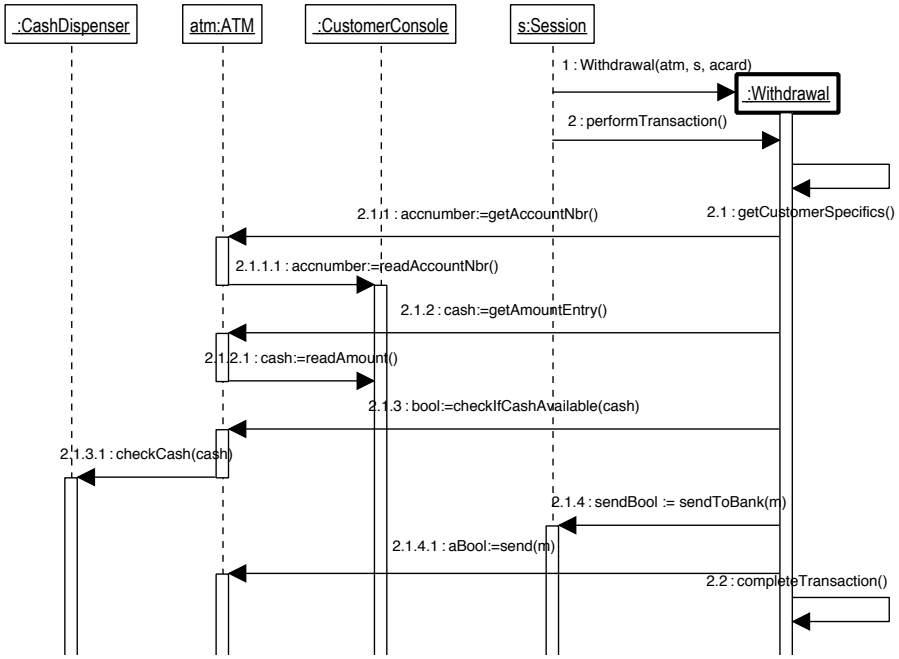

Figure 9.8: Sequence diagram after the execution of *Move Operation*.

ter. The second and third possible solutions correspond to several composed inconsistency resolution rules. The dangling feature reference inconsistency is resolved by changing the message representing the operation call in such a way that it is received by the correct object. The rules representing these resolutions are also discussed in the previous chapter. As remarked in that chapter, these resolutions introduce a dangling association reference inconsistency. In the case of this refactoring this inconsistency can be resolved by creating or using an association from the target class to the source class. Two rules correspond to these two resolution possibilities. These rules are also introduced in the previous chapter.

The third possible solution gives rise to the *Add Parameter* refactoring.

In our example, this inconsistency is resolved by sending the corresponding message to the *Session* object. A dangling association reference is introduced and this inconsistency can be resolved by sending the message over the existing association declared between the *Session* and *Withdrawal* class (see Figure 9.6).

A *dangling association reference* occurs after the second step in this refactoring, when objects are referenced in the body of the operation that are not known to the target class. Again different solutions are possible: (1) an explicit association can be added to reference the class in question; (2) a parameter can be added to the operation, which results in an implicit association; (3) as the objects are known to the source class, operation calls can be sent to the source class, but this involves the addition of different operations in the source class, which are delegating operations.

This inconsistency occurs in our example. At this moment, there is no link between the objects of the *Withdrawal* class and the *ATM* class. In our example, this inconsistency is solved by adding an association between the classes *Withdrawal* and *ATM*.

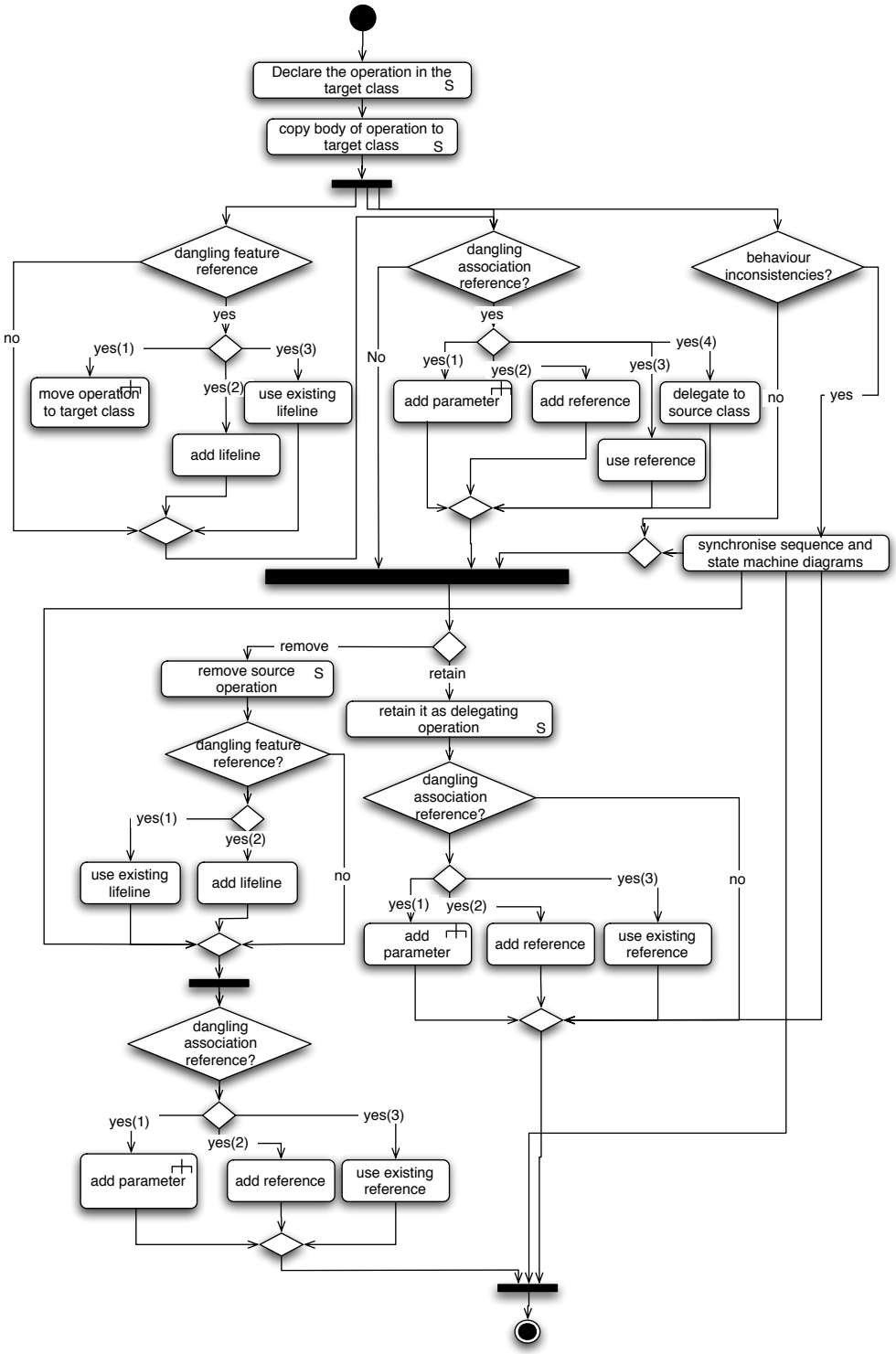

Figure 9.9: Decision activities for *Move Operation*.

The first rule introduced as a possible solution for the dangling feature reference inconsistency in the previous chapter represents the above presented solution (1). In the action of the rule a new association is created and linked to the correct classes. This association is used for typing the connector which links the relevant objects. The rule for the second possible resolution is similar to the above presented rule for resolving a dangling feature reference. The third possible solution is a composition of different solutions. First, the dangling association reference is solved by using the reference between the target object and the source object as the type of the connector and changing the object receiving the message. This is described by the rule below. The execution of this rule will again result in a dangling feature reference. By introducing the operation in the corresponding class, this inconsistency can be solved.

```
(firerule
  (and (check-DAR ?assoc ?m)
      (?m ?con connectorr)
      (?m ?mendsend sendEvent)
      (?m ?mendreceive receiveEvent)
      (?mendreceive ?lifelinereceive coveredsub)
      (?mendsend ?lifelinesend coveredsub)
      (?lifelinesend ?connectableelsend represents)
      (?connectableelsend ?classsend base)
      (?assoc ?end1 ownedAttribute)
      (?end1 ?classsend definedType)
      (?assoc ?end2 ownedAttribute)
      (?end2 ?classsend ownedAttribute)
      (?class ?end2 ownedAttribute)
      (user-option-useLifeline ?lifeline)
      (?lifeline ?connectableel represents)
      (?connectableel ?class  base)
      (user-option-useAssocObj ?assocuser ?lifeline)
      )
       ((related ?m (new-ind connector ?assocuser) connectorr)
        (related ?mendreceive ?lifeline coveredsub)
        (forget-role-assertion ?mendreceive ?lifelinereceive coveredsub)
        (forget-role-assertion ?m ?con connectorr))
)
```

Incompatible behaviour and inheritance inconsistencies occur due to the fact that the sequence diagrams and the state machine diagrams of the source and target class are no longer synchronised. These inconsistencies can be resolved in this step, or after the last step in the execution of this refactoring.

Remove the operation in the source class or retain it as a delegating operation If the operation is *removed from the source class*, a dangling feature reference can occur in the scenarios where the operation is sent to an object of the source class or in the specification of the behaviour of the source class. In this case, the receiver of the operation must be changed to the correct object that is an instance of the target class or such an object must be created. In our example, the $getCustomerSpecifics()$ operation is removed from the *Session* class. As a result, a dangling feature reference occurs. The message can be redirected to the object of type $Withdrawal$ shown in the scenario in Figure 9.7. This resolution can cause a dangling association reference if the target class cannot be reached from the object invoking the operation or if the relationship on which the message is sent, is not known between the objects. This inconsistency can be resolved by creating a reference or using an existing one or adding a parameter to the operation in question.

In case the operation is *turned into a delegating method*, a dangling association reference can occur. Solving this inconsistency boils down to answering the question *how to reference the correct target object from the source object*. Again different resolutions are possible. The correct target object can be referenced through the creation of a new reference, or using an existing reference or adding a parameter to the operation.

If there are still inheritance inconsistencies or incompatible behaviour, these inconsistencies need to be resolved at the end of this step. The resolution of these inconsistencies changes the relevant sequence and state machine diagrams.

9.5 Discussion on a Rule-Based Refactoring Approach

The previous section has illustrated with an example that executing a model refactoring boils down to resolving a number of possibly recurring inconsistencies. The decisions that have to be taken in order to execute the *Move Operation* model refactoring are depicted in the activity diagram in Figure 9.9. However, even for a model refactoring as moderately complex as this one is, this diagram is non-trivial. Note that when implementing the decision process represented in the diagram in an imperative programming language using a traditional conditional statement, this results in a program that is equally non-trivial: similar to the activity diagram, the programmer has to order and nest conditional statements manually. Every possible situation in the model to be refactored leads to a potentially different flow of inconsistency resolutions. Moreover, the same inconsistency resolution can occur multiple times in different combinations with other inconsistency resolutions. Although the previous section only illustrated one model refactoring, we observe that the same inconsistencies occur in multiple model refactorings. These decision diagrams are shown in Appendix C. As such, a first important criterion when resolving inconsistencies as part of model refactorings is *reuse of inconsistency resolutions in and across model refactorings*.

Similar to inconsistency resolution in general, each detected inconsistency can be resolved in several different ways and the selection of a particular resolution for an inconsistency can depend on the particular state of the model. For the most part, however, selecting a resolution is a matter of preference of the person who is executing the model refactoring. Therefore, again similar to inconsistency resolution in general, we require choice points in the flow of inconsistency resolutions where the refactorer is able to communicate his or her preference for a particular resolution. This choice affects the subsequent flow of inconsistency resolutions. As such, a second criterion we aim to address is *support for user-guided selection of inconsistency resolutions*.

9.5.1 Evaluation

It is clear from the previous section that the process of detecting and resolving inconsistencies until all (resulting) inconsistencies are eliminated, is useful not merely in the context of inconsistency management but also in the support for applying refactorings. In this section, we evaluate a particular set of model refactorings and the above introduced criteria.

Table 9.1 summarises which inconsistencies can be detected and resolved to support a certain model refactoring. The rows contain the different refactorings we (re)designed at model level so far. A X in a cell of the table indicates that the resolutions of the corresponding inconsistency can be used to support the execution of the corresponding refactoring.

This table indicates that executing the studied model refactorings consists indeed of resolving inconsistencies and that the same inconsistencies occur in different model refactorings. The decision diagrams of the different model refactorings mentioned in Figure 9.1 can be found in Appendix C.

	inheritance	specification incompatibility	behaviour specification incompatibility	dangling type reference	instance specification missing
Add Parameter	×		×	×	×
Change Bidirectional Association to Unidirectional	×	×	×	×	×
Extract Class	×	×	×		×
Move Attribute	×	×	×		×
Move Operation	×	×	×	×	×
Pull up Operation	×				×
Push down Operation			×		×
Extract Operation	×		×	×	×
Replace Conditional with Polymorphism	×	×	×		×

Table 9.1: Analysis of relation between model refactorings and inconsistencies.

Some refactorings however, can be executed without this process of inconsistency detection and resolution. Consider a refactoring where a class is inserted in a hierarchy. In a first step, the new class is created. This does not introduce any inconsistencies. A second step is to add a generalisation relationship between this new class and the superclass. This also does not introduce any inconsistencies. A third step is to add a generalisation relationship between the subclass and the new class. Finally, the generalisation relationship between the superclass and subclass can be removed without causing any inconsistencies.

The criteria identified in the beginning of this section, are fulfilled by our rule-based inconsistency resolution approach. Rules representing the different inconsistency resolutions are defined only once and the rule engine fires the ones that are appropriate for a certain situation guaranteeing *reuse of inconsistency resolution in and across model refactorings.*

We now show that reuse of inconsistency resolutions such as described above is crucial in the context of model refactorings, and thus motivates a rule-based approach. We present an analysis of the relation between several model refactorings and inconsistency resolutions. The results are shown in Table 9.2. This table presents the same 9 model refactorings as before (colums), but this time in relation to the concrete inconsistency resolutions (rows) that we have defined for supporting the execution of these model refactorings. These inconsistency resolutions are distilled from the decision diagrams in Appendix C and listed on the right-hand side of Table 9.2. We discovered 17 different resolutions for 4 inconsistency categories. The cells indicate if an inconsistency resolution has been employed to execute a model refactoring. In some cases a particular resolution can occur several times in the same

model refactoring, as is also illustrated in the *Move Operation* model refactoring presented in Section 9.4. It is clear from the table that there is significant reuse of resolutions in and across model refactorings. The number of times an inconsistency resolution is reused can be calculated from this table by making the sum of the numbers of the row corresponding to the inconsistency resolution. However, these numbers are lower bounds. Some resolutions cause the execution of a model refactoring, e.g., S1 and some model refactorings are composed of other refactorings. In the numbers shown in Table 9.2, we do not take into account that in the execution of some model refactorings, other model refactorings need to be executed. An exception is the *Extract Class* because this refactoring is a composition of two other model refactorings.

		Add Parameter	Change Bidirectional Association to Unidirectional	Extract Class	Move Attribute	Move Operation	Pull up Operation	Push down Operation	Extract Operation	Replace Conditional with Polymorphism	
dangling feature reference (DFR)	S1		1	1		1				1	S1 →DFR, Move operation
	S2		1	3	1	2		1	2	1	S2 →DFR, Use existing lifeline
	S3		1	3	1	2		1	2	1	S3 →DFR, Add lifeline
	S4	1									S4 →DFR, Use existing operation
	S5		1						2		S5 →DFR, Add new operation
dangling association reference (DAR)	S6			1	1						S6 →DFR, Encapsulate Attribute
	S7						1				S7 →DFR, Pull up operation
	S8										S8 →DFR, Abstract operation
	S9		1	4	1	3		1	2	1	S9 →DAR, Add parameter
	S10		1	4	1	3	1	1	2	1	S10→DAR, Add reference
	S11		1	4	1	3	1		2	1	S11→DAR, Use reference
	S12			1		1		1			S12→DAR, Delegate operations
dangling type reference (DTR)	S13	2							3		S13→DTR, Add type
	S14	2							3		S14→DTR, Use existing type
specification incompatibility (SI)	S15		1								S15→SI, Add parameter
	S16		1								S16→SI, Add reference
	S17		1								S17→SI, Use reference

Table 9.2: Analysis of reuse of inconsistency resolutions in and across model refactorings.

Note that the inconsistency resolutions belong to structural inconsistency categories. We have omitted the behavioural inconsistencies that occur in the analysed model refactorings since we did not include the resolution of these inconsistencies in our tool support (see next chapter).

As already mentioned in the previous chapter, a rule-based system separates the definition of a rule from the strategy employed by a rule engine for selecting the rules to be fired. Our rule-based approach to inconsistency resolution provides a mix between automation and user input: the rule engine automatically finds all applicable rules in each situation

and as such automatically constructs an implicit flow of inconsistency resolutions, whereas *the developer is able to select his or her preferred resolution* out of the applicable ones.

9.5.2 Open Issues

The evaluation presented in the previous section, clearly shows that our rule-based inconsistency resolution approach is useful in the context of applying a model refactoring. However, some issues need to be discussed or need to be addressed by an extension of our approach. In the remainder of this section, we will discuss two issues in depth and briefly describe the remainder of the issues. The latter issues are outside the scope of this dissertation and are described in detail as future work in the last chapter of this dissertation.

- The decision diagrams presented in Figure 9.9 and in Appendix C need to be interpreted as possible executions of the corresponding model refactorings. By drawing these diagrams, we claim that an inconsistency resolution approach can be used to support the application of these refactorings and that the same inconsistency resolutions reoccur in and across different model refactorings. We do not claim that these decision diagrams are *complete* and *correct*. These diagrams are the result of manually redesigning the corresponding refactorings and, based on, e.g., application- or system- or company-specific requirements, other resolutions than the ones specified in the decision diagrams can be added. Remark that it is not easy to find all the relevant places in the different diagrams where a new resolution needs to be added. In our rule-based approach, a resolution corresponds to a rule that needs to be defined only once and that will automatically become applicable.

- A refactoring is *correct* if it *terminates*, the model is *syntactically correct* and *some behavioural properties of the model* are preserved [Por03]. Due to the fact that resolutions can introduce new inconsistencies, a rule can be fired more than once. This can introduce cycles leading to an infinite chain of resolutions. In our tool support (see next chapter), the firing of the same rule on the same data is allowed only once preventing such cycles. Formal techniques can be used to prove termination of the rules. The syntactical correctness of the model is guaranteed because of the usage of DL. As already explained, the UML metamodel is translated into concepts and roles, called definitions, and the user-defined models are translated into assertions which can be checked for conformance with the definitions using a standard DL reasoning task. A last issue is to prove that the model refactoring preserves some behavioural properties of the model. We addressed this issue in the first part of this chapter.

- The inconsistency resolutions implemented in our approach can be too *fine-grained*. Depending on the application and system, it must be possible to group different resolutions or to define new resolutions (see Section 11.2.4).

- Different rules can be fired at the same time, user interaction is used in our approach to decide which inconsistency must be resolved first. As a result of the resolution of a certain inconsistency, it can be that other inconsistencies are resolved as well. This can happen in our approach as a side-effect of choosing a particular resolution. To find an *optimal ordering* of the different rules, analysis of the dependencies between the inconsistencies and between different resolution actions is needed (see Section 11.2.3).

- Even with an optimal ordering of the rules, it is still possible that *a lot of inconsistencies are detected* and for each inconsistency, *a lot of resolutions are possible*. The question is how to manage all these inconsistency occurrences and their resolutions. Several possibilities can be investigated (see Section 11.2.3).

- Other refactorings then the ones presented in this chapter and in Appendix C need to be analysed in the context of our rule-based inconsistency resolution approach (see Section 11.2.5).

- In the context of a refactoring, an inconsistency resolution can be the application of a certain refactoring as shown in the *Move Operation* refactoring presented in Section 9.4. The question remains how to support this by our rule-based DL inconsistency resolution approach (see Section 11.2.5).

