**Vrije Universiteit Brussel - Belgium**
**Faculty of Sciences**
**In Collaboration with Ecole des Mines de Nantes - France**
**2003**



# CONSISTENCY MAINTENANCE OF UML MODELS WITH DESCRIPTION LOGICS

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Jocelyn Paola Simmonds Wagemann

Promoter: Prof. Dr. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promotor: Dr. Tom Mens (Vrije Universiteit Brussel)

# Abstract

Nowadays, the prevailing methodology used to specify software designs is UML. This language consists of various types of diagrams, each one dedicated to a different design aspect. This variety of views, that overlap with respect to the information depicted in each, can leave the overall system design specification in an inconsistent state. Necessarily, some form of consistency checking must be performed between different UML models. This is especially true in the context of design evolution. Therefore, a means of detecting and solving inconsistencies between related UML models must be provided.

With respect to UML models, two types of consistency can be noted: consistency between models belonging to the same design version and consistency between different versions of the same model. These are known as horizontal and evolution consistency, respectively. Current UML CASE tools do not take full advantage of the relationships that exist between the different UML model elements, and, as such, provide poor support for maintaining consistency between (evolving) UML models.

By specifying consistency relationships in a formal and precise way, reasoning about model consistency will be possible. The current UML metamodel provides poor support for consistency preservation and software evolution, e.g. versions are not supported. Relationships between models cannot be established. This thesis proposes a way in which this support can be integrated in the UML metamodel, with only some minor additions.

Based on the characteristics of inconsistencies observed between UML models, a classification of inconsistencies is proposed. To be able to detect and resolve inconsistencies, both a formal specification of model consistency and a formal reasoning engine relying on this specification is needed. This is also true for design restructurings, as their detection depends on the same metamodel relationships as the inconsistencies. This thesis studies the feasibility of using the formalism of Description Logics (DL) for this purpose.

In order to determine the suitability of DL, small experiments have been carried out. Detection predicates need to be written for each inconsistency. These have then been applied to the case study prepared for the occasion. The experimental results for inconsistencies and design restructurings indicate the feasibility of approach. Finally, these predicates have been incorporated into a tool that automates the translation between UML models and logic assertions, allowing the effortless application of detection and solution predicates.

# Acknowledgments

There are so many people who I would like to thank for this year. First and foremost, my parents, especially my mom, whose thirst of knowledge I inherited. They have always stimulated my curiosity about the world, and their actions are the ones that have had the most influence in shaping me into the person I am today. To Maxs, my favourite brother ;), to Barbara, who has tried to put some order into this messy student's life, to Susan, who has managed to keep me up to date on family news, and Charlotte, who brings out the child in me. To my niece Kelly, who always lights a candle in my heart.

Thanks to all my friends at the University of Chile, who have been very supportive during this whole experience, chatting and sending news by means of instant messengers and e-mail. It has been hard living so far from home for so long, but they have helped make this burden a little easier to carry. I would also like to thank my fellow *EMOOSEers*. The "surprise" birthday parties made these dates even more special, and even funnier, as some birthday guys and gals were truly surprised.

I would also like to thank my thesis advisors Ragnhild Van Der Straeten and Tom Mens. Their dedication as investigators and advisors is incredible. They were always willing to discuss ideas, problems or just talk. They even had the patience to read and correct this thesis :), for which I am truly grateful. Thanks to Luis Mateu, who made this trip possible. Thanks also to the people at the Ecole des Mines de Nantes and the Vrije Universiteit Brussel who helped us settle in and made us feel at home. Finally, thanks to the people at Google for making such a good search engine.

*For the journeys we've made together,*
*for the times we've shared...*
*May we share many more.*

Brussels, Belgium
August 15, 2003

Jocelyn Simmonds

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Context

As software development stepped up from a one-man effort to a multiple disciplined team effort, the methodologies used to specify and document the process also evolved. As systems grew in complexity and size, it became necessary to rethink the development process. The need for a pre-coding design process was acknowledged. Various methodologies emerged, most of these using diagrams as a way to represent system designs. Over the years, the various methodologies were merged, and a heterogeneous mix of diagrams that can be used to specify the critical parts of a system was obtained. An industry standard was formed - the Unified Modelling Language (UML).

UML, as a visual modelling language, provides a family of diagrams with which aspects like the structure and behaviour of a system can be defined. In an ideal situation, a system design will be defined by a model composed of diagrams that accurately represent its critical components. Although each diagram, by definition, shows a different view of the system, as a whole, the model should be consistent. Secondly, the system model should be a "snapshot" of the actual code that implements the system. Sadly, this is far from the truth.

Systems development is now often a group effort. Various people will brainstorm over the design, until an initial model is developed. General points of the design are agreed upon, and smaller groups develop the details of sub-components of the system design. For this approach to work, it is imperative that the sub-designs be consistent. A problem that arises is that each person on the team has a different level of expertise in UML. Furthermore, different tools make different versions of the UML standard. The notation is not always clear, and some things can be written in various ways. This is further complicated by the fact that the development team is not necessarily localised in one work area. With the appearance of the Internet, distributed workplaces have become commonplace.

It is common that the system requirements change, so this initial design is usually the first of many versions. Current UML CASE tools do not provide adequate versioning capabilities [9, 4]. Each model version has to be saved to a different file, each one having its independent system definition. Ideally, these system definitions should be linked in some form through the CASE tool, so that it is possible to perform some form of consistency check every time a new model version is produced. In practice, however, version consistency has to be checked manually, which calls into play the "human error" factor.

In addition, CASE tools maintain only basic relationships between the different types of UML diagrams. The set of class diagrams in a model defines the static representation of the system structure, as a collection of classes and relationships. The system definition for the model is generated from this set of class diagrams. Once this system definition is established, the rest of the diagrams are related to it as their elements are created. Modifying a class diagram modifies the system definition, but this definition alteration usually has little or no effect on the rest of the diagrams in the model. This type of consistency is referred to as horizontal consistency[17]. For example, removing a class from the model will not cause the removal of instances of this class from interaction diagrams. So, horizontal consistency has to be checked manually.

There is also the problem of maintaining the relationship between the requirements, design and the source code. The UML models can only be useful in the system maintenance phase if they are an accurate representation of the system implementation. This means that every change made at a design level has to be reflected in the code, and vice-versa. This is known as vertical consistency or co-evolution[16]. This artefact synchronisation is usually relegated to a second plane, as the primary objective of the development team is to comply with the product deadlines. The rich information present in the design models, that is quite valuable when deciding what evolution steps to take, is no longer available, since with each implementation iteration, the gap between the models and the implementation becomes larger.

For the reasons stated above, the automatic detection of inconsistencies, also known as consistency conflicts, would greatly benefit the software development community that uses UML as a form of design specification. UML is defined through its abstract syntax, presented as class diagrams, well-formedness rules in OCL and semantics, specified in natural language. Due to this, current approaches that deal with inconsistency detection have been limited to the study of very specific problems (see Section 9.1). Moreover, as these studies are independent of each other, each one uses a different formalism. What is needed is a general framework that would allow the uniform detection of the largest amount of consistency problems possible.

## 1.2 Motivation

The relationships established in the UML metamodel, represented as associations between the classes that represent the different model elements, could be used for consistency checking. In order to use these relationships, the consistency between models will have to be specified in a precise way. The semantic links between the UML models have to established using some formalism that provides reasoning mechanisms. The partial automation of this process will provide the development team with a consistent view on the system, through the system documentation. This will be a direct benefit, as the system will become more maintainable.

Once the links between the different diagrams are established, the limits of this approach should be tested. What kinds of inconsistencies can be detected? e.g., syntactic inconsistencies, semantic inconsistencies, breaches of guidelines and conventions, etc. As there is no official consensus as to which are the consistency problems between different UML models and diagrams, this set will have to be determined by means of experimentation. After determining what inconsistencies can be produced during an evolution step, it would be interesting to know what information is needed in order to detect each inconsistency. Given the existing information, is it possible to offer an automatic resolution of the inconsistency? As a result, this approach should allow the study of more complex relations in UML, going further than the "traditional" notion of inconsistency.

The UML model represented by the diagrams presented in figures 1.1 and 1.2 shows a motivational example. In this case, the sequence of interactions shown in the sequence diagram, that are received by an instance of the *ATM* class, does not exist as a trace of the state diagram that represents the behaviour for this class. This is not detected by CASE tools. Another example of an inconsistency is that seen in the state diagram shown in figure 1.3. Various states are not reachable from the initial state because the connecting transition, between the *PINEntry* and *VerifyAccountBalance* states, has been deleted.

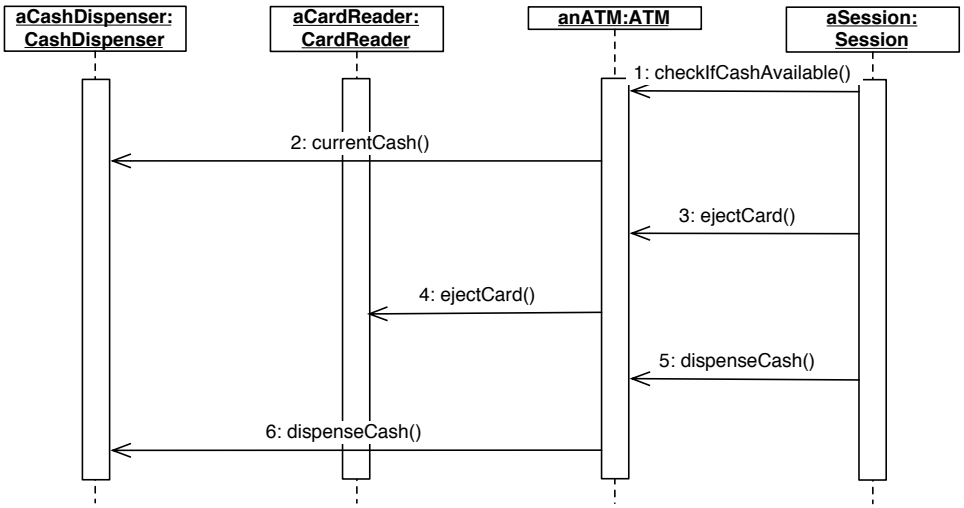


Figure 1.1: First motivational example: contradicting behaviour

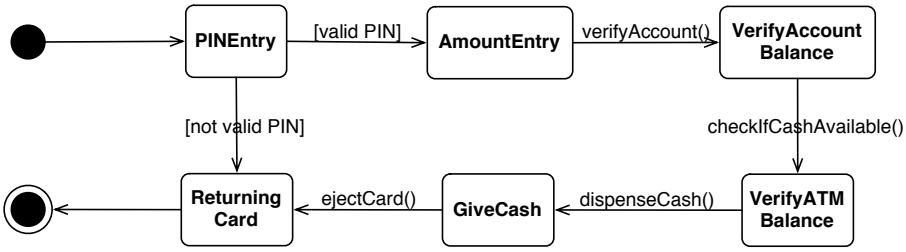


Figure 1.2: First motivational example: contradicting behaviour

Another question that motivates this experimental study is how evolution through refactorings could be detected and applied using the information explicit in the diagrams and models. A refactoring is a software transformation that preserves the external behaviour of the software and improves its internal structure[50, 54, 22]. The idea would be to use the same semantic links as in consistency checking, so as to provide a general framework that could deal with evolution-induced problems as a whole. Being able to apply refactorings at this level would be an advantage, as the designer can be more adventurous when deciding which refactorings to apply, because he will be able to see what effects each refactoring has on the system as a whole.

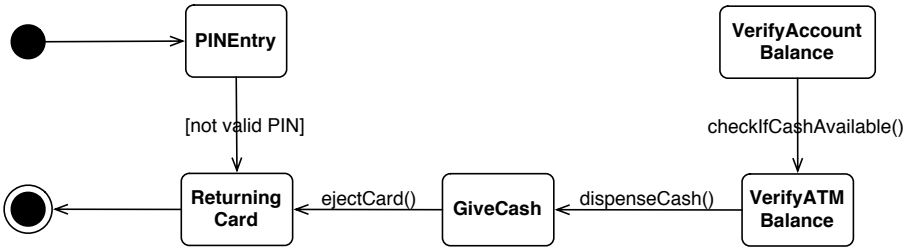


Figure 1.3: Second motivational example: disconnected model

## 1.3 Objectives

The main objective of this dissertation is to experiment with Description Logics (DL), as a formal means of detecting inconsistencies caused by evolution. Of special interest are the inconsistencies that arise between models and between model versions, after introducing evolutionary changes. Since UML lacks formal semantics, a formalism has to be chosen in order to allow manipulation of models, based on this meaning and not their visual representation as a diagram. This formalism should have an implemented reasoning engine, as one of the requirements is to benefit from the semi-automatization of the consistency checking step of the development process. Description Logics seems like the obvious candidate because of its basic reasoning tasks: subsumption, instance checking, relation checking, concept consistency and knowledge base consistency, if consistency checking is based on the UML metamodel. The subsumption task consists in determining which concepts subsume each concept defined for the knowledge base, thus organising the concepts into hierarchies. The instance checking task determines if an individual is the instance of a concept, while relation checking determines if two individuals in the knowledge base stand in a given relation. Finally concept consistency checks that the concept definitions are consistent, while knowledge base consistency verifies that the information about individuals is consistent.

As UML has nine diagram types, each one representing a different view of either the static system structure or its dynamic behaviour, the universe of study of this dissertation has been reduced for practical purposes. Three diagram types will be considered: Class, Statechart and Sequence diagrams. The information present in these diagrams permits the study of the basic structure of the system, how each element behaves in its own life-span, and how various elements interact as parts of the system, respectively. This will allow the detection of inconsistencies between structure and behaviour. Moreover, these diagrams are the ones most frequently used by system designers.

## 1.4 Overview

The dissertation proceeds as follows. Chapter 2 details the characteristics present in Description Logics that makes it a suitable candidate as a formalism for detecting and resolving inconsistency, also known as consistency conflicts. Additionally, it contains the result of the evaluation of three implementations of DL reasoning engines. In chapter 3, the UML profile of the metamodel used is described. This includes the definition of new stereotypes so as to be able to support model evolution. The environment necessary to carry out the experiments that will

validate the ideas of this dissertation, is described in chapter 4. Chapter 5 introduces the Banking Simulation, the case study used to produce test scenarios for these experiments. In chapter 6, the set of inconsistencies discovered in the study is presented. Chapter 7 is focused on design improvements tested during experimentation. In order to show the practicality of the ideas presented in this dissertation, a tool that combines the DL engine and the UML models was implemented. The details with respect to this tool are given in chapter 8. Finally, chapter 9 presents the conclusions of this dissertation and related and future work.

# Chapter 2

# Description Logics

The ideas expressed in this dissertation deal with the consistency problems that arise in UML models and diagrams. As UML lacks formal semantics, a formalism had to be chosen in order to allow manipulation of models and diagrams, based on the meaning of the diagrams and not their representation. 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. Of special interest are KRSs using Description Logics[6] (DL) as a representation language, as DL is a decidable fragment of first-order logic. Various systems based on description logics have been implemented, each with its own expressive power. The decidability and complexity of these formalisms has been investigated in detail.

## 2.1 Basic Issues

The main motivation for resorting to description logic is because it is a decidable fragment of first-order logic that possesses sound and complete reasoning mechanisms. In his PhD, Kim Mens [42] tried to maintain consistency between a software architecture and its corresponding source code using a logic programming language. Unfortunately, the algorithm to check consistency turned out to be very inefficient. Moreover, due to the fact that a full-fledged programming language was used, there was no guarantee whatsoever concerning the decidability or completeness of the consistency algorithm. Description logic solves the latter problem because it is a decidable and complete fragment of first-order predicate logic. It is also hoped that the efficiency will be better.

### 2.1.1 Concepts and Roles

A KRS based on DL has a concept specification language which is a combination of the basic DL operators, and, most importantly, automatic reasoning procedures. Each member of the DL family is characterised by the set of concept and roles constructors allowed. Different

| **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. | $\forall R.C$ | $\{d_1 \mid \forall d_2 \in \Delta^{\mathcal{I}}.(R^{\mathcal{I}}(d_1, d_2) \rightarrow d_2 \in C^{\mathcal{I}})\}$ |
| existential quant. ($\mathcal{E}$) | $\exists R.C$ | $\{d_1 \mid \exists d_2 \in \Delta^{\mathcal{I}}.(R^{\mathcal{I}}(d_1, d_2) \wedge d_2 \in C^{\mathcal{I}})\}$ |
| unqualified number restr. ($\mathcal{N}$) | $(\geq n\ R)$ | $\{d_1 \mid \lvert\{d_2 \mid R^{\mathcal{I}}(d_1, d_2)\}\rvert \geq n\}$ |
| | $(\leq n\ R)$ | $\{d_1 \mid \lvert\{d_2 \mid R^{\mathcal{I}}(d_1, d_2)\}\rvert \leq n\}$ |
| qualified number restr. ($\mathcal{Q}$) | $(\geq n\ R.C)$ | $\{d_1 \mid \lvert\{d_2 \mid R^{\mathcal{I}}(d_1, d_2) \wedge d_2 \in C^{\mathcal{I}}\}\rvert \geq n\}$ |
| | $(\leq n\ R.C)$ | $\{d_1 \mid \lvert\{d_2 \mid R^{\mathcal{I}}(d_1, d_2) \wedge d_2 \in C^{\mathcal{I}}\}\rvert \leq n\}$ |
| functional number restr. ($\mathcal{F}$) | $(\geq 2\ R)$ | $\{d_1 \mid \lvert\{d_2 \mid R^{\mathcal{I}}(d_1, d_2)\}\rvert \geq 2\}$ |
| | $(\leq 1\ R)$ | $\{d_1 \mid \lvert\{d_2 \mid R^{\mathcal{I}}(d_1, d_2)\}\rvert \leq 1\}$ |
| one-of ($\mathcal{O}$) | $\{a_1, \ldots, a_n\}$ | $\{d \mid d = a_i^{\mathcal{I}}$ for some $a_i\}$ |
| role filler ($\mathcal{B}$) | $\exists R.\{a\}$ | $\{d \mid R^{\mathcal{I}}(d, a^{\mathcal{I}})\}$ |
| role name | $R$ | $R^{\mathcal{I}}$ |
| role conjunction ($\mathcal{R}$) | $R_1 \sqcap R_2$ | $R_1^{\mathcal{I}} \cap R_2^{\mathcal{I}}$ |
| role hierarchy ($\mathcal{H}$) | $R_1 \sqsubseteq R_2$ | $R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$ |
| inverse roles ($\mathcal{I}$) | $R^{-1}$ | $\{(d_1, d_2) \mid R^{\mathcal{I}}(d_2, d_1)\}$ |

Table 2.1: Common DL operators

combinations of operators generate languages with different expressiveness. Table 2.1[3] shows a summary of the most common DL constructors, including their semantics. Using these concept and roles constructors, complex concept and role expressions can be formed.

Historically, a number of description logics have received a special name. The language $\mathcal{FL}^-$ [12] is defined as the description logic allowing universal quantification, conjunction 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 existentials 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 $\mathcal{AL}$[55] extends $\mathcal{FL}^-$ with negation of atomic concepts. It is customary to define systems by postfixing the names of these original systems with the names of the added operators from table 2.1. For example, the logic $\mathcal{ALC}$ is $\mathcal{AL}$ extended with full negation.

### 2.1.2 Knowledge bases and Inference

The concept specification language allows the definition of terminology to be used in the creation of knowledge bases, where inferences can later be performed. The knowledge base terminology is specified by a finite set of expressions composed from the concept operators allowed by the DL. These concept definition expressions are called terminological axioms. The set of concept definitions is called the Terminological-Box (*Tbox*). The part of the knowledge base that contains the individuals that instantiate the concepts defined in the *Tbox*, is called the Assertional-Box (*Abox*). The *Abox* contains extensional knowledge about the domain of interest, as a finite set of expressions relating concepts and relationships to individuals, called assertions.

In earlier DL systems, terminological axioms were thought of as definitions, and restrictions were placed on their form. The first is that axioms must be simple, i.e, any atomic concept appears at most once in the left hand side of a terminological axiom in the *Tbox* . The second restriction is the prohibition of cycles in concept definitions. These restrictions were imposed due to the fact that it was thought that lifting these restrictions would require some sort of special semantics, due to the underlying graph-like nature of the DL systems. Currently, general terminological axioms have been introduced, without the need for special semantics. As a result, these two restrictions have been lifted in current systems[31].

A system is complete if it is guaranteed to find all the valid inferences. Some systems have preferred to restrict expressiveness in order to guarantee tractability and completeness. Having a less expressive concept definition language limits the power of the system, and the situations in which these systems are applicable is reduced. Other systems have decided to include constructs that are known to be intractable or even undecidable. These are more expressive, but the classifiers are incomplete. In this case, the users have to be aware that there could be missing inferences.

### 2.1.3 Reasoning Tasks

The reasoning tasks reason about the consistency of knowledge bases. *Tbox* reasoning is used to refer to the ability to perform inferences from a *Tbox*, and similarly, *Abox* reasoning is inference for an *Abox*. The standard reasoning tasks offered by knowledge representation systems are: subsumption, instance checking, relation checking, concept consistency and knowledge base consistency. These tasks can be performed under either the closed or open world assumption. If the classifier assumes closed-world semantics, it considers that the current fillers for all roles are the only fillers for these roles, in other words, the current knowledge is complete. The open-world assumption implies that the current knowledge is not complete.

Subsumption is one of the classical reasoning tasks performed in most knowledge representation systems. It is used to organise concept definitions into "is-a" hierarchies, finding for each concept the other concepts that subsume it, also identifying the most specific subsumers for a concept. Classification algorithms rely directly on the subsumption check. The subsumption relation implicitly defines a taxonomy of concepts, which can actually be used to solve some of the other reasoning tasks; for example, a concept is unsatisfiable if it is subsumed by the empty concept $\perp$.

Instance checking is used to verify whether an individual belonging to an *Abox* is an instance of a concept belonging to a *Tbox*. An individual is inconsistent with respect to a *Tbox* if it is classified as an instance of the empty concept $\perp$. Relation checking determines if two individuals of an *Abox* stand in a given relation of a *Tbox*. These two operations can be considered the central reasoning tasks for retrieving information about individuals from a knowledge base. Finally, concept and knowledge base consistency is used for verifying whether the information contained in a knowledge base is coherent.

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.

## 2.2 Tool Support Evaluation

KL-ONE was the first DL system implemented, by Brachman[11]. This system served as a foundation block for various other systems, such as BACK[53], CLASSIC[10], LOOM [39] and K-REP [41], that are known as the "KL-ONE" family because of this relation. These systems were designed and implemented taking different positions on the requirements of expressive power, completeness of reasoning, and tractability of reasoning. These first generation systems were used for medical systems, data-mining, knowledge-enhanced search, configurators, government research and application programs.

These early systems however typically sacrificed expressive power or completeness in order to maintain efficiency or even understandability. This was done by choosing determined combinations of the basic DL operators. The more recent set of implemented description logics are expressive with respect to concept reasoning and also maintain complete reasoners with computationally efficient implementations. A few examples of implemented description logics in this class today are DLP[51], FACT[33], and RACE [29]. The aspects that will be discussed in this evaluation are the following: supported DL, classifier, query mechanism and open/closed world assumption. Some additional features per tool are also discussed.

### 2.2.1 CLASSIC v 2.3

CLASSIC[10, 2, 52] 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.

#### Supported DL

CLASSIC's concept definition language belongs to the $\mathcal{AL}$ family of languages. As such, it supports the definition of concepts and relations, has "top" and "bottom" concepts, concept conjunction, and universal and existential quantifiers. According to the constructs presented in Table 2.1, the extensions to this DL classify it as $\mathcal{ALQUIFO}$. Its terminological language is of limited expressiveness - so that *Tbox* reasoning can be performed in polynomial time. The designers of the system decided that it was essential that the system be able to respond rapidly to questions. Also, by restricting expressiveness, tractability and completeness can be guaranteed when classifying the concepts of the *Tbox*.

#### Classifier

Elements are defined using compound expressions from a simple grammar. A concept definition is coherent if it can be instantiated. If a definition is not found in the knowledge base (KB) of the model and it is coherent with the rest of the definitions, it is added. Concept definitions cannot be cyclic or recursive, due to the limited expressiveness of the language supported by the system. The system does not support transitive roles, either.

The classifier should find all the inferences, but there are some areas in which it is known that the inference capability offered by the classifier is incomplete. With respect to concept and individual processing, some inferences relative to role hierarchies are not performed to completion. Also, the properties of the individuals that are implied by their presence in descriptions

are ignored when classifying them. The reason given for this is that this last type of inference is computationally difficult. It is also possible for concept subsumption incompleteness to appear, when determining whether or not an individual satisfies a concept definition. This is caused also because of the ignored individual properties. These cases of incompleteness are known and documented.

#### Query Mechanism

CLASSIC is classifier-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.

#### Open/Closed World Assumption

CLASSIC assumes that all knowledge bases are "open worlds", that is, if something cannot be proven as false, it is considered true. The user can manually close roles using a system function. The reason why a function is provided, and not an operator, is that having an operator would make concept descriptions autoepistemic, and thus, non-monotonic. So, if a role has not been closed by the user, or deduced to be closed, it is assumed to be open.

#### Additional Features

Continual refinements and changes can be made to the KB 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 KB, whenever an inconsistent state is detected. Copies of the individuals that lead to the inconsistent state are saved. The explanation facility offered by the system can be used to find the cause of these inconsistencies, as CLASSIC can explain its deductions with respect to subsumption, rule firing and error occurrence.

### 2.2.2 Loom v 4.0

Loom[39, 13, 40, 35] was developed at University of Southern California's Information Sciences Institute. As a KRS, Loom is one of the most flexible KR systems in existence, and offers a wide range of services: reasoning, editing, validation and explanation of model structures. 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. Loom can be considered as a complete programming environment.

#### Supported DL

Loom's concept definition language belongs to the $\mathcal{AL}$ family of languages. As such, it supports the definition of concepts and relations, has "top" and "bottom" concepts, concept negation, concept conjunction, and universal and existential quantifiers. According to the constructs presented in Table 2.1, the extensions to this DL classify it as $\mathcal{ALCQRIFO}$. This terminological

language is quite powerful, but forces the use of incomplete reasoning algorithms for concept classification and subsumption. The reason why its classifier is incomplete is that the subsumption algorithms used are of the structural type, and given the expressiveness of the concept definition language, it is impossible to provide efficient, complete classifiers using the type of classification algorithms used by the KL-ONE family.

#### Classifier

Loom is more expressive by knowingly supporting intractable terminological constructs. The justification given by Loom's designers for the utility of an incomplete classifier is that complete systems are only of theoretical interest, but are too restrictive to be of use in most applications. The problem with this 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.

Complexity of the classifier is comparatively efficient with respect to other KRS using structural subsumption algorithms, but 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.

The classifier does not support partial loading of knowledge bases. Disjoint relationships between concepts have to be explicitly declared, which is contrary to the natural idea that any unrelated concepts are automatically considered as disjoint. The classifier does not support transitive roles. Assertions specifying transitivity can be made in the *Abox*, as well as about role refinements. However, models compiled with *Abox* transitivity and refinement exhibit bad performance.

#### Query Mechanism

Unlike most of its peers, Loom provides a powerful assertional reasoning facility, in the form of the query and retrieval mechanism. The query language is a predicate calculus-based language. This language is strictly more expressive than the description logics used in the concept definition language. It is possible to include predicate calculus-based expressions in description logics expressions, using special operators, making it possible to even more expressive when defining concepts. However, the classifier cannot reason about these expressions. The Loom query language does not support questions returning concepts and formulas, only sets of individuals.

#### 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 been asserted. 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 will let

Loom draw additional inferences (as opposed to the default open-world assumption). There are no operators of functions that will switch the world from open to closed or vice-versa.

#### Additional Features

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 matcher is invoked. This causes the re-computation of the types of the modified instances, and the changes are propagated throughout the KB using a forward-chaining algorithm. When a concept is redefined, the necessary reclassifications are made and the model is checked so as to ensure that model consistency 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. Assertions are propagated through the knowledge bases following certain rules.

### 2.2.3 RACER v 1.7

RACER[29, 30, 31] was initially developed at the University of Hamburg, Germany. RACER is actively supported and future releases are developed at Concordia University in Montreal, Canada, and at the University of Applied Sciences in Wedel, Germany. Of all the evaluated KR tools, this is the only one that is actively being developed and maintained.

#### Supported DL

RACER uses $\mathcal{SHIQ}$[34] logic - a very expressive logic, that includes DL constructs like qualified number restrictions, inverse roles, hierarchies, transitive roles and full negation. Simple roles, that 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 the case of this unrestricted syntax.

#### Classifier

There have been advances in the development of reasoning algorithms for expressive description logics, but mostly only considering *Tbox* reasoning. Only recently, *Abox* reasoning has summoned interest. In previous systems, *Abox* reasoning was never implemented, as the complexity of the proposed algorithms was intractable. RACER is the first system to support full-fledged *Tbox* and *Abox* reasoning.

In order to be able to provide complete reasoning for an expressive DL like $\mathcal{SHIQ}$, RACER does not use the same type of classification algorithm as the KL-ONE family. Instead, it uses tableau calculus-based algorithms. The complexity for this family of algorithms has been determined for the worst-case, and the first systems using this type of reasoning algorithms behaved in an acceptable manner in practice.

A RACER *Tbox* contains concept definitions, general concept inclusions (explicit subsumption relations) and role definitions. Cyclic concept definitions are handled correctly. Multiple

definitions can exist for a concept. These add on to the central definition of the concept, if the resulting concept is coherent. RACER is very flexible w.r.t concept definition, and even allows forward references to concepts that will be introduced later.

Role definitions must be unique, and can be arranged into hierarchies. Super roles cannot be cyclic. Like Loom, roles can be given a range and domain. Simple roles can have number restrictions that belong to $\mathbb{Z}$ and $\mathbb{Q}$, and these restrictions are effectively used when dealing with subsumption. Constraints can be established between individuals.

#### Query Mechanism

Even though RACER offers *Abox* reasoning, it only provides query functions that return predefined sets of individuals, instead of a flexible query language, like Loom. However, these sets can then be manipulated using the underlying implementation language of the DL tool. The reasoning effort used for answering *Abox* queries can be controlled, by choosing either query indexes or exploiting the query subsumption.

#### Open/Closed World Assumption

RACER operates under the open world assumption: what can't be proved as false, is assumed to be true. There are no operators or functions that allow the closing of the roles of an individual. This is because closed world assumption requires non-monotonic reasoning, currently not supported by DL systems.

#### Additional Features

This DL tool 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. The file interface of RACER supports queries given in KRSS-like syntax. 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.

## 2.3 Conclusion

After a study of the characteristics offered by each system, and interacting with each one, it was decided that Loom will be used as the DL-based reasoning tool in this dissertation. Loom has a very expressive concept definition language and a powerful query and retrieval mechanism. Although Loom has an incomplete classification algorithm, the experiments will be restricted to a subset that is complete. The query facility and production rules can be used to detect inconsistencies, provide solutions to these and propagate changes to the rest of the models.

The main reason for not choosing CLASSIC is the lack of expressiveness of the concept definition language. RACER, with its highly expressive terminological language, transitive roles and *Abox* reasoning, would have been an ideal candidate, but this choice would have represented more work, as the lack of a query language would force the implementation of a layer that could interpret the results of the query functions.

# Chapter 3

# UML Profile

In this chapter, a summarised overview of UML as a modelling language is given. This is followed by a section that discusses the subset of the UML metamodel used in this study, with respect to the diagram types used. As this study is a proof of concept, the expressiveness of these diagrams was restricted, including only the model elements being studied in the case study. Finally, the extension to the UML metamodel that allows reasoning with respect to horizontal and evolution consistency, is presented.