9.6 Related Work

Research on model refactoring is emerging. A set of basic UML refactorings is provided in [SPLJ01] to improve the software design in a stepwise fashion. Model refactorings are defined in [Por03] as a sequence of transformation rules. Reusability of refactoring steps across different refactorings is not considered, whereas in our work inconsistency resolutions are decidedly reused, not only conceptually but also their actual definitions as rules. Moreover, the transformation rules in [Por03] are executed in the order they are defined and there is no rule engine that chains rules, i.e., (re)activating rules when data is changed or created.

To the best of our knowledge, tool support for model refactorings is only discussed in [BSF02] and [Ast02]. Boger *et al.* [BSF02] show how model refactorings can be integrated in the Poseidon [gen05] UML refactoring browser. However, this plug-in is not available anymore for Poseidon. Boger *et al.* define several model refactorings such as state machine refactorings and activity diagrams. Whether these model refactorings can also be supported through inconsistency resolution needs further investigation. Astels [Ast02] uses a UML tool to perform source code refactorings more easily, and also to aid in code smell detection. However, possible inconsistencies or problems are left for detection by the source code compiler. A *Move Method* e.g., is done by just dragging the method in the corresponding UML class diagram to the target class.

Surprisingly, none of the above approaches towards model refactoring takes behaviour preservation into account. One of the reasons is that there is no generally accepted behavioural interpretation of UML models. Therefore, we consider this as an important contribution of this dissertation. Few other works have been presented in the context of behaviour preservation of refactorings. In [EHKG02], transformation rules are defined in the context of model evolution. These rules express which operation are allowed on a certain model and transform this model into another model preserving some property. However, there is no rule engine chaining the rules. Some behavioural properties of source-code refactorings have been defined in [MDJ02] and graph transformations have been defined to support them. These properties are quite general. One property defined in [MDJ02], *call preservation*, is refined by us into several possible behaviour preservation properties. Van Gorp *et al.* [VSMD03] define source-consistent model refactoring contracts. These contracts allow to compose refactorings and to check some program behaviour properties.

By complementing their contracts with our consistencies a relation can be created between the UML diagrams and source code.

9.7 Conclusion

In this chapter, we show how starting from some of our defined consistencies, similar definitions of behaviour preservation properties can be derived between a model and its refactored version. We also prove some additional properties regarding inheritance consistency and behaviour preservation between a refined model and its refactored version.

In a second part of this chapter, we argue that the rule-based approach for inconsistency resolution can be applied to the domain of model refactorings. We elaborate on a particular model refactoring, *Move Operation*. We show that in order to execute this refactoring, a chain of inconsistency detection and resolution steps is actually performed. Our rule-based approach enables *reuse of inconsistency resolutions in and across model refactorings* and *support for user-guided selection of inconsistency resolutions*. However, there are still some open issues which we briefly described in this chapter, but that will be discussed in detail in Chapter 11.

Until now, we discussed inconsistency detection, resolution and support for model refactorings on a conceptual level. How these activities are supported and how this support has been implemented in a state-of-the-art UML CASE tool is explained in the next chapter.

Chapter 10

Proof-of-concept Tool Support

In the previous chapters we have shown how inconsistencies can be detected and resolved using DLs. We observed, in the previous chapter, that these ideas can be used for the execution of model refactorings. The rule-based formalism presented, introduces some variability in the implementation of the rule engine's strategy. In this chapter, proof-of-concept tool support is presented for demonstrating the detection and resolution of inconsistencies and the execution of model refactorings. The tool support is a proof-of-concept because it is a prototype tool and only a limited number of inconsistencies and model refactorings are implemented.

First, we introduce our prototype tool (Section 10.1). Next, implementations of inconsistency detection queries are shown (Section 10.2). Finally, we illustrate the implementation of inconsistency resolution rules by using these rules for the implementation of model refactorings (Section 10.3). This chapter uses examples taken from our case study to illustrate how the ideas presented in this dissertation can be implemented.

10.1 Introduction to *RACOoN*

In this section we introduce our proof-of-concept tool, called *RACOoN*, which stands for *Resolution Actions for inCONsistencies*[1]. The tool is a *proof-of-concept* tool due to several reasons: (1) it is a prototype tool; (2) only a representative set of inconsistency resolution actions and model refactorings are implemented for demonstrating our work; (3) the user interface of the tool can still be improved. Consequently, the tool can still be extended and improved in various ways. The possible extensions and improvements are discussed in the next chapter.

Our tool is a Poseidon [gen05] plug-in using RACER as DL reasoning engine. Poseidon extended with this plug-in that uses RACER, serves as our inconsistency detection and resolution environment. In the next section, we explain the architecture of our proof-of-concept tool.

[1] Definition of racoon by the Encyclopædia Britannica: *also called ringtail, any of two to seven species (depending on the authority) of small, nocturnal carnivores constituting the genus Procyon (family Procyonidae) and characterised by bushy, ringed tails.*

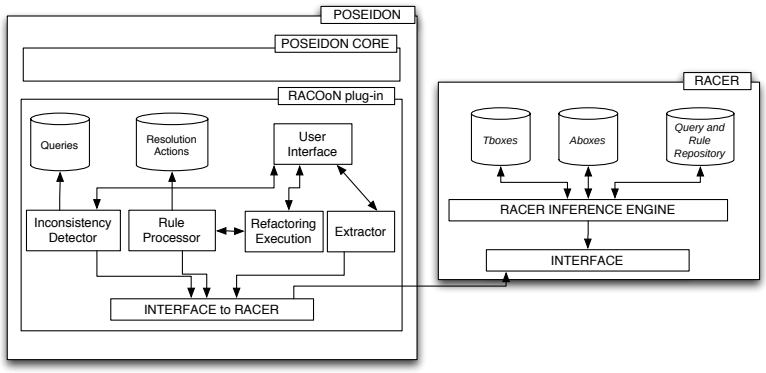

Figure 10.1: Architecture of inconsistency detection and resolution environment.

10.1.1 Architecture

In [SVJM04], we set up a preliminary tool chain for the purpose of checking inconsistencies between UML models. This tool chain has evolved into a Poseidon plug-in which was initially developed by Jocelyn Simmonds [SB05]. We extended this plug-in with various inconsistency detection queries, the ability to generate different *Tboxes* using spanning functions and a rule-based resolution approach, and finally a model refactoring execution engine.

Due to its plug-in mechanism and because it is a well-known UML CASE tool, Poseidon is a suitable basis for our inconsistency detection and resolution environment. In Chapter 5, we already motivated the choice of RACER as a DL system. The architecture of our environment is depicted in Figure 10.1. This figure shows Poseidon and RACER as starting elements of our environment. *RACOoN* is plugged into the Poseidon tool and contains six components. Each of these components will be discussed below.

User interface Through the user interface of the plug-in, the tool can be configured and the translation of the UML metamodel fragment can be loaded into RACER. Through this interface the detection and resolution of particular inconsistencies on particular UML models can be chosen and executed. Also the execution of model refactorings can be controlled in this interface.

Extractor The extractor allows for the translation of the user-defined UML models into *Abox* assertions. It also defines the spanning functions for the creation of *Tbox* statements representing different interpretations of UML model elements.

Inconsistency detector The inconsistency detector communicates with RACER through the *interface to* RACER to check whether certain inconsistencies, indicated by the user, occur. It also processes the results returned by RACER.

Rule processor Different rule sets for the resolution of an inconsistency are defined by the rule processor and activated by sending these rules to RACER. Different rule engine strategies can be implemented in this component. We will explain one possible strategy in Section 10.3.

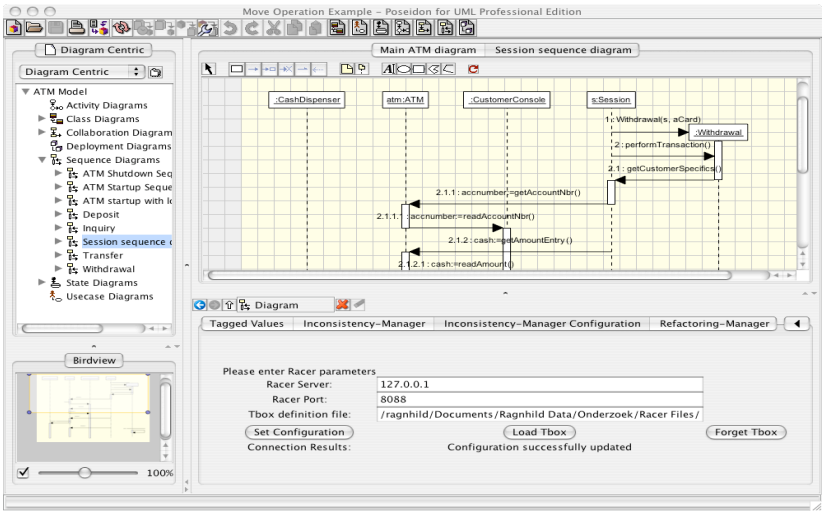

Figure 10.2: Screenshot of *RACOoN*'s configuration pane in Poseidon.

Refactoring execution This component implements the execution of the model refactorings. During this execution, a rule engine, defined in the rule processor, is called.

Interface to RACER This component enables the communication between the different above presented components of the tool and RACER. RACER is available as a server and it also offers an extensible Java client interface, called JRacer. Through this interface, interactions between the reasoning engine and the plug-in can be implemented straightforwardly.

Next to these components, the tool also contains a set of queries and resolution actions. The queries are used to detect part of our classified inconsistencies, they are also used by the spanning functions and in the resolution rules.

In Figure 10.2 a screenshot of *RACOoN*'s *configuration pane* is shown. Through this configuration pane, the RACER parameters – the IP address of the RACER server and its port – can be dynamically changed. The *Tbox* containing the UML metamodel fragment can also dynamically change.

CASE Tool Restrictions Due to the usage of Poseidon, some technical restrictions must be taken into account. Most of these restrictions also apply to other commercial UML CASE tools. Because the plug-in mechanism of Poseidon is used and Poseidon is implemented in Java, our tool is also implemented in Java. All commercial CASE tools – also Poseidon – still use the metamodel of the UML version 1.4. As a result queries on the UML metamodel, including our inconsistency detection queries, are written using the UML version 1.4. This does not derogate the proof-of-concept of our ideas. Compared to the metamodels of the UML versions 1.x, the metamodel of the UML version 2.0 is mostly changed for the representation of sequence diagrams. Lifelines are made explicit in version 2.0 while in the previous versions, a lifeline is an object of a certain type. Poseidon does not support explicitly the role interaction interpretation of sequence diagrams. We restricted

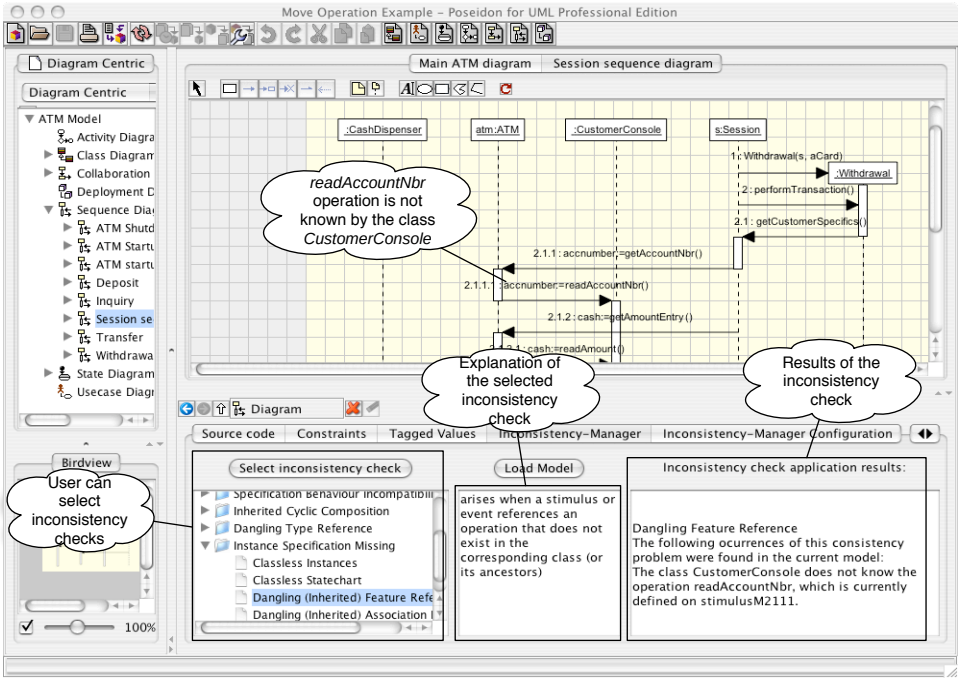

Figure 10.3: Screenshot of *RACOoN*'s inconsistency manager pane in Poseidon.

our support for inconsistency detection and resolution to sequence diagrams representing object interactions.

In the remainder of this chapter, we will demonstrate how inconsistencies are detected and how model refactorings are executed using our tool.

10.2 Inconsistency Detection in *RACOoN*

In Figure 10.3 a screenshot is shown of *RACOoN*'s *inconsistency manager pane*. Our catalogue of inconsistencies is shown in this pane. This catalogue is not hard-coded but can be changed by the user and is loaded when the tool is started. Translating models from Poseidon to RACER and checking properties on these models are independent activities. The checks are executed on user-demand.

10.2.1 Querying the Metamodel

As explained in Chapter 7 our classified inconsistencies can be checked by executing *nRQL* queries or by *Tbox* or *Abox* evaluation functions. If the user selects a check implemented by a *nRQL* query, this query is retrieved from our query repository and processed by the query processor. If an inconsistency occurs the inconsistency manager pane informs the user and lists the UML elements involved in this particular occurrence of the inconsistency.

The screenshot of Figure 10.3 shows that there is a dangling feature reference in the modelled sequence diagram. The operation named *readAccountNbr* is not known

to the *CustomerConsole* class while it is invoked on an object that is an instance of *CustomerConsole*.

Before continuing with the implementation of inconsistency detection via our DL framework, we discuss the possible impact of the different *nRQL* completeness modes on our inconsistency detection tool support. This discussion is also relevant for inconsistency detection via our DL framework.

10.2.2 Impact of *nRQL* Completeness Modes

In Chapter 7, we showed that it is possible to reason in different modes of completeness in *nRQL*. What does this mean for inconsistency detection in our tool? Recall that reasoning in an incomplete mode only uses the syntactic, told information from an *Abox*, while reasoning in a complete mode uses this told information plus exploited *Tbox* information for all concepts.

On the one hand, in *RACOoN* the translation of elements of a user-defined UML model is based on the elements modelled in Poseidon. Poseidon and all current state-of-the-art UML CASE tools allow the user to only create instances of metaclasses that are leafs of metamodel inheritance hierarchies and each user-defined model element is an instance of a certain metaclass. If we could specify in a CASE tool that, for example, an element i of a model is an instance of the metaclass *InstanceSpecfication* and that this element is connected to an instance c (the type of c is not known) via the meta-association *classifierspec*, the following query

```
(retrieve (?x ?y)
   (and (?x instancespecification)
      (?x ?y classifierspec)
      (?y classifier)))
```

returns `NIL` in an incomplete mode, and it returns the set {((?x i) (?y c))} in complete mode.

On the other hand, our queries use those leaf metaclasses to navigate over the metamodel. Using a metaclass that is not a leaf in an inheritance hierarchy together with (in)completeness of querying, has an impact on the results of the query. Consider, for example, the query detecting a dangling feature reference. If the metaclass $BehaviouralFeature$ would have been used in this query instead of *Operation*, the inconsistency detected in the previous section would not have been detected in an incomplete mode. It would have been detected in a complete mode. The operation in question (*readAcountNbr*) is modelled as an instance of the metaclass *Operation*. An incomplete mode only takes this information into account, while a complete mode also takes into account that *Operation* is a subclass of $BehaviouralFeature$.

We can conclude that because our tool uses the information delivered by Poseidon and our queries use metaclasses that are leafs in an inheritance hierarchy, the impact of reasoning in an incomplete mode or complete mode has minor impact on the inconsistencies detected. Further research is necessary to determine the precise impact of these reasoning modes.

10.2.3 DL Framework

If the inconsistency check is implemented through *Tbox* or *Abox* evaluation functions, an appropriate *Tbox* or *Abox* is generated through the use of spanning functions. The *Strategy*

```
public void spanningFunction(String instanceName) throws RacerException,
    IOException {
        processTransitions();
        if (! (mappingTable.isEmpty())){
             processSequences();
             processDisjointness();
}}
```

Figure 10.4: Implementation of the translation of call sequences of a PSM.

pattern is used to implement the generation of the different *Tboxes* representing (complete) sets of call sequences and SD traces.

The code fragment shown in Figure 10.4 shows part of the code implementing the generation of a *Tbox* representing a set of call sequences. First the transitions are processed. This activity is shown in Figure 10.5. For each transition the operation, precondition and postcondition is retrieved. *nRQL* queries are used to retrieve the operations and the preconditions and postconditions. In this version of the tool, we assume that the pre- and postconditions are DL concepts because the automatic translation of OCL constraints to DL statements is not yet implemented. Finally, the transition is translated into a DL concept. This process results for the example in the RACER Fragment 6.3 in the generation of the first 10 lines.

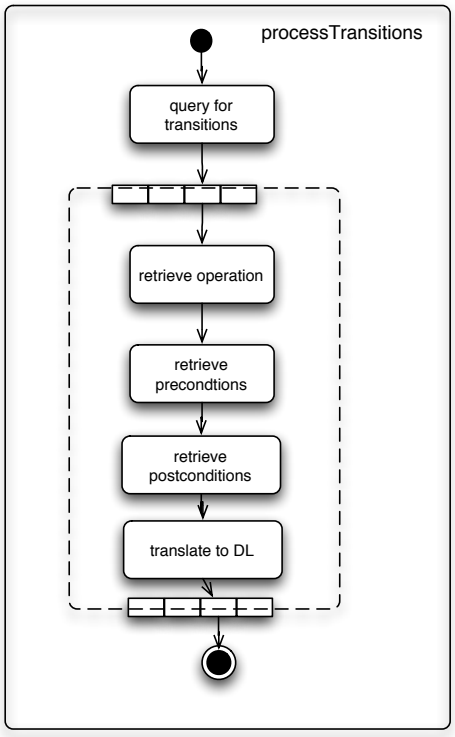

Figure 10.5: The activity of processing the transitions of a PSM.

A second step is to process the different call sequences and to put them in a sequence. The method `processSequences` modelled in the activity diagram shown in Figure 10.6, first retrieves the initial state of the PSM through a *nRQL* query. Secondly, the transitions outgoing this state are retrieved and each transition is processed by the activity `processSequence`.

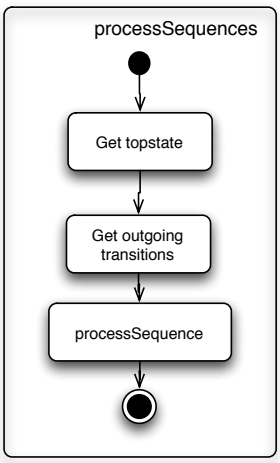

Figure 10.6: The activity of processing the call sequences.

The activity `processSequence` is implemented by two methods: `processSequence` and `processCallSequences`. Relevant fragments of these methods are shown in the code fragment shown in Figure 10.7.

The method `processSequence` retrieves the target state of a given transition `trans`. If this state is already known by the spanning function, the algorithm stops because the transitions outgoing this state are already processed and translated into *Tbox* statements. If the target state is not yet known, the transitions outgoing this target state are retrieved and processed together with the transition `trans` in the method `processCallSequences`. The method `processCallSequences` (shown in the code fragment in Figure 10.7) puts the given transition `trans` in sequence with each transition from the string `transitions`. For each such transition, the method `processSequence` is called again with the target state as source state and the transitions as starting transition.

The translation of *Tbox* statements generated by this spanning function complemented by a similar translation of SD traces can be used to check invocation or observation consistency between a PSM and sequence diagrams or to check behaviour compatibility between a PSM and relevant sequence diagrams.