## 3.1 UML

UML[48], as a visual modelling language, provides a family of diagrams with which aspects like the structure and behaviour of a system can be defined in a clear and unambiguous fashion. In an ideal situation, a system will be defined by a model composed of diagrams that accurately represent its critical components. The following is a list of the kinds of diagrams that represent the static and dynamic aspects of a system:

- Class diagram: shows a static representation of the system structure, as a collection of classes and relationships (association, aggregation and . Class definitions are included in this diagram, by specifying attributes and operations. Main diagram in a model, since it specifies the model elements that are referenced by the rest of the diagrams.
- Statechart diagram: shows the behaviour of the instances of one class, as a collection of states and transitions. These states represent the different stages of an object's lifetime in the system.
- Sequence diagram: shows the interaction between system objects, arranged in a time sequence. Interactions are message sends between the represented objects.
- Use Case diagram: shows the functionalities that the user expects in the final product. These functions are represented as sequences of actions that the system can perform, as a result of the interaction between the system users (actors) and the system.
- Collaboration diagram: shows the interaction between objects, as free-form interactions between objects. Messages are labelled chronologically, so as to maintain the ordering.
- Object diagram: shows a static view of the states of objects and their relationships, as the configuration of the system as a certain point in time.

- Activity diagrams: shows the behaviour of an operation as a set of actions.
- Component diagram: shows the system as a collection of physical components.
- Deployment diagram: shows how components will be deployed on specific hardware parts.

UML has a set of fundamental modelling concepts that represent the model elements used in the different diagrams. The UML authors have tried to make this list as comprehensive as possible, so that UML can be used in a wide range of scenarios. They avoided specialising the notation towards certain areas, and preferred to include a notation extension mechanism, in order to make the language more customisable. Groups that feel the need of a more specific notation, or believe that the current notation and model elements do not support modelling of certain aspects, are free to use this mechanism in order to extend the modelling language to their own needs.

A UML profile is a specification that identifies a subset of the UML metamodel (which may be the entire UML metamodel) that is used, and also specifies well-formedness rules beyond those included in the normative UML specification that contribute to the definition of metamodel elements. Standard elements beyond those specified by the identified subset of the UML model are specified. A profile can also include the specification of additional semantics, expressed in natural language. A UML profile was specified for this thesis, as new concepts were included so as to permit reasoning with respect to model information. Also, this profile includes a subset of the UML metamodel and well-formedness rules, as this thesis provides a proof of concept, and some model elements were not used in the case study.

The UML profile used in this study includes a subset of the *Core, Model Management, Common Behavior, Collaborations* and *State Machines* packages of the UML metamodel. This includes the corresponding subset of well-formedness rules and semantics. The *Core* package specifies the abstract syntax of Class diagrams. The concepts of models and relationships between models are present in the *Model Management* package. The *Common Behavior* and *Collaborations* packages specify the elements present in Sequence and Collaboration diagrams. Finally, State diagrams are specified by the *State Machines* package. In the following sections, more details will be given about the UML profile used.

## 3.2 Models

As mentioned in section 1.3, only three types of diagrams will be considered in this thesis: Class, Sequence and Statechart diagrams. The system elements are defined in the class diagrams, their individual behaviours appear in the statechart diagrams, and finally, their role in the execution of the system is visible in the sequence diagrams. Using these three kinds of diagrams, static and dynamic information on the system is available, and the size of the experiments is kept manageable. Subsections 3.2.1, 3.2.2 and 3.2.3 present the details of the UML metamodel subsets that correspond to these diagrams. This information is highly technical and can be skipped by readers who are not familiar with the UML metamodel.

### 3.2.1 Class Diagram

From the *Core Package - Backbone* ([48]: page 2-13), the intermediate metaclasses *StructuralFeature* and *BehavioralFeature* and the metaclass *Constraint* will not be considered in this

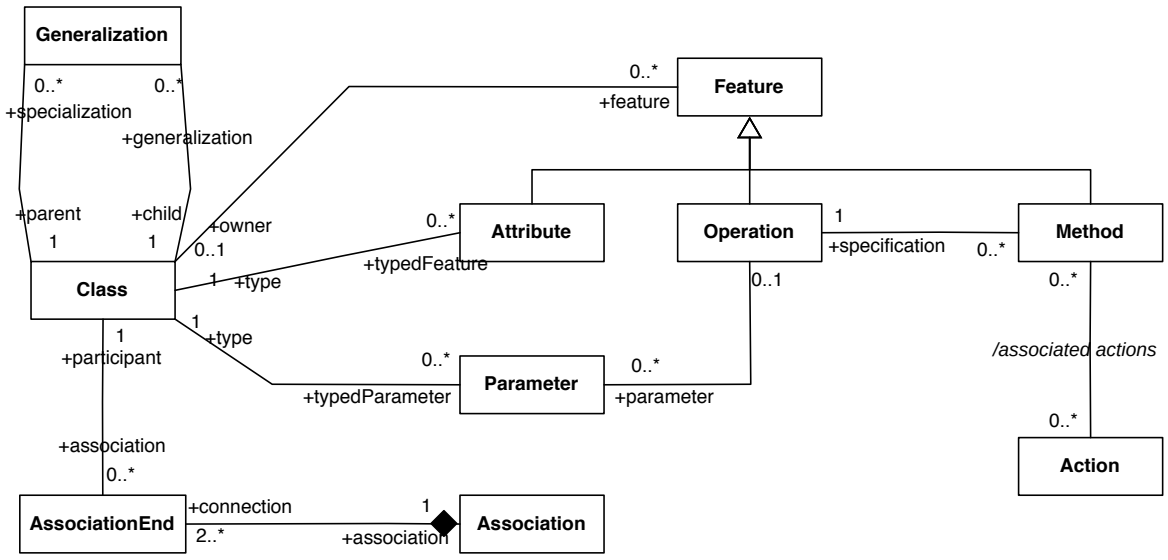


Figure 3.1: Subset of the UML *Core Package*

thesis. *BehavioralFeature* is an abstract metaclass that only represents features like operations and methods, so its omission has no effect on the type of elements that can be included in a class diagram. *StructuralFeature*, like *BehavioralFeature*, only represents class feature attributes. A *Constraint* is a semantic condition or restriction expressed in text, with respect to a model element. This metaclass has been omitted from the profile, *Constraints* cannot be included in class diagrams.

From *Core Package - Classifiers* ([48]: page 2-16), the metaclass hierarchies *Interface, DataType, Node, Component* and *Artifact* are ignored[1] as subclasses of the Classifier metaclass. Only *Class* is kept, and this metaclass replaces *Classifier* in this profile. Of these metaclasses, the only one that is used in class diagrams is the *Interface* metaclass. The omission of the rest does not influence the expressiveness of the resulting class diagram notation for the considered case study. The *Interface* metaclass was omitted in order to make the experiments more manageable.

From the *Core Package - Relationships* ([48]: page 2-14), the metaclasses *Flow, GeneralizableElement* and *AssociationClass* are not considered. A *Flow* is a relationship between two versions of an object or between an object and a copy of it. As it is a relationship between objects, it cannot be included in class diagrams, so its elimination does not restrict the expressiveness of the class diagrams allowed. A *GeneralizableElement* is a model element that may participate in a generalisation relationship. In a class diagram, this usually refers to a class or interface. Since interfaces are not being considered in these experiments, classes are supposed as being *GeneralizableElements*, and the abstract metaclass is removed. An *AssociationClass* is an association that is also a class. It is a relationship that has features of its own. These are not considered, like interfaces, as these constructs are not used in the case study.

None of the metaclasses belonging to *Core Package - Auxiliary Elements* ([48]: page 2-17) and *Extension Mechanisms* ([48]: page 2-76) are included in this profile. From *Data Types Package*

[1]This includes their descendants.

- *Main* ([48]: page 2-85), the metaclasses *Multiplicity* and *MultiplicityRange* are included, so as to permit the use of multiplicities in the example class diagrams. An overview of the *Core Package* subset used is shown in figure 3.1. The metaclass *ModelElement* is included, but not displayed in the figure for reasons of clarity, as most of the elements present in the diagram inherit from it, directly or indirectly.

### 3.2.2 Statechart Diagram

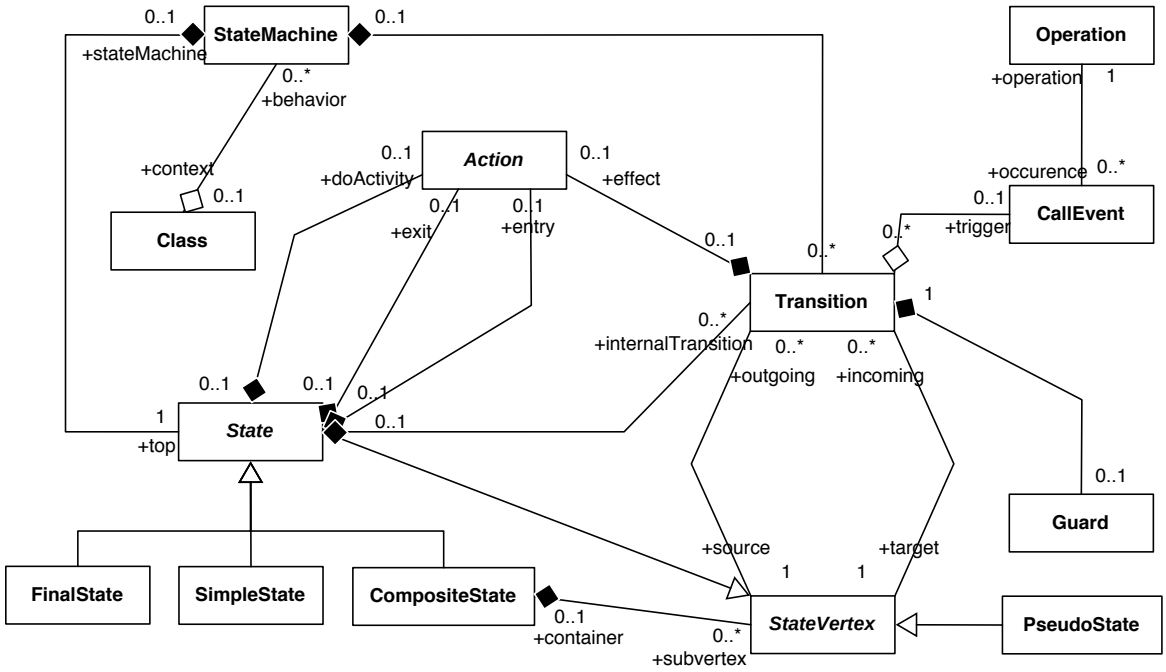


Figure 3.2: Subset of the UML *State Machines Package*

From the *State Machines Package - Main* ([48]: page 2-141), the metaclasses and properties supporting concurrent states, stubbed transitions, submachines states, synch states and time and signal events are omitted. This results in omitting the metaclasses *SynchState, SubMachineState, StubState, SignalEvent* and *TimeEvent* and the metaattribute *isConcurrent* of metaclass *CompositeState*. Additionally, the only kind of *PseudoStates* that will be allowed are *initial* pseudo states. This means that *deepHistory, shallowHistory, join, fork, junction* and *choice* pseudo states cannot be used when using this UML profile. All these characteristics have been eliminated as the state machines studied in the case study do not use these constructs. An overview of the *State Machines Package* subset used is shown in figure 3.2.

### 3.2.3 Sequence Diagram

The *Common Behavior Package - Signals* ([48]: page 2-94) is excluded as a whole, as only synchronous stimuli between instances are considered in this study. From *Common Behavior Package - Links*, the metaclass *ObjectLink* is ignored, as the corresponding model element for class diagrams, *AssociationClass*, was ignored. As only sequence diagrams are considered in

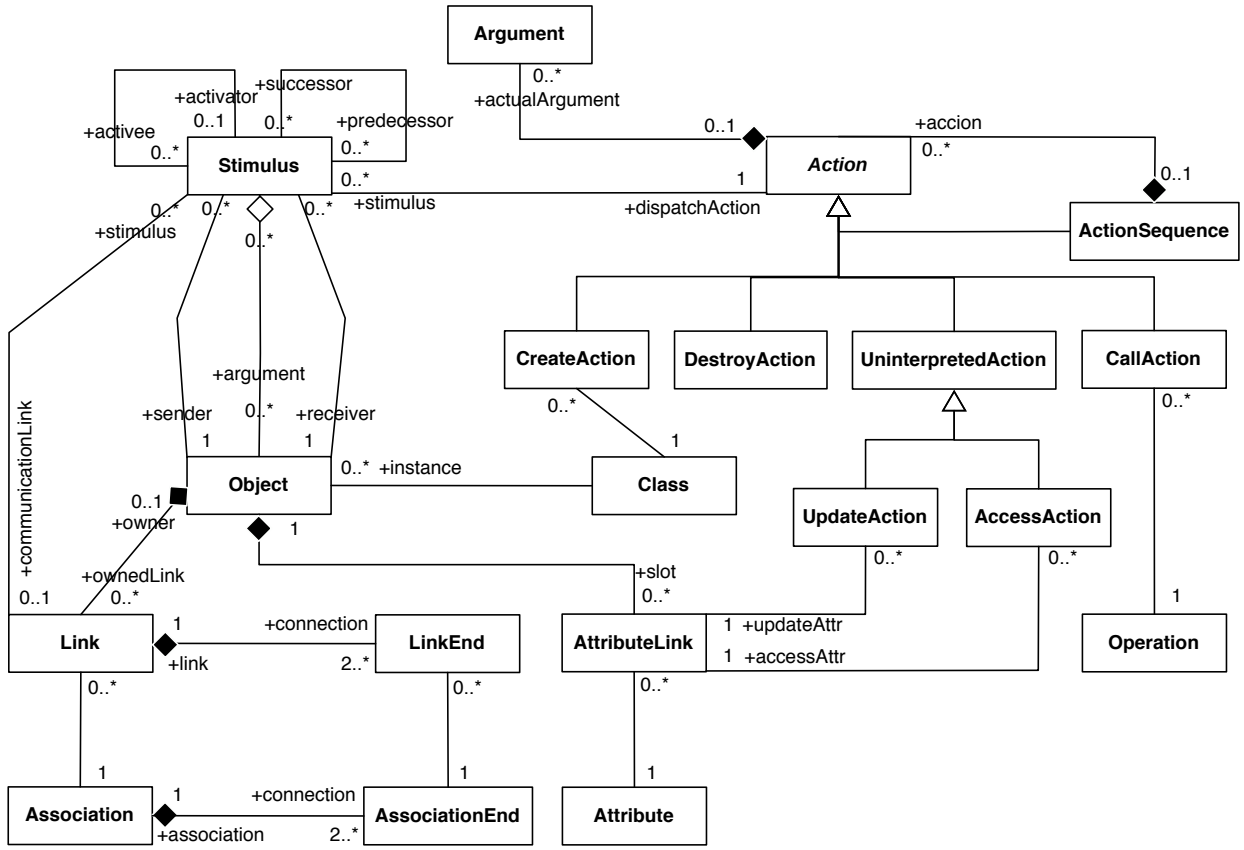


Figure 3.3: Subset of the UML *Common Behavior* and *Collaborations Packages*

this profile, the model elements used in interaction diagrams are not included in this profile. The result is that the *Collaborations Package - Interaction* subset is omitted from this profile.

From the *Common Behavior Package - Actions* ([47]: page 2-96), the metaclasses *ReturnAction, SendAction* and *TerminateAction* are ignored. As mentioned in Section 3.2.1, the metaclass *Classifier* is replaced by its descendent, the metaclass *Class*. Sequence Diagrams that comply with the metamodel of this profile cannot have return, send or terminate actions. A return action is an action that results in returning a value to a caller. A send action is an action that results in the asynchronous sending of a signal, and as signals have been omitted, this action type has to be omitted. A terminate action results in self-destruction of an object.

From *Common Behavior Package - Instances* ([48]: page 2-96), the metaclasses *DataValue, SubsystemInstance, NodeInstance* and *ComponentInstance* are ignored as subclasses of the *Instance* metaclass. Only *Object* is kept, and this metaclass replaces *Instance* in this profile. This is due to the fact that only *Class* is considered as a *Classifier*. These metaclasses are not considered in this profile, as they are not present in the case study.

From the *Collaborations Package - Instances*, only sequence diagrams presenting an *InteractionInstanceSet* are supported. An *InteractionInstanceSet* is a set of *Stimuli* between *Instances* within a *CollaborationInstanceSet*. However, the metaclasses *InteractionInstanceSet* and *CollaborationInstanceSet* have been omitted, as this information can be determined by looking at the sequence model. This decision is strengthened by the fact that some current UML CASE tools (e.g., Poseidon [26]) do not include this level of information in models produced. An overview of the *Common Behavior* and *Collaborations Packages* subset used is shown in figure 3.3.

To be able to indicate if an *AttributeLink* is accessed or updated, the *UninterpretedAction* metaclass is stereotyped into *UpdateAction* and *AccessAction*. In order to keep track of the *AttributeLink* that is accessed or updated by an instance of *AccessAction* or *UpdateAction*, different kinds of tag-value pairs are introduced. The pair (`updateAttr`, *AttributeLink*) keeps track of *UpdateActions* and the pair (`accessAttr`, *AttributeLink*) keeps track of *AccessActions*. These pairs are represented as associations in figure 3.3

In order to reason about the ordering of the *Stimuli* in a sequence diagram, two tag-value pairs have been introduced. The pair (`successor`, *Set(Stimulus)*) keeps track of the *Stimuli* that are successors of a *Stimulus*, in the scope of an activation box of an *Object*. Likewise, the pair (`predecessor`, *Set(Stimulus)*) keeps track of the *Stimuli* that are predecessors of a *Stimulus*. So as to be able to identify the *Stimuli* that belong to one activation box, two more tag-value pairs have been introduced. The pair (`activator`, *Set(Stimulus)*) keeps track of the activating *Stimulus* of a set of *Stimuli* that interact in one activation box. The pair (`activee`, *Stimulus*) keeps track of the set of *Stimuli* that are activated by one *Stimulus*, thus identifying the set of stimulus that interact in one activation box. Each pair of tag-value pairs are represented together as a bidirectional associations in figure 3.5.

## 3.3 Model Management

The existing set of model elements includes the idea of a model as a component, a subsystem or a package. Design, as the set of various models, is not present. The idea of maintaining a link between different versions of a model is also not present. The only existing *evolution* relationship is *Refinement* that specifies the refinement relationship between model elements

at different semantic levels, such as analysis and design. Due to this, the profile has been augmented with stereotypes, so as to be able to support model evolution.

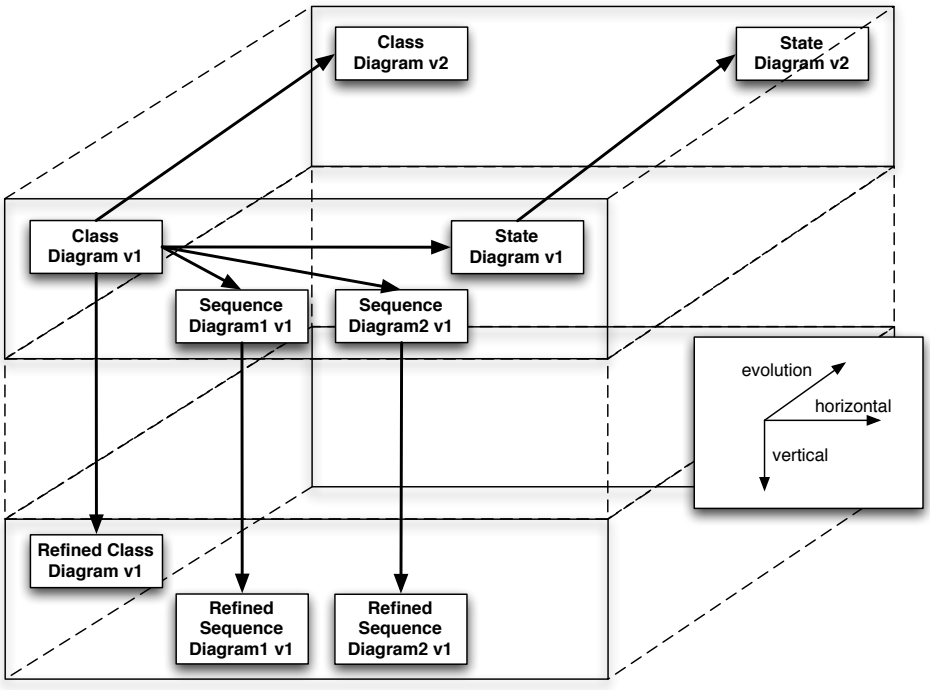


Figure 3.4: Horizontal, Vertical and Evolution Consistency

### 3.3.1 Traces

Two types of consistency[17, 16] have been distinguished in UML models. The first is horizontal consistency, and indicates consistency between different models within the same version. Evolution consistency indicates the consistency between different versions of the same model. A third type of consistency, vertical consistency, which relates model elements at different semantic levels. The three types of consistency are shown in figure 3.4: the horizontal arrows describe horizontal consistency, the diagonal arrows describe evolution consistency and the vertical arrows describe vertical consistency. Both horizontal and evolution consistency can be expressed in this UML profile, by defining two stereotypes for the Trace metaclass: *HorizontalTrace* and *EvolutionTrace* (see figure 3.5). Vertical consistency can be expressed using the existing *Refinement* stereotype.

### 3.3.2 Models

From the *Model Management Package* ([48]: page 2-182), the intermediate metaclasses *Namespace* and *Package* are ignored, as they are not explicitly used in the case study. This hierarchy has been flattened, maintaining only the subclass *Model*. The *namespace* relationship is now assumed by *Model*. The model elements that participate in a model are now directly related to the owner model. In order to be able to deal with model versions, evolution and version con-

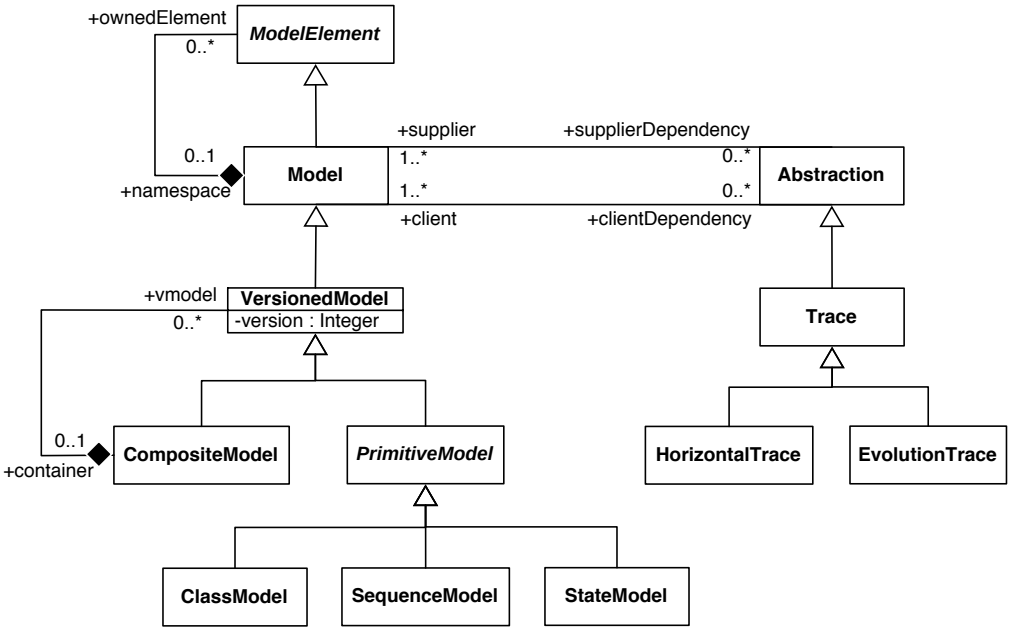


Figure 3.5: Subset of the UML *Model Management Package*

sistency, the *Model* metaclass is stereotyped into *VersionedModel*, so as to include a tag-value pair - (`version`, *Integer*) - that indicates the model version number.

To specify the kind of models that can be related by horizontal or evolution consistency arrows, the *VersionedModel* metaclass in the *Model Management Package* is stereotyped to distinguish between primitive models and composite models. For this reason, *VersionedModel* is stereotyped into *CompositeModel* and *PrimitiveModel*. *PrimitiveModel* is an abstract metaclass, that is further specialised (stereotyped), into *ClassModel, SequenceModel* and *StateModel*. *CompositeModel* is a stereotyped *VersionedModel* that is a container for all the *PrimitiveModels* that belong to the same version.

In order to keep track of the models belonging to a *CompositeModel*, a tag-value pair (`vmodel`, *Set(VersionedModel)*) is introduced. For *VersionedModel*, a tag-value pair (`container`, *CompositeModel*) is needed. Both tag-value pairs are represented together as a bidirectional association in figure 3.5. Well-formedness rules must be specified for the newly introduced *Model* and *Trace* stereotypes and tag-value pairs:

- A *CompositeModel* contains zero or more *VersionedModels*.
- A *HorizontalTrace* can only be specified between *PrimitiveModels* belonging to the same *CompositeModel*.
- An *EvolutionTrace* can only be specified between different versions of the same *CompositeModel*, or between different versions of the same *PrimitiveModel*.
- With respect to two models participating in an *EvolutionTrace*, the version number of the *client* model must be greater than that of the *supplier* model.

# Chapter 4

# Experimental Setup

In this chapter, the environment needed for the experimentation process is described. In the first section, the chosen DL tool, *Loom 4.0*, is discussed. The DL tool will be used as an inference engine, storing user-defined models as instances of the UML metamodel. In second section, the chosen UML CASE tool, *Poseidon For UML CE 1.6*, is discussed. The user-defined models will be developed in this graphical environment, which saves model data as XMI files. Finally, in section three, the *SAXON 6.5.2* package is discussed. This package is a collection of tools for processing XML documents. This will be used for translating the UML metamodel and the user-defined models into *Loom* syntax, and vice-versa. The proposed tool interaction can be seen in figure 4.1.

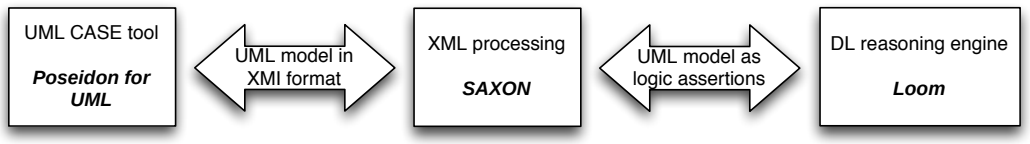


Figure 4.1: Proposed interaction between the tools used in the experimentation process

## 4.1 DL Tool - *Loom 4.0*

The experiments in this dissertation involve the detection of inconsistencies at the UML model level, using Description Logic. As a consequence of this requisite, and after evaluating various existing tools (see Section 2.2) *Loom*[13, 39, 40] was chosen as the inference engine. The main motivation for this choice is the powerful assertional reasoning facility it provides, in the form of a query and retrieval mechanism. The UML metamodel will provide the main concept definitions of the *Tbox* (see Section 2.1). The *Abox* will be populated by individuals derived from user-defined UML models.

The conjunction of the *Tbox*, *Abox* and the query mechanism will allow the query and manipulation of models. This will permit the development of detection predicates for the different inconsistencies defined at the UML level. A quick overview of Lisp functions and macros, and *Loom*'s concept definition and query language is given in this section. As an example of the concepts and relations used in the *Tbox*, the subset of the UML profile representing the class

diagrams is translated into the corresponding *Loom* syntax. As an example of the individuals that populate the *Abox*, a user-defined class diagram is translated into the corresponding *Loom* syntax. This will allow a better understanding of the experiments presented in Chapter 6.

### 4.1.1 Lisp Overview

*Loom* is implemented in Lisp[25], and allows the use of Lisp functions, macros and keywords in queries and predicates. In this section, a quick overview of the main functions and macros used in the experimentation process, is given. Special forms are generally exceptions to normal Lisp syntax that make coding easier and more convenient for the programmer. They are expressions that do not follow the normal rules of evaluation.

Functions are defined using the `defun` special form, that receives the function name, argument list and body as arguments. Another special form, `setq` is used to set variables in Lisp. `progn` is a special form that stands for "program with no arguments". It takes a number of expressions and groups them all together in to a single expression that returns the value of the last expression in the group. This is useful for writing if statements that execute more than one expression at a time. The `let` special form is used to introduce local variables. A local variable exists only for the duration of its scope, which in Lisp is the length of the `let` statement. This form is sensitive to order of evaluation, a problem that can be avoided by using `let*`, which is less efficient than `let`, but is not sensitive to order of evaluation.

The general printing function `format` is used. `format` takes at least two parameters, an output stream and a string. If the string contains formatting directives, additional parameters might be needed. When the first parameter is `T`, the result is printed to the standard output. Formatting directives all begin with `~`, like new line (`~%`) and symbol (`~s`). An additional argument has to be given for each `~s` directive, and is taken from the argument list (in order of directive) and printed in the output.

The `equalp` function is a more flexible version of equal. In particular, it compares strings case-insensitively, and it compares numbers without regard to type. Vectors are compared recursively. All other objects are compared as if by equal. Various list manipulation functions are used, specially `car` and `cdr`, that returns the value pointed to by the first pointer of the list, and returns the value pointed ti by the second pointer of the list, respectively.

The functions `first`, `second`, `third` and `fourth` access the first, second, third and fourth elements of a list, respectively. The `sort` function sorts a list stably, though destructively, and returns the sorted list. It establishes the order of the elements by comparing them using a predicate given as an argument. This predicate must be a binary function and must give reliable results for any given pair of arguments, at least within a single call to sort.

### 4.1.2 Concept and Relation Definition

*Loom*'s concept definition language belongs to the $\mathcal{AL}$ family of languages. As such, it supports the definition of concepts and relations, has "top" and "bottom" concepts, concept negation, concept conjunction, and universal and existential quantifiers. According to the constructs presented in Table 4.1, the extensions to this DL classify it as $\mathcal{ALCQRIFO}$. This same table shows the corresponding *Loom* syntax for each DL construct that is a part of the concept definition language offered by this tool.

By looking at the syntax for qualified number restrictions, it becomes visible that it is also

| **Constructor** | **DL syntax** | **Loom syntax** |
|---|---|---|
| concept name | $C$ | `(defconcept C)` |
| top | $\top$ | `\|C\|THING` |
| negation ($\mathcal{C}$) | $\neg C$ | `(defconcept D :is (:not C))` |
| conjunction | $C_1 \sqcap C_2$ | `(defconcept D :is (:and C1 C2))` |
| disjunction ($\mathcal{U}$) | $C_1 \sqcup C_2$ | `(defconcept D :is (:or C1 C2))` |
| universal quant. | $\forall R.C$ | `(defconcept D :is (:all R C))` |
| existential quant. ($\mathcal{E}$) | $\exists R.C$ | `(defconcept D :is (:some R C))` |
| qualified number restr. ($\mathcal{Q}$) | $(\geq n\ R.C)$ | `(defconcept D :is (:and (:at-least n R) (:all R C)))` |
| | $(\leq n\ R.C)$ | `(defconcept D :is (:and (:at-most n R) (:all R C)))` |
| functional number restr. ($\mathcal{F}$) | $(\geq 2\ R)$ | `(defconcept D :is (:at-least 2 R))` |
| | $(\leq 1\ R)$ | `(defconcept D :is (:at-most 1 R))` |
| one-of ($\mathcal{O}$) | $\{a_1, \ldots, a_n\}$ | `(defconcept D :is (:one-of 'A1 ... 'An))` |
| role name | $R$ | `(defrelation R)` |
| role conjunction ($\mathcal{R}$) | $R_1 \sqcap R_2$ | `(defrelation S :is (:and R1 R2))` |
| inverse roles ($\mathcal{I}$) | $R^{-1}$ | `(defrelation S :is (:inverse R))` |

Table 4.1: DL constructs and the equivalent Loom syntax

possible to represent unqualified number restrictions, by omitting the `(:all R C)` predicate in the concept definition. This construct was not used in the *Tbox* definition used in the experimentation process, as the UML metamodel did not present ambiguities that would have made it indispensable.

An equivalent manner of defining the qualified number restrictions is by setting certain relation characteristics: domain and range. This approach has been preferred by the author, as neater, more compact concept definitions that yield the same subsumption relations when the *Tbox* definitions are classified, can be written. For example, `(defconcept C (:and (:at-least 1 R) (:all R D))) (defrelation R)` can be written as `(defconcept C (:at-least 1 R)) (defrelation R :domai`
Also reducing the size of concept definitions, the predicate `(:exactly n R)` is syntactic sugar for the longer `(:and (:at-least n R) (:at-most n R))`. In addition to the `:all` and `:some` quantifiers, the `:the` keyword can be used, as shorthand for stating the fact that there is exactly one filler (`:some` only specifies that there is at least one).

Both concept and relation definitions can include first-order logic expressions, using the `:satisfies` keyword, that receives a first-order expression in the form of a query expression. In the case of concept definitions, if `C = (:satisifes (?X) Q)`, and `I` is an instance of `C`, then the *Loom* query `Q` is satisfied when the variable `?X` is bound to `I`. In the case of relation definitions, if `R = (:satisfies (?X1...?Xn) Q)`, and $< I1, ..., In >$ is an instance of `R`, then the *Loom* query `Q` is satisfied when the variables `?X1...?Xn` are bound (in order) to `I1...In`. The use of this keyword has been avoided, as reasoning with respect to these predicate calculus-based expressions is not complete. Completeness is necessary when inconsistencies are being detected, so as to assure that the model as a whole is consistent.

The following example defines an elementary *Tbox* that represents some concepts associated to people:

```
(defrelation gender :characteristics single-valued :domain Person :range Gender)
```

```
(defrelation age :is (:and Integer) :characteristics single-valued)
(defrelation wife :domain Man :range Woman)
(defrelation husband :inverse wife)
(defrelation child :domain Parent :range Person)
(defrelation daughter :domain Parent :range Female)
(defrelation son :domain Parent :range Male)
(defrelation pet :domain Person :range Pet)

(defconcept Person :roles (gender age))
(defconcept Adult)
(defconcept Gender :partitions $Gender$)
(defconcept Female :in-partition $Gender$)
(defconcept Male :in-partition $Gender$)
(defconcept Woman :is (:and Person Adult (:the gender Female)))
(defconcept Man :is (:and Person Adult (:the gender Male)))
(defconcept Parent :is (:and Person (:some child Person)))
(defconcept Mother :is (:and Woman Parent))
(defconcept Father :is (:and Man Parent))
(defconcept Teacher :is (:and Person Adult))
(defconcept Pet)
```

Age and gender are defined as attributes of the Person concept, each allowing only one value. Wife, husband, child, daughter, son and pet are modelled as relationships between concepts, that are specified as the domain and range. Gender defines a disjoint partition, which only has the concepts Male and Female as participants. The rest of the concepts are defined as conjunctions of the previously defined concepts.

### 4.1.3 Individual Creation and Description Macros

Individuals are created with the `(create identifier concept)` function. The `identifier` argument is a symbol or string which identifies the new instance. If the identifier is `NIL`, the individual is given a unique system-generated identifier. The `concept` argument is the concept, or the name of the concept, to be used as the creation type of the new instance. When no concept is supplied, the creation type is `Thing`. Assertions about the individuals are then made using the `tellm` macro.

The `(tellm propositions)` macro asserts one or more propositions, and then calls the *Loom* matcher to recompute the types of the affected instances. Each proposition in the `propositions` argument above has the following syntax:

```
proposition ::=
( concept instance ) |
( relation instance+ value ) |
( :CREATE ?Var concept [:CLOS] ) |
( :SAME-AS instance instance ) |
( :ABOUT instance about-clause* )
```

The `[:CLOS]` switch present in the `create` form above informs *Loom* as to whether it should look in the *Tbox* for the concept parameter, or if a Common Lisp Object System (CLOS) class should be looked up.

Each about-clause in a proposition has the form:

```
about-clause ::=
concept |
( concept ) |
( relation value ) |
( :FILLED-BY relation value+ ) |
( :FILLED-BY-LIST relation List ) |
( {:AT-LEAST | :AT-MOST | :EXACTLY} Integer relation ) |
( {:ALL | :SOME | :THE} relation concept )
```

The following examples show some elementary combinations of propositions, that create individuals that belong to the universe defined in the example *Tbox* of Section 4.1.2[1]:

```
(tellm (Person Joe))
(tellm (age Joe 30) (wife Joe Sue) (Teacher Sue))
(tellm (age (wife Joe) (+ (age Joe) 1)))
(tellm (Person Honey) (:same-as (wife Joe) Honey))
(tellm (:create ?X Dog) (pet Joe ?X))
(tellm (:about Mark Person (wife Jane)))
(tellm (:about Mark (child Fred) (child Mary)))
(tellm (:about Mark (:filled-by child Fred Mary Don)))
(tellm (:about Mark (:at-least 3 child) (:at-most 1 daughter) (:exactly 2 son)))
```

The `(forgetm propositions)` macro is used to retract propositions from the *Loom* database. These propositions may state that an instance belongs to a given concept or has a particular feature, or that a certain relationship holds between two or more instances. Each proposition in the propositions argument above has the following syntax:

```
proposition ::=
( concept instance ) |
( relation instance+ value ) |
( :CREATE ?Var concept [:CLOS] ) |
( :ABOUT instance about-clause* )
```

Each about-clause in a proposition has the form:

```
about-clause ::=
concept | ( concept ) | ( relation value ) |
( :FILLED-BY relation value+ ) |
( {:AT-LEAST | :AT-MOST | :EXACTLY} Integer relation ) |
( {:ALL | :SOME | :THE} relation concept )
```

The `forgetm` macro acts as the inverse of the `tellm` macro. As such, the propositions are constructed in the same way. For example, to undo the assertion `(tellm (Person Joe))`, the assertion `(forgetm (Person Joe))` has to be executed. When it is necessary to retract all the fillers for a role, the `*` symbol can be used. For example, `(forgetm (child Joe *))` retracts all the fillers of the `child` role, for the individual `Joe`. In order to partially retract fillers, the specific *Loom* identifications of these individuals should be known.

All the macros used in this section will automatically advance to a new knowledge base state. In addition to this, all *Aboxes* defined during the course of the experimentation process have,

[1] Asserting the same fact more than once in a knowledge base will have no effect on the existing state of the knowledge base

as their creation policy, `:classified-instance`. This means that all instances are classified as soon as possible. As a result, all knowledge bases automatically advance state and incorporate all assertions immediately, updating the necessary individuals.

### 4.1.4 Query Language

The query and retrieval mechanism offered by *Loom* is very powerful. Expressions of arbitrary complexity can be formed by using certain operators. The query processor evaluates these expressions, and returns a list of instances or tuples that satisfy them. The inference capabilities of the query processor have been limited by the system architects, in order to gain performance speed. If the classifier is enabled, it discovers the inferences missed by the query processor.

Two data-retrieval macros are available: `(do-retrieve variables query body)` and `(retrieve variables query-expr)`. The `variables` argument is a symbol, or list of symbols, each beginning with the character `?`. The `query` argument is an expression in the Loom query language. This language has the expressive power of the first-order predicate calculus. The difference between both macros is that the `do-retrieve` macro executes a `body` of Lisp code once for each instance or tuple retrieved by a given query.

The query expressions appearing in the `query` argument above have the following syntax:

```
query-expr ::=
( {:AND | :OR} query-expr+ ) |
( {:NOT | :FAIL} query-expr ) |
( :IMPLIES query-expr query-expr ) |
( {:FOR-SOME | :FOR-ALL} ( ?Var+ ) query-expr ) |
( :COLLECT ( ?Var ) query-expr ) |
( concept instance ) |
( relation instance+ value ) |
( :SAME-AS instance instance ) |
( :SUBSET instance instances ) |
( :PREDCALL LispPredicate value+ ) |
( :ABOUT instance about-clause* )
```

Each about-term in a query `:about` clause has the form:

```
about-clause ::=
concept |
( concept ) |
( relation value ) |
( :FILLED-BY relation value+ ) |
( {:AT-LEAST | :AT-MOST | :EXACTLY} Integer relation ) |
( {:ALL | :SOME | :THE} relation concept )
```

The following examples show some queries relative to the *Tbox* and *Abox* given as examples in Sections 4.1.2 and 4.1.3, respectively:

```
(retrieve ?x (age Joe ?x))
(retrieve ?x (max (age (child Joe)) ?x))
(retrieve (?x ?y) (wife ?x ?y))
(retrieve ?x (:and (pet Joe ?x) (:not (Dog ?x))))
(do-retrieve (?x ?y) (wife ?x ?y) (format t "~S is married to ~S~%" ?x ?y))
```

### 4.1.5 Tbox

In figure 4.2, the subset of the UML profile that specifies the properties of the class diagrams is shown[2]. The complete translation of this subset into *Loom* syntax is detailed in this section. The translation of the rest of the UML profile can be found in Appendix A.1. The UML profile has been translated by package (see Chapter 3), with each metaclass translated as a concept, each association translated as a relation and each attribute translated as a role of the concept that models the corresponding metaclass. Inheritance is modelled through the subsumption relationship, and also by establishing disjoint partitions. The set of OCL well-formedness rules associated to each subset of the UML profile has also been specified in *Loom*, using logic rules written in a combination of *Lisp* and *Loom*. The concept definitions of elements that appear in more than one subset are later combined into single concept definitions.

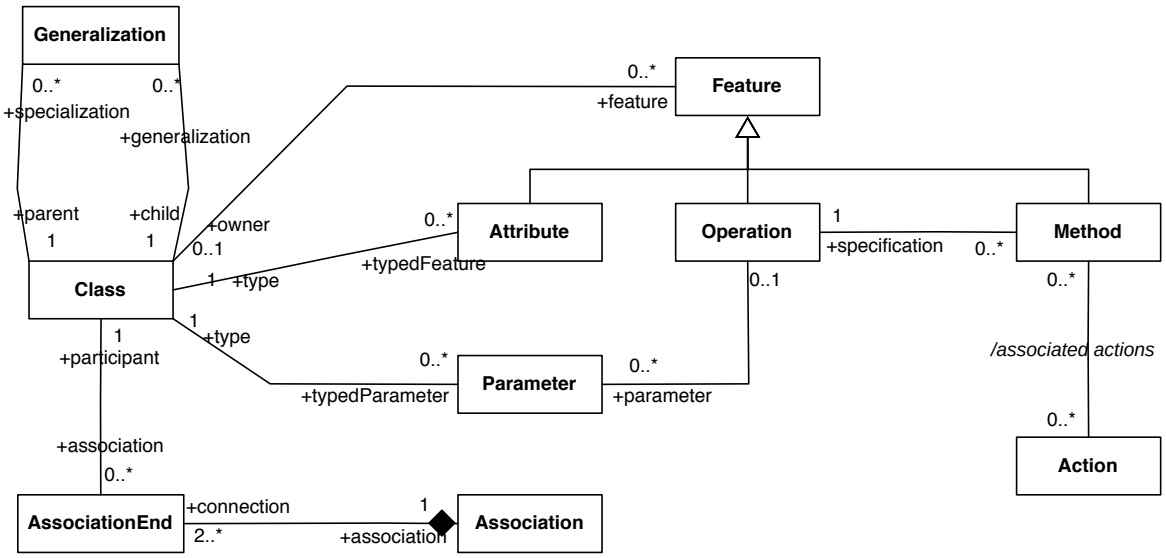


Figure 4.2: Subset of the UML *Core Package*

#### Network Initialisation

The creation of a knowledge base is known as the "network initialization" in *Loom*. This terminology derives from the type of subsumption algorithm used by *Loom*, which is structural, based on graphs. Thus, before facts can be asserted, the graph network has to be initialised. A package must always be created for a knowledge base. Then, *Loom* is instructed to use this newly created package for storing the *Abox* and *Tbox* assertions. In this case, the name of the package is "UML".

Various features can be set and unset for a knowledge base. The following list describes the options used during the experimentation process, but a comprehensive list can be can be found in the *Loom* manual[13] and site[35]:

- If the `:display-match-changes` feature is set, whenever a production memory match or

[2]Although it is not visible in this diagram, all the classes are descendants of the *ModelElement* class.

instance classification occurs, *Loom* indicates which concepts have entered or exited the type of each affected instance.

- If the `:emit-match-stars` feature is set, an asterisk is printed whenever the type of an instance changes. Also, *Loom* prints a `T` whenever a new type is created as a result of unifying a newly-matched concept with the concepts already in an instance's type.
- If the `:emit-classify-dots` feature is set, a dot, minus, or plus is printed when a concept is classified, unclassified, or completed, respectively, and an exclamation mark is printed when a concept is sealed (i.e., ready to participate in data manipulation operations). Also, the character `M` is printed when concepts merge during classification, `U` is printed when they unmerge, and `C` is printed when a CLOS class is created.
- If the `:ask-for-concept-names` feature is set, *Loom* prompts the user for concept names whenever system-defined concepts are generated. The prompting is done after the new concepts have been completed. If the user declines to supply a name, the concept is assigned a system-generated name.

A instance creation policy must be chosen for the knowledge base. This is done by calling the `creation-policy` function. Legal policy values are `:classified-instance` - classifies instances as soon as possible, `:lite-instance` - classifies instances when forced by the user and `:clos-instance` - classifies instances as CLOS objects. The `initialize-network` function clears all knowledge bases by destroying the concepts, relations, instances, and behaviours that they contain. It also rebuilds the top-level knowledge base, populating it with built-in concepts such as `Thing` and `Constant`. The following initialisation settings were used during the experimentation process:

```
(defpackage "UML" (:use "COMMON-LISP"))
(in-package "UML")
(LOOM:use-loom "UML")
(set-features :display-match-changes :emit-match-stars :emit-classify-dots)
(unset-features :ask-for-concept-names)
(creation-policy :classified-instance)
(initialize-network)
```

### Relation definition

- Attribute *name*, belongs to the *ModelElement* metaclass
  `(defrelation name :is (:and String) :characteristics :single-valued :domain ModelElement :range String)`
- Attribute *the-prev-ver*, belongs to the *ModelElement* metaclass
  `(defrelation the-prev-ver :characteristics :single-valued :domain ModelElement :range ModelElement)`
- Attribute *kind*, belongs to the *Parameter* metaclass
  `(defrelation kind :characteristics :single-valued :domain Parameter :range String)`
- Attribute *lower*, belongs to the *MultiplicityRange* metaclass
  `(defrelation lower :is (:and Integer) :characteristics :single-valued :domain MultiplicityRange :range Integer)`

- Attribute *upper*, belongs to the *MultiplicityRange* metaclass
```
(defrelation upper :is (:and String) :characteristics :single-valued :domain
 MultiplicityRange :range String)
```
- Association between *Action* and *Method* metaclasses
```
(defrelation Action-method :domain Action :range Method)
(defrelation Method-action :is (:inverse Action-method))
```
- Association between *Operation* and *Method* metaclasses
```
(defrelation Is-specification-of :domain Operation :range Method)
(defrelation Has-specification :is (:inverse Is-specification-of))
```
- Association between *Parameter* and *Operation* metaclasses
```
(defrelation Has-parameter :domain Operation :range Parameter)
(defrelation Is-parameter-of :is (:inverse Has-parameter))
```
- Association between *Class* and *Parameter* metaclasses
```
(defrelation Parameter-class :domain Parameter :range Class)
(defrelation Class-parameter :is (:inverse Parameter-class))
```
- Association between *Class* and *Attribute* metaclasses
```
(defrelation Is-of-type :domain Attribute :range Class)
(defrelation Has-typed-feature :is (:inverse Is-of-type))
```
- Association between *Class* and *Feature* metaclasses
```
(defrelation Has-feature :domain Class :range Feature)
(defrelation Is-owned-by :is (:inverse Has-feature) :characteristics
 :single-valued)
```
- Association between *Class* and *Generalization* metaclasses
```
(defrelation Is-generalization-of :domain Generalization :range Class)
(defrelation Is-child-of :is (:inverse Is-generalization-of))
```
- Association between *Class* and *Generalization* metaclasses
```
(defrelation Is-specialization-of :domain Generalization :range Class)
(defrelation Is-parent-of :is (:inverse Is-specialization-of))
```
- Association between *Class* and *AssociationEnd* metaclasses
```
(defrelation Is-participant-of :domain Class :range AssociationEnd)
(defrelation Has-participant :is (:inverse Is-participant-of))
```
- Association between *Association* and *AssociationEnd* metaclasses
```
(defrelation Association-associationEnd :domain Association :range
 AssociationEnd)
(defrelation AssociationEnd-association :is (:inverse Association-
associationEnd))
```
- Association between *Multiplicity* and *MultiplicityRange* metaclasses
```
(defrelation Has-range :domain Multiplicity :range MultiplicityRange)
(defrelation Range-of :is (:inverse Has-range))
```
- Association between *ModelElement* and *Multiplicity* metaclasses
```
(defrelation Has-multiplicity :domain ModelElement :range Multiplicity)
```

**Concept definition**

- *ModelElement* metaclass
```
(defconcept ModelElement
 :roles (name the-prev-ver))
(tell (:about ModelElement (:at-most 1 In-namespace)))
```
- *MultiplicityRange* metaclass
```
(defconcept MultiplicityRange
 :is-primitive (:and
  (:exactly 1 Range-of))
 :roles (lower upper))
```
- *Multiplicity* metaclass
```
(defconcept Multiplicity
 :is-primitive (:and
  (:at-least 1 Has-range)
  (:all Has-range MultiplicityRange)))
```
- *Action* metaclass
```
(defconcept Action
 :is (:and ModelElement)
 :roles (isAsynchronous recurrence target))
```
- *Class* metaclass
```
(defconcept class
 :is (:and ModelElement
  (:all Instantiated Object)
  (:all Is-child-of Generalization)
  (:all Is-parent-of Generalization)
  (:all Is-participant-of AssociationEnd))
 :roles (isAbstract))
```
- *Feature* metaclass
```
 (defconcept Feature
 :is (:and ModelElement
  (:at-most 1 Is-owned-by))
 :partitions $Feature$
 :roles (ownerScope visibility))
```
- *Operation* metaclass
```
(defconcept Operation
 :is (:and Feature
  (:all Has-specification Method)
  (:all Has-parameter Parameter))
 :in-partition $Feature$
 :roles (concurrency isAbstract isLeaf isRoot isQuery))
```
- *Method* metaclass
```
(defconcept Method
 :is (:and Feature
```

```
  (:exactly 1 Is-specification-of))
 :in-partition $Feature$
 :roles (body isQuery))
```

- *Attribute* metaclass

```
(defconcept Attribute
 :is (:and Feature
  (:all Is-of-type Class)
  (:exactly 1 Is-of-type)
  (:exactly 1 Has-multiplicity))
 :in-partition $Feature$
 :roles (initialValue changeability ordering targetScope))
```

- *Parameter* metaclass

```
(defconcept Parameter
 :is (:and ModelElement
  (:at-most 1 Is-parameter-of)
  (:all Parameter-class Class)
  (:exactly 1 Parameter-class))
 :roles (defaultValue kind))
```

- *Generalization* metaclass

```
(defconcept Generalization
 :is (:and ModelElement
  (:exactly 1 Is-generalization-of)
  (:exactly 1 Is-specialization-of))
 :roles (discriminator))
```

- *AssociationEnd* metaclass

```
(defconcept AssociationEnd
 :is (:and ModelElement
  (:exactly 1 Has-participant)
  (:exactly 1 AssociationEnd-association)
  (:exactly 1 Has-multiplicity))
 :roles (aggregation ordering isNavigable visibility))
```

- *Association* metaclass

```
(defconcept Association
 :is (:and ModelElement
  (:all Association-associationEnd AssociationEnd)
  (:at-least 2 Association-associationEnd)))
```

### 4.1.6 Abox

#### Model Example

User-defined UML models must be translated into *Loom* individuals. Each element of these models must be specified as an instance of the appropriate concept, filling the corresponding associations and attributes of the UML metamodel. In this section, a partial translation of the

class diagram presented in figure 4.3 is given. Full examples of translations of class, state and sequence diagrams can be seen in Appendix A.2.

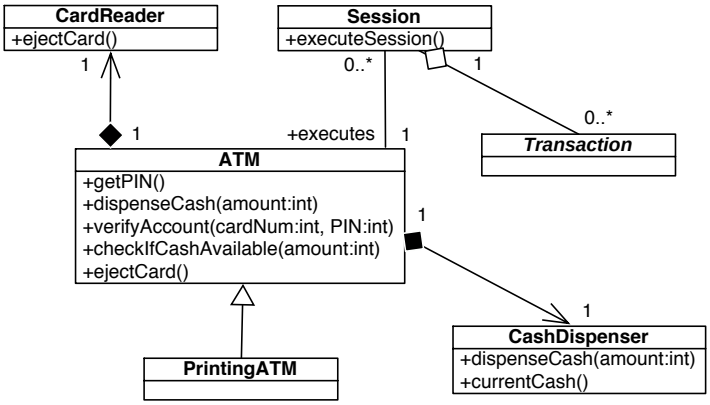


Figure 4.3: Class diagram example for UML to *Loom* translation

- Instances of the *MultiplicityRange* metaclass

```
(tellm (:about mr1-1.0 (lower 1) (upper 1)))
(tellm (:about mr2-1.0 (lower 1) (upper 1)))
```

- Instances of the *Multiplicity* metaclass

```
(tellm (:about m1-1.0 (Has-range mr1-1.0)))
(tellm (:about m2-1.0 (Has-range mr2-1.0)))
```

- Instance of the *ClassModel* metaclass

```
(tellm (:about Class-Diagram-1.0 (version 1.0)))
```

- Instances of the *Class* metaclass

```
(tellm (:about ATM-1.0
 (name ATM)
 (isAbstract false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about PrintingATM-1.0
 (name PrintingATM)
 (isAbstract false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about CardReader-1.0
 (name CardReader)
 (isAbstract false)
 (In-namespace Class-Diagram-1.0)))
```

- Instance of the *Parameter* metaclass

```
(tellm (:about amount-dispenseCash-ATM-1.0
 (name amount)
 (Is-parameter-of dispenseCash-ATM-1.0)
 (Parameter-class int)
 (In-namespace Class-Diagram-1.0)))
```

- Instances of the *Operation* metaclass

```
(tellm (:about getPIN-ATM-1.0
 (name getPIN)
 (Is-owned-by ATM-1.0)
 (IsAbstract false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about dispenseCash-ATM-1.0
 (name dispenseCash)
 (Is-owned-by ATM-1.0)
 (IsAbstract false)
 (In-namespace Class-Diagram-1.0)))
```

- Instance of the *Generalization* metaclass

```
(create 'ATM-PrintingATM-1.0 'Generalization)
(tellm (:about ATM-PrintingATM-1.0
 (name ATM-PrintingATM)
 (Is-generalization-of PrintingATM-1.0)
 (Is-specialization-of ATM-1.0)
 (In-namespace Class-Diagram-1.0)))
```

- Instances of the *AssociationEnd* metaclass

```
(tellm (:about ATM-end1-1.0
 (name ATM-end1)
 (Has-multiplicity m1-1.0)
 (Has-participant ATM-1.0)
 (aggregation composite)
 (isNavigable false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about CashDispenser-end1-1.0
 (name CashDispenser-end1)
 (Has-multiplicity m2-1.0)
 (Has-participant CashDispenser-1.0)
 (aggregation none)
 (isNavigable true)
 (In-namespace Class-Diagram-1.0)))
```

- Instance of the *Association* metaclass

```
(tellm (:about CashDispenser-ATM-1.0
 (name CashDispenser-ATM)
 (In-namespace Class-Diagram-1.0)
 (Association-associationEnd CashDispenser-end1-1.0)
 (Association-associationEnd ATM-end1-1.0)))
```

### Model Management Example

The model and trace information (see Section 3.3), which is not present in the user-defined models as this is not part of the UML metamodel, but part of the extension proposed for dealing with model evolution and consistency, has to be added manually. In this section, a translation of the model and trace information presented in figure 4.4 is given.

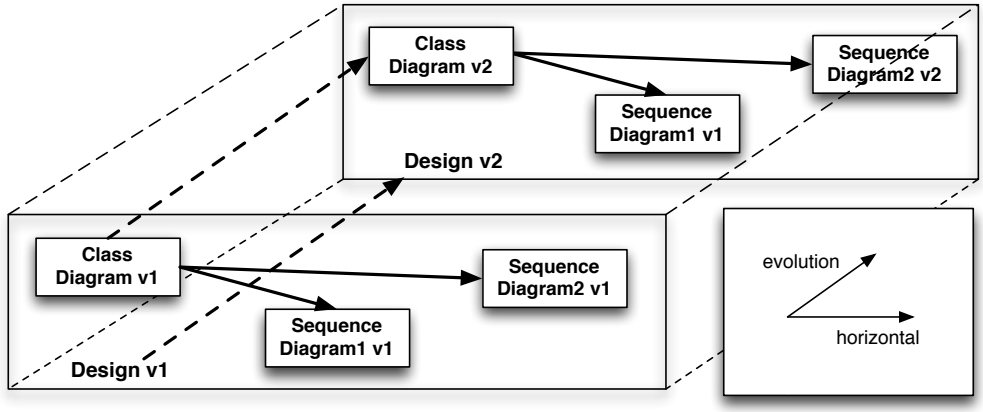


Figure 4.4: Model Evolution Example

- Instances of the *CompositeModel* metaclass

```
(tellm (:about Design1
 (name Design)
 (version 1)
 (Contains-model Class-Diagram-1)
 (Contains-model Sequence-Diagram1-1)
 (Contains-model Sequence-Diagram2-1)))
(tellm (:about Design2
 (name Design)
 (version 2)
 (Contains-model Class-Diagram-2)
 (Contains-model Sequence-Diagram1-2)
 (Contains-model Sequence-Diagram2-2)))
```

- Instances of the *EvolutionTrace* metaclass

```
(tellm (:about D1-D2
 (name D1-D2)
 (Supplier Design1)
 (Client Design2)))
(tellm (:about CD1-CD2
 (name CD1-CD2)
 (Supplier Class-Diagram-1)
 (Client Class-Diagram-2)))
```

- Instances of the *HorizontalTrace* metaclass

```
(tellm (:about CD1-SD1-1
 (name CD1-SD1-1)
 (Supplier Class-Diagram-1)
 (Client Sequence-Diagram1-1)))
(tellm (:about CD1-SD2-1
 (name CD1-SD2-1)
 (Supplier Class-Diagram-1)
```

```
 (Client Sequence-Diagram2-1)))
(tellm (:about CD2-SD1-2
 (name CD2-SD1-2)
 (Client Class-Diagram-2)
 (Supplier Sequence-Diagram1-2)))
(tellm (:about CD2-SD2-2
 (name CD2-SD2-2)
 (Client Class-Diagram-2)
 (Supplier Sequence-Diagram2-2)))
```

## 4.2 UML CASE Tool - *Poseidon For UML CE 1.6*

*Poseidon for UML*[26, 9] is a commercial tool, that has its origins in the ArgoUML[4] open-source project. However, *Poseidon for UML* is more mature, provides additional features and is more stable. This software suite has various *editions*, each one targeted at different audiences, so each edition offers a different set of functionalities. The edition used in this study is the Community Edition (CE). This is the base version, with respect to the functionality offered and is available without cost.

This edition is fully usable for modelling UML 1.4, is not restricted to non-commercial projects or to evaluation periods. It contains all UML diagrams and all implemented diagram elements. Projects can be created, saved and loaded, existing models can be browsed and even Java code can be generated. The main restrictions of this tool is that printing and copy-and-paste to external applications is not supported. Also, this edition can not load plug-ins. This is available in other editions, like the Standard and Professional editions.

*Poseidon for UML* supports open standards extensively. This is true for the saving format as well. UML is standardised by the Object Management Group (OMG)[49]. Part of the official UML specification by the OMG is a mechanism for the exchange of models between different tools. This mechanism is based on XML and has special extensions and rules to better represent object-oriented structures as well as metadata. The OMG has specified a concrete application of XML for this purpose, that is called the XML Metadata Interchange, or simply XMI for short.

This version of the tool makes use of the latest version of this format (XMI 1.2). While most other tools can only import or export XMI, *Poseidon for UML* uses XMI as default saving and loading mechanism. The current version of XMI is by itself not sufficient to save all aspects of an UML model. It can be used to transport the names and properties of all model elements, but the diagram information (layout, colours etc.) is currently not included. This is included in *Poseidon For UML*-specific files.

This XMI support makes *Poseidon For UML* the ideal tool for the experiments that are carried out in this study. Using XML manipulation tools, the translation of user-defined models into other formats, like *Loom* code, can be done automatically. *Poseidon for UML* is known to produce one of the cleanest XMI files for its models, and many tools have chosen to support its variant of XMI.

## 4.3 Additional tools - *Saxon 6.5.2*

The SAXON[36] package is a collection of tools for processing XML documents. The main components are:

- An XSLT[63] processor, which implements the Version 1.0 XSLT and XPath[62] Recommendations from the World Wide Web Consortium[61], with a number of powerful extensions.
- A Java library, which supports a similar processing model to XSL, but allows full programming capabilities, that are necessary if complex data processing or access external services such as relational databases, will be performed.
- A slightly improved version of the Ælfred parser from Microstar, but *SAXON* can be used with any SAX-compliant XML parser.

*SAXON* provides a set of services that are particularly useful when converting XML data into other formats. The output format may be XML, or HTML, or some other format such as data in a relational database. In the specific case of the experiments carried out in this study, it is necessary to convert XMI files, that contain the names and properties of all model elements of user-defined models, into logic assertions. This will allow the automated population of the DL knowledge bases, which can later be queried so as to detect inconsistencies. This would make the approach used in the tool support transparent to any user of the detection system, as knowledge of logic rules and predicates would not be necessary.

# Chapter 5

# Case Study - The Banking Simulation

In this chapter, the case study used to provide examples for the experiments, is detailed. This design of an ATM simulation is an adapted version of the ATM simulation example by Russell C. Bjork [7]. The static structure of the system is documented by means of three class diagrams that combined, give the overall structure of the system. The dynamic behaviour of the controlling objects of the system is documented by means of state diagrams for the classes of the ATM hierarchy. The rest of the behaviour specified here, is documented as sequence diagrams.

## 5.1 Static Structure

The class diagrams shown in figures 5.1, 5.2 and 5.3, combined, give the overall structure of the banking system described in this chapter. The details of the class interfaces can be see in Appendix B.

## 5.2 Dynamic Behaviour

The state and sequence diagrams shown in this section describe portions of the behaviour of the banking system described in this chapter. As in the case of the static system description, this documentation is not absolute - various diagrams have been simplified because of space and clarity reasons. The behaviour described in this section is limited to what is needed to the examples that will be given in Chapters 6 and 7.

The state diagram for the *ATM* class is shown in figure 5.4. This state diagram presents a summary of the behaviour of an ATM machine, when a customer chooses to withdraw money from his account. Supposing that the client has inserted a valid ATM card, he is prompted to enter the associated PIN number. If the PIN is not valid, the card is returned to the user. If a valid PIN has been entered, the ATM prompts the user to enter the amount to withdraw from his account. First, the ATM checks that the client's account has sufficient funds. If so, the ATM proceeds to check if it can locally dispense this amount. Once these checks have been passed, the ATM dispenses the money and ejects the card. If either the account, or the

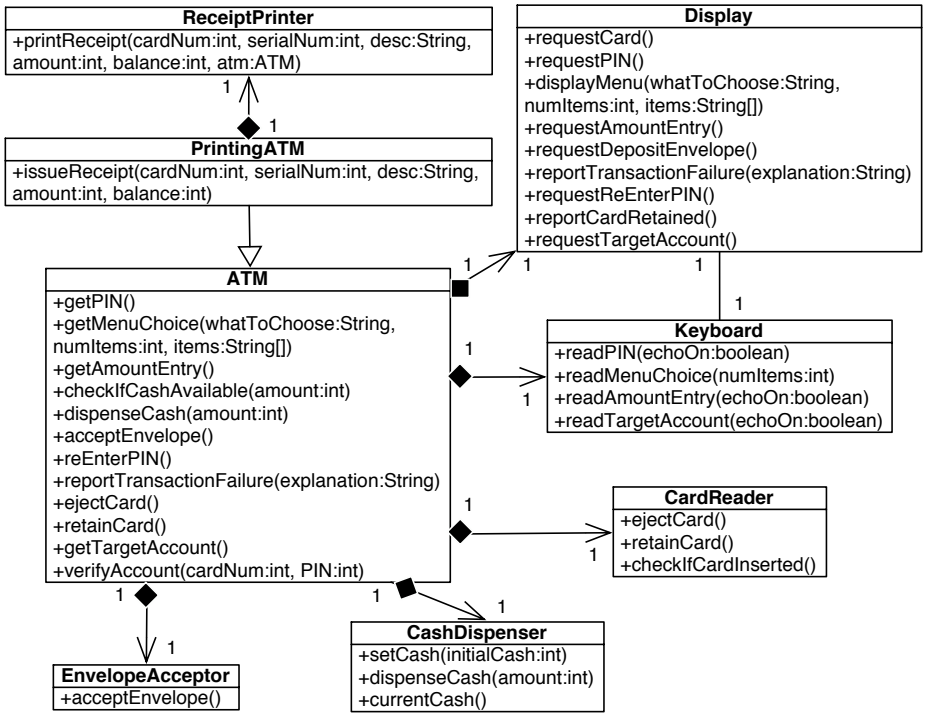


Figure 5.1: Class Diagram - Part I, version 1

ATM, do not have sufficient funds for completing the withdrawal, the card is returned (these transitions are not shown).