10.3 Supporting Refactorings in *RACOoN*

Through the step-by-step execution of the model refactoring *Move Operation*, we will not only show how our rule-based inconsistency resolution approach can be used in the context of model refactorings, but also how inconsistencies are resolved in our approach.

```
private void processSequence(State sourcestate, Transition trans) throws
  RacerException, IOException{
 String query = queryRepository.getQuery(_CC_TargetTrans(trans.getName()));
 String res = client.synchronousSend(query);
 String targetName = format(res);                    First, get the target state
 State targetstate = null;                              of transition trans
 if (! (States.containsKey(targetName))){
   targetstate = new State(targetName);
   States.put(targetName, targetstate);
   String q = queryRepository.getQuery(_CC_TransState(targetName));
   String transitions = client.synchronousSend(q);
   processCallSequences(sourcestate, trans, targetstate, transitions);
 }                                          If target state is not yet processed,
}                                        get the outgoing transitions, otherwise
                                                translation stops

private void processCallSequences(State sourcestate, Transition trans,
              State targetstate, String transitions) throws RacerException,
                   IOException{
 //for each transition
 if (!(mappingTable.containsKey(transName))){
   targettrans = new Transition(transName);
   mappingTable.put(transName, targettrans);
   } else{
     targettrans = (Transition)mappingTable.get(transName);
     }
 sourcestate.translateSequencetoDL(trans, targetstate, targettrans, getFlag());
 processSequence(targetstate, targettrans);

  String DLresult = sourcestate.finishTranslation();
  writer.write(DLresult);                  For each transition, first translate the sequence
}                                             and next, process the next sequence
```

Figure 10.7: Code implementing the processing of a sequence.

In Figure 10.9 a screenshot is shown of *RACOoN*'s *refactoring-manager pane*. The catalogue of refactorings is shown in this pane. This catalogue is not hard-coded but can be changed by the user and is loaded when the tool is started.

10.3.1 *Move Operation* Step-by-Step in *RACOoN*

We use in this section the same example as used in Section 9.4. The operation $getCustomerSpecifics$ is moved to the $Withdrawal$ class.

If the user selects the *Move Operation* refactoring in the refactoring manager pane, the first step of this refactoring is executed. The user is asked for the operation to be moved and the target class. The specified operation is automatically added to the specified target class.

In a second step the body (as specified in a certain sequence diagram) of the operation is copied to the target class. This is done automatically. First, the user is asked for the object that contains the body of this operation. Secondly, the user is asked for the object where the body of this operation should be copied to. After copying the operation body, a rule engine is invoked with the necessary *nRQL* rules. Each *nRQL* rule represents a possible resolution of a certain inconsistency.

Remember that current *nRQL* rules do not support expressions prompting for user input in their condition. How do we implement this user interaction? Because the condition of a *nRQL* rule is an ordinary *nRQL* query, the conditions of all the rules are matched against the data of the *Abox*. The list of returned bindings for each of the different rules is presented to the user. This functionality is implemented by the method *runIt* modelled as an activity in Figure 10.8.

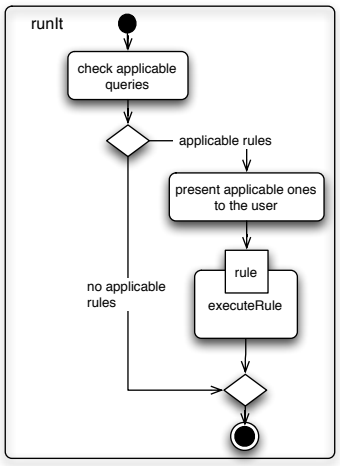

Figure 10.8: Running a rule engine.

In the example, one occurrence of the dangling feature reference inconsistency and four occurrences of the dangling association reference inconsistency are detected. These occurrences together with the proposed solutions are shown to the user (see right bottom corner of Figure 10.9).

Three different resolutions for the dangling feature reference inconsistency are proposed: (1) add the operation to the class to which the operation is sent; (2) send the operation to an existing object that is an instance of a class owning the operation; (3) send the operation to a new object that is an instance of a class owning the operation. The different possible resolutions for this inconsistency are shown in the right bottom corner of Figure 10.9.

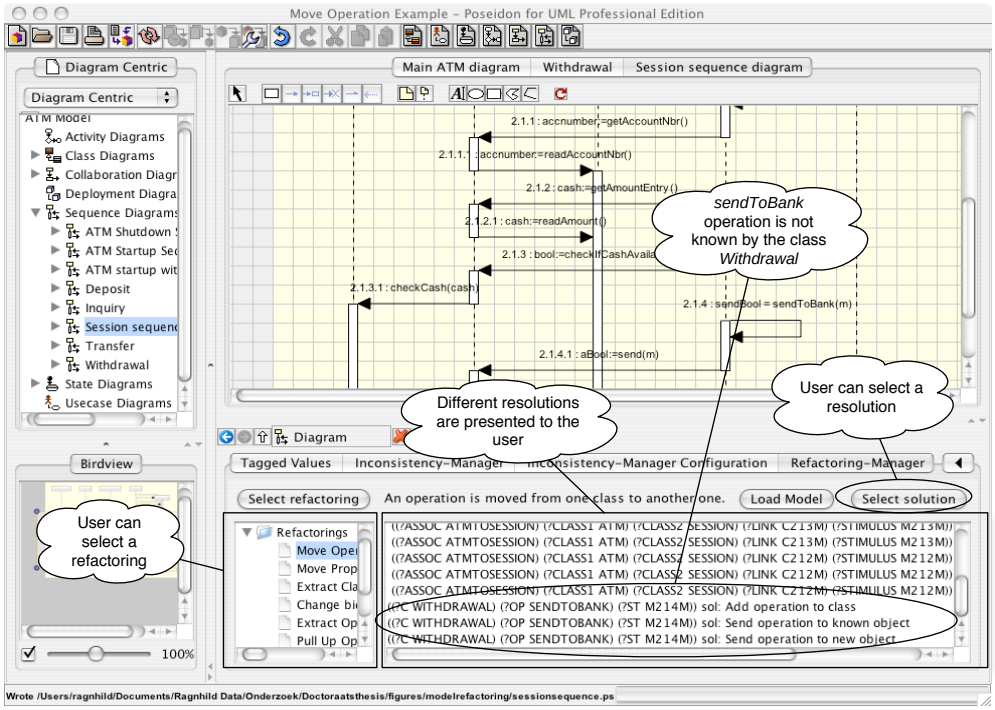

Figure 10.9: Screenshot of rule instantiations.

The user has to select which rule instantiation, i.e., which inconsistency resolution, will be chosen for a particular binding through the plug-in interface. We choose to send the operation to the existing object of type *Withdrawal*. After the user has selected a possible resolution, the method *executeRule* is executed (modelled as an activity in Figure 10.10).

First, user input is asked. In our example, we need to know to which existing object (or lifeline) the operation must be sent. This user input is asserted as a new concept. The rule is prepared and the correct rule instantiation is executed. The consequent of this particular rule instantiation is added to the *Abox*. The user input assertions are removed from the *Abox*. Because only one occurrence is resolved and resolution actions can introduce new inconsistencies, the conditions of the rules are again matched against the changed *Abox* by calling the method *runIt* again. Consequently, after each inconsistency resolution, the applicable inconsistency resolutions are updated by the rule engine.

As a result of the resolution of this dangling feature reference inconsistency in our example, an occurrence of the dangling association reference inconsistency is introduced. In Figure 10.11, four different resolutions are proposed to the user: (1) add a new association; (2) add a parameter; (3) add delegating operations, and, finally, (4) use an existing asso-

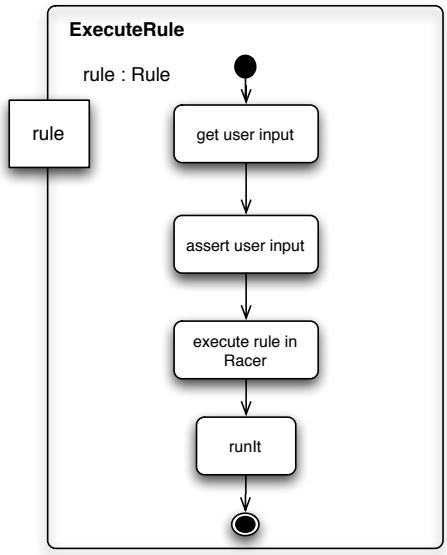

Figure 10.10: Execution of a rule.

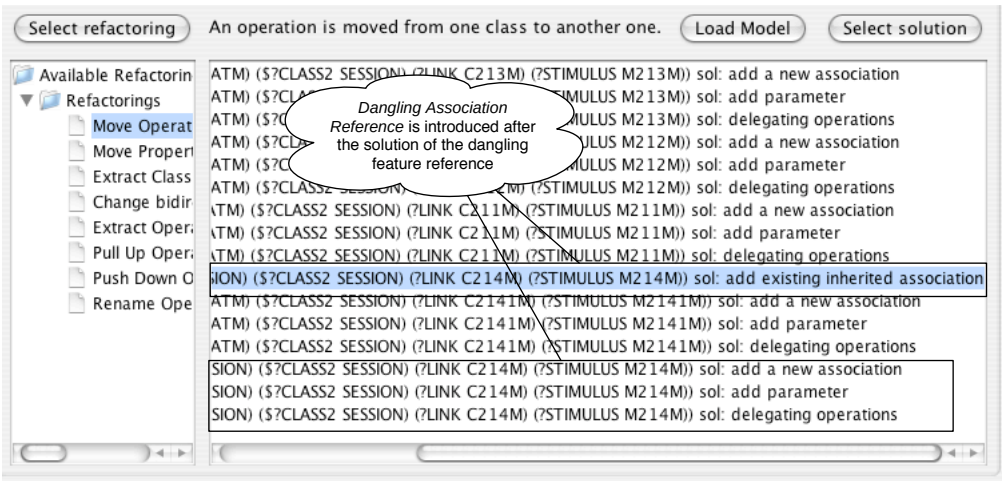

Figure 10.11: Screenshot of possible resolutions for the *dangling association reference* inconsistency occurrences.

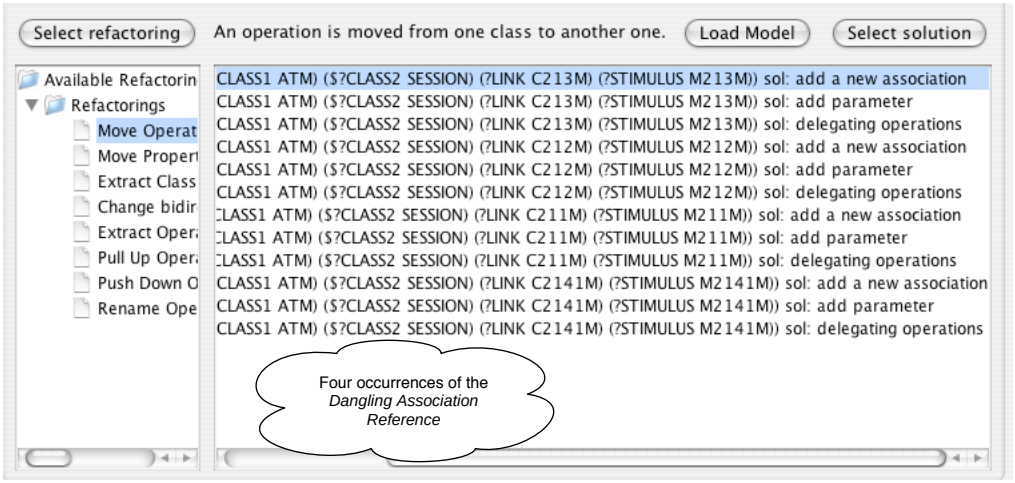

Figure 10.12: Still four occurrences of the *dangling association reference* inconsistency.

ciation. Again, the user has to select which inconsistency resolution will be chosen for a particular binding. We choose to use the existing association defined between the classes $Transaction$ and $Session$.

After this inconsistency resolution, the applicable inconsistency resolutions are updated by the rule engine. This results in the detection of four occurrences of the dangling association reference inconsistency. Three different resolutions for each occurrence of the inconsistency are proposed: (1) add a new association; (2) add a parameter to the operation; (3) send operation calls to the source class. The possible resolutions are shown in the right bottom corner of Figure 10.12. The fourth possible resolutions – use an existing association – does not occur in the list of possible resolutions because there is no existing association between the classes ATM and $Withdrawal$. In our example, we choose the first resolution shown in Figure 10.12. First, user input is asked. In our example, we need to know the name of the new association. This user interaction is shown in Figure 10.13. As a result, a new association is created between the classes ATM and $Withdrawal$ and that particular occurrence of the dangling association reference is resolved.

By re-executing the rules after this resolution, three occurrences of the dangling association reference inconsistency are still detected. There is one occurrence of this inconsistency less because it was resolved in the previous step, but, next to the three possible resolutions as proposed for the previous four occurrences of this inconsistency, a new possible resolution is added. Due to the previous resolution action, there exists an association between the classes ATM and $Withdrawal$. The four possible resolutions for each remaining occurrence of the dangling association reference are shown in Figure 10.14. This association can now be used to resolve the other occurrences of the dangling association inconsistency.

As a last step of this refactoring the user has to decide whether to remove the operation from the source class or retain it as a delegating method. Suppose the user decides to remove the operation from the source class. This is done automatically. As a next step, the rule engine is called again with the necessary rules. The different resolution steps are similar to the ones discussed above and shown in the different figures (Figure 10.9, Figure 10.11,

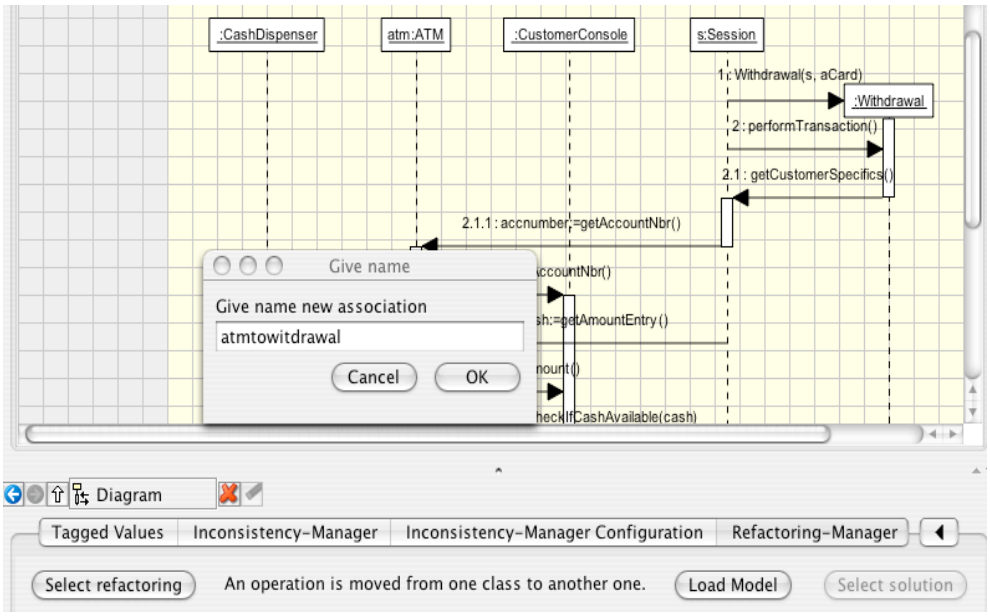

Figure 10.13: Asking user input for a certain inconsistency resolution.

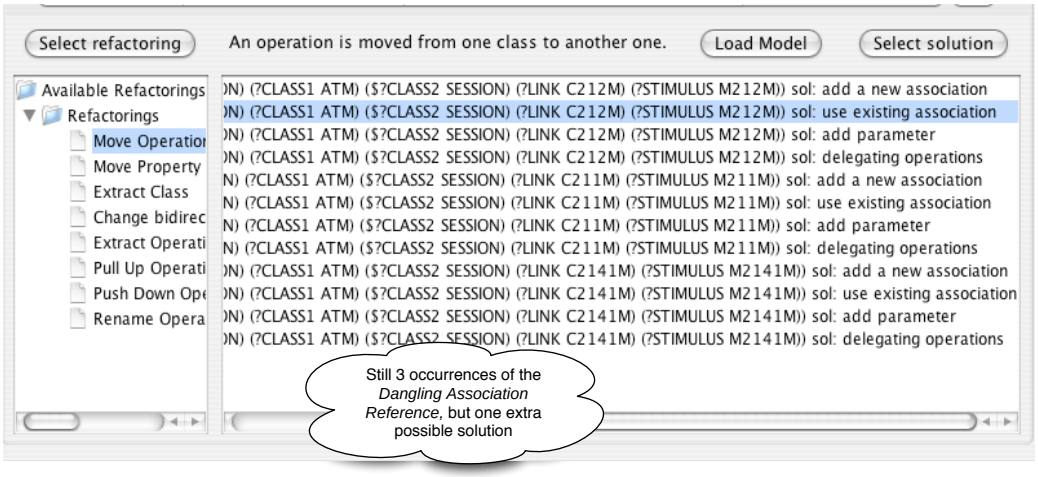

Figure 10.14: Extra possible resolution for the remaining three occurrences of the *dangling association reference* inconsistency.

Figure 10.12, Figure 10.14).

In this section, we showed how the strategy presented in the previous chapter can be implemented. Remark that at this moment, only the *Abox* assertions are changed. The implementation of the visual changes in Poseidon is under construction. Issues not addressed by this strategy, such as fine-grained resolutions, an optimal ordering of rules, managing a lot of detected inconsistencies and the associated possible resolutions, are detailed in the next chapter.

10.4 Conclusion

In this chapter, we introduced our prototype tool support for inconsistency management. Due to the important role of inconsistency management in MDE, tool support for the different activities of this process becomes indispensable. Our inconsistency management environment consists of our tool *RACOoN* that uses RACER and that is plugged into Poseidon. First, we introduced the architecture of *RACOoN*, our proof-of-concept inconsistency detection and resolution tool. Because this tool is a proof-of-concept of the ideas presented in this dissertation, no experiments with end-users can be done due to the rudimentary user interface of our tool. In the next chapter, we will discuss possible improvements and extensions to our tool.

Despite the different limitations of this environment, we were able to show (1) how our inconsistency detection approach – as introduced in Chapter 7 – can be implemented and captured by tool support; (2) how our inconsistency resolution approach applied to model refactorings – discussed in Chapter 9 – can be implemented and captured by tool support.

Chapter 11

Conclusion

In this chapter, the conclusions of this dissertation are presented. First, the ideas and work presented in this dissertation are summarised stressing our contributions (Section 11.1). Finally, directions for future research are discussed (Section 11.2).

11.1 Summary and Contributions

The goal of this dissertation was to investigate inconsistencies in (evolving) object-oriented software models and to address their definition, detection and resolution by developing a coherent inconsistency management framework using a declarative formalism.

MDE is an approach to software engineering where the primary focus is on models. Different views of the software system are covered by different models and these different models can get refined or can evolve. Managing and synchronising the different models is a complex task and inconsistencies can arise easily. As declarative formalism, Description Logics are evaluated for the purpose of inconsistency definition, detection and resolution. As a result, this work combines two fields from two different computer science disciplines, inconsistency management, that is a research field in software engineering, and Description Logics, that is a research field in artificial intelligence. In this summary the same sequence of steps is followed as in the dissertation: first, the modelling language and the important activities in the inconsistency management process are synthesised, next different criteria for an inconsistency management environment are summarised, then Description Logics and relevant systems are revisited, finally, we go back to the different identified criteria and review to which extent DLs and their systems can be used to address the requirements identified for each of these criteria. In each of the steps, we clearly indicate the contributions of this dissertation.

The UML became the standard state-of-the-art modelling language and is heavily used in current MDE approaches. Because of the standardisation and different revision cycles of the OMG, the UML has become a huge language, full of compromises. Some people consider this as an advantage because the UML has the potential to cover a broad range of systems. However, this advantage is also a disadvantage. Because the UML is so huge, it is difficult for a human to know it completely and inconsistencies can easily arise in its abstract syntax and it is easy to make mistakes to its abstract syntax. UML also lacks a precise semantics. In this dissertation, we go back to the basic features of the UML used in the context of object-oriented modelling. We only consider a limited fragment of the UML

abstract syntax. This fragment allows the expression of class diagrams, sequence (communication) diagrams and protocol state machine diagrams. We use relations to describe the various abstract syntax elements. We also introduce a possible semantics for the described behaviour of (interacting) objects. The goal of this formalisation is to have a clear, coherent and unambiguous description of the UML fragment considered. This formalisation will serve as a basis for the definition, detection and resolution of inconsistencies. This results in our first contribution:

Contribution 1: A lightweight formalisation of a UML fragment representative for the basic features of an object-oriented modelling language and serving as a basis for the definition of certain activities of the inconsistency management process [VJM04], [VMJ06].