A second state diagram for the *ATM* class is shown in figure 5.5. This state diagram presents a summary of the behaviour of an ATM machine, when a customer chooses to withdraw money from his account, and includes the behaviour that results when the user repeatedly enters an incorrect PIN number. Supposing that the client has inserted a valid ATM card, he is prompted to enter the associated PIN number. If the PIN is not valid, the user is prompted again for his PIN number. He has a maximum of 3 tries, and if these are used, the card is retained instead of returned. If a valid PIN has been entered, the behaviour continues as that described for the previous diagram.

The state diagram for the *PrintingATM* class is shown in figure 5.6. This state diagram presents a summary of the behaviour of a PrintingATM machine, when a customer chooses to withdraw money from his account. Supposing that the client has inserted a valid ATM card, he is prompted to enter the associated PIN number. If the PIN is not valid, the card is returned to the user. If a valid PIN has been entered, the ATM prompts the user to enter the amount to withdraw from his account. First, the ATM checks that the client's account has sufficient funds. If so, the ATM proceeds to check if locally, it can dispense this amount. Once these checks have been passed, the ATM dispenses the money. The *PrintingATM* class, unlike its parent the *ATM* class, can print receipts, and the user must now indicate if he wants one for the current transaction. Afterwards, the card is ejected. If either the account, or the ATM, do not have sufficient funds for completing the withdrawal, the card is returned (these transitions are not shown).

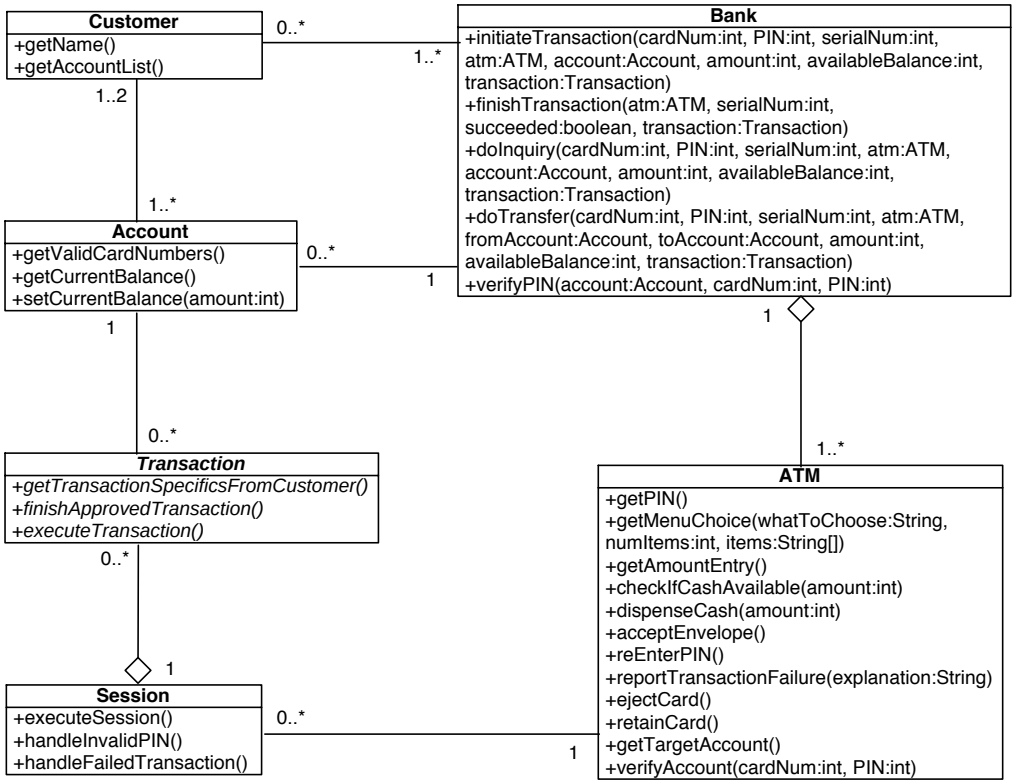


Figure 5.2: Class Diagram - Part II, version 1

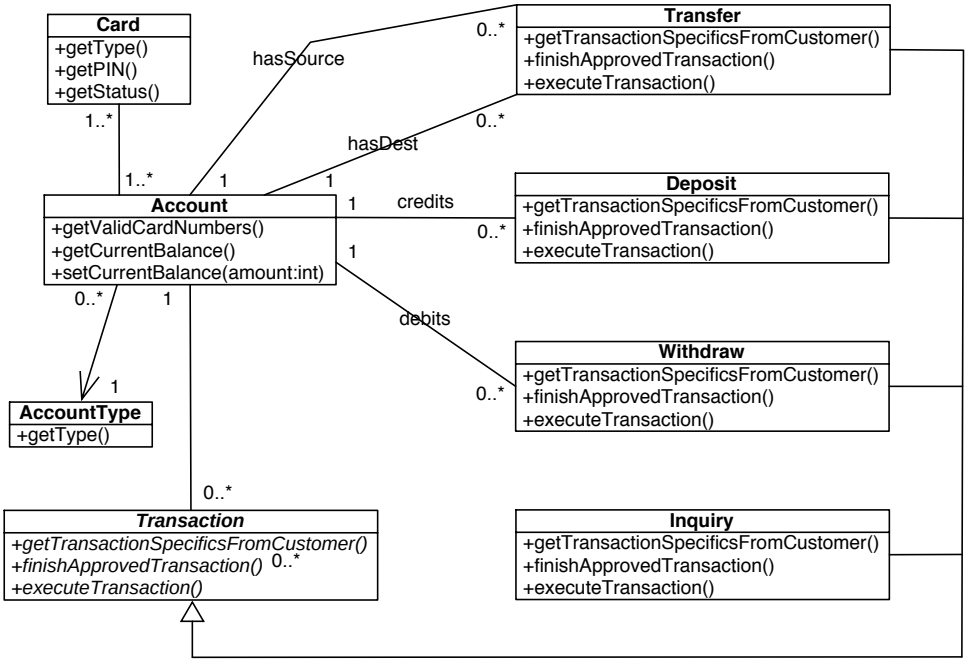


Figure 5.3: Class Diagram - Part III, version 1

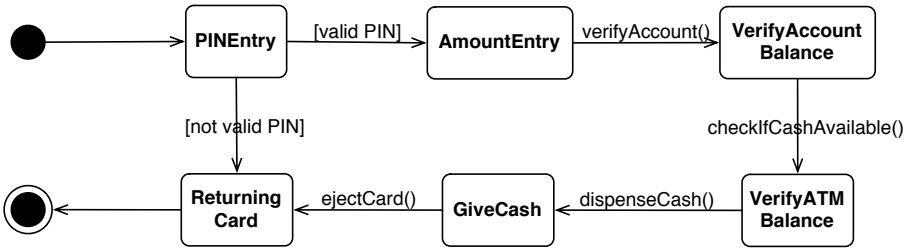


Figure 5.4: ATM State Diagram, version 1

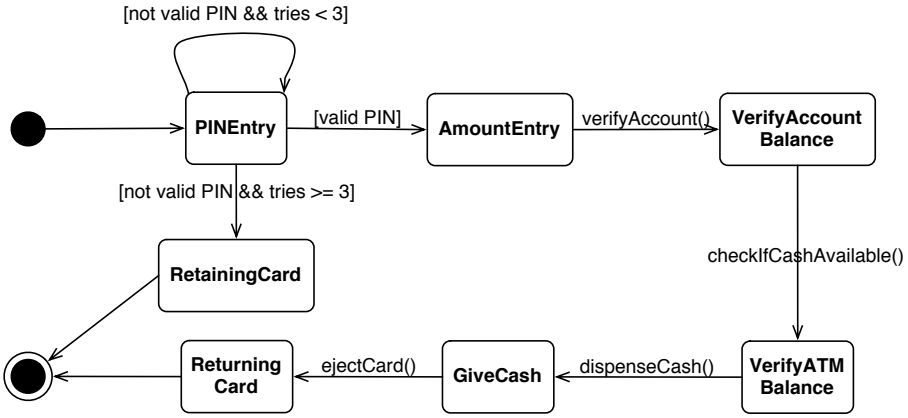


Figure 5.5: ATM State Diagram, version 2

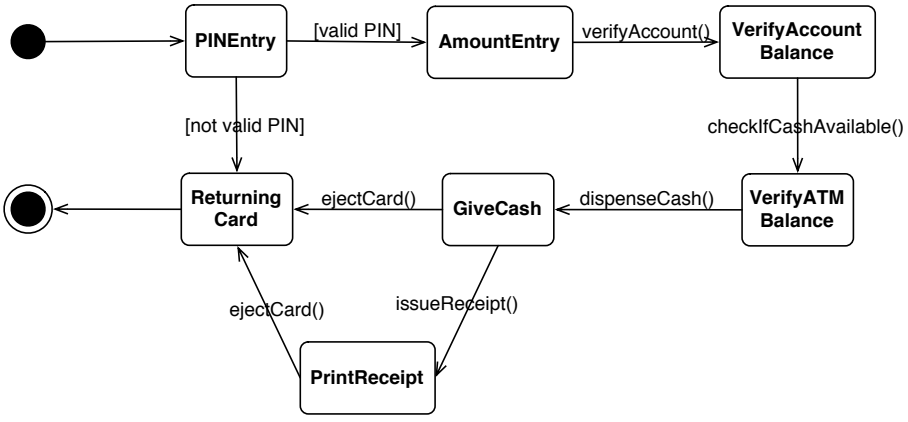


Figure 5.6: PrintingATM State Diagram, version 1

The sequence diagram shown in figure 5.7 represents the general interaction between the classes *ATM*, *Session*, *Bank*, *Display* and *CardReader*, when a card is retained after the user has entered two invalid PIN numbers in a row. Once the *ATM* has detected that a card has been inserted, a new *Session* is created, that will log all the actions carried out by the user, until the card is either ejected or retained. The *Session* starts by validating the user. This is done by asking for the PIN number associated to the card that was inserted. This number is then sent to the *Bank*, that verifies if the PIN - card number pair is valid. If the pair is not valid, the *Bank* asks the *Session* to handle this. The *Session* will store, in an internal variable, that the user has entered an invalid PIN, and will tell the *ATM* to ask the user for his PIN number again. The *ATM* does this through the *Display*. Once a new PIN is obtained, it is again checked by the *Bank*. As it is again invalid, the *Session* has to order the retention of the card. This is done by sending a message to the *ATM*, that delegates the physical task of retaining the card to the *CardReader*. The user is informed, again through the *Display*, that his card has been retained by the *ATM*.

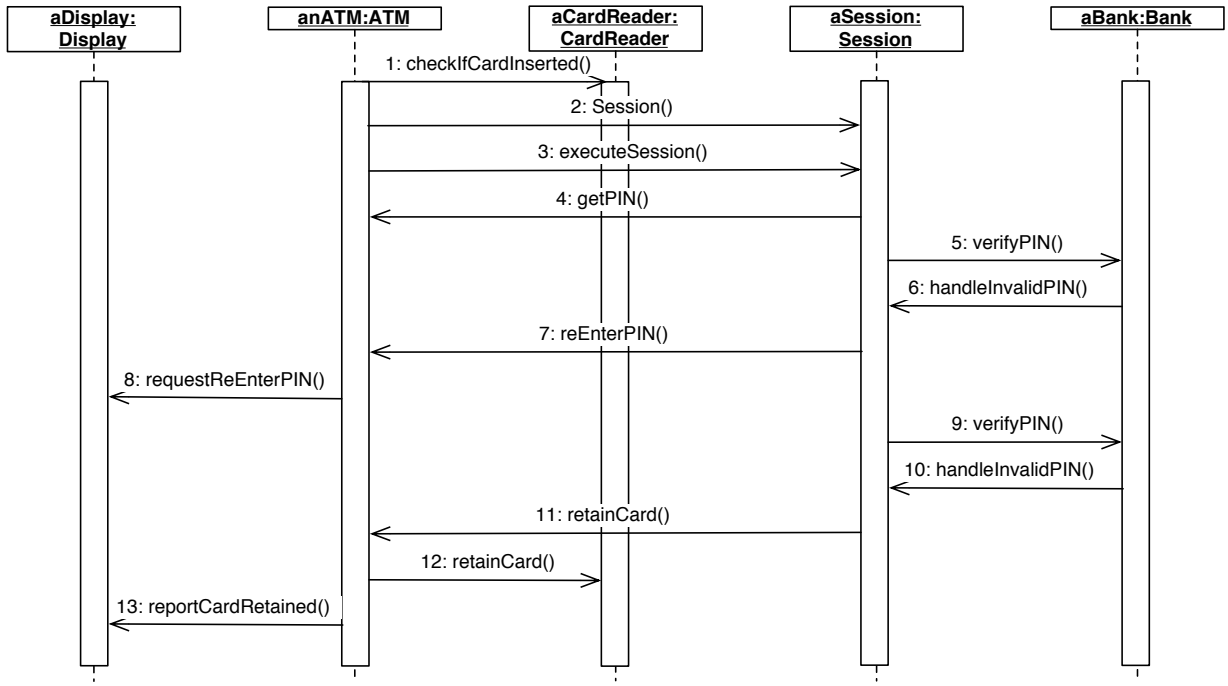


Figure 5.7: Sequence Diagram: checking card and PIN number, version 1

The sequence diagram shown in figure 5.8 represents the general interaction between the classes *ATM*, *Session*, *Withdraw*, *Bank* and *CashDispenser*, when a user decides to make a withdrawal. The PIN - card number pair validity must be checked before carrying out these steps. As the user wants to make a withdrawal, the *Session* creates a new *Withdraw* transaction. The first step is obtaining the transaction-specific information from the user. In this case, the user is prompted to enter the amount he wishes to withdraw. Secondly, the transaction is initiated at the *Bank*, and carried out by the *ATM*, if there are enough funds locally. Finally, the *Bank* is sent the *finishTransaction* message, notifying it that the transaction was either carried out successfully, or failed because, in this case, the *ATM* did not have sufficient funds with which to satisfy the withdrawal request.

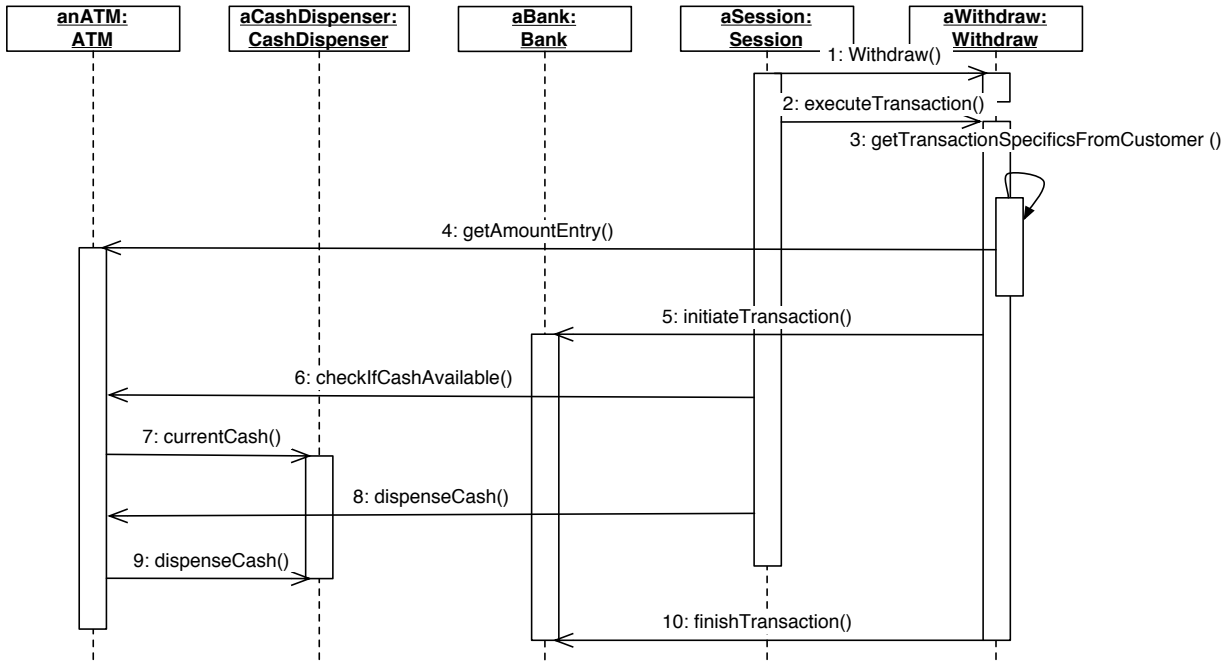


Figure 5.8: Sequence Diagram for withdraw transactions, version 1

The sequence diagram shown in figure 5.9 represents the general interaction between the classes *ATM*, *Session*, *Deposit*, *Bank* and *Keyboard*, when a user decides to make a deposit. The PIN - card number pair validity must be checked before carrying out these steps. As the user wants to make a deposit, the *Session* creates a new *Deposit* transaction. The first step is obtaining the transaction-specific information from the user. In this case, the user is prompted to enter the amount he wishes to deposit. Secondly, the transaction is initiated at the *Bank*, and carried out by the *ATM*, that should receive an envelope with the deposit in cash. Finally, the *Bank* is sent the *finishTransaction* message, notifying it that the transaction was either carried out successfully, or failed because, in this case, the *ATM* did not receive a deposit envelope.

The sequence diagram shown in figure 5.10 represents the general interaction between the classes *PrintingATM*, *Session*, *Inquiry* and *Bank*, when a user decides to print an inquiry of the state of his account. The PIN - card number pair validity must be checked before carrying out these steps. As the user wants to inquire the state of the account, the *Session* creates a new *Inquiry* transaction. The first step is obtaining the transaction-specific information from the user, for example, print only balance, print all transactions for the week, etc. Afterwards, the *Bank* is notified that the inquiry has taken place, and is carried out by the *PrintingATM*, that prints out the information requested by the user.

The sequence diagram shown in figure 5.11 represents the general interaction between the classes *PrintingATM*, *Session*, *Transfer* and *Bank*, when a user decides to make a transfer. The PIN - card number pair validity must be checked before carrying out these steps. As the user wants to make a transfer, the *Session* creates a new *Transfer* transaction. The first step is obtaining the transaction-specific information from the user. In this case, the user is prompted to enter the identification of the account to which he wants to make the transfer to. Secondly, the transaction is initiated at the *Bank*, and carried out by the *ATM*, that should receive an

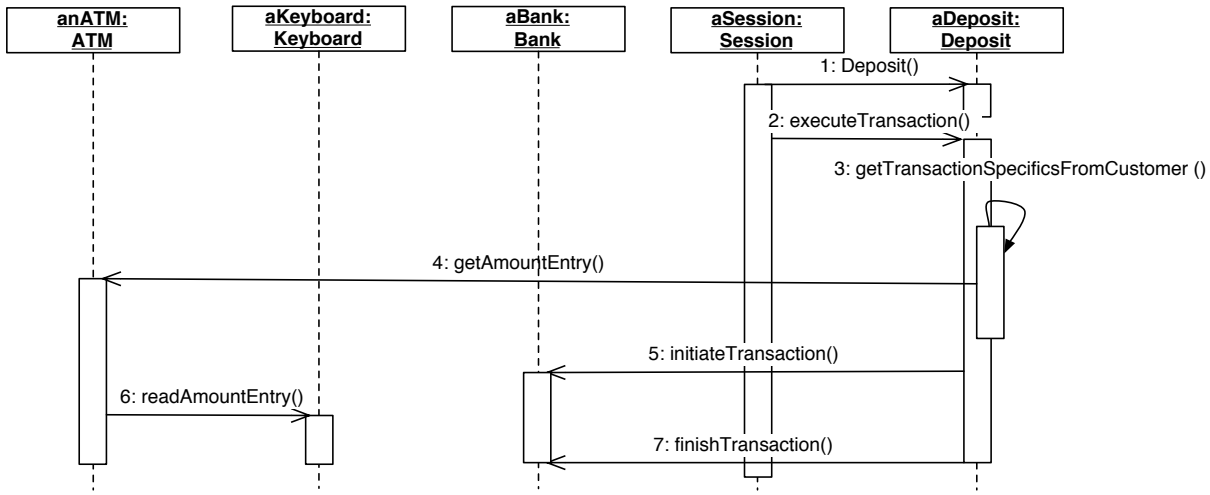


Figure 5.9: Sequence Diagram for deposit transactions, version 1

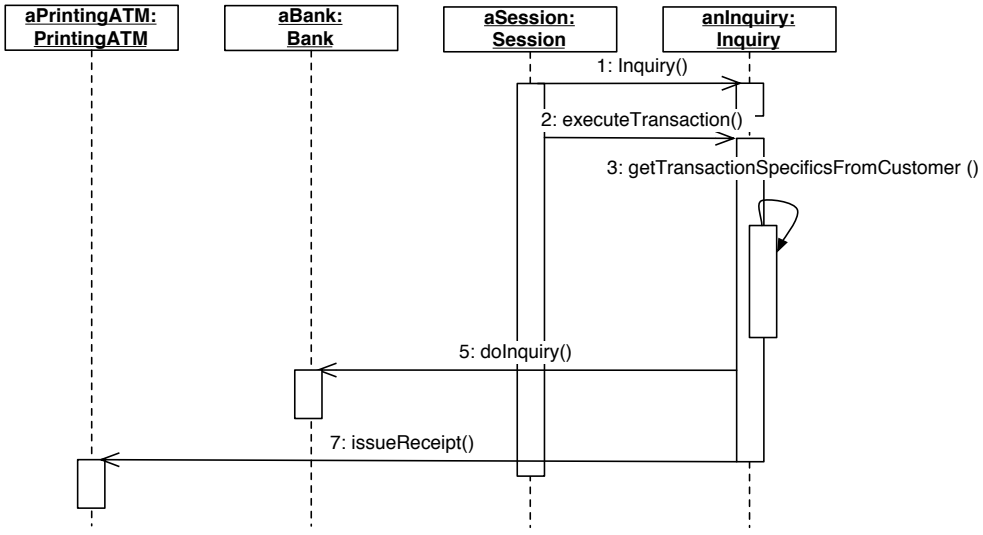


Figure 5.10: Sequence Diagram for inquiry transactions, version 1

envelope with the deposit in cash. Afterwards, the *Bank* is notified that the transfer has taken place, and the *PrintingATM* prints out a receipt.

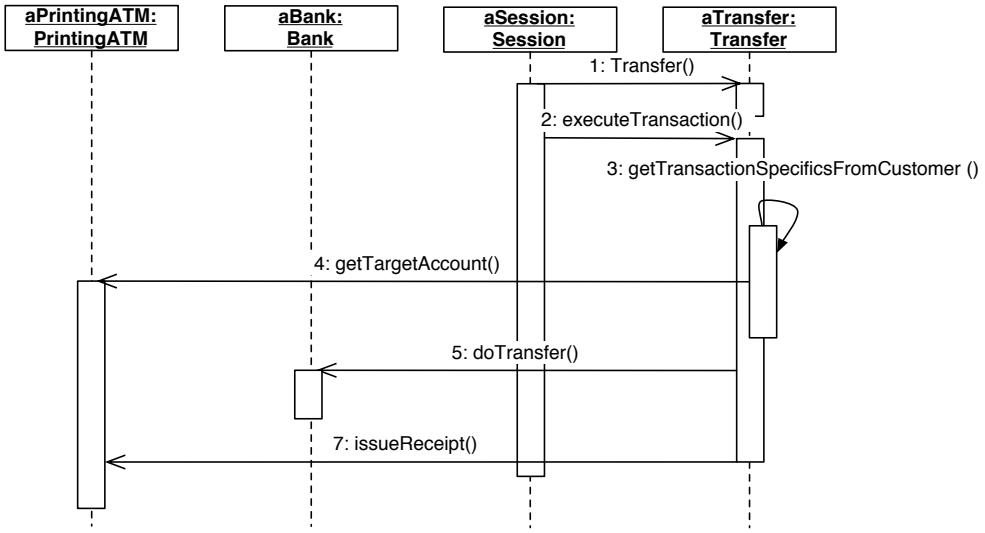


Figure 5.11: Sequence Diagram for transfer transactions, version 1

# Chapter 6

# Classification of Inconsistencies

In this chapter, the set of inconsistencies discovered in this study are presented. In the first section, a classification is proposed for these inconsistencies. This is a two-dimensional classification, based on the type of model that is affected and whether the inconsistencies are of the structural or behavioural type. In the second section, each inconsistency is presented by means of a definition, a UML example, the predicates used for its detection and some experimentation results. Some solution predicates are also presented.

## 6.1 Proposed Classification

The proposed classification has been developed by categorising the inconsistencies by their characteristics. The inconsistencies considered in this thesis are those that affect the structural and behavioural aspects of the models. Another aspect, not considered in this study because current UML CASE tools already deal with it, is compliance with UML well-formedness rules. The inconsistencies that belong to the last aspect are usually caused by the violation of a UML well-formedness rules, like duplicate class names in a class model.

In addition to classifying the inconsistencies by aspect, they are also classified by the type of model that is affected. As mentioned in section 3.2, the types of diagrams considered in this thesis are Class, State and Sequence diagrams. This decision is justified by the time limits imposed on the development of this thesis, which makes it impossible to cover UML as a whole, and only a proof of concept can be provided. In this study, two types of models are considered, as defined by the Model Driven Architecture (MDA)[46]:

- Model: the elements included in this type of model serve as definitions for the elements included in other models. Class diagrams are of this type.
- Instance: the elements included in this type of model are instances of elements defined in other models. Sequence and State diagrams are of this type.

The resulting classification is shown in table 6.1, where the conflicts found are represented by more general categories. For the purpose of the thesis, conflict is considered a synonym of inconsistency. Note that this is only one of possible classifications. On the following pages, an overview of these categories is given. In section 6.1.2, concrete detection predicates and examples are given for the actual conflicts. The roles used in the queries correspond to those

defined by the UML metamodel in the profile created for these experiments, which are available in Appendix A.1. All detection predicates only inform the user about the inconsistencies that are detected, as some inconsistencies are introduced deliberately by designers and their solution is not always desired.

| | **Behavioural** | **Structural** |
|---|---|---|
| model-model conflict | | - inherited association conflict<br>- dangling (type) reference |
| model-instance conflict | - incompatible definition | - instance definition missing |
| instance-instance conflict | - invocable behaviour conflict<br>- observable behaviour conflict<br>- incompatible behaviour conflict | - disconnected model |

Table 6.1: Two-dimensional inconsistency conflict table

### 6.1.1 Behavioural Inconsistencies

The inconsistencies considered in this category refer to situations in which the behaviour of the system, as defined in Sequence and State models, is incomplete, incompatible or inconsistent with respect to existing behaviour or definitions.

#### Model-Model Conflicts

No inconsistencies were found in this category. This is due to the fact that the only type of Model model considered in this dissertation are class diagrams, and inconsistencies between diagrams of this kind are syntactical.

#### Model-Instance Conflicts

**I)** Incompatible Definition

The inconsistencies included in this category are those that are related to model elements that do not comply with the definitions that apply to them.

- Multiplicity: arises when a link between objects in a sequence diagram does not respect the multiplicity restrictions imposed by the class diagrams in the corresponding VersionedModel.
- Navigation: arises when a link between objects in a sequence diagram does not respect the navigability restrictions imposed on the associations in the class diagrams of the corresponding VersionedModel.
- Abstract Object: arises when an abstract class that has no concrete subclasses in the class diagrams of the corresponding VersionedModel, is instantiated in a sequence diagram.

**Instance-Instance Conflicts**

**I)** Invocable Behaviour Conflicts

The inconsistencies included in this category are those that are related violations of invocation consistency[18]. Invocation consistency requires that an object of a subclass of class A can be used wherever an object of class A is required. These state diagrams have to be protocol state machines[48]. A transition in a protocol state machine has as its trigger a call event that references an operation of the class, and an empty action sequence.

- State vs. State: each sequence of the superclass state diagram should be contained in the set of sequences of the state diagram for the subclass.
- Sequence vs. Sequence: the ordered collection of stimuli sent by an object of superclass should be contained in the ordered collection of stimuli sent by an object of the subclass.
- State vs. Sequence: the ordered collection of stimuli received by an object of the superclass in a sequence diagram, should exist as a sequence of the state diagram for the subclass.

**II)** Observable Behaviour Conflicts

The inconsistencies included in this category are those that are related violations of observable behaviour consistency. Observation consistency[18] requires that an object of the subclass always behave like an object of the superclass, when viewed according to the superclass description. These state diagrams have to be protocol statecharts.

- State vs. State: after hiding all new events, each sequence of the subclass state diagram should be contained in the set of sequences of the superclass state diagram.
- Sequence vs. Sequence: after hiding stimuli that are associated to newly introduced operations, the ordered collection of stimuli sent by an object of the subclass should be contained in the ordered collection of stimuli sent by an object of the superclass.
- State vs. Sequence: after hiding stimuli that are associated to newly introduced operations, the ordered collection of stimuli received by an object of the subclass in a sequence diagram, should exist as a sequence of the state diagram for the superclass.

**III)** Incompatible Behaviour Conflicts

The inconsistencies included in this category are those that are related to conflicting behaviour definitions for a class in a VersionedModel.

- State vs. Sequence: arises when the ordered collection of stimuli received by an object in a sequence diagram does not exist as a sequence of events in the protocol state machine of the object's class.

### 6.1.2 Structural Inconsistencies

The inconsistencies considered in this category refer to situations in which the structure of the system, as detailed in class diagrams, is incomplete, incompatible or inconsistent with respect to existing behaviour or definitions.

### Model-Model Conflicts

**I)** Inherited Association Conflicts

The inconsistencies included in this category are those that are related to inherited associations between different classes in the class diagrams of a VersionedModel.

- Infinite Containment: arises when the composition (the multiplicity constraints of these composition relationships are important) and inheritance relationships between classes in the class diagrams of a VersionedModel form a cycle that produces infinite containment of the instances of the affected classes.

**II)** Dangling (Type) Reference

Arises when a parameter's or attribute's type refers to a class that does not exist in any of the class diagrams of the corresponding VersionedModel.

### Model-Instance Conflicts

**I)** Instance Definition Missing

The inconsistencies included in this category are those that are related to the fact that an element definition does not exist in a VersionedModel, as it has either been removed from a diagram or not included yet.

- Classless Instances: arises when an object in a sequence diagram is the instance of a class that does not exist in any of the class diagrams of the corresponding VersionedModel.
- Classless Statechart: arises when the state machine of a state diagram is associated to a class that does not exist in any of the class diagrams of the corresponding VersionedModel.
- Dangling (Inherited) Feature Reference: arises when a stimulus, event, guard or action references an attribute or operation that does not exist in the corresponding class (or its ancestors).
- Dangling (Inherited) Association Reference: arises when a link in a sequence diagram is related to an association that does not exist between the classes of the linked objects (or between the ancestors of these classes).

### Instance-Instance Conflicts

**I)** Disconnected Model

The inconsistencies included in this category are those that are related to the topography of the diagrams, specifically, when there are one or various elements that are not connected to the main diagram.

- State: arises when a state or transition has been deleted/omitted, resulting in a set of states that are not reachable from the initial state.
- Sequence: arises when an object or link has been deleted/omitted, resulting in a set of objects that are not reachable.

## 6.2 Experimental Results

On the following pages, the inconsistencies that were briefly introduced in section 6.1, are now explained in greater detail. Each inconsistency example includes a UML example, the detection code used and practical results obtained when the detection code was applied to the UML examples. The examples have been extracted from the Banking Simulation (see Chapter 5).

### 6.2.1 Incompatible Definition

#### Multiplicity

Affected Diagrams: Class, Sequence

This inconsistency arises when an object in a sequence diagram does not respect the multiplicity restrictions imposed in the class diagrams of the corresponding `VersionedModel`. Each multiplicity consists of a range and the multiplicity restrictions that can be checked are with respect to the top and bottom limits of this range. The top and bottom limits indicate the maximum and minimum number, respectively, of instances of the associated class with which instances of the class in question can interact with.

***Example***

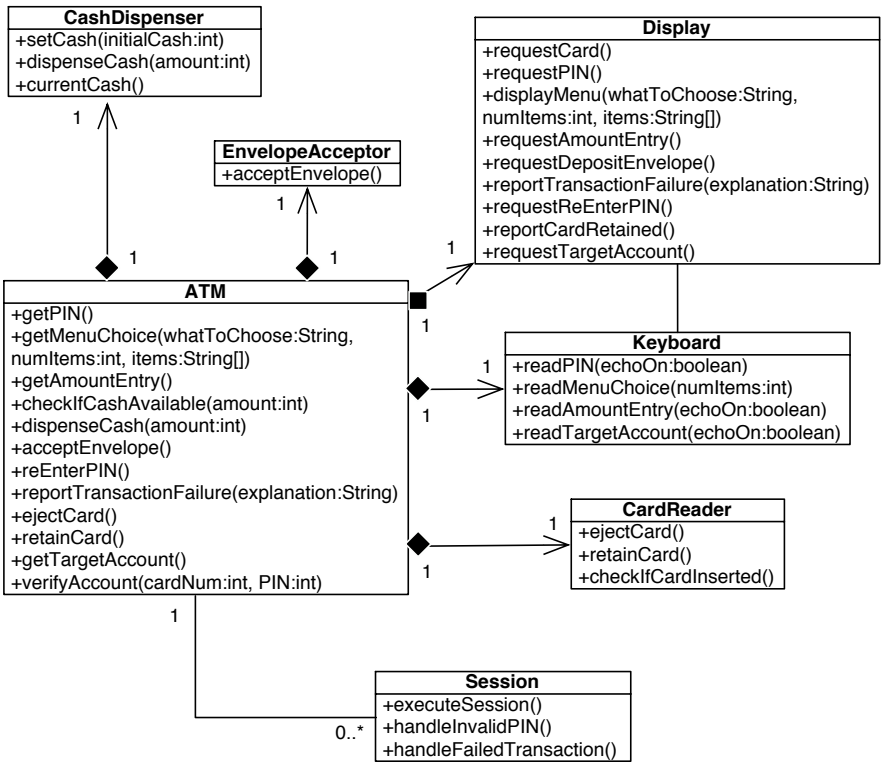


Figure 6.1: Class Diagram - Part I, version 2

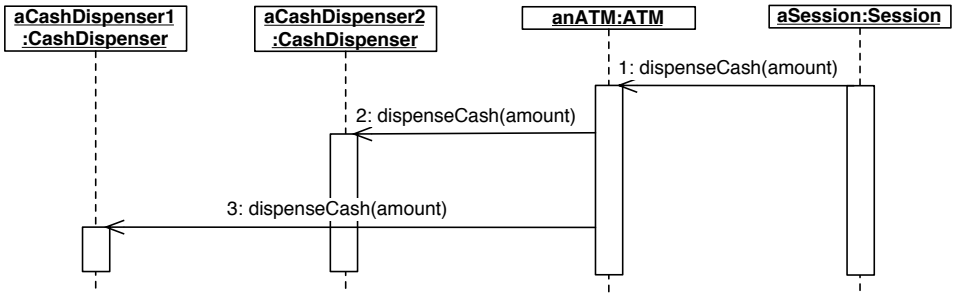


Figure 6.2: Sequence Diagram: multiplicity conflict

In the UML model that consists of the diagrams shown in figures 6.1 and 6.2 an example of this inconsistency can be seen. According to the class diagram, an instance of the `ATM` class can be associated with at most one instance of the `CashDispenser` class. In the sequence diagram, an instance of the `ATM` class is associated to two instances of the `CashDispenser` class, disobeying the limit set by the multiplicity of the association.

***Predicates***

The `multiplicity` function calculates the information needed in order to determine if an object in a sequence diagram is violating the multiplicity constraints imposed by an association in a class diagram. This information consists of the number of objects related to a specific object through the same association, and the top limit of the multiplicity range of this association. If there are more objects than allowed by the range, the user is notified.

```
(defun multiplicity (?object1 ?class1 ?class2 ?seq-model ?association)
 (let* ((?count1 (length (retrieve (?object1 ?object2 ?stimulus ?link)
   (:and
    (Instance-of-class ?object1 ?class1)
    (In-namespace ?object1 ?seq-model)
    (Instance-of-class ?object2 ?class2)
    (In-namespace ?object2 ?seq-model)
    (Sender-of ?object1 ?stimulus)
    (Receiver-of ?object2 ?stimulus)
    (Stimulus-link ?stimulus ?link)
    (Link-association ?link ?association)))))
  (?count2 (first (first (retrieve (?bound ?assocEnd ?multiplicity ?mult-range)
   (:and
    (Association-associationEnd ?association ?assocEnd)
    (Has-participant ?assocEnd ?class2)
    (Has-multiplicity ?assocEnd ?multiplicity)
    (Has-range ?multiplicity ?mult-range)
    (upper ?mult-range ?bound)))))))
  (if (> ?count1 ?count2)
   (format t "Multiplicity conflict: ~S accesses ~S instances of ~S,
   only ~S permitted~%" ?class1 ?count1 ?class2 ?count2))))
```

***Results***

The predicate was applied to the model presented in the example. The results that were obtained indicate that the object `aCardReader`, that belongs to `Sequence Diagram 5` (see figure

6.2), accesses 2 instances of the `CashDispenser` class, where the existing association between the classes `ATM` and `CashDispenser` restricts this number to 1 instance.

```
Multi. conflict: |I|ANATM-1.0 accesses 2 instances of |I|CASHDISPENSER-1.0,
 only 1 permitted
```

### Navigation

Affected Diagrams: Class, Sequence

This inconsistency arises when an object in a sequence diagram does not respect the navigability restrictions imposed on the object's class, by specifying the sense of navigation on the associations in the class diagrams of the corresponding VersionedModel.

***Example***

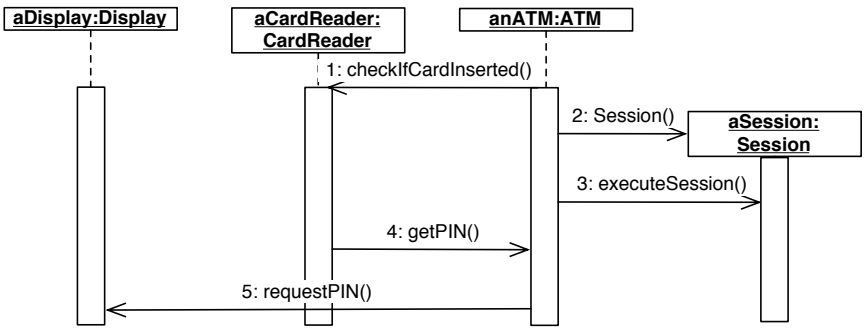


Figure 6.3: Sequence Diagram: navigability conflict

In the UML model that consists of the diagrams shown in figures 6.1 and 6.3 an example of this inconsistency can be seen. According to the class diagram, an instance of the `ATM` class can control an instance of the `CardReader` class. An `ATM` object will be able to send messages to the `CardReader` object, and to receive replies to these messages. `CardReader` objects cannot send messages to `ATM` objects. In the example, a `CardReader` object sends a `getPIN` message to an `ATM` object, and thus, is inconsistent with the association definition.

***Predicates***

This predicate queries the knowledge base directly, looking for objects that send messages to objects when forbidden by the navigation constraints imposed on the related association. The objects and the associations that they violate are printed on screen.

```
(do-retrieve (?object1 ?object2 ?seq-model ?class1 ?class2 ?stimulus ?operation
 ?association ?associationEnd2)
 (:and
  (Object ?object1)
  (In-namespace ?object1 ?seq-model)
  (Instance-of-class ?object1 ?class1)
  (Object ?object2)
  (In-namespace ?object2 ?seq-model)
  (Instance-of-class ?object2 ?class2)
```

```
  (Sender-of ?object1 ?stimulus)
  (Receiver-of ?object2 ?stimulus)
  (Sends-operation ?stimulus ?operation)
  (Has-feature ?class2 ?operation)
  (Related-to ?stimulus ?association)
  (AssociationEnd-Association ?associationEnd2 ?association)
  (Is-participant-of ?class2 ?associationEnd2)
  (isNavigable ?associationEnd2 false)
 )
 (format t "Nav. Inc: association ~S navigation is violated by object ~S~%"
  (get-value ?association 'name) (get-value ?object1 'name)))
```

***Results***

The predicate was applied to the model presented in the example. The results that were obtained indicate that the `aCardReader` object sends a message to an instance of the `ATM` class, when this is prohibited by the navigation constraint of the related association.

```
Nav. Inc: association |I|CARDREADER-ATM navigation is violated by object
 |I|ACARDREADER
```

### Abstract Object

Affected Diagrams: Class, Sequence

This inconsistency arises when an abstract class, that has no concrete subclasses, is instantiated in a sequence diagram. If the abstract class has concrete subclasses, the behaviour specified in the sequence diagram would apply to the instances of these subclasses. This is equivalent to declaring that the abstract class is the static type of the object, and during execution, dynamically typing it as one of the concrete subclasses of the abstract class. This is done so as to take advantage of polymorphism offered by object-oriented languages.

***Example***

In the UML model that consists of the diagrams shown in figures 6.4 and 6.5, the `Transaction` class is abstract and has no concrete subclasses. However, it is instantiated in the sequence diagram, which is inconsistent with the definition.

***Predicates***

The `abstract-objects` function calculates the information needed in order to determine if an abstract class, that has no concrete subclasses, is instantiated. This information consists of the list of inheritance relationships in which the abstract class participates as the *parent* class, and the list of objects that instantiate the abstract class. If the abstract class does not participate in any inheritance relationship as a parent, it has no subclasses. The user is notified of the inconsistency, and the list of objects that instantiate the class is printed.

```
(defun abstract-objects (?class)
 (let* ((?list-gen
   (retrieve (?generalization)
    (:and
     (isAbstract ?class true)
     (Is-generalization-of ?class ?generalization))))
```

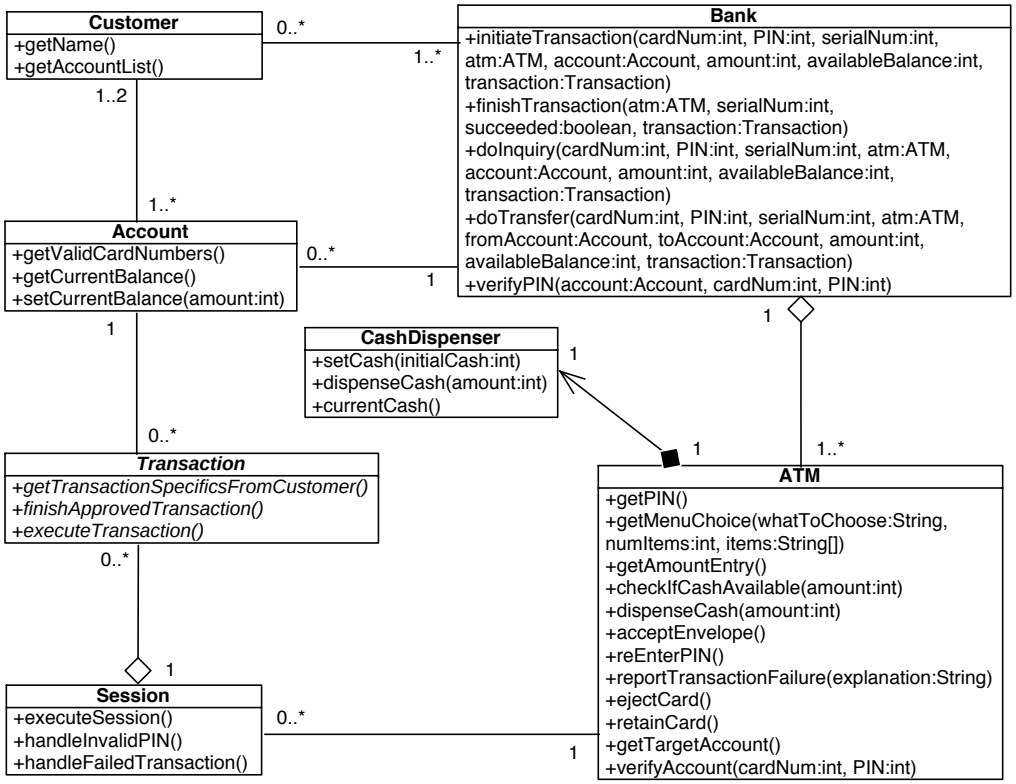


Figure 6.4: Class Diagram - Part II, version 2

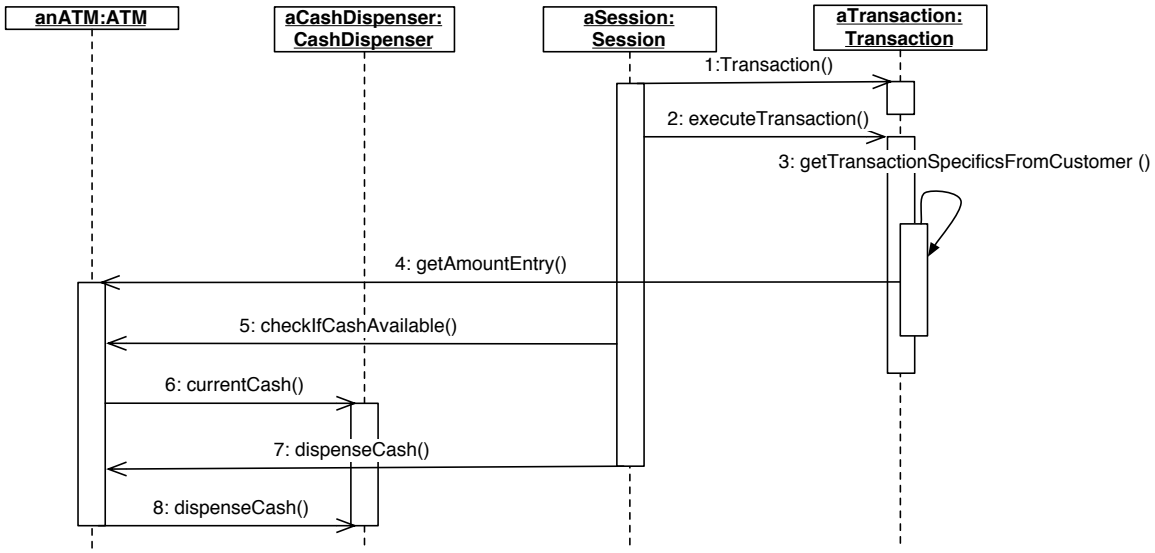


Figure 6.5: Sequence Diagram: abstract object inconsistency

```
(?list-obj
 (retrieve (?object)
  (:and
   (Instance-of-class ?object ?class)))))
(if (and (equalp NIL ?list-gen) (< 0 (length ?list-obj)))
 (format t "~S has abstract objects ~S~%" ?class ?list-obj))))
```

***Results***

The predicates were applied to the model presented in the example. The results that were obtained indicate that the `Transaction` class has been instantiated in a sequence diagram, even though it has no concrete subclasses. `fi` is a Loom function that finds the individual associated to the Loom identifier given as an argument.

```
UML(315): (abstract-objects (fi transaction-1.0))
|I|TRANSACTION-1.0 has abstract objects ((|I|ATRANSACTION-1.0))
NIL
```

### 6.2.2 Invocable Behaviour Conflicts

#### Invocable Behaviour - State vs. State

Affected Diagrams: State

This inconsistency arises when the behaviour of a subclass is not invocation consistent[18] with that of its superclass. In this case, class behaviour is specified using protocol state machines[48]. Invocation consistency asserts that each sequence of the superclass state diagram should be contained in the set of sequences of the state diagram for the subclass. The corresponding notion of behaviour inheritance is based on the substitution principle[37]. This principle requires that an object of a subclass of class A can be used wherever an object of class A is required.

***Example***

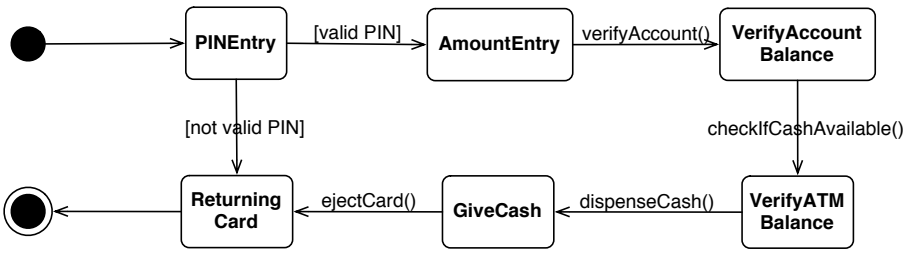


Figure 6.6: ATM State Diagram, version 1

In the UML model that consists of the diagrams shown in figures 5.1, 5.2, 5.3, 6.6 and 6.7, the behaviour of the `PrintingATM` class is invocation consistent with respect to that of the `ATM` class. This is because each trace of the superclass' state diagram can be found in the state diagram of the subclass. Hence, there are no inconsistencies.

***Predicates***

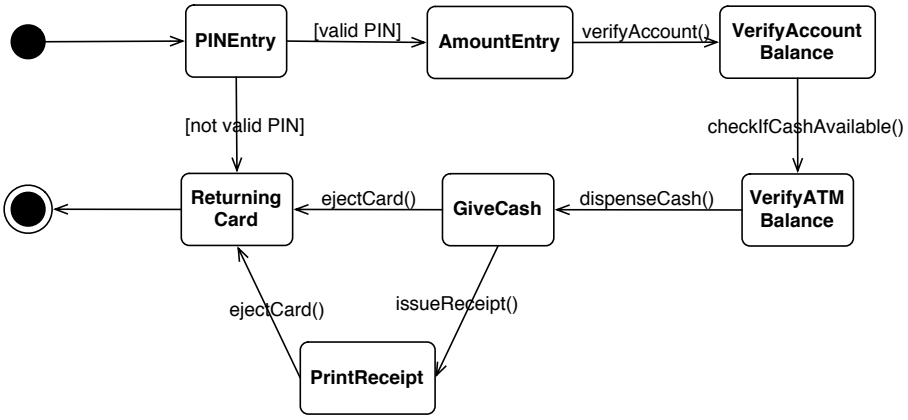


Figure 6.7: PrintingATM State Diagram, version 1

New concepts and relationships were introduced into the *Tbox*, which are used for storing state machine traces. The `SMTrace` represents one trace of a state machine, and is associated to the default `top` state of the machine, using the `Belongs-to-top` relationship.

```
(LOOM:defrelation Belongs-to-top
 :domain SMTrace
 :range VersionedState)
(LOOM:defrelation Has-trace
 :is (:inverse Belongs-to-top))

(LOOM:defrelation trace-value :characteristics :single-valued)

(LOOM:defconcept SMTrace
 :is-primitive (:exactly 1 Belongs-to-top)
 :roles (trace-value))
```

The `generate-traces` function recursively generates the traces of a state machine. A trace[56] is a sequence of states, starting from the *initial* state, with the events that trigger the transitions from state to state intercalated between the different states. The first part of the predicate assigns the current trace to a newly created `SMTrace` instance, and then associates it with the current state machine. The second part finds all the transitions from the current state, and recursively follows each one of these paths, adding the current trigger information to the trace.

```
(defun generate-traces (?top-state ?state1 ?aux)
 (let ((?trace (create 'nil 'SMTrace)))
  (tellm (about ?trace
   (Belongs-to-top ?top-state)
   (trace-value ?aux))))
 (do-retrieve ()
  (:and
   (Is-container-of ?top-state ?state1)
   (for-some (?transition ?state2 ?callevent)
    (:and
     (Is-source-of ?state1 ?transition)
```

```
    (Is-target-of ?state2 ?transition)
    (Is-container-of ?top-state ?state2)
    (Triggered-by ?transition ?callevent)
    (:predcall #'generate-traces ?top-state ?state2
     (:predcall #'cons (name ?state2)
      (:predcall #'cons (name ?callevent) ?aux))))))
  (T)))
```

The `check-traces` function checks that the traces associated to `?top-state1` exist in `?top-state2`. As these experiments are conducted under the Open World assumption, it is necessary to manually close the `Has-trace` role, so as to be able to reason with respect to all the existing traces of a machine. The first part of the function does this, by counting how many traces a machine has, and then asserting that the machine has exactly that amount of traces. The second part checks that each trace of the first machine has a counterpart in the second machine. Afterwards, the `Has-role` is re-opened, by cancelling the assertion made previously about the amount of fillers of the role, and the result of the comparison is returned.

```
(defun check-traces (?top-state1 ?top-state2)
 (let* ((?flag)
        (?count1 (first (retrieve ?x (count (Has-trace ?top-state1) ?x))))
        (?count2 (first (retrieve ?x (count (Has-trace ?top-state2) ?x)))))
  (tellm (about ?top-state1 (:exactly ?count1 Has-trace)))
  (tellm (about ?top-state2 (:exactly ?count2 Has-trace)))
  (do-retrieve (?x)
   (:and
    (name ?top-state1 ?x)
    (for-all (?trace1)
     (:implies
      (Has-trace ?top-state1 ?trace1)
      (for-some (?trace2)
       (:and
        (Has-trace ?top-state2 ?trace2)
        (:same-as (trace-value ?trace1) (trace-value ?trace2)))))))
   (setq ?flag "OK"))
  (forgetm (about ?top-state1 (:exactly ?count1 Has-trace)))
  (forgetm (about ?top-state2 (:exactly ?count2 Has-trace)))
  (if (equalp ?flag "OK")
   (format NIL "Consistency OK"))))
```

The `check-states` and `check-transitions` functions are auxiliary functions used by the `check-invocable-consistency` in order to perform some checks before actually comparing the two sets of traces. The `check-states` function checks that the states of `?top-state1` exist in `?top-state2`. The `check-transitions` function checks that the transitions of `?machine1` exist in `?machine2`. Finally, the `check-invocable-consistency` uses the `check-traces` function so as to check the traces. The `?machine1` parameter is associated with the parent class.

```
(defun check-invocable-consistency (?machine1 ?machine2 ?top-state1 ?top-state2)
 (format t "~S   ~S~%" ?machine1 ?machine2)
 (let ((?states (check-states ?top-state1 ?top-state2)))
  (format t "~%~S~%" ?states)
  (if (not (eql ?states NIL))
   (let ((?transitions (check-transitions ?machine1 ?machine2)))
```


(a) ATM object (b) PrintingATM object

Figure 6.8: Sequence diagram extracts, showing the behaviour of ATM and PrintingATM objects

```
(format t "~%~S~%" ?transitions)
(if (not (eql ?transitions NIL))
 (let ((?traces (check-traces ?top-state1 ?top-state2)))
  (format t "~%Invocable ~S~%" ?traces)))))))
```

***Results***

The predicates were applied to the model presented in the example. The results that were obtained indicate that the behaviour defined for the classes `ATM` and `PrintingATM`, through protocol state machines, is invocation consistent.

```
|I|ATM-STATEMACHINE-1.1   |I|PRINTINGATM-STATEMACHINE-1.1
"States OK"
"Transitions OK"
Invocable "Consistency OK"
NIL
```

### Invocable Behaviour - Sequence vs. Sequence

Affected Diagrams: Sequence

This inconsistency arises when the behaviour of a subclass is not invocation consistent[18] with that of its superclass. In this case, class behaviour is specified by the ordered collection of stimuli that its instances send, activated by the reception of a stimulus. Invocation consistency asserts that the set of sequences of messages sent by the instances of the superclass should be contained in the set of sequences of messages sent by the instances of the subclass. The corresponding notion of behaviour inheritance is based on the substitution principle[37]. This principle requires that an object of a subclass of class A can be used wherever an object of class A is required.

***Example***

In the UML model that consists of the diagrams shown in figures 5.1, 5.2, 5.3, 6.8(a) and 6.8(b), the behaviour of the `aPrintingATM` object is invocation consistent with respect to the `anATM` object. Figure 6.8(a) shows an extract of a sequence diagram that presents the behaviour

of the ATM class when a dispenseCash message is received. Figure 6.8(b) shows the same situation, with respect to the PrintingATM class. Even though the subclass incorporates new behaviour, in this case, sending a printReceipt message, it still offers the behaviour established by the superclass, which is sending a dispenseCash message.

***Predicates***

The generate-sent-operations function generates the ordered collection of stimuli sent when an activator stimulus is received. Using a query, the list of stimuli sent by the corresponding object when the stimuli (given as the function argument) is received, is generated. Then, the stimuli are ordered by name, so it is necessary to correctly label each one, using integers. The ordered list is returned.

```
(defun generate-sent-operations (?activator)
 (let* ((?ordered-rec-ops)
  (?received-ops (retrieve (?name1 ?name2 ?activee ?callaction ?operation)
   (:and
    (Stimulus-Activee ?activator ?activee)
    (name ?activee ?name1)
    (Initiates ?activee ?callaction)
    (CallAction-operation ?callaction ?operation)
    (name ?operation ?name2)))))
  (setq ?ordered-rec-ops (sort ?received-ops '< :key 'first))))
```

The invoc-seq function checks for invocation consistency between the behaviour that is observed when the same message is received by objects that instantiate classes related by inheritance. The list of messages sent by each object, in response to the activator stimulus received by each one, is generated using the generate-sent-operations, and is given to the invoc-seq function as arguments. The function then compares these two lists, in order to determine if the behaviour specified by the parent object is present in the child object. The user is notified about the result of this comparison.

```
(defun invoc-seq (?parent-seq ?child-seq)
 (let* ((?current-parent (first ?parent-seq))
   (?current-child (first ?child-seq)))
  (if (equalp NIL ?parent-seq)
   (format t "invocation consistent")
   (if (equalp NIL ?child-seq)
    (format t "not invocation consistent")
    (if (equalp (second ?parent-seq) (second ?child-seq))
     (invoc-seq (cdr ?parent-seq) (cdr ?child-seq))
     (invoc-seq ?parent-seq (cdr ?child-seq)))))))
```

***Results***

The predicate was applied to the model presented in the example. The results that were obtained indicate that an object of the subclass PrintingATM can substitute an object of the superclass ATM, given the behaviour defined in the sequence diagrams. The PrintingATM objects can behave like an ATM object, if required. fi is a Loom function that finds the individual associated to the Loom identifier given as an argument.

```
UML(27): (invoc-seq (generate-sent-operations (fi stima2-1.0))
```

```
 (generate-sent-operations (fi stimb2-1.0)))
invocation consistent
NIL
```

### Invocable Behaviour - State vs. Sequence

Affected Diagrams: Sequence, State

This inconsistency arises when the behaviour of a subclass is not invocation consistent[18] with that of its superclass. In this case, the superclass behaviour is specified by the ordered collection of stimuli that its instances receive and the subclass behaviour is specified using a protocol state machine[48]. Invocation consistency asserts that the sequences of messages received by the instances of the superclass are contained in the set of sequences of the state diagram for the subclass. The corresponding notion of behaviour inheritance is based on the substitution principle[37]. This principle requires that an object of a subclass of class A can be used wherever an object of class A is required.

***Example***

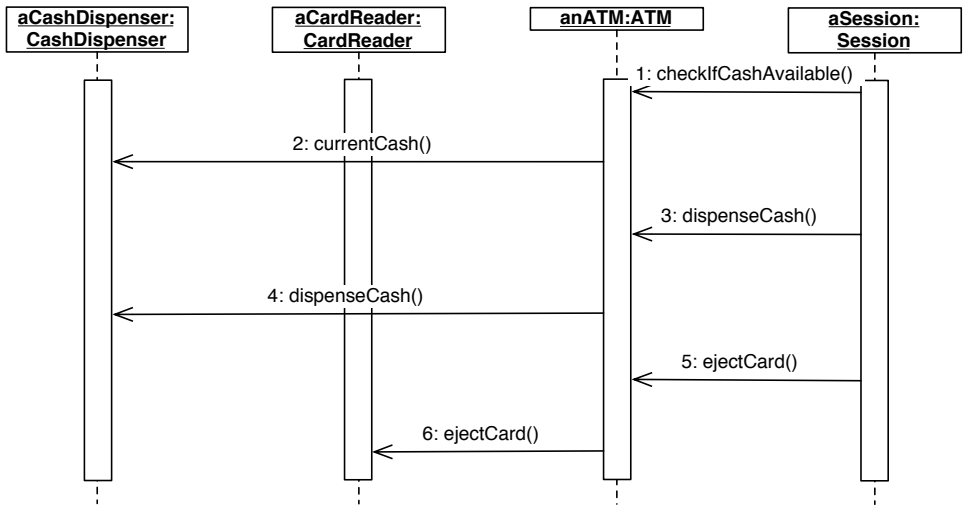


Figure 6.9: Sequence Diagram: invocation consistency

In the UML model that consists of the diagrams shown in figures 5.1, 5.2, 5.3, 6.9 and 6.10, the behaviour of the `PrintingATM` class is not invocation consistent with respect to the object representing the `ATM` class. This is because the `ejectCard` transition from the `GiveCash` state to the `ReturningCard` state has been accidentally deleted (see figure 6.7 for the correct version of the state diagram).

***Predicates***

The `generate-received-operations` function generates the ordered collection of stimuli received by the object received as a parameter. Using a query, the list of stimuli received by the parameter object is generated. Then, the stimuli are ordered by name, so it is necessary to correctly label each one, using integers. The ordered list is returned.