Different dimensions of *consistencies* are recognised in literature. One such dimension is syntactic versus semantic consistency. Depending on the modelling language, application domain and so on, syntax and semantics are distinctly defined. We define syntactic consistency as the guarantee that a specification conforms to the abstract syntax of the modelling language. Semantic consistency is defined as the complement of syntactic consistency. We define violations of semantic consistency, i.e., semantic inconsistencies as violations that can not be defined as conformance violations to the abstract syntax.

Based on literature studies and on the UML fragment considered, we define a two-dimensional classification of domain-independent inconsistencies. The first dimension is: structural versus behavioural, while the second dimension is: instance versus instance/specification versus specification. Our classified inconsistencies are defined using our formalisation of the UML fragment under consideration. The set of classified inconsistencies is not exhaustive but we believe that it is a representative set. This belief is strengthened by the work of Lange *et al.* [LCM+03]. This work reports on an empirical investigation of the occurrences of inconsistencies in the UML designs of six large-scale industrial cases. Our classified inconsistencies are a superset of the inconsistencies identified in their research. This leads to our second contribution:

Contribution 2: An open, two-dimensional classification of precisely defined, domain-independent inconsistencies [VMSJ03], [SVJM04], [MVS05].

Inconsistency management is an important issue in the context of MDE. Tool support is needed for the different activities of this process. In our opinion, this tool support must rely on a powerful formalism enabling the precise definition, detection and resolution of inconsistencies. Inconsistency resolution is, as recognised in literature, a difficult issue. We distill a set of inconsistency resolution challenges.

We propose the declarative formalism Description Logics to support static inconsistency checking and resolution of UML models. Because the usage of DLs and their systems for inconsistency definition, detection and resolution is an unexploited field, we first establish possible criteria by which such formalisms can be evaluated.

Next, we introduce Description Logics, a family of logic languages. DLs have roots in frames and semantic networks. It was recognised that frames could be given a semantics relying on fragments of first-order logic. The basic inference task in Description Logics is subsumption. DLs are useful in the design of knowledge-based applications. A DL knowledge base has two components, a *Tbox* and an *Abox*. A *Tbox* contains general knowledge about the problem domain (also called terminological knowledge) while an *Abox* contains

knowledge specific to a particular domain (also called assertional knowledge). Most research focuses on *Tbox* reasoning as opposed to *Abox* reasoning. Several DL systems were developed. We give a survey of the most important ones and choose the RACER system to be used in our approach. We show that DL systems can play an important role in the context of inconsistency management and that *Tbox* and *Abox* reasoning are both important in this context.

A first step is to investigate the suitability of DLs as a representation language for the abstract syntax of the UML fragment under consideration and as a possible semantic domain for PSMs and traces modelled by sequence diagrams. This investigation resulted in the definition of a DL representation framework. This leads to our third contribution:

Contribution 3: The representation of the abstract syntax of the UML fragment and the representation of the semantics of PSMs and traces modelled by sequence diagrams by well-defined DL *Tboxes* and *Aboxes* [VMSJ03], [VSM03], [Van04], [VMJ06].

Secondly, we investigate whether the standard DL reasoning tasks are sufficient for the detection of our classified inconsistencies. We conclude that these reasoning tasks are sufficient, but that enhanced reasoning on *Aboxes* using the standard reasoning tasks, is necessary. DL systems can be equipped with DL query languages but at the time of our survey of DL systems, no state-of-the-art DL system was equipped with such a query language. This observation has led to the development of a sophisticated query language for the RACER system by the RACER developers. Consequently, through our work we contributed to the research in the field of Description Logics and associated systems.

Contribution 4: The definition of the detection of our classified inconsistencies through standard DL reasoning tasks and a sophisticated query language developed by the RACER authors based on, among other things, our concrete input [VSM03], [HMSW04].

In this dissertation, we focus on resolution actions as a means to resolve inconsistencies. The resolution of inconsistencies introduces some particular challenges. Some of these challenges are: the particular set of resolution actions that will be used to resolve an inconsistency can be dependent on the cause of the inconsistency; the execution of resolution actions on a model can introduce new inconsistencies. A final step is to investigate whether DLs and their systems offer sufficient support to cope with these challenges. We conclude that the standard reasoning tasks on DL terminological and assertional knowledge are not sufficient, and we propose an additional rule-based approach. This approach allows us to manage the different inconsistency resolution scenarios semi-automatically. This results in the following contribution:

Contribution 5: A rule-based DL inconsistency resolution approach that automatically chains the different possible inconsistency resolutions through the automatic detection of inconsistencies, the proposition of solutions to the user and the execution of the selected solution.

A last issue of research in this dissertation is the application of our ideas in the context of model refactorings. Support for inconsistency management is not only useful in the context of model development, but also in an evolution context. Model refactorings are the design-level equivalent of source code refactorings. Refactorings restructure a model or source code

improving some quality attributes of the software specification and preserving its behaviour. Few works have been presented in the context of behaviour preservation of refactorings. In this dissertation, we observe that there are behaviour preserving properties that correspond to our defined behavioural consistencies. These consistencies are originally defined between a superclass and a subclass in an inheritance hierarchy. We redefine them as behaviour preservation properties between a class and its refactored version. If a certain consistency exists between a superclass and its subclass and a certain behaviour preservation property holds between this superclass, respectively subclass, and its refactored version, then certain properties between the refactored version and the subclass, respectively superclass, can be proven.

Contribution 6: Investigation of the correspondence between consistencies and behaviour preserving properties leading to the proof of behaviour preserving properties between refactored classes in an inheritance hierarchy [VJM04], [VMJ06].

We also show that support for inconsistency resolution can be used in the execution of model refactorings. Refactorings can be executed stepwise [Fow99]. Each step consists of some particular model or source code modifications. In a source code refactoring, after each step, the code is compiled and tested [Fow99]. Many of the source code refactorings can be rephrased into model refactorings. Some of the model modifications constituting a model refactoring can be supported by inconsistency resolution. The "compile and test" step defined in source code refactorings is replaced in the context of model refactorings by the detection and resolution of inconsistencies. We show how our inconsistency resolution approach can be used in the execution of model refactorings and in some cases, allows for the semi-automatic execution of the refactoring.

Contribution 7: Application of our rule-based inconsistency resolution approach to the execution of model refactorings, allowing a semi-automatic execution of these refactorings.

Tool support for inconsistency management is indispensable. Because tool support is not a central issue in this dissertation, we developed a proof-of-concept inconsistency detection and resolution tool, called *RACOoN*. The tool is integrated in Poseidon, a state-of-the-art UML CASE tool, using Poseidon's plug-in mechanism and it uses Racer as DL reasoning engine. Because Poseidon is implemented in Java, our plug-in is also implemented in Java. This proof-of-concept tool support shows how the ideas developed and explained in this dissertation can be implemented. In particular, we show how inconsistencies can be detected and how refactorings can be implemented using our inconsistency resolution approach.

Contribution 8: Development of an inconsistency detection and resolution plug-in for the state-of-the-art CASE tool Poseidon relying on Racer and acting as a proof-of-concept of our ideas.

11.2 Future Work

In this section, some possible extensions and improvements to our work are identified. New research areas related to this work are presented too. These areas are interesting to

pursue, but are outside the scope of this dissertation. We discuss the relevant issues by theme described in this dissertation: issues concerning the UML in Section 11.2.1, concerning model refactorings in Section 11.2.5, concerning management of inconsistencies in Section 11.2.3, concerning tool support in Section 11.2.2 and Section 11.2.4, concerning DLs in Section 11.2.6.

11.2.1 Larger Set of UML Elements

In this dissertation, we restricted ourselves to the basic features of the UML in the context of object-oriented software engineering. As previously stated, the UML is a huge language and therefore other language elements can be taken into account. On the one hand, we believe that our approach can be straightforwardly extended to the language elements of structural diagrams such as component diagrams and composite structure diagrams. On the other hand, some interesting research questions can arise for behavioural diagrams, e.g., for UML activity diagrams. They are the most studied UML diagram after class diagrams, sequence diagrams and state diagrams. Activity modelling is described in [Obj04e] as: *"activity modelling emphasises the sequence and conditions for coordinating lower-level behaviours, rather than which classifiers own those behaviours. These are commonly called control flow and object flow models. The actions coordinated by activity models can be initiated because other actions finish executing, because objects and data become available, or because events occur external to the flow."*. The part of the UML metamodel describing these activity diagrams can be translated using the translation defined in Section 6.1. User-defined activity diagrams can be interpreted as instances of this part of the UML metamodel. Using a DL query language, it will be possible to detect certain inconsistencies. However, the question arises which inconsistencies are important within these diagrams and which inconsistencies can be defined between these diagrams and other kinds of UML diagrams. To be able to answer this question, the different interpretations, i.e., the different possible semantics, of this kind of diagram and their overlap with the other kinds of diagrams need to be determined. Another question is whether DLs are sufficient to represent these semantics and to verify the relevant inconsistencies.

11.2.2 Validation on Large-scale (Industrial) Cases

Until now, we only carried out experiments on small examples. It would be useful to validate our approach on some large-scale industrial cases. This validation would provide us with some empirical data. It would give DL researchers a better insight on the performance of the inference algorithms. The data relevant for the software engineering community would be the number of occurrences of particular inconsistencies in particular applications, the definition of (domain-dependent) inconsistencies that are important to some applications and some companies, which inconsistencies are the most important ones for which kind of application. However, we believe that this validation would not contribute much to fundamental research questions in inconsistency management. One exception is the question how to manage the different inconsistencies and their many resolution possibilities (see Section 11.2.3).

11.2.3 Management of Inconsistencies and Inconsistency Resolutions

In Chapter 4, we introduced possible resolution actions and showed that there are dependencies between the different resolution actions of inconsistencies. In [MTR05] critical pair analysis is used to detect and analyse possible conflicts between refactorings. A similar exercise can be done for the different resolution actions. Critical pair analysis can be used to determine possible dependencies and conflicts between resolution actions.

In Chapter 8, we presented our inconsistency resolution approach. This approach detects the inconsistencies automatically and, based on additional information, proposes resolution actions to the user. This approach contributes to the management of the different detected inconsistencies and their possible resolutions. However, it is still possible that a lot of inconsistencies are detected and for each inconsistency, a lot of resolutions are possible. The question is how to manage all these inconsistency occurrences and their resolutions. Several possibilities can be investigated. One possibility is to use learning techniques where the selection of particular resolution actions by the user in a particular context is learned by the resolution approach. Another possibility is to use predefined resolution alternatives. The user has to define different alternative resolution strategies depending on the context. A constraint system can be used to determine the resolution strategy to be chosen. However, we believe none of both approaches enables the automatic resolution of inconsistencies and we even believe that full automatic inconsistency resolution is not desirable. For example, if a new model element needs to be added, the user has to specify its name. A computer-generated name can be used, but in that case, the user still has to edit the model if he/she does not agree on the name.

11.2.4 Extending and Improving Tool Support

Our tool can be extended by implementing more inconsistencies and more model refactorings.

Improvements can be made to the tool increasing the usability of the tool. Grundy *et al.* [GHM98] claim that a key requirement for supporting inconsistency management is the facilities for developers to configure when and how inconsistencies are detected, monitored, stored, presented and possibly automatically resolved. Section 4.5 introduces such tool support requirements that are briefly recapitulated here:

- It must be possible to create and remove inconsistency detection rules.
- The software developer must be able to decide when to check for inconsistencies and which inconsistencies must be checked.
- It must be possible to add, remove or modify inconsistency resolution rules.
- The ordering and grouping of inconsistency resolution rules must be customisable.
- The resolution of inconsistencies demands a certain amount of user interaction.

The implementation of the first requirement results in an editor that allows the creation and removal of inconsistency detection functions. How the user can edit inconsistency detection queries or functions without DL knowledge must be investigated. In our tool, the user can decide which inconsistencies must be checked and when the inconsistencies must be checked,

fulfilling the second requirement. The interface implementing this functionality should still be enhanced. The ordering and grouping of inconsistency resolution rules also requires an editor. The user interaction demanded by the resolution of inconsistencies is supported by our tool. This is demonstrated in Chapter 10. The ultimate goal of these requirements is to make the tool as usable as possible for the average developer. An average developer is a developer that is not acquainted with DLs, and that is not an expert in inconsistency management. This requires a study of the interface by measuring the interface's impact on the quality of the activities executed by the developer. Empirical studies are necessary.

In Section 6.7, we discussed the relation between OCL and DLs and the translation, if possible, from OCL constraints to DL expressions. We did not take into account this translation in our tool support. As OCL version 2.0 has a metamodel that is integrated in the UML metamodel, it must be possible to straightforwardly implement this translation.

11.2.5 Model Refactorings

Other refactorings than the ones presented in Section 9.5, in particular state machine refactorings, need to be analysed and, if possible, implemented using our rule-based inconsistency resolution approach. By analysing and implementing refactorings, we can determine if other inconsistencies then the classified ones are needed and we will get a better idea on the number of resolution actions and rules that are necessary in the execution of certain model refactorings. For each inconsistency occurring during the execution of a refactoring, a certain amount of resolution actions are presented to the user. As already discussed in Section 11.2.3, management strategies resulting in a more automatic execution of the refactorings, need further investigation.

Another issue is the definition of other behaviour preservation properties. The question also arises whether the fact that a certain behaviour property between a class and its refactored version is required, can be used in the process of executing the refactoring. This is particularly interesting for refactorings on state machines. Consider the refactoring *Merge States* that merges several states to one state. During the execution of this refactoring only a behaviour incompatibility can occur. To be able to support this refactoring in the same way as we support, e.g., *Move Operation*, the required behaviour preservation properties can be checked during refactoring. If these properties are violated, resolution actions for these properties can be executed leading, together with user interaction, to the refactored model.

11.2.6 Extensions to DLs and their Systems

In Section 7.4, the lack of proper feedback when using the reasoning tasks *Tbox* coherence and *Abox* consistency was mentioned as one of the disadvantages of checking inconsistencies through these reasoning tasks. Further research is necessary to cope with this problem. The work done in [SC03] marks a promising start. It reports on reasoning services for the debugging of $\mathcal{ALC}$ terminologies.

In Chapter 8, two kinds of DL rules are introduced. We also argued that current DL systems do not support *Tbox* rules. Further research is necessary to support these rules. However, using current techniques, it is possible to support these rules to a certain extent. A first approach consists of weakening our behavioural inconsistencies in such a way that they can be detected via the metamodel interpretation of the involved UML models. In

this case, it is not possible to take into account the consistency of the specified constraints. Another way is to convert the *Tbox* coherence reasoning task to the instance checking task on *Abox* level by techniques such as *Tbox* internalisation [BCM+03]. *Tbox* internalisation reduces the problem of reasoning with respect to a general *Tbox* to reasoning with a single concept for expressive logics allowing the definition of a universal role [BCM+03]. $\mathcal{SHIQ}$, for example, is a logic allowing the definition of a universal role. The consequent of a *nRQL* rule has as syntax a set of *Abox* assertional axioms. As a consequence no *Tbox* assertions can be added or retracted from a certain *Tbox*. A work-around is to resolve such inconsistencies via the metamodel interpretation of the involved UML models. This can only be achieved if a metamodel for the used constraint language is integrated in the UML metamodel. However, this involves an extra translation step, which involves storing tracking information and introducing a possible performance penalty.