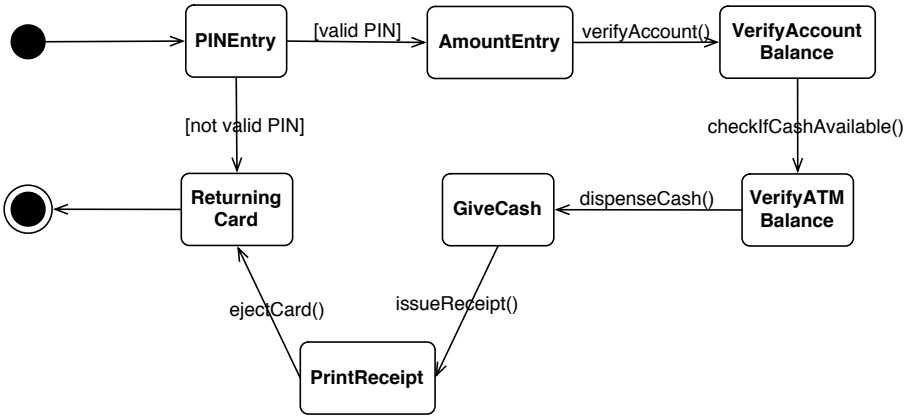


Figure 6.10: PrintingATM State Diagram, version 2

```
(defun generate-received-operations (?object)
 (let* ((?ordered-rec-ops)
  (?received-ops (retrieve (?name1 ?name2 ?stimulus ?callaction ?operation)
   (:and
    (Receiver-of ?object ?stimulus)
    (name ?stimulus ?name1)
    (Initiates ?stimulus ?callaction)
    (CallAction-operation ?callaction ?operation)
    (name ?operation ?name2)))))
  (setq ?ordered-rec-ops (sort ?received-ops '< :key 'first))))
```

The `traverse-sm` function recursively traverses a state machine, using the ordered collection of stimuli as a guide when choosing transitions. The traversal does not necessarily start at the *initial* state, as the behaviour depicted in a sequence diagram usually has omissions, in contrast to protocol state machines, that should be as complete as possible. The first part of the function is the end condition for the recursion: either a *final* state has been reached, or all the behaviour specified by the sequence diagram has been examined. If this is not the case, a query is used in order to find the transitions exiting from the current state. Using the sequence diagram behaviour as a guide, one of these is chosen, and the recursion continues. If there are no suitable transitions from the current state, an inconsistency has been found, and the user is notified.

```
(defun traverse-sm (?from ?from-name ?seq-list)
 (let* ((?aux) (?current (second (first ?seq-list))))
  (format t "Current operation: ~S~%" ?current)
  (if (or (equalp (format NIL "~S" (get-value ?from 'name)) "|I|FINAL")
          (equalp ?seq-list NIL))
   (format t "No behaviour consistency problems")
   (do-retrieve (?transition ?operation ?name ?callevent ?state)
    (:and
     (Is-source-of ?from ?transition)
     (Triggered-by ?transition ?callevent)
     (Is-occurence-of ?callevent ?operation)
     (name ?operation ?name)
```

```
 (Is-target-of ?state ?transition)
)
(if (equalp ?operation NIL)
 (traverse-sm ?state (get-value ?state 'name) ?seq-list)
 (if (equalp ?name ?current)
  (traverse-sm ?state (get-value ?state 'name) (cdr ?seq-list))
  (format t "Inconsistency found at state: ~S~%" ?from-name)))))))
```

***Results***

The predicate was applied to the model presented in the example. The results that were obtained indicate that instances of the subclass `PrintingATM` cannot be substituted by instances of the superclass `ATM` if the behaviour of the subclass is that described by the state diagram of the example. This is due to the fact that there are no transitions exiting from the `GiveCash` state that have the `ejectCard` event as a trigger. `fi` is a Loom function that finds the individual associated to the Loom identifier given as an argument.

```
UML(49): (traverse-sm (fi VerifyAccountBalance-SM-1.0) "VerifyAccountBalance"
 (generate-received-operations (fi anATM-1.0)))
Current operation: |I|CHECKIFCASHAVAILABLE
Current operation: |I|DISPENSECASH
Current operation: |I|EJECTCARD
Inconsistency found at state: |I|GIVECASH
NIL
```

### 6.2.3 Observable Behaviour Conflicts

#### Observable Behaviour - State vs. State

Affected Diagrams: State

This inconsistency arises when the behaviour of a subclass is not observation consistent[18] with that of its superclass. In this case, class behaviour is specified using protocol state machines. Observation consistency means that, after hiding all new events, each sequence of the subclass state diagram should be contained in the set of sequences of the superclass state diagram. So, an object of the subclass should always behave like an object of the superclass, if viewed only according to the superclass description.

***Example***

In the UML model that consists of the diagrams shown in figures 5.1, 5.2, 5.3, 6.6 and 6.7, the behaviour of the `PrintingATM` class is observation consistent with respect to the `ATM` class. This is because each trace of the subclass' state diagram, omitting the events associated to the operations introduced in the subclass, can be found in the state diagram of the superclass.

***Predicates***

The `generate-traces2` function recursively generates the traces of a state machine. A trace[56] is a sequence of events. In this specific case, traces are stored as state-event sequences, starting from the *initial* state, with the events that trigger the transitions from state to state intercalated between the different states. The difference between this function and the `generate-traces` function is that this function generates traces *omitting* events that reference operations that have been introduced in the subclass. The first part of the predicate assigns the current trace

to a newly created `SMTrace` instance, and then associates it with the current state machine. The second part finds all the transitions from the current state, and recursively follows each one of these paths, adding the current trigger information to the trace, if the associated operation can be found in the superclass.

```
(defun generate-traces2 (?machine1 ?machine2 ?top-state1 ?top-state2 ?state1
 ?aux ?aux-inst)
 (let* ((?f "OK")
        (?trace (create 'nil 'SMTrace)))
  (tellm (about ?trace
   (Belongs-to-top ?top-state2)
   (trace-value ?aux)))
  (do-retrieve (?x)
   (:and
    (Belongs-to-top ?x ?top-state2)
    (:not (:same-as ?x ?trace))
    (:same-as (trace-value ?x) (trace-value ?trace)))
   (setq ?f "NOT OK"))
  (if (equalp ?f "NOT OK")
   (forget-all-about ?trace)))
 (do-retrieve (?generalization ?parent ?child)
  (:and
   (Is-container-of ?top-state2 ?state1)
   (Is-statemachine-of ?machine2 ?child)
   (Is-statemachine-of ?machine1 ?parent)
   (Is-specialization-of ?generalization ?parent)
   (Is-generalization-of ?generalization ?child)
   (for-some (?transition ?state2 ?state3 ?callevent ?operation ?operation2)
    (:and
     (Is-source-of ?state1 ?transition)
     (Is-target-of ?state2 ?transition)
     (Is-container-of ?top-state2 ?state2)
     (Is-container-of ?top-state1 ?state3)
     (:same-as (name ?state2) (name ?state3))
     (Triggered-by ?transition ?callevent)
     (Has-feature ?parent ?operation)
     (Has-feature ?child ?operation2)
     (:or
      (:and
       ;operation is owned by the parent class
       (Is-occurence-of ?callevent ?operation)
       (:predcall #'generate-traces2 ?machine1 ?machine2 ?top-state1
        ?top-state2 ?state2
        (:predcall #'cons (name ?state2)
        (:predcall #'cons (name ?callevent) ?aux)) ?aux-inst)
      )
      (:and
       ;operation is owned by the child and parent classes
       (Is-occurence-of ?callevent ?operation2)
       (:same-as (name ?operation) (name ?operation2))
       (:predcall #'generate-traces2 ?machine1 ?machine2 ?top-state1
        ?top-state2 ?state2
        (:predcall #'cons (name ?state2)
        (:predcall #'cons (name ?callevent) ?aux)) ?aux-inst)
```

```
      )
      (:and
       ;operation is owned by the child class, not owned by parent
       (Is-occurence-of ?callevent ?operation2)
       (for-all (?z)
        (:implies
         (Has-feature ?parent ?z)
         (:not (:same-as (name ?z) (name ?operation2)))))
       (:predcall #'generate-traces2 ?machine1 ?machine2 ?top-state1
        ?top-state2 ?state2 (:predcall #'cons (name ?state2)
        (:predcall #'cons (name ?aux-inst) ?aux)) ?aux-inst)
      )
      (:and
       ;transition has no event
       (Is-occurence-of ?callevent NIL)
       (:predcall #'generate-traces2 ?machine1 ?machine2 ?top-state1
        ?top-state2 ?state2
        (:predcall #'cons (name ?state2)
        (:predcall #'cons (name ?callevent) ?aux)) ?aux-inst))))))
  (T)))
```

The `check-observable-consistency` uses the `generate-traces` function in order to generate the traces for the superclass' state machine. Afterwards, the `generate-traces2` function is used to generate the traces for the subclass' state machine. This is due to definition of observation consistency, where the traces of the subclass must be a subset of the superclass' set of traces. In order to be able to make this comparison, the events that reference new operations must be hidden. Finally, the `check-traces` function is used to compare the two sets of traces. The `?machine1` argument is associated with the parent class.

```
(defun check-observable-consistency (?parent ?machine1 ?machine2 ?top-state1
 ?top-state2 ?initial-state1 ?initial-state2)
 (let ((?aux-inst (create 'nil 'CallEvent))
       (?count1 (first (retrieve ?x (count (Has-feature ?parent) ?x)))))
  (tellm (about ?parent (:exactly ?count1 Has-feature)))
  (tellm (about ?aux-inst (name none)))
  (generate-traces ?top-state1 ?initial-state1 '(initial))
  (generate-traces2 ?machine1 ?machine2 ?top-state1 ?top-state2 ?initial-state2
   '(initial) ?aux-inst)
  (format t "~S   ~S~%" ?machine1 ?machine2)
  (let ((?traces (check-traces ?top-state1 ?top-state2)))
   (format t "~%Observable ~S~%" ?traces))
  (forgetm (about ?parent (:exactly ?count1 Has-feature)))))
```

***Results***

The predicates were applied to the model presented in the example. The results that were obtained indicate that the behaviour defined for the classes `ATM` and `PrintingATM`, through protocol state machines, is observation consistent.

```
|I|ATM-STATEMACHINE-1.1   |I|SATM-STATEMACHINE-1.1
Observable "Consistency OK"
```

### Observable Behaviour - Sequence vs. Sequence

Affected Diagrams: Sequence

This inconsistency arises when the behaviour of a subclass is not observation consistent[18] with that of its superclass. In this case, class behaviour is specified by the ordered collection of stimuli that its instances send, activated by the reception of a stimulus. Observation consistency means that, after hiding all the stimuli that reference new operations, the set of sequences of messages sent by the instances of the subclass are contained in the set of sequences of messages sent by the instances of the superclass. So, an object of the subclass should always behave like an object of the superclass, if viewed only according to the superclass description.

***Example***

In the UML model that consists of the diagrams shown in figures 5.1, 5.2, 5.3, 6.8(a) and 6.8(b), the behaviour of the `aPrintingATM` object is observation consistent with respect to the `anATM` object. Figure 6.8(a) shows an extract of a sequence diagram that presents the behaviour of the `ATM` class when a `dispenseCash` message is received. Figure 6.8(b) shows the same situation, with respect to the `PrintingATM` class. According to the definition of observation consistency, the stimuli associated to new behaviour are not considered, as is the case of stimulus `3: printReceipt`. The result is that the remaining sequence {`2:dispenseCash`} exists in the parent state diagram. The detection predicates for this inconsistency have not yet been developed.

### Observable Behaviour - State vs. Sequence

Affected Diagrams: Sequence, State

This inconsistency arises when the behaviour of a subclass is not observation consistent[18] with that of its superclass. In this case, the superclass behaviour is specified using a protocol state machine[48] and the subclass behaviour is specified by the ordered collection of stimuli that its instances receive. Observation consistency means that, after hiding all new events, each sequence of the subclass state diagram should be contained in the set of sequences[56] of the superclass state diagram. So, an object of the subclass should always behave like an object of the superclass, if viewed only according to the superclass description.

***Example***

In the UML model that consists of the diagrams shown in figures 5.1, 5.2, 5.3, 6.6 and 6.11, the behaviour of the `PrintingATM` object is observation consistent with respect to the behaviour defined for the `ATM` class. According to the definition of observation consistency, the stimuli associated to new behaviour are not considered, as is the case of stimulus {`5: issueReceipt`}. The result is that the remaining sequence, {`1:checkIfCashAvailable` → `3:dispenseCash` → `6:ejectCard`}, exists in the parent state diagram.

***Predicates***

This predicate queries the knowledge base directly, looking for operations that have to be passed along in inheritance relationships. If an operation is directly owned by the parent class, or has been inherited by the parent class, it is passed onto the child class, as an `Inherited-feature`. As this predicate only considers one level of inheritance, it has to be executed as many times as there are levels in the class hierarchy.

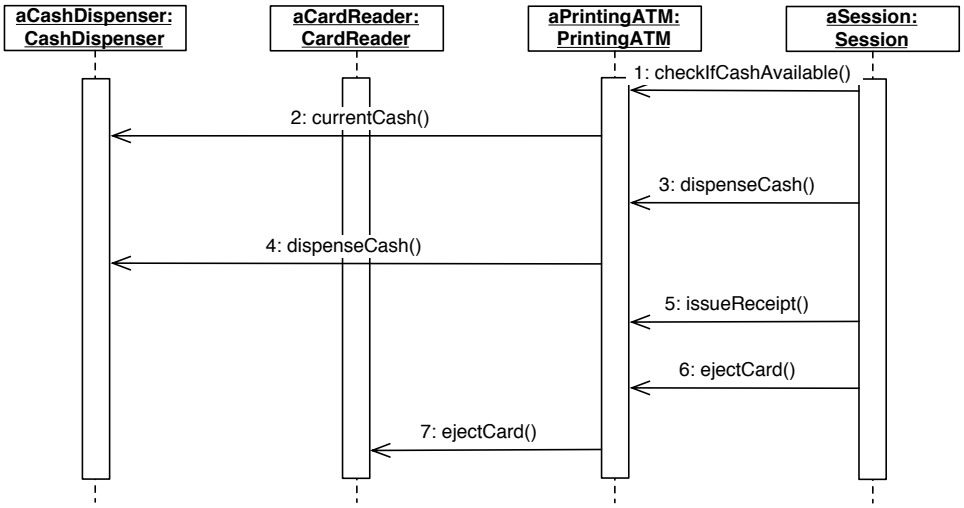


Figure 6.11: Sequence Diagram: observation consistency

```
(do-retrieve (?generalization ?parent ?child ?operation)
 (:and
  (Generalization ?generalization)
  (Is-specialization-of ?generalization ?parent)
  (Is-generalization-of ?generalization ?child)
  (or
   (Has-feature ?parent ?operation)
   (Inherited-feature ?parent ?operation)))
 (tellm (Inherited-feature ?child ?operation)))
```

The `generate-received-operations2` function generates the ordered collection of stimuli received by the object received as a parameter, including only the stimuli associated to inherited operations. Using a query, the list of stimuli received by the parameter object is generated. Then, the stimuli are ordered by sequence, so it is necessary to correctly label each one, using integers. The ordered list is returned.

```
(defun generate-received-operations2 (?object)
 (let* ((?ordered-rec-ops)
  (?received-ops (retrieve (?name1 ?name2 ?stimulus ?callaction ?operation
    ?class)
   (:and
    (Receiver-of ?object ?stimulus)
    (name ?stimulus ?name1)
    (Initiates ?stimulus ?callaction)
    (CallAction-operation ?callaction ?operation)
    (Instance-of-class ?object ?class)
    (Inherited-feature ?class ?operation)
    (name ?operation ?name2)))))
  (setq ?ordered-rec-ops (sort ?received-ops '< :key 'first))))
```

Finally, the `traverse-sm` function is used to compare the behaviour of the superclass behaviour is specified using a protocol state machine and the subclass behaviour is specified by the ordered

collection of stimuli that its instances receive. The ordered collection of stimuli is generated with the `generate-received-operations2`. The `traverse-sm` function used is the one presented in section 6.2.2.

***Results***

The predicates were applied to the model presented in the example. The results that were obtained indicate that the behaviour defined for the classes `ATM` and `PrintingATM`, through the superclass protocol state machine and messages received by subclass objects, is observation consistent. `fi` is a Loom function that finds the individual associated to the Loom identifier given as an argument.

```
UML(49): (traverse-sm (fi VerifyAccountBalance-SM-1.0) "VerifyAccountBalance"
 (generate-received-operations (fi aPrintingATM-1.0)))
Current operation: |I|CHECKIFCASHAVAILABLE
Current operation: |I|DISPENSECASH
Current operation: |I|EJECTCARD
Current operation: NIL
No behaviour consistency problems
NIL
```

### 6.2.4 Incompatible Behaviour Conflicts

#### Incompatible Behaviour - State vs. Sequence

Affected Diagrams: Sequence, State

This inconsistency arises when the sequence of messages received by an object in a sequence diagram is inconsistent with the behaviour established by the protocol state machine of the object's class. A sequence of messages is inconsistent with respect to a protocol state machine if it is not possible to find a path in the state machine that follows the order established by the sequence. It is only possible to alert the user about the existence of this inconsistency, the user must determine whether the problem lies in the protocol state machine or the sequence diagram.

***Example***

In the UML model that consists of the diagrams shown in figures 5.1, 5.2, 5.3, 6.6 and 6.12 an inconsistency of this type is detected. Using the protocol state machine as a template that indicates the permitted behaviour sequences, the ordering of the stimuli received by the instance of the `ATM` class is perceived as inconsistent with the established behaviour, since it receives a `ejectCard` message after an `checkIfCashAvailable` message. The protocol state machine of this class has no sequence modelling this behaviour.

***Predicates***

The `generate-received-operations` and `traverse-sm` functions are used in the detection of this type of inconsistency. Both these functions are presented in Section 6.2.2 - Predicates. The `generate-received-operations` function generates the ordered collection of stimuli received by the object received as a parameter. In this case, it is used generate the list of stimuli of the objects that instantiate the class in question. The `traverse-sm` function recursively traverses the state machine associated to the class in question, using the ordered collection of stimuli as a guide

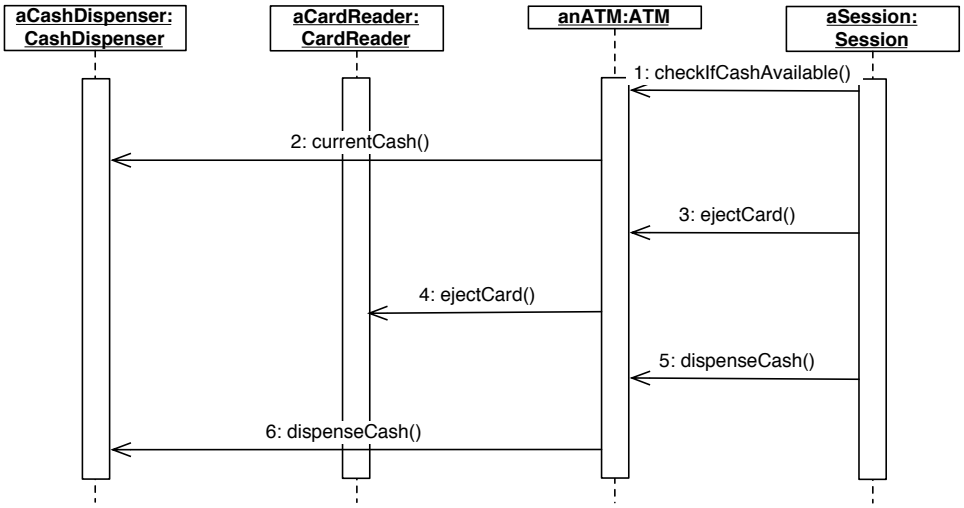


Figure 6.12: Sequence Diagram: incompatible behaviour

when choosing transitions. If an inconsistency is found, the user is notified.

***Results***

The predicate was applied to the model presented in the example. The results that were obtained indicate that the messages received by the object `anATM`, presents behaviour that is not consistent with that defined for the object's class, `ATM` (see figure 6.6), as there are no transitions exiting from the `VerifyATMBalance` state that have `ejectCard` as a trigger. `fi` is a Loom function that finds the individual associated to the Loom identifier given as an argument.

```
UML(49): (traverse-sm (fi VerifyAccountBalance-SM-1.0) "VerifyAccountBalance"
 (generate-received-operations (fi anATM-1.0)))
Current operation: |I|CHECKIFCASHAVAILABLE
Current operation: |I|EJECTCARD
Inconsistency found at state: |I|VERIFYATMBALANCE
NIL
```

### 6.2.5 Inherited Association Conflicts

#### Infinite Containment

Affected Diagrams: Class

This inconsistency arises when the composition and inheritance relationships between classes in the class diagrams of a VersionedModel form a cycle and combined with the multiplicities of the compositions, define a class whose instances will, directly or indirectly, be forced to contain at least one instance of the same class, causing an infinite chain of instances. This is probably due to the fact that the designer has overlooked an existing inheritance relationship.

***Example***

In the UML model that consists of the diagram shown in figure 6.13, an example of this

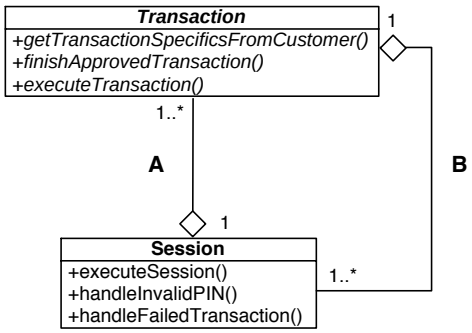


Figure 6.13: Infinite containment

inconsistency can be seen. The association marked as `A` is the original one, visible in figure 5.2. The association marked as `B` has been accidentally added to this diagram. The aggregation and multiplicity constraints cause an infinite containment inconsistency because each instance of the `Transaction` class must have at least one instance of the `Session` class, whose instances must have at least one instance of the `Transaction` class. The detection predicates for this inconsistency have not yet been developed.

### 6.2.6 Dangling (Type) Reference

Affected Diagrams: Class

This inconsistency arises when the type of an attribute does not exist as a class in the corresponding VersionedModel. It also arises when the type of a parameter of an operation does not exist as a class in the corresponding VersionedModel. The type of the parameter or attribute may have been removed, or may have not been included in a class diagram yet.

#### Example

In the UML model that consists of the diagram shown in figure 6.14, an inconsistency of this type is detected. The `Account` class has been deleted, but is still being used as the type of various operation parameters in the `Bank` class (see figures 5.1, 5.2 and 5.3 to see the details of the class interfaces).

#### Predicates

This predicate queries the knowledge base directly, looking for parameters that have types that are classes that do not belong to any class diagrams. The parameters and their invalid types are printed on screen.

```
(do-retrieve (?operation ?parameter ?class1 ?class2)
 (:and
  (Operation ?operation)
  (Is-owned-by ?operation ?class1)
  (Has-parameter ?operation ?parameter)
```

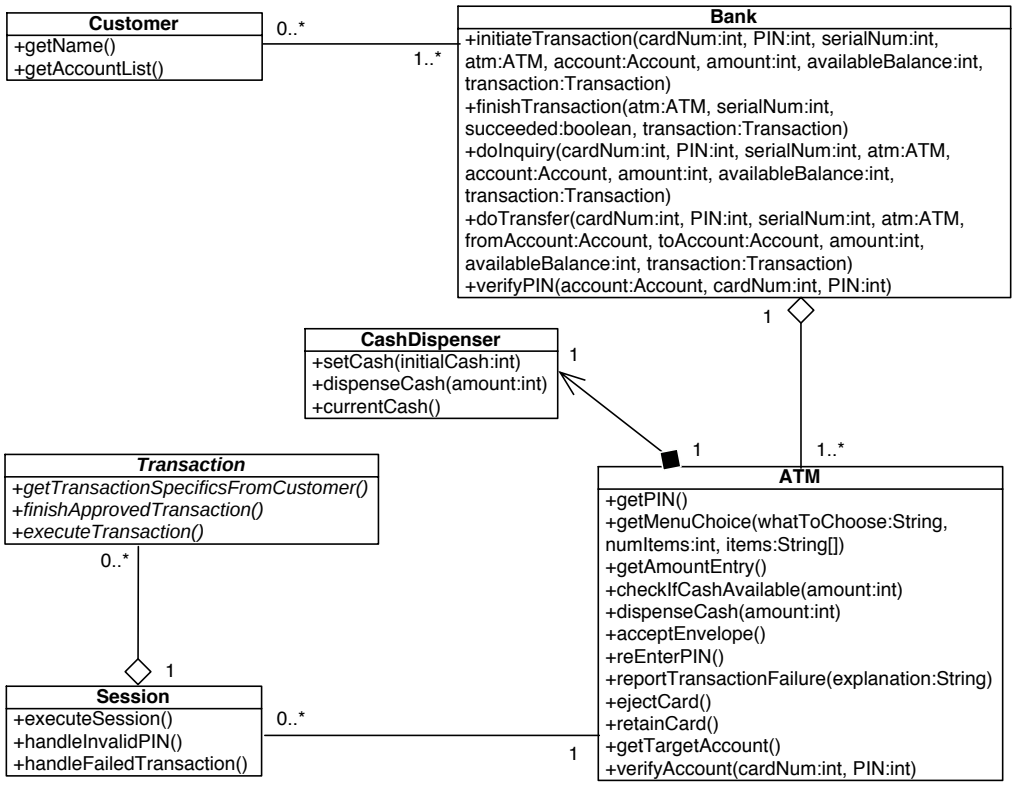


Figure 6.14: Class Diagram - Part II, version 3

```
  (Parameter-class ?parameter ?class2)
  (In-namespace ?class2 NIL))
 (format t "Type of the parameter ~S of operation ?S of class ~S, does not
  exist in any class diagram" (get-value ?parameter 'name) (get-value
  ?operation 'name) (get-value ?class1 'name)))
```

This predicate queries the knowledge base directly, looking for attributes that have types that are classes that do not belong to any class diagrams. The attributes and their invalid types are printed.

```
(do-retrieve (?attribute ?class)
 (:and
  (Attribute ?attribute)
  (Is-of-type ?attribute ?class)
  (In-namespace ?class NIL))
 (format t "The type of the ~S attribute of class ?S does not exist in any
  class diagram" (get-value ?attribute 'name) (get-value ?class 'name)))
```

#### Results

The first predicate was applied to the model presented in the example. The results that were obtained indicate that various operation parameter types of the `Bank` class are invalid, as reference the `Account` class, that has been deleted from the model.

```
The type of the parameter |I|ACCOUNT of operation |I|INITIATETRANSACTION of
 class |I|BANK, does not exist in any class diagram
The type of the parameter |I|ACCOUNT of operation |I|DOINQUIRY of class
 |I|BANK, does not exist in any class diagram
The type of the parameter |I|FROMACCOUNT of operation |I|DOTRANSFER of class
 |I|BANK, does not exist in any class diagram
The type of the parameter |ITO|ACCOUNT of operation |I|DOTRANSFER of class
 |I|BANK, does not exist in any class diagram
The type of the parameter |I|ACCOUNT of operation |I|VERIFYPIN of class |I|BANK,
 does not exist in any class diagram
```

### 6.2.7 Instance Definition Missing

#### Classless Instances

Affected Diagrams: Class, Sequence

This inconsistency arises when an object in a sequence diagram is the instance of a class that does not exist in any of the class diagrams of the corresponding VersionedModel. This can be caused by the fact that the designer has eliminated the class from the class diagrams, without updating the sequence diagrams; or has forgotten to include this new class in the class diagrams.

***Example***

In the UML model that consists of the diagrams shown in figures 6.15 and 6.16, this consistency problem arises as the designer has deleted the `CardReader` class in the class diagram, and has forgotten to update the sequence diagram, that still has instances of this class.

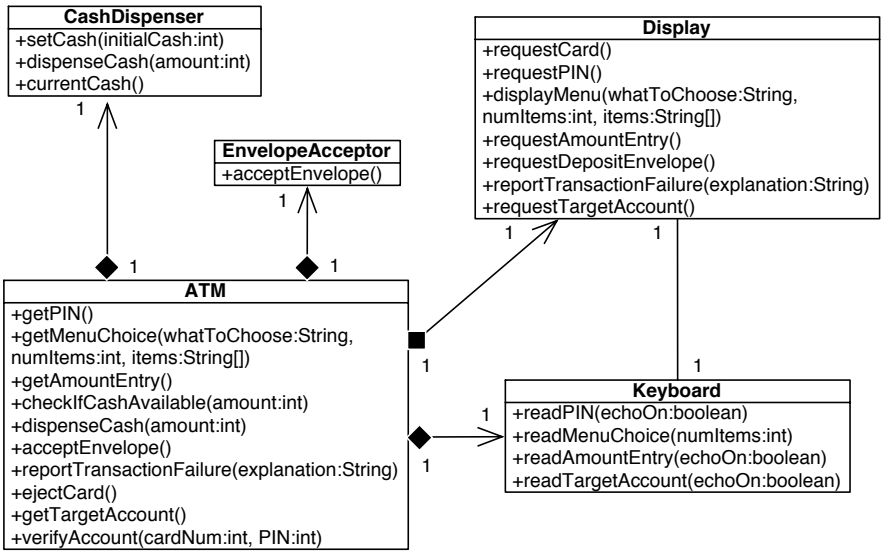


Figure 6.15: Class Diagram - Part I, version 3

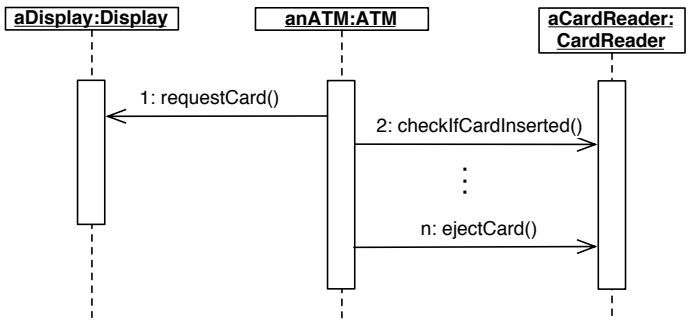


Figure 6.16: Sequence Diagram: classless instance

***Predicates***

This predicate queries the knowledge base directly, looking for objects that instantiate classes that do not belong to any class diagrams. The objects and their invalid classes are printed.

```
(do-retrieve (?object ?class ?seq-model ?model)
 (:and
  (Object ?object)
  (Instance-of-class ?object ?class)
  (In-namespace ?class NIL)
  (In-namespace ?object ?seq-model)
  (Contains-model ?model ?seq-model))
 (format t "Object ~S instantiates ?S class, that does not exist in any
  class diagram" (get-value ?object 'name) (get-value ?class 'name)))
```

***Results***

The predicate was applied to the model presented in the example. The results that were obtained indicate that the object `aCardReader` is the instance of the `CardReader` class, a class that does not belong to any class diagrams of the corresponding VersionedModel.

```
Object |I|ACARDREADER instantiates |I|CARDREADER class, that does not exist in
 any class diagram
```

### Classless Statechart

Affected Diagrams: Class, State

This inconsistency arises when a StateModel is associated to a class that does not exist in any of the class diagrams of the corresponding VersionedModel. This can be caused by the fact that the designer has eliminated the class from the class diagrams, without deleting the StateModel; or has forgotten to include this new class in the class diagrams.

***Example***

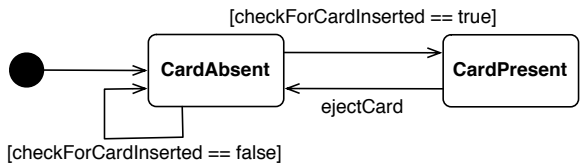


Figure 6.17: Card Reader State Diagram, version 1

In the UML model that consists of the diagrams shown in figures 6.15 and 6.17, this consistency problem arises as the designer has deleted the `CardReader` class from the class diagram, and has forgotten to delete the associated StateModel from the UML model.

***Predicates***

This predicate queries the knowledge base directly, looking for StateModels that model the behaviour of classes that do not belong to any class diagrams. The StateModels and their invalid classes are printed.

```
(do-retrieve (?state-model ?statemachine ?class ?model)
 (:and
  (StateModel ?state-model)
  (In-namespace ?statemachine ?state-model)
  (Is-stateMachine-of ?statemachine ?class)
  (In-namespace ?class NIL)
  (Contains-model ?model ?state-model))
 (format t "StateModel ~S defines behaviour of ?S class, that does not exist
  in any class diagram" (get-value ?state-model 'name) (get-value ?class
  'name)))
```

***Results***

The predicate was applied to the model presented in the example. The results that were obtained indicate that the state machine `CardReader-StateMachine` is the machine of the `CardReader` class, that does not belong to any class diagrams of the corresponding Versioned model.

```
StateModel |I|CARDREADER-STATE-DIAGRAM defines behaviour of
 |I|CARDREADER class, that does not exist in any class diagram
```

### Dangling (Inherited) Feature Reference

Affected Diagrams: Class, Sequence, State

This inconsistency arises when a stimulus, trigger, action or guard references a non-existing attribute or operation in a class diagram. This includes inherited features that are lost when the inheritance links between classes are broken. This inconsistency can also be caused by the removal of existing features, or by their omission when creating the class diagram.

***Example***

In the UML model that consists of the diagrams shown in figures 6.18, 6.19 and 6.20, various of examples of this consistency problem can be seen. The diagram shown in figure 6.18 is a modified version of that shown in figure 5.1. In this new version, the interfaces of some of the classes have been modified. Concretely, the following modifications have been made:

- `Session`
  `handleInvalidPIN` operation has been removed
- `Display`
  `requestReEnterPIN`, `reportCardRetained` operations have been removed
- `CardReader`
  `retainCard` operation has been removed
- `ATM`
  `retainCard`, `reEnterPIN` operations have been removed

The sequence diagram shown in figure 6.19 includes various examples of this consistency problem, as it was not updated so as to include the modifications made to the class diagram. Links A through G are associated to actions that refer to operations that are no longer in the definition of the system. If a member of the development team is not informed of the latest

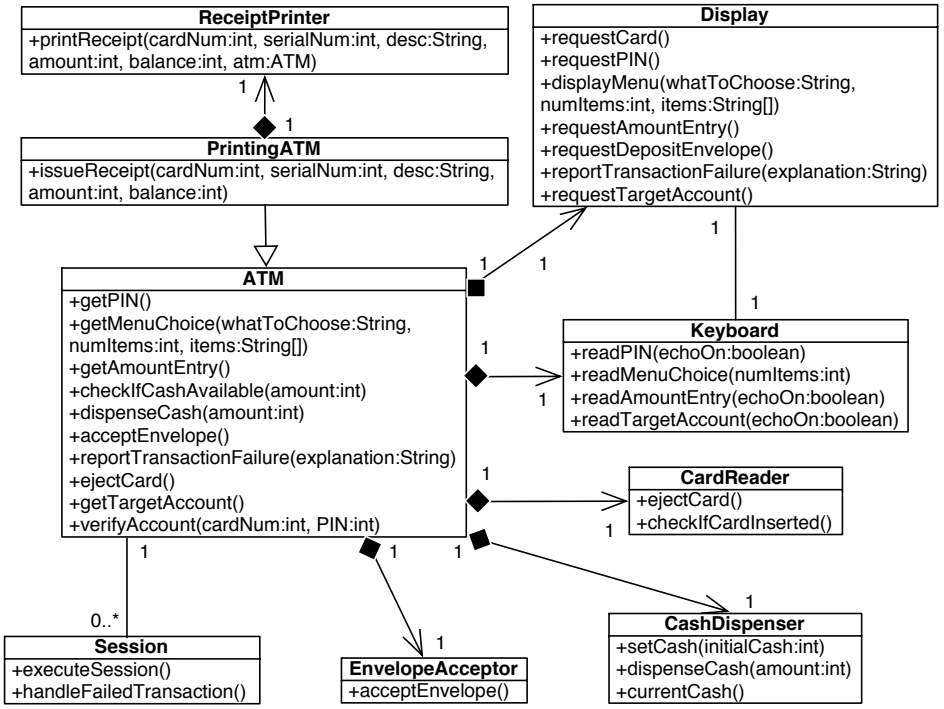


Figure 6.18: Class Diagram - Part I, version 4

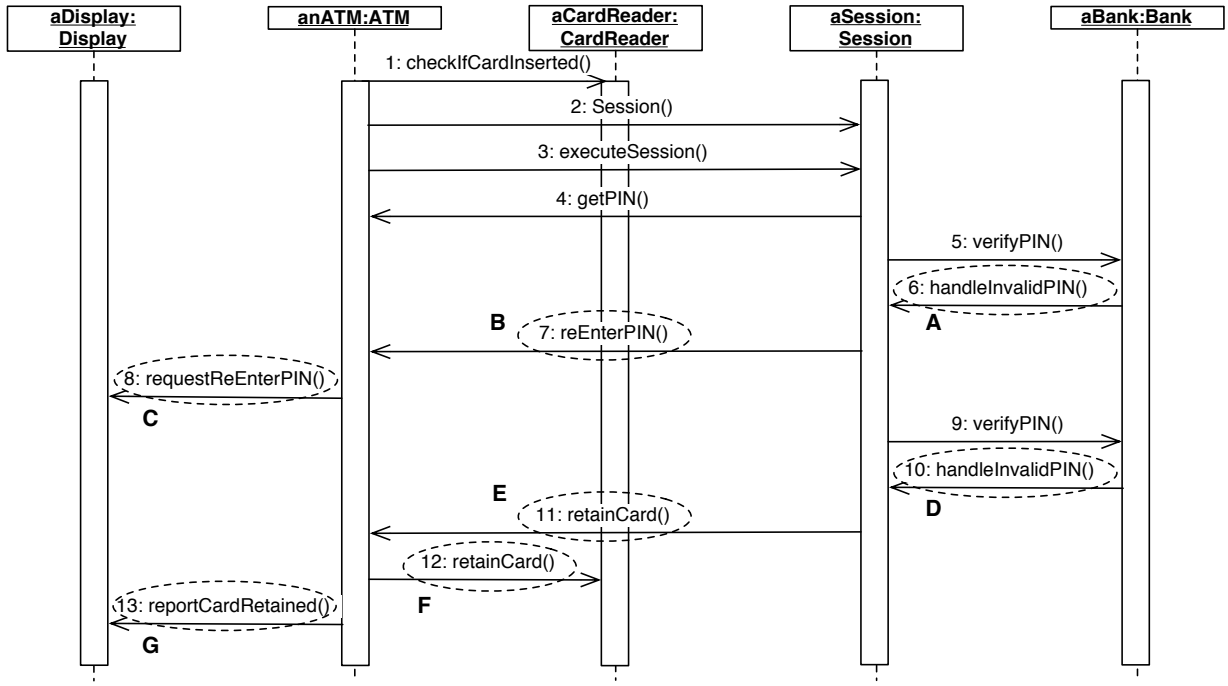


Figure 6.19: Sequence Diagram: dangling (inherited) feature references

changes in the static system definition, he may assume that this behaviour is valid and build on top of it, only to realise that this behaviour was not supported by the system when the system code is finally executed.

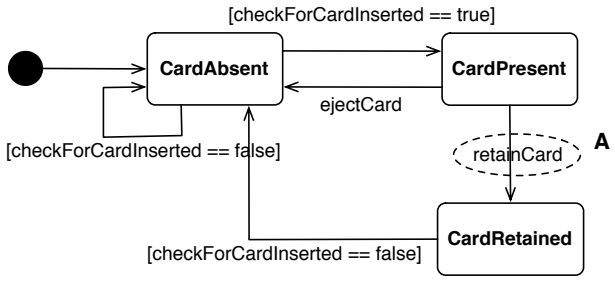


Figure 6.20: Card Reader State Diagram, version 2

The state diagram shown in figure 6.20 includes an example of this consistency problem, as it was not updated so as to include the modifications made to the class diagram. The event that triggers transition A refers to an operation that no longer exists in the definition of the system. There are no other transitions to the `CardRetained` state, and the existing one is invalid, so this state is unreachable, and should be removed from this diagram. If a member of the development team is not informed of the latest changes in the static system definition, he may assume that this state is reachable for instances of this class, and therefore will develop the code needed to process this state. This leads to wasted effort, since this code will never be executed.

***Predicates***

The predicates for this inconsistency are of two types: the ones that deal with stimuli, triggers, actions and guards that refer to non-existing operations, and the ones that deal with references to existing operations, but to which the class in question has no access, as there is no inheritance relationship between the class in question and the owner of the referenced operation. The first three predicates deal with non-existing operations. The last three predicates deal with existing operations. An auxiliary function is used in order to determine the if two classes are related through inheritance.

The following predicate queries the knowledge base directly, looking for stimuli that reference non-existing operations. The stimuli and their invalid operations are printed.

```
(do-retrieve (?link ?stimulus ?call-action ?operation)
 (:and
  (Link ?link)
  (Link-stimulus ?link ?stimulus)
  (Initiates ?stimulus ?call-action)
  (CallAction-operation ?call-action ?operation)
  (In-namespace ?operation NIL))
 (format t "Stimulus ~S in link ~S references non-existing operation ~S"
  ?stimulus ?link ?operation))
```

The following predicate queries the knowledge base directly, looking for transitions that are triggered by events that reference non-existing operations. The transitions, events and their invalid operations are printed.

```
(do-retrieve (?transition ?call-event ?operation)
 (:and
  (Transition ?transition)
  (Triggered-by ?transition ?call-event)
  (Is-occurence-of ?call-event ?operation)
  (In-namespace ?operation NIL))
 (format t "Transition ~S triggered by ~S references non-existing operation ~S"
  ?transition ?call-event ?operation))
```

The following predicate queries the knowledge base directly, looking for transitions that have effects that reference non-existing operations. The transitions and their invalid operations are printed.

```
(do-retrieve (?transition ?call-action ?operation)
 (:and
  (Transition ?transition)
  (Transition-action ?transition ?call-action)
  (CallAction-operation ?call-action ?operation)
  (In-namespace ?operation NIL))
 (format t "Transition ~S references non-existing operation ~S" ?transition
  ?operation))
```

An auxiliary function, `related`, returns T if the two classes are related by inheritance relationships, with the first argument being the supposed ancestor of the second argument. A class is considered as being "related" to itself, as this simplifies the conditional code of the predicates that uses it. If two classes are not related by inheritance relationships, NIL is returned.

```
(defun related (?parent ?class)
 (if (equalp ?parent ?class)
  T
  (do-retrieve (?child ?generalization)
   (:and
    (Is-specialization-of ?generalization ?parent)
    (Is-generalization-of ?generalization ?child)
    (:predcall #'related ?child ?class))
   (return T))))
```

The following predicate queries the knowledge base directly, looking for stimuli that reference existing operations to which the class in question has no access, as there is no inheritance relationship between the class in question and the owner of the referenced operation. The stimuli and their invalid operations are printed.

```
(do-retrieve (?link ?stimulus ?object ?class1 ?call-action ?operation ?class2)
 (:and
  (Link ?link)
  (Link-stimulus ?link ?stimulus)
  (Received-by ?stimulus ?object)
  (Instance-of-class ?object ?class1)
  (Initiates ?stimulus ?call-action)
  (CallAction-operation ?call-action ?operation)
  (not (In-namespace ?operation NIL))
```

```
  (Is-owned-by ?operation ?class2))
 (if (not (related (?class2 ?class1)))
  (format t "~S stimulus in link ~S references operation ~S, which is not
   available through inheritance" ?stimulus ?link ?operation)))
```

The following predicate queries the knowledge base directly, looking for transitions that are triggered by events that reference existing operations to which the class in question has no access, as there is no inheritance relationship between the class in question and the owner of the referenced operation. The transitions, events and their invalid operations are printed.

```
(do-retrieve (?transition ?state-machine ?class1 ?call-event ?operation
 ?class2)
 (:and
  (Transition ?transition)
  (Transition-stateMachine ?transition ?state-machine)
  (Is-stateMachine-of ?state-machine ?class1)
  (Triggered-by ?transition ?call-event)
  (Is-occurence-of ?call-event ?operation)
  (not (In-namespace ?operation NIL))
  (Is-owned-by ?operation ?class2))
 (if (not (related (?class2 ?class1)))
  (format t "Transition ~S triggered by ~S references operation ~S, which is
   not available through inheritance" ?transition ?call-event ?operation)))
```

The following predicate queries the knowledge base directly, looking for transitions that have effects that reference existing operations to which the class in question has no access, as there is no inheritance relationship between the class in question and the owner of the referenced operation. The transitions and their invalid operations are printed.

```
(do-retrieve (?transition ?state-machine ?class1 ?call-action ?operation
 ?class2)
 (:and
  (Transition ?transition)
  (Transition-stateMachine ?transition ?state-machine)
  (Is-stateMachine-of ?state-machine ?class1)
  (Transition-action ?transition ?call-action)
  (CallAction-operation ?call-action ?operation)
  (not (In-namespace ?operation NIL))
  (Is-owned-by ?operation ?class2))
 (if (not (related (?class2 ?class1)))
  (format t "Transition ~S references ~S, which is not available through
   inheritance" ?transition ?operation)))
```

***Results***

The first predicate was applied to the first example, in which various stimulus in a sequence diagram reference non-existing operations, as these were removed from the class diagram.

```
Stimulus |I|STIM6-1.0 in link |I|LINK6-1.0 references non-existing operation
 |I|HANDLEINVALIDPIN-SESSION-1.0
Stimulus |I|STIM7-1.0 in link |I|LINK7-1.0 references non-existing operation
 |I|REENTERPIN-ATM-1.0
```

```
Stimulus |I|STIM8-1.0 in link |I|LINK8-1.0 references non-existing operation
 |I|REQUESTREENTERPIN-DISPLAY-1.0
Stimulus |I|STIM10-1.0 in link |I|LINK10-1.0 references non-existing operation
 |I|HANDLEINVALIDPIN-SESSION-1.0
Stimulus |I|STIM11-1.0 in link |I|LINK11-1.0 references non-existing operation
 |I|RETAINCARD-ATM-1.0
Stimulus |I|STIM12-1.0 in link |I|LINK12-1.0 references non-existing operation
 |I|RETAINCARD-CARDREADER-1.0
Stimulus |I|STIM13-1.0 in link |I|LINK13-1.0 references non-existing operation
 |I|REPORTCARDRETAINED-DISPLAY-1.0
NIL
```

The second predicate was applied to the second example, in which a trigger in a state diagram references a non-existing operation, as it was removed from the class diagram.

```
Transition |I|TRANS4-1.0 triggered by |I|ACTION2-1.0 references non-existing
 operation  |I|RETAINCARD-CARDREADER-1.0
NIL
```

### Dangling (Inherited) Association Reference

Affected Diagrams: Class, Sequence

This inconsistency arises when a link in a sequence diagram is related to a non-existing association in a class diagram. This includes inherited associations that are lost when the inheritance links between classes are broken - these are referred to as *illegal* associations. This inconsistency can also be caused by the removal of existing associations, or by the omission of the necessary associations, when creating the class diagram - these are referred to as *non-existing* associations.

***Example***

In the UML model that consists of the diagrams shown in figures 6.21 and 6.22, two examples of this consistency problem can be seen. The first example appears when the inheritance relationship marked as `A` in figure 6.21 is deleted. As a result, the `PrintingATM` class does not inherit the association between the `ATM` and `CashDispenser` classes, which leaves the links `A` and `B` of the sequence diagram referencing an illegal association. The second example appears when the association between the `ATM` and `CashDispenser` classes, marked as `B` in the class diagram, is deleted. Now, links `A` and `B` of the sequence diagram refer to a non-existing association. The detection of this inconsistency will draw the attention of the designer to the missing association in the class diagram, allowing its correction.

***Predicates***

The predicates for this inconsistency are of two types: the ones that deal with links that refer to non-existing associations, and the ones that deal with references to existing associations, but to which the class in question has no access, as there is no inheritance relationship between the class in question and the owner of the referenced operation. The first predicate deals with non-existing associations. The last function deals with existing associations.

The following predicate queries the knowledge base directly, looking up links that reference non-existing associations. The links and their invalid associations are printed.

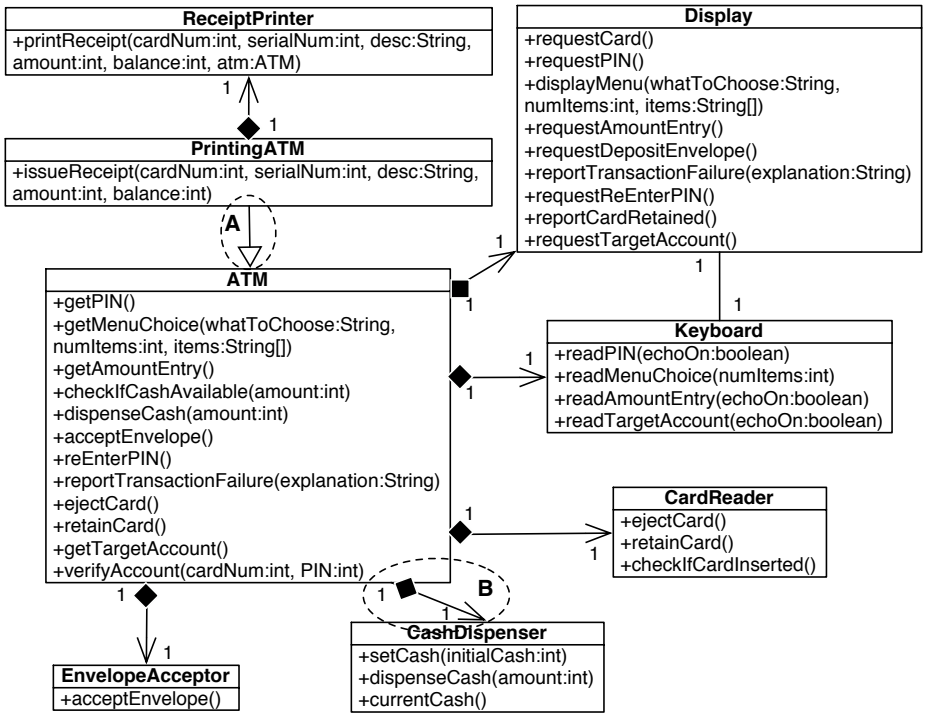


Figure 6.21: Class Diagram - Part I, version 5

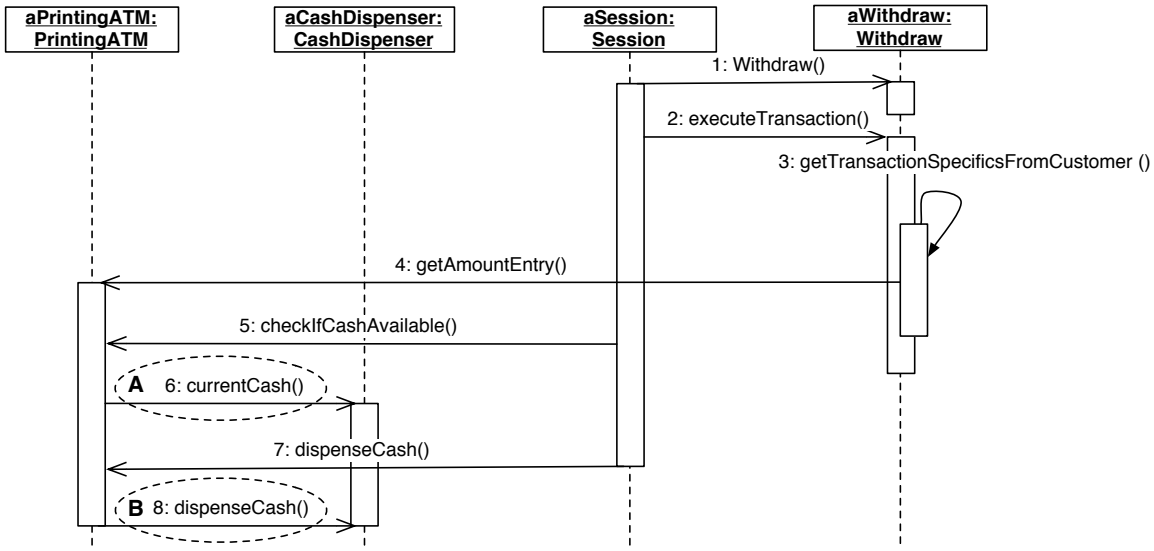


Figure 6.22: Sequence Diagram for withdraw transactions, version 2

```
(do-retrieve (?link ?association)
 (:and
  (Link ?link)
  (Link-association ?link ?association)
  (In-namespace ?association NIL))
 (format t "~S association does not exist in any class diagram" ?association))
```

An auxiliary function, `pred1`, is used in order to execute the conditions necessary so as to determine if the classes involved are related by inheritance. The `dangling-link` function calculates the information needed in order to determine if a link is referencing an existing association to which the class in question has no access, as there is no inheritance relationship between the class in question and the owner of the referenced operation. This information consists of the list of classes that participate in the association and the classes that are instantiated by the objects related by the link. Various combinations are given by Loom, as it matches the variables with all the possible sets of values, and for this reason, the list of classes has to be manipulated so as to check if there is at least one valid combination classes. If no valid combination of classes if found, the link cannot reference the association.

```
(defun pred1 (?list)
 (let ((?flag1 (related (first ?list) (third ?list)))
   (?flag2 (related (second ?list) (fourth ?list)))
   (?flag3 (related (first ?list) (fourth ?list)))
   (?flag4 (related (second ?list) (third ?list))))
  (or (and ?flag1 ?flag2) (and ?flag3 ?flag4))))

(defun dangling-link (?link ?association)
 (let* ((?list
  (retrieve (?class1 ?class2 ?class3 ?class4 ?assocEnd1 ?assocEnd2 ?stimulus
   ?object1 ?object2)
   (:and
    (Association-associationEnd ?association ?assocEnd1)
    (Association-associationEnd ?association ?assocEnd2)
    (Has-participant ?assocEnd1 ?class1)
    (Has-participant ?assocEnd2 ?class2)
    (Link-stimulus ?link ?stimulus)
    (Received-by ?stimulus ?object1)
    (Sent-by ?stimulus ?object2)
    (Instance-of-class ?object1 ?class3)
    (Instance-of-class ?object2 ?class4))))
   (?list2 (mapcar 'my-subseq ?list))
   (?list3 (mapcar 'pred1 ?list2)))
  (if (equalp NIL (or (first ?list3) (second ?list3) (third ?list3)
    (fourth ?list3)))
   (format t "~S association exists, but not available through inheritance"
    ?association)
   (format t "No problem"))))
```

***Results***

The first predicate was applied to the second example, where the association between the `ATM` and `CashDispenser` classes was removed. The results that were obtained indicate that the association between the `ATM` and `CashDispenser` does not exist.

```
|I|CASHDISPENSER-ATM-1.0 association does not exist in any class diagram
NIL
```

The second predicate was applied to the first example, where the inheritance link between the `ATM` and `PrintingATM` classes was removed. The results that were obtained indicate that, even though the association between the `ATM` and `CashDispenser` exists in the class diagram, the `PrintingATM` class no longer inherits it, and cannot reference it. `fi` is a Loom function that finds the individual associated to the Loom identifier given as an argument.

```
UML(270): (dangling-link (fi linka1-1.0) (fi cashdispenser-atm-1.0))
|I|CASHDISPENSER-ATM-1.0 association exists, but not available through
 inheritance
NIL
```

### 6.2.8 Disconnected Model

#### Disconnected Model - State

Affected Diagrams: State

This inconsistency arises when a diagram has one or various states or transitions that are not connected to the main diagram. This usually occurs when the connecting states or transitions have been removed from the diagram. It can also occur when the designer himself creates a disconnected diagram, for example, by forgetting to include a transition in a state diagram.

***Example***

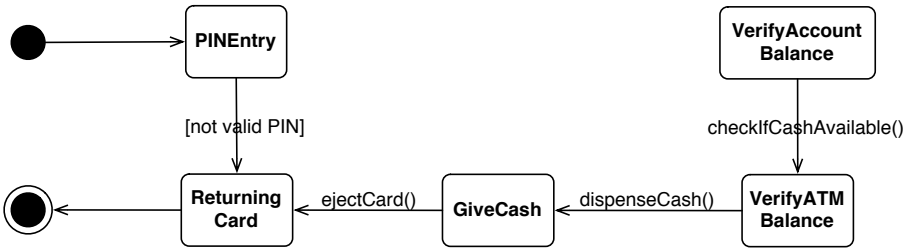


Figure 6.23: ATM State Diagram, version 3

In the UML model that consists of the diagram shown in figure 6.23, this consistency problem arises as the designer has deleted the `AmountEntry` state from the model (visible in figure 6.6). This causes the automatic deletion of the transitions to and from this state. The result is a disconnected diagram, as the states `VerifyAccountBalance`, `VerifyATMBalance` and `GiveCash` are not reachable from the initial state.

***Predicates***

In the case of state diagrams, this inconsistency is detected by checking that all the states in the diagram are reachable from the *initial* state. This is equivalent to checking if there exists at least one path from the initial state to each state. The `find-path` function does this recursively, using a query to find the transitions exiting from the current state, following each one until

either reaching the state in question, or arriving at a state that has no outgoing transitions, that is not a *final* state.

```
(defun find-path (?top-state ?state1 ?state2)
 (let* ((?cond (ask (:same-as ?state1 ?state2))))
  (if (equal ?cond T)
   (format t "State ~S is reachable~%" ?state2)
   (do-retrieve ()
    (:and
     (Is-container-of ?top-state ?state1)
     (for-some (?transition ?state)
      (:and
       (Is-source-of ?state1 ?transition)
       (Is-target-of ?state ?transition)
       (Is-container-of ?top-state ?state)
       (:predcall #'find-path ?top-state ?state ?state2))))
    ()))))
```

***Results***

The predicate was applied to the model presented in the example. The results that were obtained indicate that the states `VerifyAccountBalance`, `VerifyATMBalance` and `GiveCash` are not reachable, as no path could be found from the initial state to each one of them, and therefore, the model is disconnected. `fi` is a Loom function that finds the individual associated to the Loom identifier given as an argument.

```
UML(40): (find-path (fi top-asm-1.0) (fi initial-asm-1.0)
 (fi PINEntry-asm-1.0))
State |I|PINENTRY-ASM-1.0 is reachable
NIL
UML(41): (find-path (fi top-asm-1.0) (fi initial-asm-1.0)
 (fi VerifyAccountBalance-asm-1.0))
NIL
UML(42): (find-path (fi top-asm-1.0) (fi initial-asm-1.0)
 (fi VerifyATMBalance-asm-1.0))
NIL
UML(43): (find-path (fi top-asm-1.0) (fi initial-asm-1.0)
 (fi GiveCash-asm-1.0))
NIL
UML(44): (find-path (fi top-asm-1.0) (fi initial-asm-1.0)
 (fi ReturningCard-asm-1.0))
State |I|RETURNINGCARD-ASM-1.0 is reachable
NIL
UML(45): (find-path (fi top-asm-1.0) (fi initial-asm-1.0)
 (fi final-asm-1.0))
State |I|FINAL-ASM-1.0 is reachable
NIL
```

### Disconnected Model - Sequence

Affected Diagrams: Sequence

This inconsistency arises when a diagram has one or various objects that are not connected to the main diagram. This usually occurs when the connecting objects have been removed, which leads to the removal of the associated links. It can also be caused by the removal or omission of a link.

***Example***

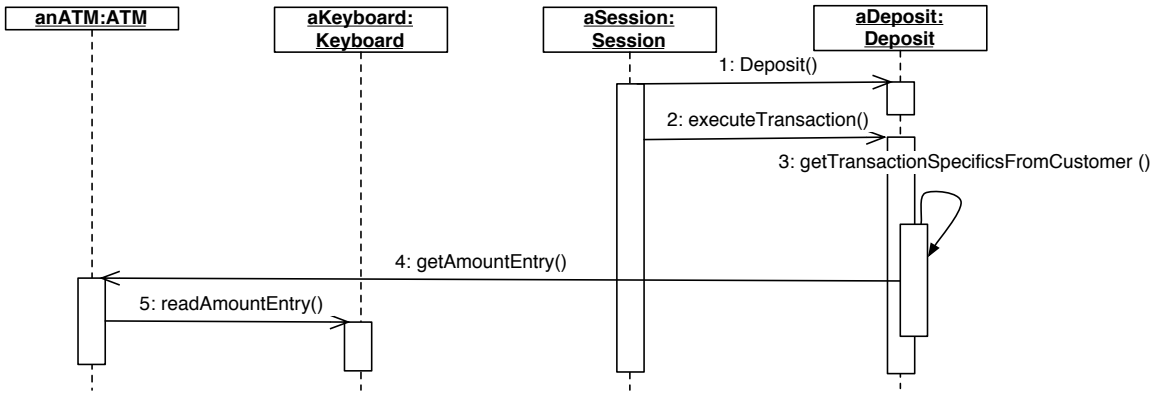


Figure 6.24: Sequence Diagram for deposit transactions, version 2

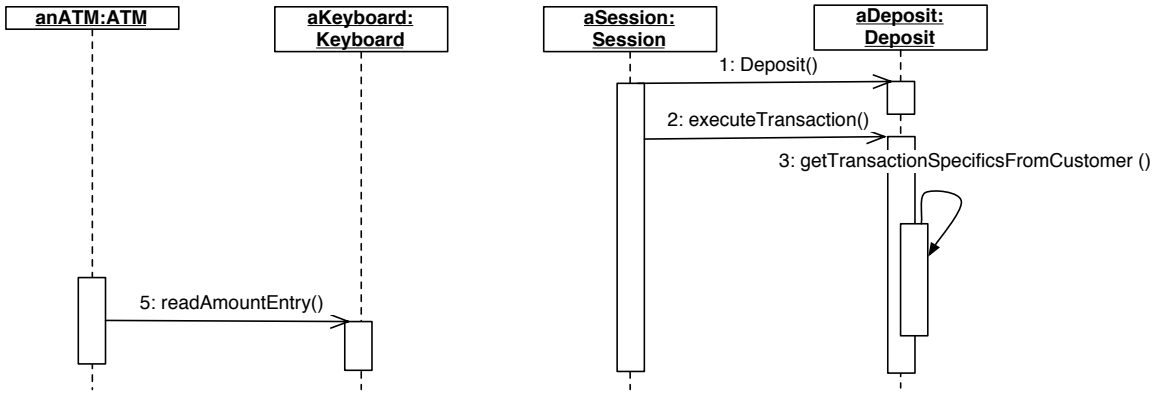


Figure 6.25: Sequence Diagram for deposit transactions, version 3 - deleted link

A valid sequence diagram is shown in figure 6.24, representing the interaction between system objects when the ATM user decides to make a deposit. In the first example, visible in figure 6.25, the diagram becomes disconnected when the link between the `aDeposit` and `anATM` objects is removed. In the second example, visible in figure 6.26, the diagram becomes disconnected when the `anATM` object is removed. This causes the removal of all the links to and from this object, resulting in a disconnected diagram, as the `aKeyboard` object has no interaction with the rest of the diagram.

***Predicates***

In the case of sequence diagrams, this inconsistency is detected by checking that all objects receive at least one stimulus. The only exception to this rule is the object that sends the first stimulus. This object will be detected as a false positive. The `check-dd-seq` function checks if

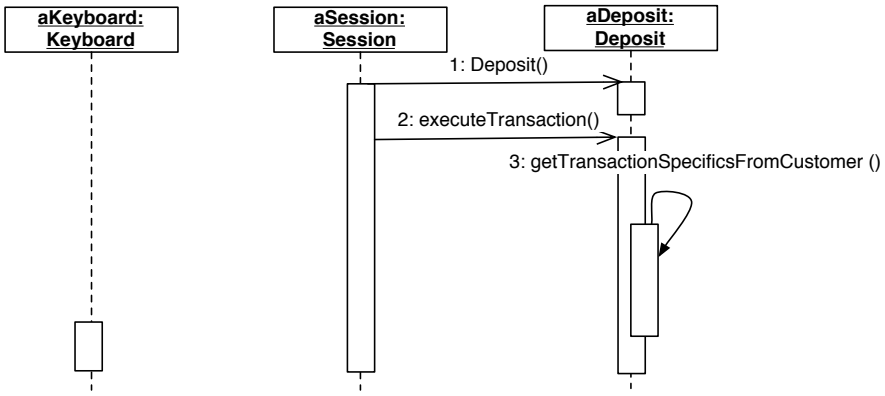


Figure 6.26: Sequence Diagram for deposit transactions, version 4 - deleted object

each object in a sequence diagram receives at least one stimulus. Objects that receive external stimuli in order to initiate the object interaction are detected as false positives.

```
(defun check-dd-seq (?diagram)
 (let* ((?received-stim
  (retrieve (?object ?stim)
   (:and (Object ?object)
    (In-namespace ?object ?diagram)
    (receiver-of ?object ?stim)))))
  (if (equalp ?received-stim NIL)
   (format t "~S receives no stimuli~%" ?object))))
```

***Results***

The predicate was applied to the model presented in the example. The results obtained from the first example indicate that the objects `anATM` and `aSession` receive no stimuli. In the case of the `anATM` object, this is true, as the removal of the link from the `aDeposit` object causes the division of the original model into two separate models. The detection of the `aSession` object is a false positive, as it receives an external stimulus that is not included in the model, in order to start the object interaction.

```
|I|ANATM-A-1.0 receives no stimuli
|I|ASESSION-A-1.0 receives no stimuli
NIL
```

The results obtained from the second example indicate that the objects `aKeyboard` and `aSession` receive no stimuli. In the case of the `aKeyboard` object, this is true, as the removal of the `anATM` object causes the division of the original model into two separate models. The detection of the `aSession` object is again a false positive, as it receives an external stimulus.

```
|I|AKEYBOARD-A-1.0 receives no stimuli
|I|ASESSION-A-1.0 receives no stimuli
NIL
```

## 6.3 Discussion

The lack of transitive roles is notorious in the case of inheritance hierarchies and inherited features. In order to reason about inherited features, the auxiliary role `Inherited-feature` was created. This role has to be filled manually (the user has to execute a query) and after being filled, needs to be truth maintained manually. This is cumbersome and error-prone. A similar problem can be seen when dealing with class inheritance relationships, as only one level of inheritance is visible at a time - class is either the parent or child of a `Generalization`. In this case, the approach that was taken was to query the knowledge base as to what the relationship between two classes was, instead of creating an `Ancestors` role that would have to be truth maintained. This second approach has the advantage that it does not need to be truth maintained, but the disadvantage that a recursive query has to be carried out on the knowledge base. The size of the examples used make the use of either approach feasible, but no real comparison has been made.