Appendix A

RACER Statements Representing our UML 2.0 Fragment

In this appendix, the RACER statements representing our UML 2.0 fragment of the UML 2.0 metamodel are stated. The relevant elements of the different relevant packages of the UML 2 Superstructure document are translated into RACER statements. For each set of RACER statements, a reference to the corresponding, translated package is included.

```
(delete-all-tboxes)
(in-tbox UML2MetaModelTbox)
;
;The Classes package contains subpackages that deal with the basic
;modeling concepts of UML, and in particular classes and their relationships.
; pg. 25
;
;Subpackage KERNEL
;
;KERNEL - the Root diagram
; pg. 27
(define-primitive-role abstractownedelement :domain element :range element :inverse abstractowner
:transitive t :reflexive f)
(define-primitive-role ownedelement :domain element :range element :inverse owner)
(implies-role ownedelement abstractownedelement)
(implies element
  (at-most 1 owner element))
(define-primitive-role relatedElement :domain relationship :range element)
(define-primitive-role source :domain directedrelationship :range element)
(define-primitive-role target :domain directedrelationship :range element)
;define-concept vervangen door implies in beide statements hieronder
(implies relationship
   (and element
      (at-least 1 relatedelement element)))
(implies directedrelationship
   (and relationship
      (at-least 1 source element)
      (at-least 1 target element)))
;
;KERNEL - the Namespaces diagram
; pg. 31
;(define-primitive-role member :domain namespace :range namedelement)
;member in commentaar omdat het de verzameling (transitief) is van alle ownedmembers
(define-primitive-role ownedmember :domain namespace :range namedelement :inverse ownednamespace
:parent ownedelement)
(define-concrete-domain-attribute name :type string :domain namedelement)
(implies namedelement
```

```
  (and element
       (at-most 1 ownednamespace namespace)))
(implies namespace namedelement)
(define-primitive-role visibility :feature t :domain namedelement :range visibilitykind)
(equivalent visibilitykind (or public private))
(disjoint public private)
;

;KERNEL - the Multiplicities diagram
; pg. 40
(implies multiplicityelement element)
(define-concrete-domain-attribute lower :type integer :domain multiplicityelement)
(define-primitive-role upper :domain multiplicityelement :range (or Unlimitednatural LiteralInteger)
:feature t)
(define-concrete-domain-attribute value :type integer :domain LiteralInteger)
(disjoint Unlimitednatural LiteralInteger)
(define-primitive-role definedtype :domain typedelement :range type)
(implies typedelement
  (and namedelement (at-most 1 definedtype type)))
(implies type namedelement)
(disjoint type typedelement)
;
;KERNEL - the Expressions diagram
;pg. 45
(define-primitive-role operand :domain expression :range valuespecification :parent ownedelement)
(implies expression valuespecification)
(define-concrete-domain-attribute symbol :type string :domain expression)
;
;KERNEL - the Constraints diagram
;pg. 53
(define-primitive-role constrainedelements :domain constraint :range element)
;
;KERNEL - the Instances diagram
;pg. 57
(define-primitive-role owningslot :domain instancespecification :range slot :inverse owningInstance
:parent ownedelement)
(define-primitive-role classifierspec :domain instancespecification :range classifier)
(define-primitive-role specification :domain instancespecification :range valuespecification
:parent ownedelement)
(define-primitive-role instance :domain instancevalue :range instancespecification)
(define-primitive-role slotvalue :domain slot :range valuespecification :parent ownedelement)
(define-primitive-role definingfeature :domain slot :range structuralfeature)
;define-concept vervangen door implies in de onderstaande 3 statements
(implies instancespecification (and namedelement (at-least 1 classifierspec classifier)
(at-most 1 specification valuespecification)))
(implies instancevalue (and valuespecification (exactly 1 instance instancespecification)))
(implies slot (and element (exactly 1 definingfeature structuralfeature)
(exactly 1 owningInstance Instancespecification)))
(disjoint instancespecification instancevalue slot)

;
;KERNEL - the Classifiers diagram
;pg. 61
(define-primitive-role featuremembers :domain classifier :range feature :inverse featuringclassifier
:parent ownedmember)
(implies classifier namespace)
(implies classifier type)
;define-concept vervangen door implies
(implies feature (and namedelement (at-least 1 featuringclassifier classifier)))
(define-primitive-role isgeneralization :domain classifier :range generalization :inverse specific
:parent ownedelement)
(implies-role specific source)
(implies-role specific owner)
(define-primitive-role generalclass :domain generalization :range classifier :parent target)
(implies generalization (and directedrelationship (exactly 1 generalclass classifier)))
(define-primitive-role direct-general :domain classifier :range classifier :parent general)
```

```
(define-primitive-role general :domain classifier :range classifier :transitive t)
(define-primitive-role inheritedmember :domain classifier :range namedelement
:parent ownedmember)
;
;KERNEL - the Features diagram
;pg. 71
(implies structuralfeature (and typedelement feature))
(define-primitive-role hasparameter :domain behaviouralfeature :range parameter :parent ownedmember)
;member verandert in ownedmember
(implies behaviouralfeature (and feature namespace))
(implies parameter (and typedelement namedelement (at-most 1 (inv parameter) behaviouralfeature)))
(disjoint structuralfeature behaviouralfeature)
;
;KERNEL - the Operations diagram
;pg. 76
(define-primitive-role formalparameter :domain behaviouralfeature :range parameter
:parent hasparameter :inverse ownerformalparam)
(define-primitive-role returnresult :domain behaviouralfeature :range parameter
:parent hasparameter :inverse ownerreturnparam)
(implies parameter (and (at-most 1 ownerformalparam behaviouralfeature)
                        (at-most 1 ownerreturnparam behaviouralfeature)))
(implies (and (some formalparameter paramater) (some returnparameter parameter)) bottom)
;
(define-primitive-role typeoperation :domain operation :range type)
(define-primitive-role isabstract :domain (or operation class))
(implies operation (and behaviouralfeature (at-most 1 typeoperation type)))
(define-primitive-role formalparameterop :domain operation :range parameter :inverse owneroperation
 :parent formalparameter)
(define-primitive-role preconditionop :domain operation :range constraint :inverse precontext)
(implies-role preconditionop ownedmember)
(implies-role precontext ownednamespace)
(define-primitive-role postconditionop :domain operation :range constraint :inverse postcontext)
(implies-role postconditionop ownedmember)
(implies-role postcontext ownednamespace)
(implies (and (some precontext operation) (some postcontext operation)) bottom)
(implies constraint (and (at-most 1 postcontext operation) (at-most 1 precontext operation)))
(define-primitive-role redefinedelement :domain redefinableelement :range redefinableelement)
(define-primitive-role redefinedoperation :domain operation :range operation :parent redefinedelement)

;KERNEL - the Classes diagram
;pg. 80
(define-primitive-role allattribute :domain classifier :range property :inverse owningclassifier
:parent featuremembers)
(implies-role redefinedattribute redefinitioncontext)
(define-primitive-role redefinitioncontext :domain redefinableelement :range classifier)
(implies redefinableelement namedelement)
(define-primitive-role ownedattribute :domain class :range property :inverse owningclass
:parents(ownedmember allattribute))
(implies-role owningclass redefinedattribute)
(implies-role owningclass ownednamespace)
(implies-role owningclass featuringclassifier)
(implies property (and structuralfeature (some definedtype (or class primitivetype))))
(define-primitive-role subsettedproperty :domain property :range property)
(define-primitive-role redefinedproperty :domain property :range property
:parent redefinedelement)
(define-primitive-role direct-superclass :domain class :range class :parent general)
(define-primitive-role nestedclassifier :domain class :range classifier :inverse nestingclass
:parent ownedmember)
(implies-role classnestingclass ownednamespace)
(implies-role classnestingclass redefinitioncontext)
(define-primitive-role ownedoperation :domain class :range operation :inverse ownedclass
:parents(featuremembers ownedmember))
(implies-role ownedclass owningnamespace)
(implies-role ownedclass redefinitioncontext)
(implies-role ownedclass featuringclassifier)
(define-primitive-role memberassociation :domain property :range association :inverse memberend)
```

```
(implies-role memberend ownedmember)
;member verandert in ownedmember
(define-primitive-role owningassociation :domain property :range association :inverse ownedend
:parents(memberassociation owningnamespace featuringclassifier))
(define-primitive-role defaultvalue :domain property :range valuespecification :inverse owningproperty
:parent ownedelement)
(define-primitive-role opposite :domain property :range property)
(implies property (and structuralfeature (at-most 1 owningclass class)
(at-most 1 owningclassifier classifier)
(at-most 1 memberassociation association) (at-most 1 owningassociation association)
(at-most 1 defaultvalue valuespecification) (at-most 1 opposite property)))
(implies association (and relationship classifier (at-least 2 memberend property)
(at-least 1 endtype type)))
(define-primitive-role endtype :domain association :range type)
(implies class classifier)
(define-primitive-role aggregation :domain Property :range AggregationKind)
(equivalent AggregationKind (or none shared composite))
(disjoint none shared composite)
(disjoint association class property operation constraint instancespecification)
;
;KERNEL - the Datatypes diagram
;pg. 94
; the datatypes correspond to the concrete domains of Racer
(implies datatype classifier)
(implies primitivetype datatype)
(equivalent complexType (or primitivetype class))
;
;KERNEL - the Packages diagram
;pg. 99
(implies package namespace)

;
;Subpackage COMPOSITE STRUCTURES
;pg. 151
;
;COMPOSITE STRUCTURES - Connectors
;pg. 153
(define-primitive-role connectorends :domain connectableelement :range connectorend
:inverse roleelement)
(define-primitive-role connectortoends :domain connector :range connectorend)
(implies connectorend multiplicityelement)
(implies connector feature)
(define-primitive-role definingend :domain connectorend :range property)
(define-primitive-role associationtype :domain connector :range association)
(implies connectorend (and (at-most 1 roleelement connectableelement)
(at-most 1 definingend property)))
(implies connector (and (at-most 1 associationtype association)
(at-least 2 connectortoends connectorend)))
(define-primitive-role base :domain connector :range classifier :inverse roleofclassifier)
(disjoint connector connectorend operation class property)
;
;The Interaction package describes the concepts needed to express Interactions.
;pg. 404
(define-primitive-role ownedbehavior :domain classifier :range behavior :inverse contextbehavior)
(implies behavior (and class (at-most 1 contextbehavior)))
;pg. 405
(define-primitive-role fragment :domain interaction :range interactionfragment :inverse
enclosinginteraction :parent ownedmember)
(implies interaction (and behavior interactionfragment))
(implies executionoccurrence interactionfragment)
(implies eventoccurrence interactionfragment)
(implies stop eventoccurrence)
(implies interactionfragment (and namedelement (at-most 1 enclosinginteraction interaction)
 (or interaction executionoccurrence eventoccurrence)))
;pg. 406
(define-primitive-role coveredlifeline :domain interaction :range lifeline :inverse ownedinteraction
```

```
:parent ownedmember)
(define-primitive-role coveredsub :domain eventoccurrence :range lifeline :inverse eventoccurrences
:parent covered)
(define-primitive-role covered :domain interactionfragment :range interactionfragment
:inverse coveredby)
;(define-primitive-role represents :domain lifeline :range object)
(define-primitive-role represents :domain lifeline :range connectableelement)
(define-primitive-role instance-of :domain object :range class)
(implies lifeline (and namedelement (exactly 1 ownedinteraction interaction)
 (exactly 1 represents connectableelement)))
(implies eventoccurrence (exactly 1 coveredsub lifeline))
;pg. 407
(define-primitive-role sendevent :domain message :range messageend :inverse sendmessage)
(define-primitive-role receiveevent :domain message :range messageend :inverse receivemessage)
(define-primitive-role argumentv :domain message :range valuespecification :parent ownedelement)
(define-primitive-role signature :domain message :range namedelement)
(define-primitive-role connectorr :domain message :range connector)
(define-primitive-role messager :domain interaction :range message :parent ownedmember)
(define-primitive-role finishexec :domain eventoccurrence :range executionoccurrence :inverse finish)
(define-primitive-role startexec :domain eventoccurrence :range executionoccurrence :inverse start)
(define-primitive-role tbefore :domain message :range message :transitive t :inverse tafter)
(define-primitive-role before :domain message :range message :inverse after)
(implies-role before tbefore)
(implies message (and namedelement (exactly 1 interactionr interaction)
                            (at-most 1 sendevent messageend)
                            (at-most 1 receiveevent messageend)
                            (at-most 1 signature namedelement)
                            (at-most 1 connectorr connector)))
(implies messageend (and namedelement (at-most 1 sendmessage message)
 (at-most 1 receivemessage message)))
(implies valuespecification (at-most 1 argumentv valuespecification))
(implies eventoccurrence (and messageend (at-most 1 before eventoccurrence)
(at-most 1 after eventoccurrence)))
(implies executionoccurrence  (and (exactly 1 finish) (exactly 1 start)))
(disjoint interaction executionoccurrence eventoccurrence)
(disjoint lifeline message eventoccurrence)

;
;The StateMachine package defines a set of concepts that can be used for modelling
;discrete behaviour through finite state-transition systems.
;pg. 455
(implies statemachine behavior)
(implies protocolstatemachine statemachine)
(implies finalstate state)
(implies state vertex)
(implies vertex namedelement)
(implies initialstate vertex)
(define-primitive-role outgoing :domain vertex :range transition :inverse sourcevertex)
(define-primitive-role incoming :domain vertex :range transition :inverse targetvertex)
(define-primitive-role successor :domain vertex :range vertex :transitive t)
(define-primitive-role direct-successor :domain vertex :range vertex :parent successor)
(implies initialstate (at-most 0 incoming transition))
(implies finalstate (at-most 0 outgoing transition))
(implies region namedelement)
(define-primitive-role subvertex :domain region :range vertex :parent ownedelement)
(define-primitive-role transitions :domain region :range transition :parent ownedelement)
(define-primitive-role regions :domain statemachine :range region :parent ownedmember)
(define-primitive-role extendedregion :domain region :range region :transitive t)
(implies transition (and (exactly 1 sourcevertex vertex) (exactly 1 targetvertex vertex)
namedelement))
(implies protocoltransition (and transition (at-most 1 precondition constraint)
(at-most 1 postcondition constraint)))
(define-primitive-role precondition :domain protocoltransition :range constraint :inverse pre)
(define-primitive-role postcondition :domain protocoltransition :range constraint :inverse post)
(implies constraint (and (at-most 1 pre protocoltransition) (at-most 1 post protocoltransition)))
(implies (and (some pre protocoltransition) (some post protocoltransition)) bottom)
```

```
(define-primitive-role referredoperation :domain protocoltransition :range operation)
(disjoint protocolstatemachine protocoltransition state region constraint)
(disjoint initialstate finalstate)

;
;PACKAGE AUXILIARY CONSTRUCTS
;pg. 531
;
;Subpackage models
;pg. 535
(implies model package)
```

Appendix B

nRQL Inconsistency Detection Queries

In this appendix, the *nRQL* queries that defined and detect the inconsistencies specified in the second column of Table 7.2, and that are not yet described in Chapter 7 are described. These queries are defined based on the concepts and roles defined in the *Tbox* shown in Appendix A.

B.1 Dangling Type Reference

An operation has some formal parameters or return parameters that do not belong to a certain model.

$$
\begin{aligned}
&ans(op, p, atype)\\
&\quad \leftarrow \quad operation(op) \wedge formalparameter(op, p) \wedge definedtype(p, atype)\\
&\qquad \wedge\ member(op, amodel) \wedge not(member(atype, amodel))
\end{aligned}
$$

The queries for the return parameters are similar.

$$
\begin{aligned}
&ans(op, r, atype)\\
&\quad \leftarrow \quad operation(op) \wedge returnresult(op, r) \wedge definedtype(r, atype)\\
&\qquad \wedge\ member(op, amodel) \wedge not(member(atype, amodel))
\end{aligned}
$$

B.2 Connector Specification Missing

B.2.1 Classless Connectable Element

The base class(es) of the connectable element are unknown.

$$
\begin{aligned}
&ans(el)\\
&\quad \leftarrow \quad connectableelement(el) \wedge \backslash(has\_known\_successor(el, base))
\end{aligned}
$$

$$ans(el, c, amodel)$$
$$\leftarrow \quad connectableelement(el) \wedge base(el, c) \wedge (not(member(c, amodel)))$$

B.2.2 Dangling Connectable Feature Reference

The operation called in a message is not known by any of the base classes (or by the ancestors of the base classes) of the connectable element.

$$ans(m, op, c)$$
$$\leftarrow \quad message(m) \wedge signature(m, op) \wedge operation(op) \wedge receiveevent(m, mend) \wedge$$
$$coveredsub(mend, lifeline) \wedge lifeline(lifeline) \wedge represents(lifeline, connectableel) \wedge$$
$$base(connectableel, c) \wedge class(c) \wedge (\backslash(has\_known\_successor(ownedoperation, op)) \vee$$
$$((not(ownedoperation(c, op))) \wedge (not(general(superc, c))) \wedge ownedoperation(superc, op)))$$

B.2.3 Dangling Connectable Association Reference

This inconsistency can occur if or the association typing the connector is not an element of the model or there is no association typing the connector or the association typing the connector does not exist between the base classes of the connectable elements involved in the connector.

$$ans(c)$$
$$\leftarrow \quad connector(c) \wedge (\backslash(has\_known\_successor(c, associationtype)) \vee$$
$$(associationtype(c, assoc) \wedge (not(member(assoc, m))) \vee$$
$$\backslash(has\_known\_successor(assoc, member))))$$

$$ans(c, assoc, cl)$$
$$\leftarrow \quad connector(c) \wedge associationtype(c, assoc) \wedge base(c, cl) \wedge class(cl) \wedge$$
$$owningassociation(end, assoc) \wedge (not(definedType(end, cl))) \wedge$$
$$(not(general(supercl, cl))) \wedge definedType(end, supercl)$$

B.3 Instance Specification Missing

Classless Instance

A certain instance is not an instance of a class or it is an instance of a class not known by the model.

$$
\begin{aligned}
&ans(obj)\\
&\quad \leftarrow \quad instancespecification(obj) \wedge (\backslash(has\_known\_successor(obj, classifierspec)) \vee\\
&\qquad (classifierspec(obj, cl) \wedge ((not(member(cl, m))) \vee \backslash(has\_known\_successor(cl, member)))))
\end{aligned}
$$

B.3.1 Classless Protocol State Machine

A PSM is associated to a class not known to the model.

$$
\begin{aligned}
&ans(psm, cl, model)\\
&\quad \leftarrow \quad protocolstatemachine(psm) \wedge contextbehavior(psm, cl) \wedge (not(member(cl, model))))
\end{aligned}
$$

B.3.2 Dangling Feature Reference

The query searches for operations that are not known to the class of the state machine.

$$
\begin{aligned}
&ans(psm, cl, op)\\
&\quad \leftarrow \quad protocolstatemachine(psm) \wedge contextbehavior(psm, cl) \wedge\\
&\qquad regions(psm, r) \wedge transitions(r, t) \wedge protocoltransition(t) \wedge\\
&\qquad referredoperation(t, op) \wedge \ (\backslash(has\_known\_successor(ownedoperation, op)) \ \vee\\
&\qquad ((not(ownedoperation(cl, op))) \wedge (not(general(superc, c))) \wedge ownedoperation(superc, op)))
\end{aligned}
$$

Preconditions are specified on a certain model element, e.g., a transition, and the involved elements in the conditions belong to a class not belonging to the model.

$$
\begin{aligned}
&ans(pt, cl, el, m)\\
&\quad \leftarrow \quad protocoltransition(pt) \wedge precondition(pt, constraint) \wedge\\
&\qquad constrainedelements(constraint, el) \wedge feature(el) \wedge featuringclassifier(el, cl) \wedge\\
&\qquad (not(member(cl, m)))
\end{aligned}
$$

A similar query can be defined for the postconditions specified on a certain model element.

B.3.3 Dangling Association Reference

The association typing the connector is not an element of the model or there is no association typing the connector.

$$
\begin{aligned}
&ans(l, assoc, m)\\
&\quad \leftarrow \quad instancespecification(l) \wedge (\backslash(has\_known\_successor(l, classifierspec)) \ \vee\\
&\qquad classifierspec(l, assoc) \wedge association(assoc) \wedge (not(member(assoc, m))))
\end{aligned}
$$

The association typing the connector does not exist between the classes of the objects connected through the connector.

$$
\begin{aligned}
&ans(c, assoc, cl)\\
&\quad \leftarrow \quad connector(c) \wedge associationtype(c, assoc) \wedge base(c, cl) \wedge class(cl) \wedge\\
&\qquad owningassociation(end, assoc) \wedge (not(definedType(end, cl))) \wedge\\
&\qquad (not(general(supercl, cl))) \wedge definedType(end, supercl)
\end{aligned}
$$

B.4 Disconnected Model

A disconnected PSM is defined and detected by:

$$
\begin{aligned}
&ans(state, psm)\\
&\quad \leftarrow \quad protocolstatemachine(psm) \wedge regions(psm, r) \wedge subvertex(r, initial) \wedge\\
&\qquad initialstate(initial) \wedge subvertex(r, state) \wedge (not(successor(initial, state)))
\end{aligned}
$$

A disconnected sequence diagram is defined and detected by:

$$
\begin{aligned}
&ans(l)\\
&\quad \leftarrow \quad lifeline(l) \wedge coveredsub(mend, l) \wedge messageend(mend) \wedge\\
&\qquad (not((some\ receiveevent\ top)(mend)))
\end{aligned}
$$

B.5 Specification Incompatibility

B.5.1 Multiplicity Incompatibility

In Chapter 7 a query is defined retrieving the lower bound of the multiplicity for a certain association. The following query retrieves the objects connected through this association. Remark that the upper bound of the association is verified by checking the consistency between a *Tbox* and corresponding *Abox*.

$$
\begin{aligned}
&ans(senderobj, objectsrec, assoc, connector)\\
&\quad \leftarrow \quad represents(lifelinesend, objects) \wedge classifierspec(objects, sendclass) \wedge\\
&\qquad eventoccurrences(lifelinesend, sendevents) \wedge receivemessage(sendevents, message) \wedge\\
&\qquad represents(lifelinereceive, objectsrec) \wedge classifierspec(objectsrec, receiveclass) \wedge\\
&\qquad eventoccurrences(lifelinereceive, receiveevents) \wedge sendmessage(receiveevents, message) \wedge\\
&\qquad connectorr(message, connector) \wedge associationtype(connector, assoc)
\end{aligned}
$$

$ans(senderobj, receivingcl, assoc)$
$\leftarrow$ $represents(l, obj1) \wedge connector(c) \wedge connectorends(obj1, end1) \wedge$
$connectortoends(c, end1) \wedge represents(l, obj2) \wedge connectorends(obj2, end2) \wedge connectortoends(c, end2) \wedge$
$associationtype(c, assoc)$

B.5.2 Navigation Incompatibility

Messages are sent over a certain association that is not navigable.

$ans(senderobj, receivingcl, assoc, assocend)$
$\leftarrow$ $represents(lifelinesend, objects) \wedge classifierspec(objects, sendclass) \wedge$
$eventoccurrences(lifelinesend, sendevents) \wedge receivemessage(sendevents, message) \wedge$
$represents(lifelinereceive, objectsrec) \wedge classifierspec(objectsrec, receiveclass) \wedge$
$eventoccurrences(lifelinereceive, receiveevents) \wedge sendmessage(receiveevents, message) \wedge$
$connectorr(message, connector) \wedge associationtype(connector, assoc) \wedge$
$memberEnd(assoc, assocend) \wedge ownedAttribute(receivingcl, assocend) \wedge (not(Navigable(assocend)))$

B.5.3 Abstract Object

An abstract class is instantiated.

$ans(obj, abstractcl)$
$\leftarrow$ $classifierspec(obj, abstractcl) \wedge isAbstract(abstractcl) \wedge (not(general(abstractcl, subclasses))$

Appendix C

Decision Diagrams for Execution of Model Refactorings

In this appendix, the decision diagrams of the 8 model refactorings specified in Table 9.1 are shown. As already stated in Chapter 9, the goal of these decision diagrams is to show that the same inconsistency resolutions reoccur in and across different model refactorings. We stress that these diagrams have been designed manually and are not claimed to be complete. These decision diagrams are modelled as UML 2.0 activity diagrams.

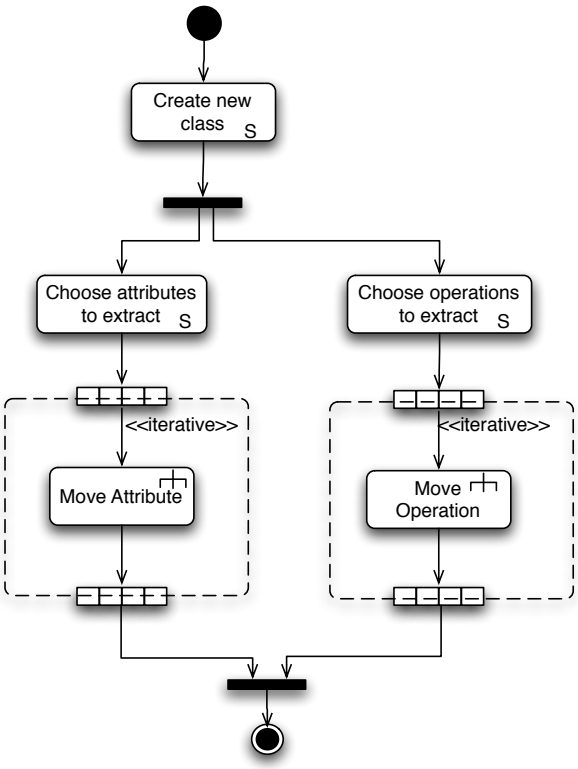

Figure C.1: Decision activities for *Extract Class*.

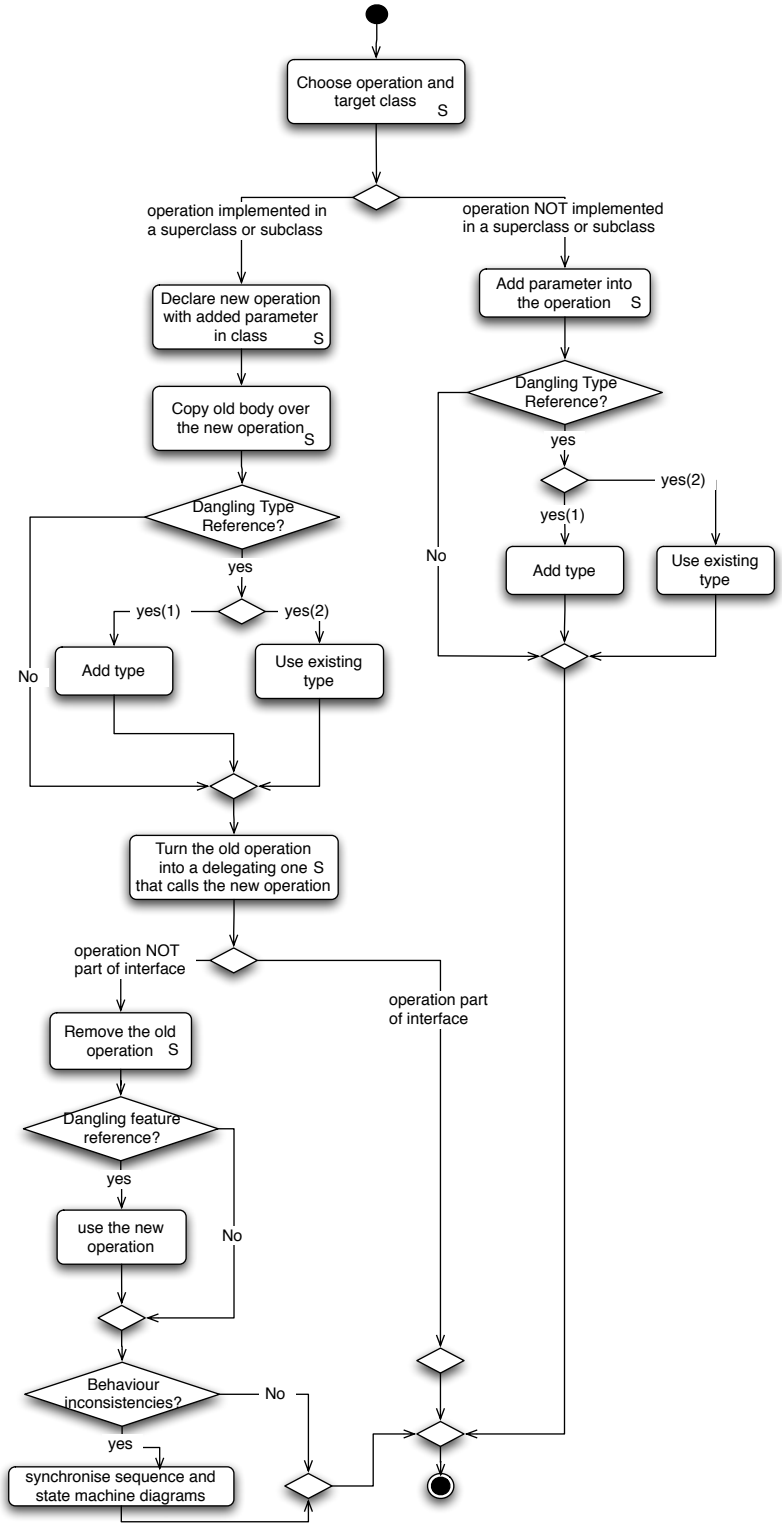

Figure C.2: Decision activities for *Add Parameter*.

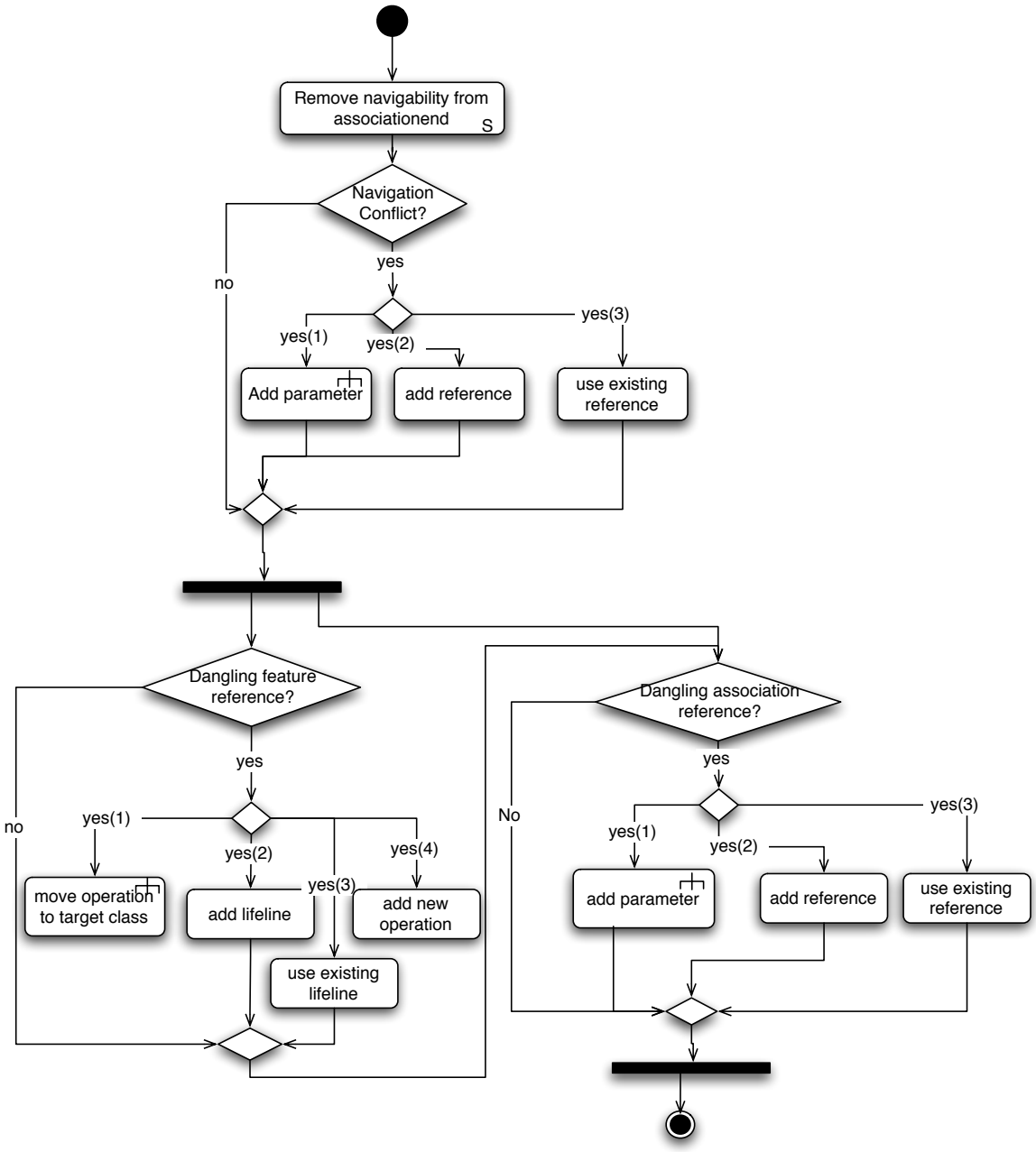

Figure C.3: Decision activities for *Change Bidirectional to Unidirectional Association*.

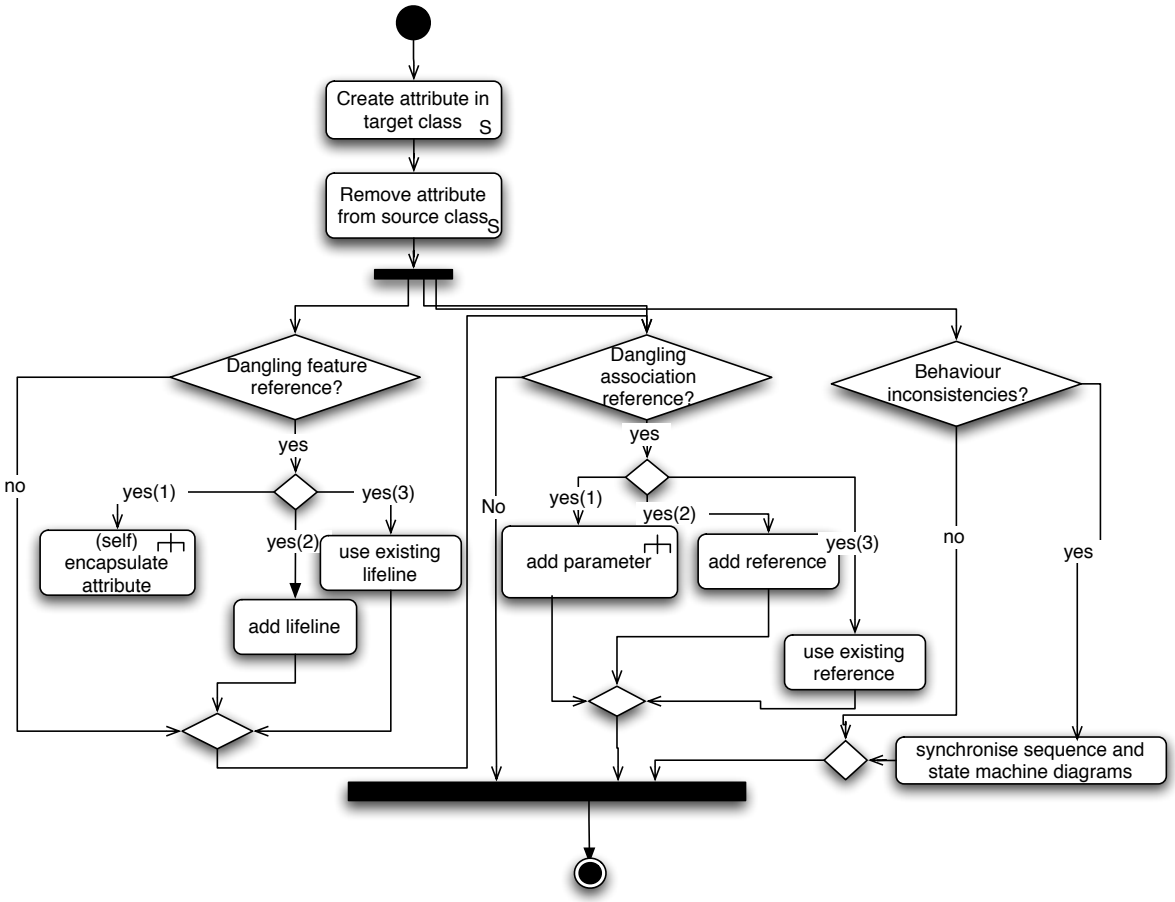

Figure C.4: Decision activities for *Move Attribute*.

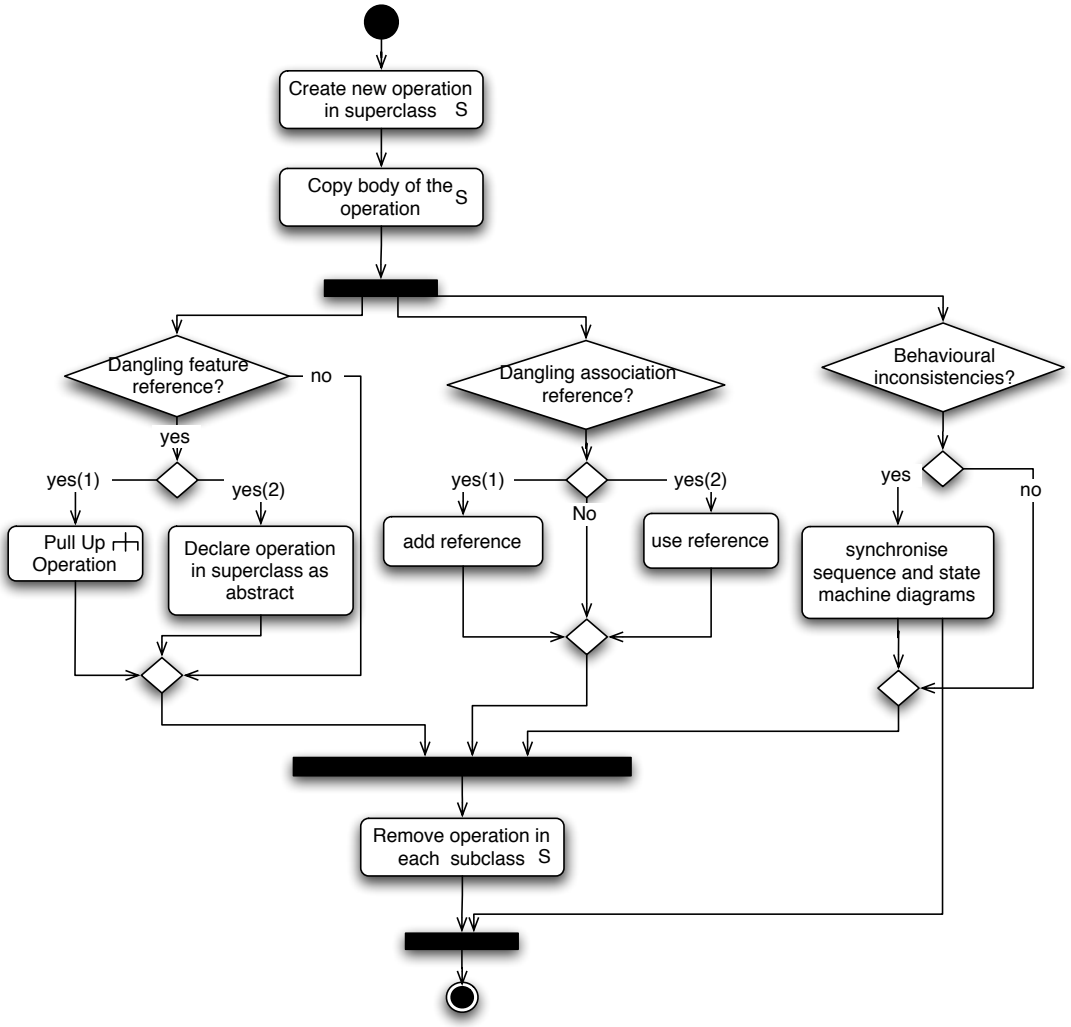

Figure C.5: Decision activities for *Pull Up Operation*.

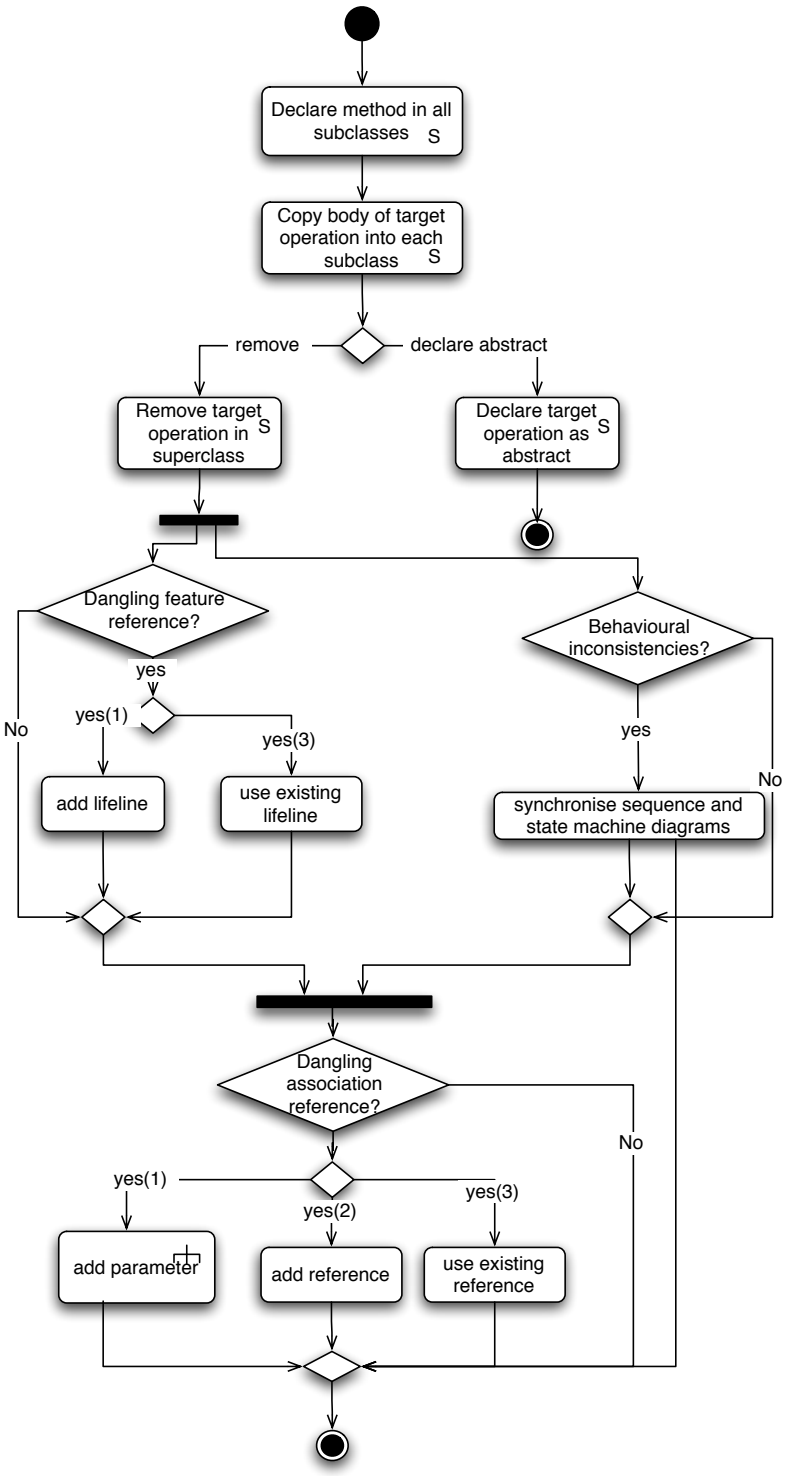

Figure C.6: Decision activities for *Push Down Operation*.

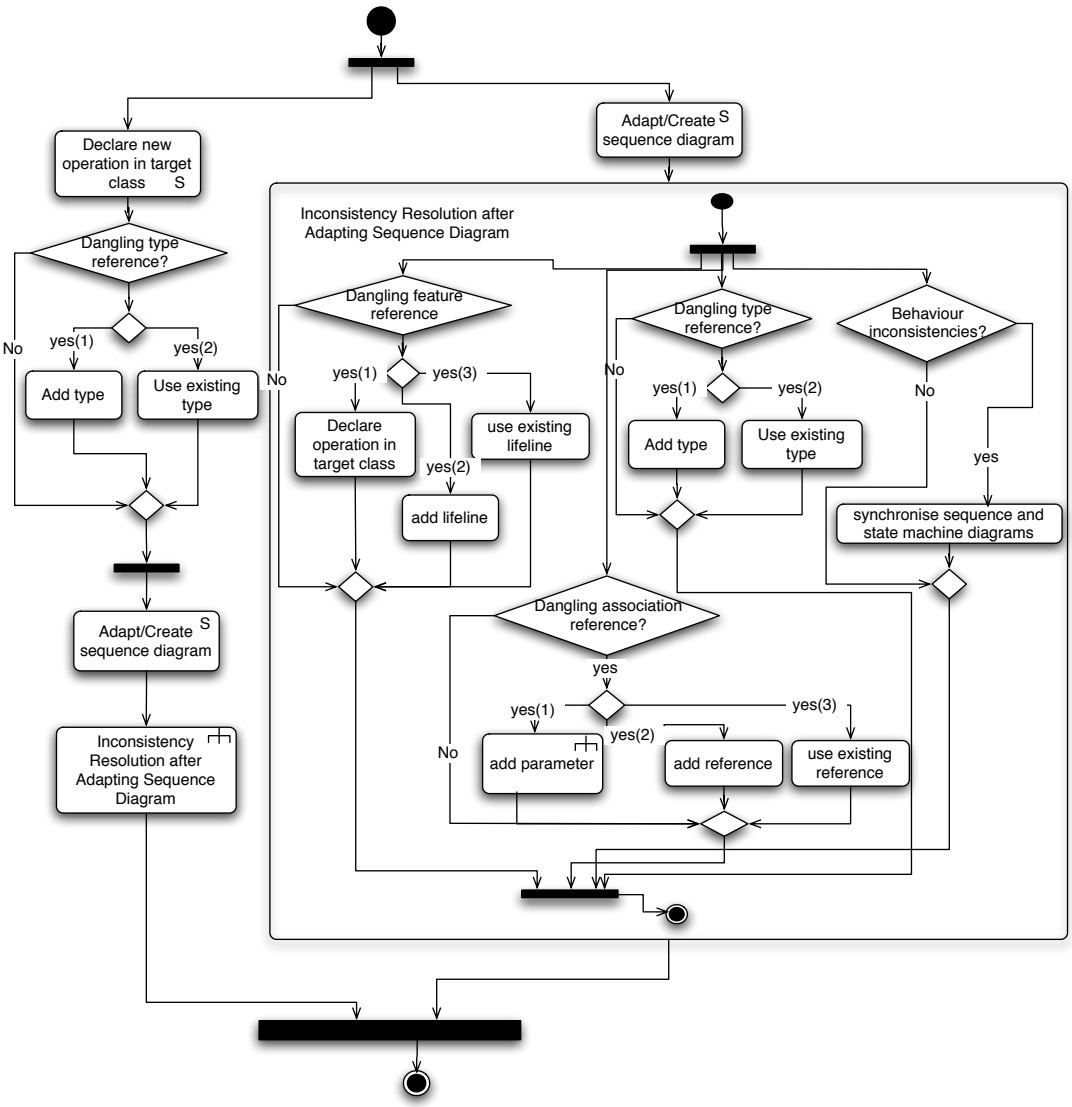

Figure C.7: Decision activities for *Extract Operation*.

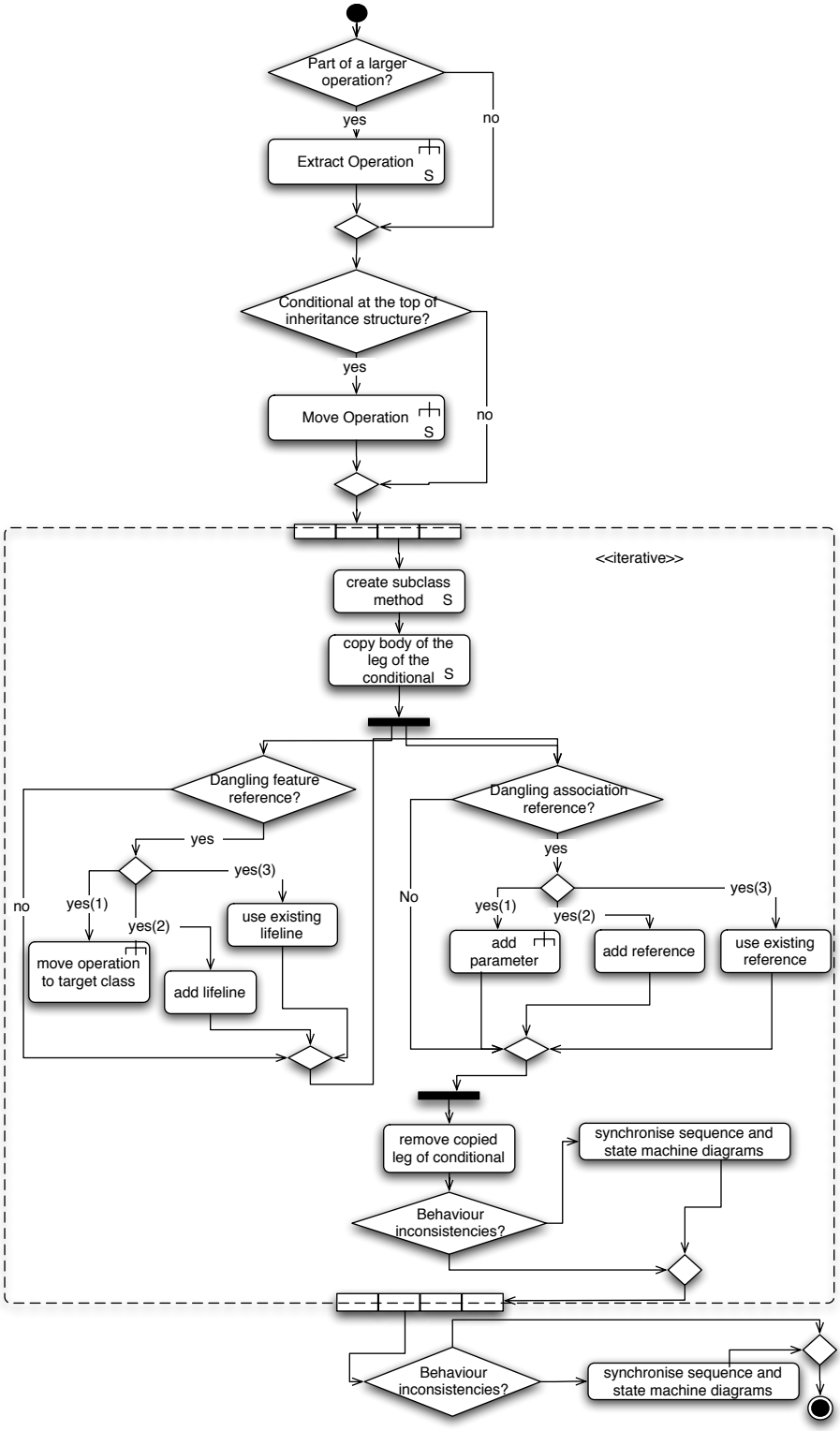

Figure C.8: Decision activities for *Replace Conditional with Polymorphism.*

Bibliography

[AM00] João Araújo and Ana Moreira. Specifying the behaviour of UML collaborations using Object-Z. In Michael Chung, editor, *2000 Americas Conference on Information, Systems (AMCIS)*. AIS, 2000. Long Beach, California, USA. 152

[Ara98] João Araújo. Formalizing sequence diagrams. In Luis Andrade, Ana Moreira, Akash Deshpande, and Stuart Kent, editors, *Proceedings of OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998. Vancouver, Canada. 116

[Are00] Carlos Areces. *Logic Engineering*. PhD thesis, Institute for Logic Language and Computation, University of Amsterdam, 2000. Amsterdam, The Netherlands. 16, 103, 105, 109

[Ast02] Dave Astels. Refactoring with UML. In M. Marchesi and G. Succi, editors, *Proceedings International Conference eXtreme Programming and Flexible Processes in Software Engineering*, pages 67–70, 2002. Alghero, Sardinia, Italy. 207

[Baa91] Franz Baader. Augmenting concept languages by transitive closure of roles: An alternative to terminological cycles. In *Proceedings of twelfth International Joint Conference on Artificial Intelligence (IJCAI1991)*, pages 446–451. Morgan Kaufmann Publishers, August 1991. Sydney, Australia. 105

[BBH93] Franz Baader, Martin Buchheit, and Bernhard Hollunder. Cardinality restrictions on concepts. DFKI Research Report RR-93-48, Deutsches Forschungszentrum für Künstliche Intelligenz, 1993. Kaiserslautern, Germany. 149

[BCM+03] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003. 15, 103, 108, 110, 113, 115, 116, 119, 121, 178, 230

[BdV01] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, 2001. 105, 116

[BE89] Alex Borgida and David W. Etherington. Hierarchical knowledge bases and efficient disjunctive reasoning. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *Proceedings of first International Conference on Principles of Knowledge Representation and Reasoning (KR1989)*, pages 33–43. Morgan Kaufmann Publishers, May 1989. Toronto, Canada. 118

[Ber02] Daniela Berardi. Using DLs to reason on UML class diagrams. In Günther Görz, Volker Haarslev, Carsten Lutz, and Ralf Möller, editors, *Proceedings of International Workshop on Applications of Description Logics (ADL2002)*, volume 63 of *CEUR Workshop Proceedings*, pages 1–11, 2002. Aachen, Germany. 128, 138, 154

[BH91] Franz Baader and Philipp Hanschke. A scheme for integrating concrete domains into concept languages. DFKI Research Report RR-91-10, Deutsches Forschungszentrum für Künstliche Intelligenz, 1991. Kaiserslautern, Germany. 116

[BHGS01] Sean Bechhofer, Ian Horrocks, Carole Goble, and Robert Stevens. OilEd: a Reason-able Ontology Editor for the Semantic Web. In Franz Baader, Gerhard Brewka, and Thomas Eiter, editors, *Proceedings of Advances in Artificial Intelligence, Joint German/Austrian Conference on Artificial Intelligence*, volume 2174 of *Lecture Notes in Computer Science*, pages 396–408. Springer, September 2001. Vienna. 122

[BHS05] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description Logics as ontology languages for the semantic web. In Dieter Hutter and Werner Stephan, editors, *Festschrift in honor of Jörg Siekmann*, volume 2605 of *Lecture Notes in Artificial Intelligence*. Springer, 2005. 16, 103

[Bjo04] Russell C. Bjork. ATM Simulation. www.math-cs.gordon.edu/local/courses/cs211/ATMExample/, December 2004. 19

[BKBM99] Franz Baader, Ralf Küsters, Alexander Borgida, and Deborah L. McGuinness. Matching in Description Logics. *Journal of Logic and Computation*, 9(3):411–447, 1999. 181

[BL84] Ronald Brachman and Hector Levesque. The tractability of nsubsumption in frame-based description languages. In Ronald Brachman, editor, *Proceedings of National Conference on Artificial Intelligence*, pages 34–37. AAAI Press, August 1984. 105

[BMP+02] Jean-Paul Bodeveix, Thierry Millan, Christian Percebois, Christophe Le Camus, Pierre Bazes, and Louis Ferraud. Extending OCL for verifying UML model consistency. In Ludwig Kuzniarz, Gianna Reggio, Jean Louis Sourrouille, and Zbigniew Huzar, editors, *Blekinge Institute of Technology, Research Report 2002:06. UML 2002 Workshop on Consistency Problems in UML-Based Software Development. Workshop Materials*, 2002. 105

[Bra77] Ronald J. Brachman. What's in a concept: Structural foundations for semantic networks. *International Journal of Man-Machine Studies*, 9(2):127–152, 1977. 117

[Bri93] David Brill. *Loom Reference Manual*. University of Southern California, version 2.0 edition, December 1993. 120

[BSF02] Marko Boger, Thorsten Sturm, and Per Fragemann. Refactoring browser for UML. In M. Marchesi and G. Succi, editors, *Proceedings International Conference*

eXtreme Programming and Flexible Processes in Software Engineering, pages 77–81, 2002. Alghero, Sardinia, Italy. 189, 207

[CC90] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990. 8

[CCDL01] Andrea Calí, Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Reasoning on UML class diagrams in Description Logics. In Bernhard Beckert, Robert France, Reiner Hähnle, and Bart Jacobs, editors, *Proceedings of IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development (PMD2001)*, 2001. 104

[CGL98] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. On the decidability of query containment under constraints. In *Proceedings of seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 149–158. ACM Press, June 1-3 1998. Seattle, Washington. 138

[CGLN01] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Daniele Nardi. Reasoning in expressive Description Logics. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1581–1634. Elsevier Science Publishers and MIT Press, 2001. 115

[CT04] Jordi Cabot and Ernest Teniente. Determining the structural events that may violate an integrity constraint. In Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors, *Proceedings seventh International Conference UML 2004*, volume 3273 of *Lecture Notes in Computer Science*, pages 320–334. Springer, October 2004. Lisbon, Portugal. 148, 149

[Eas91] Steve Easterbrook. Handling conflict between domain descriptions with computer-supported negotiation. *Knowledge Acquisition*, 3(3):255–289, 1991. 98, 184

[EE95] Jürgen Ebert and Gregor Engels. Specialization of object life cycle definitions. Fachbericht Informatik 19/95, Universität Koblenz-Landau, Fachbereich Informatik, 1995. Koblenz. 58, 72, 85, 95, 99, 193

[EFA+99] Wolfgang Emmerich, Anthony Finkelstein, Stefano Antonelli, Stephen Armitage, and Richard Stevens. Managing standards compliance. *IEEE Transactions on Software Engineering*, 25(6):836–851, 1999. 104

[Egy01] Alexander Egyed. Scalable consistency checking between diagrams-the viewintegra approach. In M. Feather and M. Goedicke, editors, *Proceedings of sixteenth IEEE International Conference on Automated Software Engineering (ASE2001)*, pages 387–390. IEEE Computer Society, November 2001. Coronado Island, San Diego, CA, USA. 170

[EHHS02] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Testing the consistency of dynamic UML diagrams. In *Proceedings sixth biennal world conference on Integrated Design and Process Technology (IDPT 2002)*. Society for Design and Process Science, June 2002. Pasadena, CA, USA. 3, 26, 98

[EHK01] Gregor Engels, Reiko Heckel, and Jochen Malte Küster. Rule-based specification of behavioral consistency based on the UML meta-model. In Martin Gogolla and Cris Kobryn, editors, *Proceedings fourth International Conference UML 2001*, number 2185 in Lecture Notes in Computer Science, pages 272–286. Springer, October 2001. Toronto, Canada. 3, 80, 81, 151

[EHKG02] Gregor Engels, Reiko Heckel, Jochen Malte Küster, and Luuk Groenewegen. Consistency-preserving model evolution through transformations. In Jean-Marc Jézéquel, Heinrich Hußmann, and Stephen Cook, editors, *Proceedings fifth International Conference UML 2002*, number 2460 in Lecture Notes in Computer Science, pages 212–227. Springer, October 2002. Dresden, Germany. 3, 99, 207

[EKHG01] Gregor Engels, Jochen Malte Küster, Reiko Heckel, and Luuk Groenewegen. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In *Proceedings of eighth European Software Engineering Conference held jointly with nineth ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE2001)*, pages 186–195. ACM Press, September 2001. Vienna, Austria. 12, 58

[EKHG02] Gregor Engels, Jochen Malte Küster, Reiko Heckel, and Luuk Groenewegen. Towards consistency-preserving model evolution. In Katsuro Inoue, Václav Rajlich, and Mikio Aoyama, editors, *Proceedings International Workshop on Principles of Software Evolution*, pages 129–132. ACM Press, 2002. Orlando, Florida. 12

[ET00] Hartmut Ehrig and Aliki Tsiolakis. Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In Hartmut Ehrig and Gabriele Taentzer, editors, *Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems*, pages 77–86, March 2000. Berlin, Germany. 3, 26, 58, 84, 170

[FGH+93] Anthony Finkelstein, Dov M. Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multi-perspective specifications. In Ian Sommerville and Manfred Paul, editors, *Proceedings of fourth European Software Engineering Conference (ESEC1993)*, volume 717 of *Lecture Notes in Computer Science*, pages 84–99. Springer, September 1993. Garmisch-Partenkirchen, Germany. 104, 169, 184

[Fin00] Anthony Finkelstein. A foolish consistency: Technical challenges in consistency management. In I. Ibrahim, J. Küng, and N. Revell, editors, *Proceedings International Conference Database and Expert Systems Applications (DEXA2000)*, volume 1873 of *Lecture Notes in Computer Science*, pages 1–5. Springer, September 2000. London, UK. 3, 88

[FL79] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Science*, 18(2):194–211, 1979. 105, 115

[FMP99] Pascal Fradet, Daniel Le Métayer, and Michaël Périn. Consistency checking for multiple view software architectures. In Oscar Nierstrasz and M. Lemoine, editors, *Proceedings seventh European Software Engineering Conference, held jointly*

with the seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering, volume 1687 of *Lecture Notes in Computer Science*, pages 410–428. Springer, September 1999. Toulouse, France. 3, 170

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999. 2, 7, 8, 189, 197, 199, 226

[Fow03] Martin Fowler. *UML Distilled, A Brief Guide to the Standard Object Modeling Language, Third Edition*. Addison-Wesley, 2003. 24

[FST96] Anthony Finkelstein, George Spanoudakis, and David Till. Managing interference. In *Joint proceedings of second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, pages 172–174. ACM Press, 1996. New York, NY, USA. 10

[gen05] gentleware.com. Poseidon. http://www.gentleware.com/, April 8 2005. 14, 207, 209

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems*. Addison-Wesley, 1994. 89

[GHM98] John C. Grundy, John G. Hosking, and Warwick B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engineering*, 24(11):960–981, 1998. 99, 228

[GLF03] Gonzola Genova, Juan Llorens, and Jose M. Fuentes. The baseless links problem. In Ludwik Kuzniarz, Zbigniew Huzar, Gianna Reggio, Jean Louis Sourrouille, and Miroslaw Staron, editors, *Blekinge Institute of Technology, Research Report 2003:06. UML 2003 Workshop on Consistency Problems in UML-Based Software Development II. Workshop Materials*. Department of Software Engineering and Computer Science, Blekinge Institute of Technology, 2003. San Francisco, CA, USA. 62

[Gli04] Birte Glimm. A query language for web ontologies. Technical report, Hamburg University of Applied Sciences, June 2004. 162

[HDF00] Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting OCL. In Andy Evans, Stuart Kent, and Bran Selic, editors, *Proceedings third International Conference UML 2000*, volume 1939 of *Lecture Notes in Computer Science*, pages 278–293. Springer, October 2000. York, UK. 23

[HHS02] Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Extended model relations with graphical consistency conditions. In Ludwik Kuzniarz, Gianna Reggio, Jean Louis Sourrouille, and Zbigniew Huzar, editors, *Blekinge Institute of Technology, Research Report 2002:06. UML 2002 Workshop on Consistency Problems in UML-Based Software Development. Workshop Materials*, pages 61–74. Department of Software Engineering and Computer Science, Blekinge Institute of Technology, 2002. 3, 98

[HK99] David Harel and Orna Kupferman. On the inheritance of state-based object behavior. Technical report mcs99-12, Weizmann Institute of Science, Faculty of Mathematics and Computer Science, 1999. 72

[HKS01] Jan Hendrik Hausmann, Jochen Malte Küster, and Stefan Sauer. Identifying semantic dimensions of (UML) sequence diagrams. In Andy Evans, Robert France, Ana Moreira, and Bernhard Rumpe, editors, *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists. UML2001 Workshop of the pUML-Group*, volume P-7 of *Lecture Notes in Informatics*, pages 142–157. German Informatics Society, October 2001. Toronto, Canada. 136, 137

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000. 116

[HLM99] Volker Haarslev, Carsten Lutz, and Ralf Möller. A Description Logic with concrete domains and role-forming predicates. *Journal of Logic and Computation*, 9(3):351–384, 1999. 113, 114, 118, 122

[HM00] Volker Haarslev and Ralf Möller. Expressive abox reasoning with number restrictions, role hierarchies, and transitively closed roles. In A.G. Cohn, F. Giunchiglia, and B. Selman, editors, *Proceedings of seventh International Conference on Knowledge Representation and Reasoning (KR2000)*, pages 273–284. Morgan Kaufmann Publishers, April 2000. Breckenridge, COL, USA. 114

[HM01] Volker Haarslev and Ralf Möller. RACER system description. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Proceedings Automated Reasoning, First International Joint Conference, (IJCAR2001)*, volume 2083 of *Lecture Notes in Computer Science*, pages 701–706. Springer, June 2001. Siena, Italy. 118, 122

[HM03] Volker Haarslev and Ralf Möller. *RACER User's Guide and Reference Manual*, version 1.7 edition, March 2003. 122

[HMSW04] Volker Haarslev, Ralf Möller, Ragnhild Van Der Straeten, and Michael Wessel. Extended query facilities for racer and an application to software-engineering problems. In Volker Haarslev and Ralf Möller, editors, *Proceedings of 2004 International Workshop on Description Logics (DL2004), Whistler, British Columbia, Canada, June 6-8, 2004*, volume 104 of *CEUR Workshop Proceedings*, 2004. 14, 161, 162, 225

[HMW04] Volker Haarslev, Ralf Möller, and Michael Wessel. *RACER User's Guide and Reference Manual*, version 1.7.19 edition, April 2004. 130, 153

[Hor99] Ian Horrocks. FaCT and iFaCT. In Patrick Lambrix, Alexander Borgida, Maurizio Lenzerini, Ralf Möller, and Peter F. Patel-Schneider, editors, *Proceedings of 1999 International Workshop on Description Logics (DL1999)*, volume 22 of *CEUR Workshop Proceedings*, pages 133–135, July 1999. Linköping, Sweden. 118, 121

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science. Modelling and Reasoning about Systems*. Cambridge University Press, 2004. 152

[HS99] Ian Horrocks and Ulrike Sattler. A Description Logic with transitive and inverse roles and role hierarchies. *Journal of Logic and Computation*, 9(3):385–410, 1999. 114

[HS03] Ian Horrocks and Ulrike Sattler. Decidability of SHIQ with complex role inclusion axioms. In Georg Gottlob and Toby Walsh, editors, *Proceedings of eighteenth International Joint Conference on Artificial Intelligence (IJCAI2003)*. Morgan Kaufmann Publishers, August 2003. Acapulco, Mexico. 112, 116

[HST99] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for expressive Description Logics. In Harald Ganzinger, David McAllester, and Andrei Voronkov, editors, *Proceedings sixth International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, number 1705 in Lecture Notes in Artificial Intelligence, pages 161–180. Springer, September 1999. 115

[IBM04] IBM. The Eclipse Project. www.eclipse.org/downloads/, June 2004. 197

[Jac86] Peter Jackson. *Introduction to Expert Systems*. Addison-Wesley, 1986. 182

[KÖ4] Jochen M. Küster. *Consistency Management of Object-Oriented Behavioral Models*. PhD thesis, University of Paderborn, March 2004. Paderborn, Germany. 85, 99

[KB01] Torger Kielland and Jon Arvid Borretzen. UML consistency checking. Research Report SIF8094, Institutt for datateknikk og informasjonsvitenskap, Oslo, Norway, 2001. 85, 170

[Kna99] Alexander Knapp. A formal semantics for UML interactions. In Robert France and Bernhard Rumpe, editors, *Proceedings second International Conference UML 1999*, volume 1723 of *Lecture Notes in Computer Science*, pages 116–130. Springer, October 1999. Fort Collins, CO, USA. 152

[KRSH02] Ludwig Kuzniarz, Gianna Reggio, Jean Louis Sourrouille, and Zbigniew Huzar. Consistency problems in UML-based software development: Workshop materials. Research Report 2002-06, 2002. Available at `http://www.ipdt.bth.se/uml2002/RR-2002-06-.pdf`, Department of Software Engineering and Computer Science, Blekinge Institute of Technology, October 2002. 11, 12, 58

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model-Driven Architecture: Practice and Promise*. Addison-Wesley, 2003. 7

[KZ04] Alexander Kozlenkov and Andrea Zisman. Discovering, recording, and handling inconsistencies in software specifications. *International Journal of Computer and Information Science*, 5(2), June 2004. 3, 99, 185

[Lan03] Christian F.J. Lange. Empirical investigations in software architecture completeness. Master's thesis, Technische Universiteit Eindhoven, September 2003. Eindhoven, The Netherlands. 58, 170

[LCM+03] Christian F.J. Lange, Michel R.V. Chaudron, Johan Muskens, L.J. Somers, and H.M. Dortmans. An empirical investigation in quantifying inconsistency and incompleteness of UML designs. In Ludwik Kuzniarz, Zbigniew Huzar, Gianna Reggio, Jean Louis Sourrouille, and Miroslaw Staron, editors, *Blekinge Institute of Technology, Research Report 2003:06. UML 2003 Workshop on Consistency Problems in UML-Based Software Development II. Workshop Materials*. Department of Software Engineering and Computer Science, Blekinge Institute of Technology, 2003. 26, 58, 85, 170, 224

[LL99] Xuandong Li and Johan Lilius. Timing analysis of UML sequence diagrams. In Robert France and Bernhard Rumpe, editors, *Proceedings second International Conference UML 1999*, volume 1723, pages 661–674. Springer, October 1999. Fort Collins, CO, USA. 26

[LMM99] Diego Latella, István Majzik, and Mieke Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999. 151

[LRW+97] Meir M. Lehman, Juan F. Ramil, P.D. Wernick, Dewayne E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *Proceedings fourth IEEE International Software Metrics Symposium (METRICS 1997)*, pages 20–32. IEEE Computer Society Press, 1997. 2, 8

[Lut01] Carsten Lutz. NExpTime-complete Description Logics with concrete domains. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Proceedings of International Joint Conference on Automated Reasoning (IJCAR2001)*, number 2083 in Lecture Notes in Artifical Intelligence, pages 45–60. Springer, 2001. Siena, Italy. 116

[Mac91] Robert MacGregor. Inside the LOOM description classifier. *SIGART Bulletin*, 2(3):88–92, 1991. 118, 120

[MCF03] Stephen Mellor, Anthony Clark, and Takao Futagami. Guest editor's introduction: Model-driven development. *IEEE Software*, 20(5):14–18, September/October 2003. 5

[MCV05] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. A taxonomy of model transformations. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. 7

[MD94] Jonathan P. Munson and Prasun Dewan. A flexible object merging framework. In John B. Smith, F. Don Smith, and Thomas W. Malone, editors, *CSCW '94: Proceedings of 1994 ACM conference on Computer supported cooperative work*, pages 231–242. ACM Press, 1994. New York, NY, USA. 92

[MDJ02] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kre-

owski, and Grzegorz Rozenberg, editors, *Proceedings of first International Conference on Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer, October 2002. Barcelona, Spain. 9, 207

[Men99] Tom Mens. *Formal Foundation for Object-Oriented Software Evolution*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, 1999. Brussel, Belgium. 58

[Men02] Tom Mens. A state-of-the-art survey on software merging. *IEEE Transactions Software Engineering*, 28(5):449–462, 2002. 92

[MT04] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004. 2, 8, 9, 197

[MTR05] Tom Mens, Gabriele Taentzer, and Olga Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113–128, 2005. 228

[MVS05] Tom Mens, Ragnhild Van Der Straeten, and Jocelyn Simmonds. A framework for managing consisency of evolving UML models. In H. Yang, editor, *Software Evolution with UML and XML*, chapter 1. Idea Group Inc., 2005. 14, 58, 224

[Neb91] Bernhard Nebel. Terminological cycles: Semantics and computational properties. In J. F. Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, pages 331–361. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1991. chapter 11. 104

[NER00] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Leveraging inconsistency in software development. *IEEE Computer*, 33(4):24–29, 2000. 10, 11, 88

[NJJ+96] Hans Nissen, Manfred Jeusfeld, Matthias Jarke, Georg Zemanek, and Harald Guber. Managing multiple requirements perspectives with metamodels. *IEEE Software*, 13(2):37–47, March 1996. 104

[NKF94] Bashar Nuseibeh, Jef Kramer, and Anthony Finkelstein. A framework for expressing the relationship between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, 1994. 104

[Obj04a] Object Management Group. Unified Modeling Language 2.0. http://www.omg.org/, December 2004. 19

[Obj04b] Object Management Group. Unified Modeling Language 2.0 Diagram Interchange Final Adopted Specification. http://www.omg.org/cgi-bin/doc?ptc/03-09-01, December 2004. 23

[Obj04c] Object Management Group. Unified Modeling Language 2.0 Infrastructure Final Adopted Specification. http://www.omg.org/cgi-bin/doc?ptc/03-09-15, December 2004. 22, 24

[Obj04d] Object Management Group. Unified Modeling Language 2.0 OCL Final Adopted Specification. http://www.omg.org/cgi-bin/doc?ptc/03-10-14, December 2004. 21, 22, 147, 149

[Obj04e] Object Management Group. Unified Modeling Language 2.0 Superstructure Final Adopted Specification. http://www.omg.org/cgi-bin/doc?ptc/03-08-02, February 2004. 22, 31, 37, 42, 46, 50, 72, 146, 227

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992. Urbana-Champaign, IL, USA. 2, 8, 9

[Por03] Ivan Porres. Model refactorings as rule-based update transformations. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *Proceedings sixth International Conference UML 2003*, volume 2863 of *Lecture Notes in Computer Science*, pages 159–174. Springer, October 2003. San Francisco, CA, USA. 206, 207

[RBB+95] Lori Alperin Resnick, Alex Borgida, Ronald J. Brachman, Deborah L. McGuinness, Peter F. Patel-Schneider, and Kevin C. Zalondek. *CLASSIC Description and Reference Manual for the Common Lisp implementation*. AT&T Bell Labs, Murray Hill, NY, USA, December 1995. Version 2.3. 118

[Rob97] Don Roberts. *Pratical Analysis for Refactoring*. PhD thesis, University of Illinois, 1997. Urbana-Champaign, IL, USA. 9

[Sat96] Ulrike Sattler. A concept language extended with different kinds of transitive roles. In G. Görz and S. Hölldobler, editors, *20. Deutsche Jahrestagung für Künstliche Intelligenz*, number 1137 in Lecture Notes in Artificial Intelligence. Springer, 1996. 115

[Sat03] Ulrike Sattler. Description Logics for ontologies. In Aldo de Moor, Wilfried Lex, and Bernhard Ganter, editors, *Proceedings of eleventh International Conference on Conceptual Structures (ICCS2003)*, volume 2746 of *Lecture Notes in Computer Science*. Springer, July 21-25 2003. Dresden, Germany. 16, 103, 106, 107, 110, 112, 115, 130

[SB05] Jocelyn Simmonds and Maria Cecilia Bastarrica. Description Logics for consistency checking of architectural features in UML 2.0 models. DCC Technical Report TR/DCC-2005-1, Departamento de Ciencias de la Computacion, Santiago, Chile, 2005. 210

[SC03] Stefan Schlobach and Ronald Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In Georg Gottlob and Toby Walsh, editors, *Proceedings of eighteenth International Joint Conference on Artificial Intelligence (IJCAI2003)*, pages 355–362. Morgan Kaufmann Publishers, August 2003. Acapulco, Mexico. 229

[Sch91] Klaus Schild. A correspondence theory for terminological logics: Preliminary report. In *Proceedings of twelfth International Joint Conference on Artificial Intelligence (IJCAI1991)*, pages 466–471. Morgan Kaufmann Publishers, August 1991. Sydney, Australia. 115, 117

[SF97] George Spanoudakis and Anthony Finkelstein. Reconciling requirements: a method for managing interference, inconsistency and conflict. *Annals of Software Engineering*, 3:433–457, 1997. 98, 184

[Sim03] Jocelyn Simmonds. Consistency maintenance of UML models with Description Logics. Master's thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium and Ecole des Mines de Nantes, France, September 2003. 157, 161

[SK03] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, September/October 2003. 1

[SKM01] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3), 2001. 3, 151

[SPLJ01] Gerson Sunyé, Damien Pollet, Yves LeTraon, and Jean-Marc Jézéquel. Refactoring UML models. In Martin Gogolla and Cris Kobryn, editors, *Proceedings fourth International Conference UML 2001*, volume 2185 of *Lecture Notes in Computer Science*, pages 134–138. Springer, October 2001. Toronto, Canada. 197, 207

[SS89] Manfred Schmidt-Schauß. Subsumption in KL-ONE is undecidable. In Ronald Brachman, Hector Levesque, and Raymond Reiter, editors, *Proceedings of first International Conference on Principles of Knowledge Representation and Reasoning (KR1989)*, pages 421–431. Morgan Kaufmann Publishers, May 1989. Toronto, Canada. 112

[SS00] Markus Stumptner and Michael Schrefl. Behavior consistent inheritance in UML. In Alberto H. F. Laender et al. editor, *Proceedings of nineteenth International Conference on Conceptual Modeling (ER2000)*, volume 1920 of *Lecture Notes in Computer Science*, pages 527–542. Springer, October 2000. Salt Lake City, Utah, USA. 48, 72, 85

[SS02] Michael Schrefl and Markus Stumptner. Behavior consistent specialization of object life cycles. *ACM Transactions on Software Engineering and Methodology*, 11(1):92–148, January 2002. 72, 85, 99

[SSS91] Manfred Schmidt-Schauss and Gert Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991. 106, 115

[ST03] Perdita Stevens and Jennifer Tenzer. Modelling recursive calls with UML state diagrams. In Mauro Pezzé, editor, *Proceedings sixth International Conference Fundamental Approaches to Software Engineering (FASE2003), held as part of the Joint European Conferences on Theory and Practice of Software (ETAPS2003)*, volume 2621 of *Lecture Notes in Computer Science*, pages 135–149. Springer, April 2003. Warsaw, Poland. 48

[SVJM04] Jocelyn Simmonds, Ragnhild Van Der Straeten, Viviane Jonckers, and Tom Mens. Maintaining consistency between UML models using Description Logic. *L'Objet logiciel, bases de données, réseaux*, 11(1-2):231–244, March 2004. Lille, France. 14, 58, 210, 224

[SZ01] George Spanoudakis and Andrea Zisman. Inconsistency management in software engineering: Survey and open research issues. In Chang S. K., editor, *Handbook of Software Engineering and Knowledge Engineering*, volume 1, pages 329–380. World Scientific Publishing Co., 2001. 2, 10, 11, 53, 87, 88, 104, 169

[Tob01] Stephan Tobies. *Complexity Results and Practical algorithms for Logics in Knowledge Representation.* PhD thesis, RWTH-Aachen, 2001. Aachen, Germany. 115

[Tsi01] Aliki Tsiolakis. Semantic analysis and consistency checking of UML sequence diagrams. Master's thesis, Technische Universität Berlin, April 2001. Technical Report No. 2001-06. 3, 170

[Uni04a] Universitatea Babes-Bolyai, Cluj - Napoca, Romania. OCLE, Object Constraint Language Environment version 2.0. http://lci.cs.ubbcluj.ro/ocle/index.htm, December 2004. 23

[Uni04b] University of Southern California. Loom Knowledge Representation System 4.0. http://www.isi.edu/isd/LOOM/, December 2004. 120

[Uni04c] University of Southern California. PowerLoom Knowledge Representation System 4.0. http://www.isi.edu/isd/LOOM/PowerLoom/index.html, December 2004. 120

[van02a] Wil M.P. van der Aalst. Inheritance of dynamic behaviour in UML. In Daniel Moldt, editor, *Proceedings of second International Workshop on Modelling of Objects, Components and Agents (MOCA2002)*, pages 105–120, August 2002. 72, 85

[Van02b] Ragnhild Van Der Straeten. Using Description Logic in object-oriented software development. In Ian Horrocks and Sergio Tessaris, editors, *Proceedings of 2002 International Workshop on Description Logics (DL2002)*, volume 53 of *CEUR Workshop Proceedings*. CEUR Workshop Proceedings, April 2002. Toulouse, France. 14

[Van04] Ragnhild Van Der Straeten. Inconsistency detection between UML models using Racer and nRQL. In Sean Bechhofer, Volker Haarslev, Carsten Lutz, and Ralf Möller, editors, *Proceedings of 2004 third International Workshop on Applications of Description Logics (ADL2004)*, volume 115 of *CEUR Workshop Proceedings*, September 2004. 14, 225

[vdB01] Michael van der Beeck. Formalization of UML-statecharts. In Martin Gogolla and Cris Kobryn, editors, *Proceedings fourth International Conference UML 2001*, volume 2185 of *Lecture Notes in Computer Science*, pages 406–421. Springer, October 2001. Toronto, Canada. 26

[Ver01] Kurt Verschaeve. *UML - SDL Round-Trip Engineering Through Incremental Translation of Changes.* PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, February 2001. Brussels, Belgium. 26

[VJM04] Ragnhild Van Der Straeten, Viviane Jonckers, and Tom Mens. Supporting model refactorings through behaviour inheritance consistencies. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen Mellor, editors, *Proceedings seventh International Conference UML 2004*, volume 3273 of *Lecture Notes in Computer Science*, pages 305–319. Springer, October 2004. Lisbon, Portugal. 14, 224, 226

[vLLD98] Axel van Lamsweerde, Emmanual Letier, and Robert Darimont. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering*, 24(11):908–926, 1998. 3, 98, 184

[VMJ06] Ragnhild Van Der Straeten, Tom Mens, and Viviane Jonckers. A formal approach to model refactoring and model refinement. *Software and Systems Modeling*, 2006. to be published. 14, 224, 225, 226

[VMSJ03] Ragnhild Van Der Straeten, Tom Mens, Jocelyn Simmonds, and Viviane Jonckers. Using Description Logics to maintain consistency between UML models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *Proceedings sixth International Conference UML 2003*, volume 2863 of *Lecture Notes in Computer Science*, pages 326–340. Springer, October 2003. San Francisco, CA, USA. 14, 58, 224, 225

[VSM03] Ragnhild Van Der Straeten, Jocelyn Simmonds, and Tom Mens. Detecting inconsistencies between UML models using description logic. In Diego Calvanese, Giuseppe De Giacomo, and Enrico Franconi, editors, *Proceedings of 2003 International Workshop on Description Logics (DL2003)*, volume 81 of *CEUR Workshop Proceedings*, September 2003. Rome, Italy. 14, 158, 161, 225

[VSMD03] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent UML refactorings. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *Proceedings sixth International Conference UML 2003*, volume 2863 of *Lecture Notes in Computer Science*, pages 144–158. Springer, October 2003. San Francisco, CA, USA. 207

[Wel95] Christopher A. Welty. *An Integrated Representation for Software Development and Discovery.* PhD thesis, Rensselaer Polytechnic Institute, 1995. Troy, NY, USA. 134

[Wes04] Michael Wessel. *nRQL, http://www.sts.tu-harburg.de/~f.moeller/racer/racer-queries.pdf*, March 30 2004. 147, 158, 161, 162

[WF94] Christopher A. Welty and David A. Ferrucci. What's in an instance? Technical Report 18, RPI Computer Science Department, 1994. 134, 150

[WGN03] Robert Wagner, Holger Giese, and Ulrich A. Nickel. A plug-in for flexible and incremental inconsistency management. In Ludwik Kuzniarz, Zbigniew Huzar, Gianna Reggio, Jean Louis Sourrouille, and Miroslaw Staron, editors, *Blekinge Institute of Technology, Research Report 2003:06. UML 2003 Workshop on Consistency Problems in UML-Based Software Development II. Workshop Materials*, pages 78–88. Department of Software Engineering and Computer Science, Blekinge Institute of Technology, 2003. 94

[Wyd01] Bart Wydaeghe. *PaCoSuite Component Composition Based on Composition Patterns and Usage Scenarios.* PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, November 2001. Brussels, Belgium. 26

Index