Whenever the results of a query have to be manipulated, the query has to be wrapped in a Lisp function, so that Lisp's list management facilities are available. Loom has some list manipulation predicates, but these are hard to use in comparison to the ones offered by Lisp. Queries can be quite long, as it is necessary, in some cases, to pass through various levels of indirection in order to obtain the information that is needed. In this study, the use of long queries has usually been favoured, as the task of truth maintaining auxiliary roles requires more effort.

Some functions are hard to read, as these make recursive calls when making predicate calls to themselves in the queries that represent the body of these functions. For example, `generate-trace2` (see Section 6.2.2), which is used to generate a special form of state machine traces. It is quite long, with various conditions in the function queries. With respect to the open world assumption, a work-around was used in order to switch from an open world to a closed world semantics and vice-versa, for individuals. This was necessary when wanting to reason about all the existing fillers of a role. This was done by manually closing a role by stating that the instance can have exactly as many role-fillers as it currently has. After executing the queries that needed a closed world, these assertions were removed from the knowledge base, which permits the addition of new role-fillers.

# Chapter 7

# Design Restructurings

In this chapter, the results of detecting design restructurings in UML models are presented and discussed. In the first section, the approach used for restructuring detection in this thesis, which relies solely on UML models, is explained. An excellent overview of current research on formalisms, techniques, languages and tools that apply to this problem has been compiled by Mens et. al [43]. Later in the section, the concrete application of the technique proposed in this thesis is presented, with respect to the detection of the *Pull Up Method* and *Push Down Method* refactorings [22].

## 7.1 Proposed Approach

Traditionally, refactorings and restructurings are applied at code level[50, 54]. An emerging trend is to deal with refactorings at a higher level of abstraction than source code, for example, with UML design models[5, 8]. One reason for this change is due to the advantages offered by working at this level. The source code has a lot of information, distributed through various locations, in a format in which abstractions are not easily deduced at a glance. Applying refactorings to system designs, in UML - for example, by representing the system design as diagrams, these abstractions are more apparent.

Another reason for this shift to the design level is that the UML abstract syntax is defined by means of a metamodel. This metamodel establishes the relationships between the different model elements, thus linking all the information present in a user-defined model. The DL approach followed in this study takes advantage of these relationships, as the UML metamodel is used as the base definition of the concepts and relations that define the logic world used (see Section 4.1), and user-defined model elements are created as instances of these concepts, ensuring well-formedness and consistency of the models through the use of predicates.

Current refactoring tools focus on applying refactorings when solicited by the user. For example, Boger et. al [8] have developed the *Refactoring Browser* for *Poseidon for UML*[26], a popular UML CASE tool. This CASE tool was used to develop the examples presented in this thesis. The *Refactoring Browser* allows the direct application of refactorings to class, state and activity diagrams. Before performing the refactoring, the browser issues warnings if the refactoring is likely to alter the model's behaviour. Error messages are generated if a modification will result in a inconsistent model. The refactorings offered are basic restructurings, for example, decomposing a composite state or deleting an attribute, corresponding to state and

class diagrams, respectively.

This still leaves the problem of deciding when and where restructurings should be applied. As the user has to state the model management information (see Section 3.3), the relationships between the model elements that belong to different models can be established, through the horizontal and evolution traces that link the different models. This information can be used to deduce where a refactoring should be applied, by determining the existence of certain necessary pre-conditions.

For example, a pre-condition for recommending a *Safe Remove Method*[22] refactoring is that the associated operation can not be called by any objects, meaning that it is not used. This is determined by analysing the sequence diagrams associated to the class diagram in which the operation's owner appears. If there are no instances of the owner class (or one of its descendants) that receives or sends a stimulus that calls the operation in question, the operation is not used, and could be removed from the design. In this way, it is possible to recommend restructurings, but it will always depend on the user if the recommendation is accepted or ignored.

## 7.2 Examples

### 7.2.1 Detecting a *Pull Up Method* Refactoring

The Banking Simulation described in Chapter 5 was slightly modified for this example. Figure 7.1 shows a modification that was made to the *Bank* class, where the *doTransaction()* operation was added, as a way of standardising bank notifications. As a result, the sequence diagrams that involve instances of the subclasses of the *Transaction* class have been modified. The sequence diagrams describing the new behaviour of the *Transfer* and *Inquiry* classes can be seen in figures 7.2 and 7.3 (the diagrams for *Withdraw* and *Deposit* have been omitted, but are similar). The original diagrams can be located in Section 5.2.

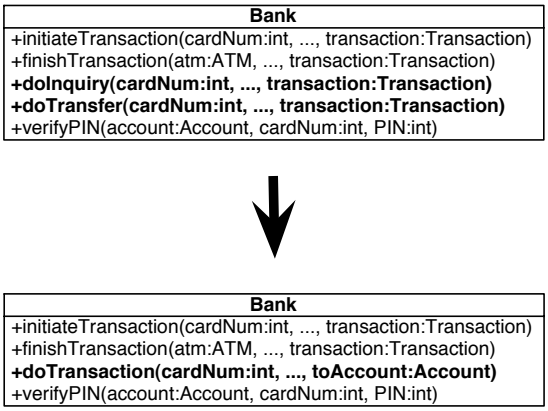


Figure 7.1: Bank class, versions 1 and 2

Improvements on the design of the class diagram based on information available in the sequence diagrams, can be suggested. More specifically, a *Pull Up Method* refactoring[22] can be proposed. A recurring factor in this set of sequence diagrams is that the operation *executeTransaction()* first calls the operation *getTransactionSpecificsFromCustomer()*, then the

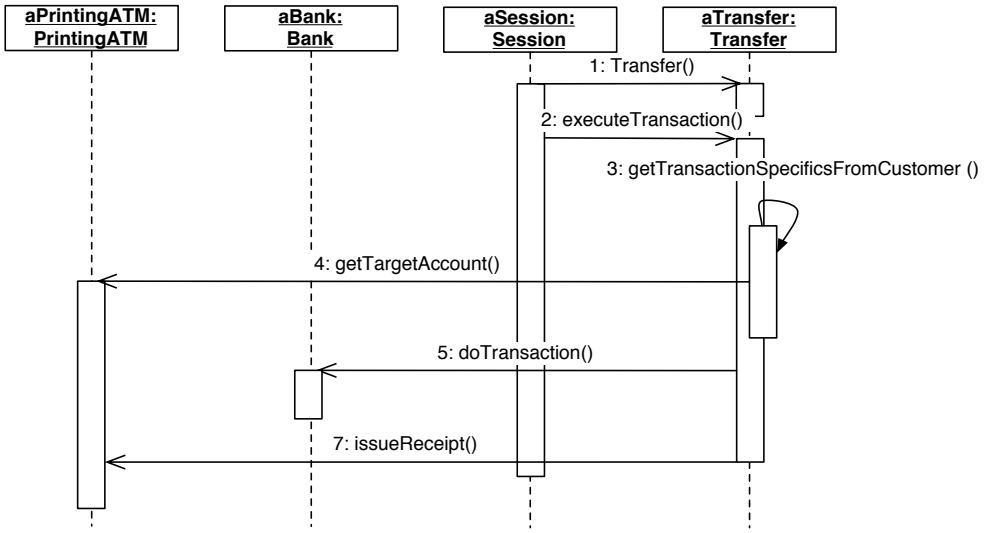


Figure 7.2: Sequence Diagram for transfer transactions, version 2

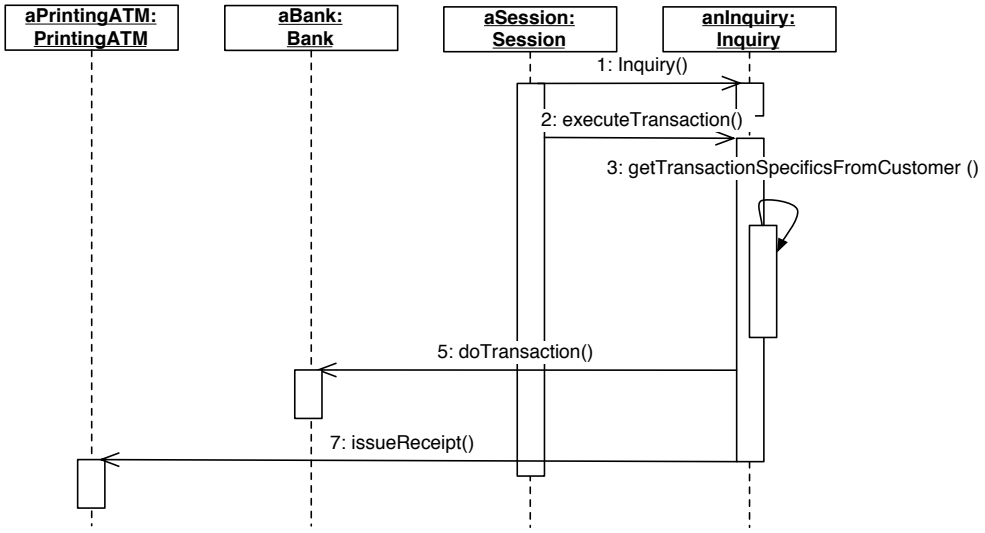


Figure 7.3: Sequence Diagram for inquiry transactions, version 2

*doTransaction()*, and afterwards, *issueReceipt()*. The body of the operation *getTransaction-SpecificsFromCustomer()* is different for each kind of transaction, while the body of the operation *executeTransaction()* is the same for each kind of transaction. This fact allows the conclusion that the operation *executeTransaction()* can be pulled up and become a concrete method of the abstract class *Transaction*.

### Predicates

The sequence diagrams considered show communicating instances. Subsequent arrows on the same lifeline map into *Stimuli* obeying the *predecessor* association in the UML metamodel. Each *Stimulus* has an instance of the *activator* association in the UML metamodel to the *Stimulus* corresponding to the incoming arrow of the activation. The *Stimulus* holds the sender and receiver *Instance*. To be able to detect a candidate operation to pull up, the roles *activator*, *predecessor* and *successor* (which is the inverse of *predecessor*) on the class *Stimulus* in the UML metamodel are used. The *activator*, *predecessor* and *successor* information had to be manually added to the knowledge base, because this information is not available in the XMI files representing the sequence diagrams generated by Poseidon.

The `get-activees` function generates the ordered collection of stimuli that are activated in response to receiving the activating stimulus. In a sequence diagram, this ordered collection (called *activees* in these experiments) is equivalent to the method body. Using a query, the list of stimuli that are activated by the parameter stimulus, is generated. Then, the stimuli are ordered sequentially. The ordered list is returned.

```
(defun get-activees (?stim1)
 (sort
  (retrieve (?name-action ?action ?name-stim ?stim2)
   (:and (Stimulus ?stim2)
   (Stimulus-activee ?stim1 ?stim2)
   (name ?stim2 ?name-stim)
   (Initiates ?stim2 ?action)
   (name ?action ?name-action)))
  '< :key 'third))
```

The following predicate finds and associates the corresponding *activees* to every stimulus in the knowledge base.

```
(do-retrieve (?stim)
 (Stimulus ?stim)
 (tellm (about ?stim (activees (:predcall #'get-activees ?stim)))))
```

The `pull-up-method` method receives a class as a parameter, and compares the *activee* sets of the *activator* stimulus corresponding to an operation, for all the instances of its direct subclasses. This is done by first finding an instance of one of these subclasses. Once an *activator* stimulus for this instance is found, it is necessary to check if similar *activator* stimuli exist for instances of the other subclasses. The pre-condition for this refactoring is that these *activator* stimuli must have the same *activee* set, for all the instances of the subclasses of the parent class that receive the same *activator* stimuli. If this condition is met, the user is then informed of the candidate methods that could be pulled up. Remark that the parent - child relationship has

to be manually closed[1], in order for the `for-all` statement to have effect. For example, in the *Transaction* example, that has four subclasses, the following assertion must be added to the knowledge base: `(tellm (about Transaction-1.0 (:exactly 4 Is-parent-of)))`

```
(defmethod pull-up-method (?parent ?oper ?gen ?child ?obj ?stim ?act)
 :situation
 (:and
  (Class ?parent)
  (Generalization ?gen)
  (Is-specialization-of ?gen ?parent)
  (Operation ?oper)
  (Is-generalization-of ?gen ?child)
  (Has-feature ?child ?oper)
  (Instance-of-class ?obj ?child)
  (Receiver-of ?obj ?stim)
  (Initiates ?stim ?act)
  (:same-as (name ?act)(name ?oper))
  (:about ?stim (:at-least 1 Stimulus-Activee))
  (for-all (?gen1)
   (:implies
    (Is-specialization-of ?gen1 ?parent)
    (for-some (?child2 ?obj2 ?stim2 ?act2)
     (:and
      (Is-generalization-of ?gen1 ?child2)
      (Instance-of-class ?obj2 ?child2)
      (Receiver-of ?obj2 ?stim2)
      (Initiates ?stim2 ?act2)
      (:same-as (name ?act2)(name ?oper))
      (:about ?stim2 (:at-least 1 Stimulus-Activee))
      (:same-as (activees ?stim)(activees ?stim2)))))))
 :response ((LOOM::say "Operation" (get-value ?oper 'name) "should be pulled up from
  subclass" (get-value ?child 'name) "to parent class" (get-value ?parent 'name))))
```

The following predicate queries the knowledge base so as to find the necessary arguments for the `pull-up-method` method.

```
(do-retrieve (?parent ?oper ?gen ?child ?obj ?stim ?act)
 (:and
  (Class ?parent)
  (Generalization ?gen)
  (Is-specialization-of ?gen ?parent)
  (Is-generalization-of ?gen ?child)
  (Has-feature ?child ?oper)
  (Instance-of-class ?obj ?child)
  (Receiver-of ?obj ?stim)
  (Initiates ?stim ?act))
 (perform (pull-up-method ?parent ?oper ?gen ?child ?obj ?stim ?act)))
```

### Results

The class diagram shown in figure 7.4 shows the portion of the Banking Simulation static structure that illustrates the *Transaction* class hierarchy *before* any changes were applied to

[1]This tells the reasoning engine that the current fillers of the relationship are all the fillers it will have.

the *Bank* class and the sequence diagrams (see figures 7.1, 7.2 and 7.3). The class diagram shown in figure 7.5 shows the same portion as figure 7.4, *after* including the recommended changes, that were derived from the information contained in the sequence diagrams. As expected, after applying the predicates to the example, it has been recommended that the *executeTransaction()* method be pulled up from the *Transaction* subclasses to the *Transaction* class.

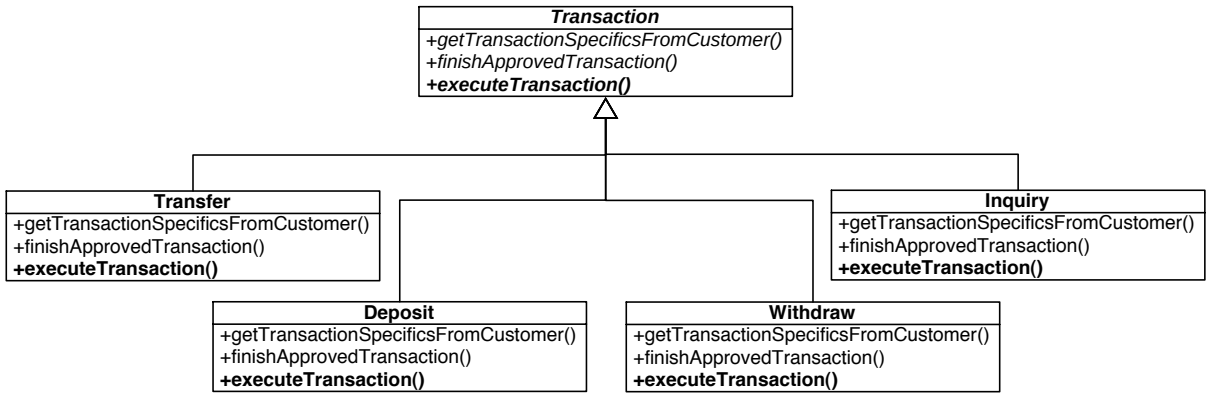


Figure 7.4: Transaction hierarchy, version 1

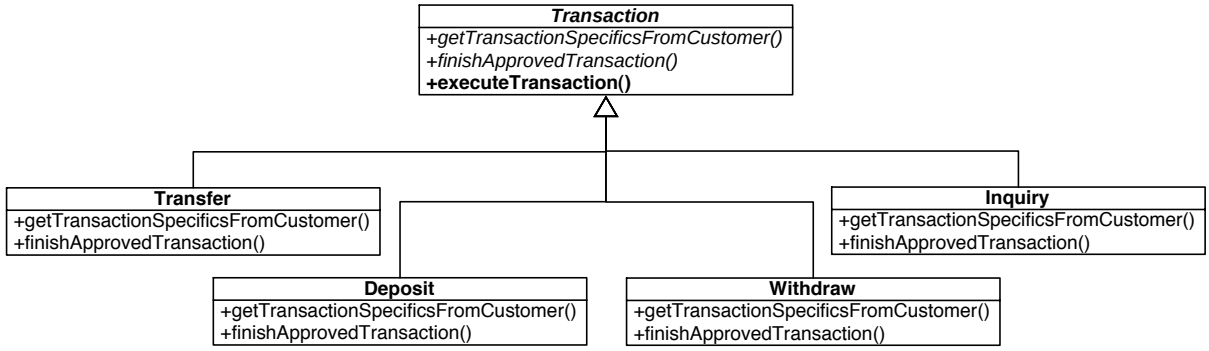


Figure 7.5: Transaction hierarchy, version 2

```
Operation |I|EXECUTETRANSACTION should be pulled up from the subclass |I|INQUIRY to
 parent class |I|TRANSACTION
Operation |I|EXECUTETRANSACTION should be pulled up from the subclass |I|DEPOSIT to
 parent class |I|TRANSACTION
Operation |I|EXECUTETRANSACTION should be pulled up from the subclass |I|WITHDRAW to
 parent class |I|TRANSACTION
Operation |I|EXECUTETRANSACTION should be pulled up from the subclass |I|TRANSFER to
 parent class |I|TRANSACTION
```

### 7.2.2 Detecting a *Push Down Method* Refactoring

The Banking Simulation described in Chapter 5 was slightly modified for this example. Figure 7.6 shows a modified version of the *ATM* class hierarchy, were the *issueReceipt()* operation is

owned by the *ATM* class, and not by the *PrintingATM* class, as in the original example. The sequence diagrams that involve instances of the classes of the *ATM* class hierarchy have not been modified. The sequence diagrams describing the behaviour of the *Deposit* class, one with respect to the *ATM* class, and the other with respect to the *PrintingATM* class, are shown in figures 7.7 and 7.8, respectively.

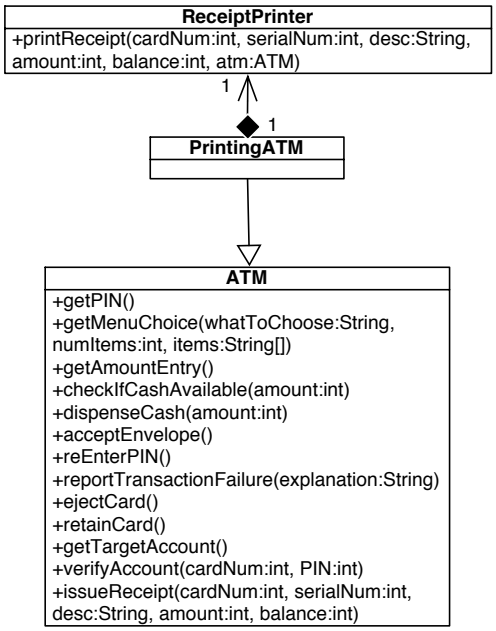


Figure 7.6: ATM hierarchy, version 1

Improvements on the design of the class diagram based on information available in the sequence diagrams, can be suggested. More specifically, a *Push Down Method* refactoring[22] can be proposed. In this set of sequence diagrams, the operation *issueReceipt()* is only sent to instances of the *PrintingATM* class, and never instances of the *ATM* class, that is the owner of the operation. This fact allows the conclusion that the operation *issueReceipt()* can be pushed down and become a method of the class *PrintingATM*.

### Predicates

The `generate-rec-sent-operations` function generates the set of stimuli that are sent or received by all the instances of a class, that is passed as a parameter. This list includes the information about the operations that are called by the stimuli.

```
(defun generate-rec-sent-operations (?class)
 (retrieve (?name ?operation ?object ?stim ?action)
  (:and
   (Instance-of-class ?object ?class)
   (:or
    (Receiver-of ?object ?stim)
    (Sender-of ?object ?stim))
   (Initiates ?stim ?action)
```

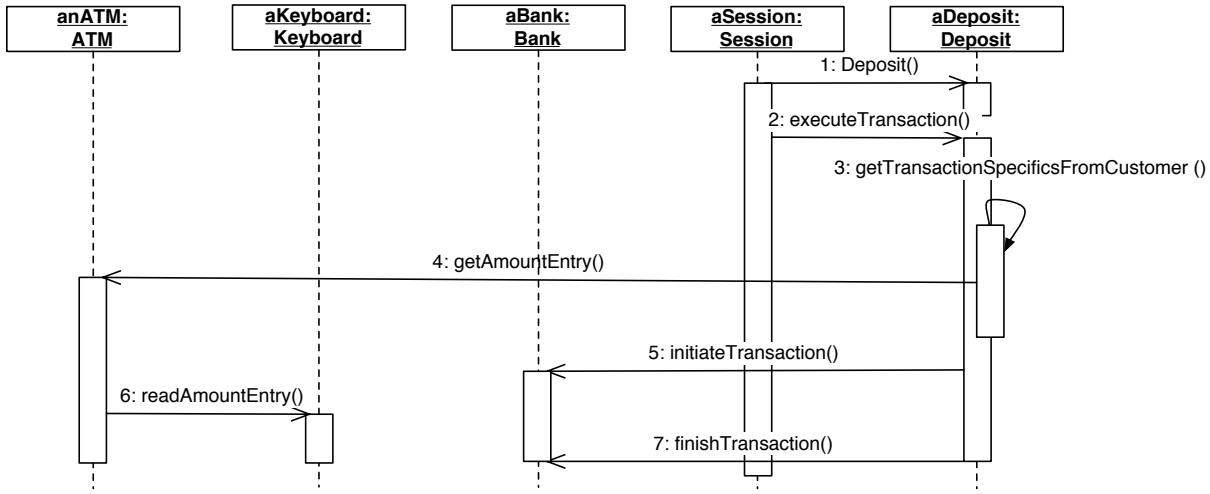


Figure 7.7: Sequence Diagram for deposit transactions, version 1

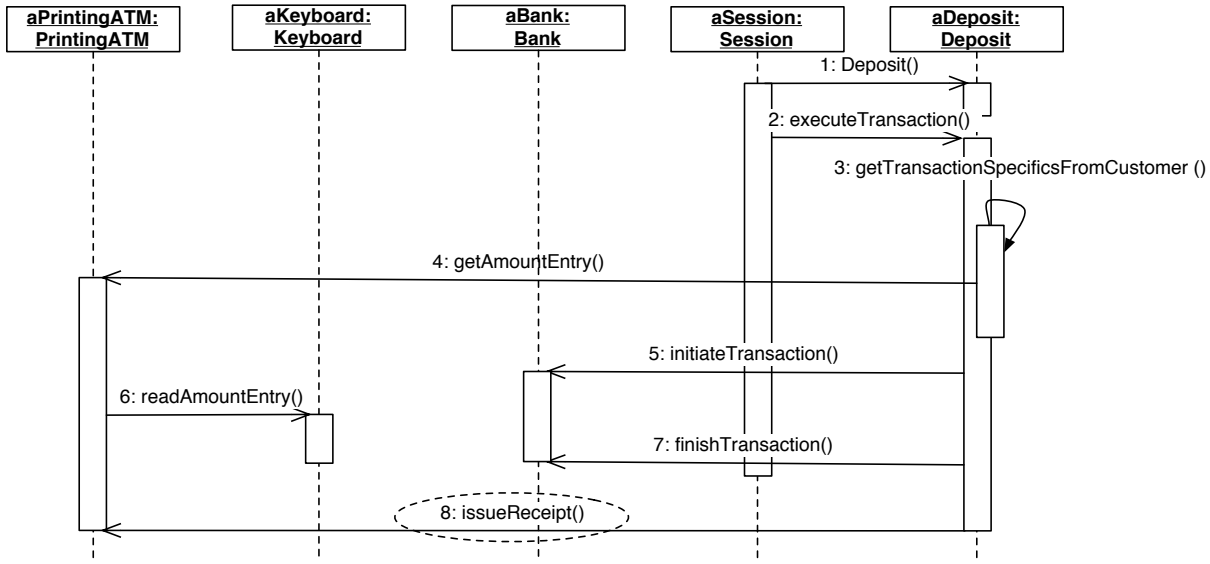


Figure 7.8: Sequence Diagram for deposit transactions, version 2

```
   (CallAction-operation ?action ?operation)
   (name ?operation ?name))) )
```

The `check-use` function recursively checks if an operation has been used by the instances of a class. This is done by generating the list of stimuli that are sent and received by all the instances of the class, using the `generate-rec-sent-operations`, and passing it to this function as a parameter. This function then takes the first element of the list, compares it to the operation passed as a parameter. If these match, the operation is being used by instances of the class. If not, the function is re-invoked, with the original operation and rest of the list as parameters.

```
(defun check-use (?oper ?list)
 (if (equalp NIL ?list)
  (format NIL "NOT USED")
  (if (equalp (first (car ?list)) (get-value ?oper 'name))
   (format NIL "USED")
   (check-use ?oper (cdr ?list)))))
```

The `push-down-method` function checks whether an operation is invoked exclusively by direct subclasses of the operation's owner. This is done by first checking if the operation is used by the instances of the owner class. Then, a query is used to find the direct subclasses of the owner class. As with the owner class, each subclass is checked, so as to determine if the operation is received or sent by at least one instance of the subclass. Finally, a condition is checked for each subclass found, so as to determine if the operation is used by the subclass. If this condition is met, the user is then informed of the candidate methods that could be pushed down, and into which subclasses.

```
(defun push-down-method (?parent ?oper)
 (let ((?list2) (?flag2) (?list1) (?flag1))
  (setq ?list1 (generate-rec-sent-operations ?parent))
  (setq ?flag1 (check-use ?oper ?list2))
  (do-retrieve (?gen ?child)
   (:and
    (Is-generalization-of ?gen ?child)
    (Is-specialization-of ?gen ?parent))
   (progn ()
    (setq ?list2 (generate-rec-sent-operations ?child))
    (setq ?flag2 (check-use ?oper ?list2))
    (if (and (equalp ?flag1 "NOT USED") (equalp ?flag2 "USED"))
     (format t "Operation ~S in class ~S should be pushed down to class ~S~%"
      (get-value ?oper 'name) ?parent ?child))))))
```

The following predicate queries the knowledge base so as to find the necessary arguments for the `push-down-method` operation.

```
(do-retrieve (?parent ?oper)
 (:and
  (Class ?parent)
  (Has-feature ?parent ?oper))
 (push-down-method ?parent ?oper))
```

### Results

The class diagram shown in figure 7.6 shows the portion of the Banking Simulation static structure that illustrates the *ATM* class hierarchy, as modified for this example. The class diagram shown in figure 7.9 shows the same portion as figure 7.6, *after* including the recommended changes, that were derived from the information contained in the sequence diagrams. As expected, after applying the predicates to the example, it has been recommended that the *issueReceipt()* method can be pushed down from the *ATM* class to the *PrintingATM*.

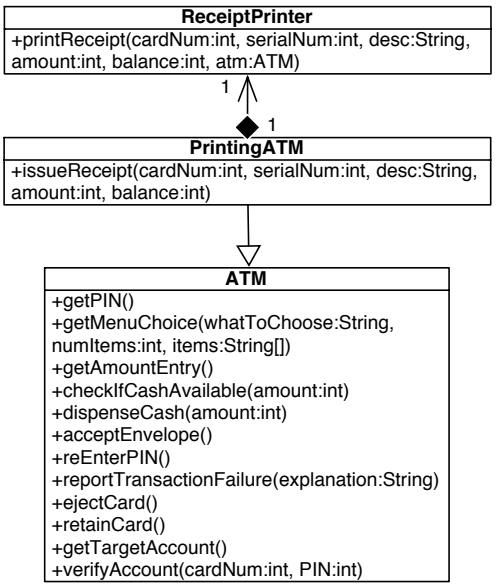


Figure 7.9: ATM hierarchy, version 2

```
Operation |I|ISSUERECEIPT in class |I|ATM-1.0 should be pushed down to class
 |I|PRINTINGATM-1.0
NIL
```

# Chapter 8

# Tool Support

In this chapter, *Conan - Consistency Analyser for UML* is introduced. This tool implements the ideas presented in this thesis, allowing users to inspect and analyse UML models in a formal manner. It receives as input UML models, provided in XMI format by *Poseidon for UML*[26], which are then translated into logic code and asserted into a knowledge base maintained by the *Loom*[35] DL reasoning engine. This tool has been implemented in Java, and uses SAXON[36] for the XMI - logic code translation. Figure 8.1 shows the main functionalities offered by *Conan*: assertion of model evolution information, model element information and detection and solution predicates.

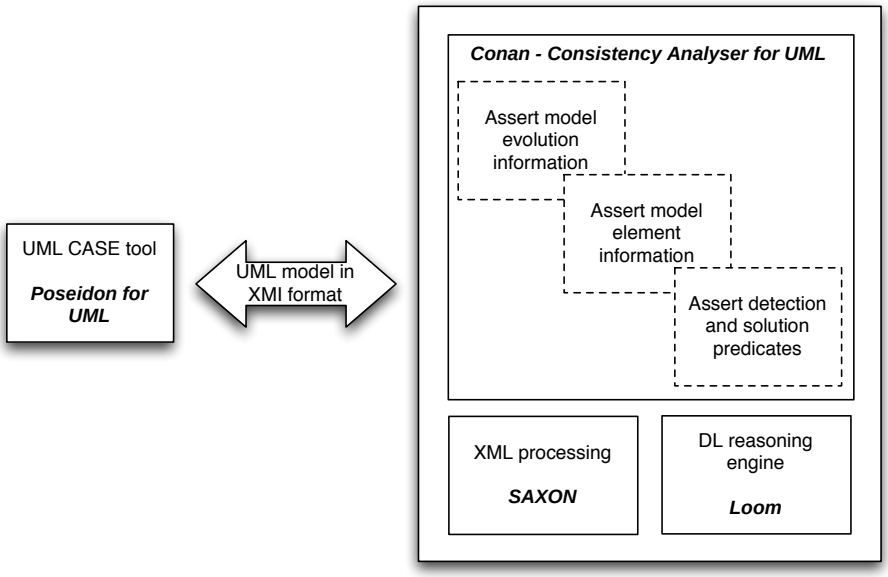


Figure 8.1: Conan - Consistency Analyser for UML

## 8.1 Asserting Model Element Information

UML models are created by users with *Poseidon for UML*. This application stores the information present in these diagrams in XMI format. The user must then load his model files into *Conan*, using the `Model` menu. This causes the translation of the XMI data into logic assertions. These are automatically loaded into a knowledge base, maintained by *Loom*. The user can immediately start analysing the consistency of these models individually, or can create relationships (see Section 8.2) between the currently loaded models, and analyse the consistency of models with respect to other models.

In the case of certain model elements, additional information has to be given, in order for the detection predicates to work. This is because this information is not present in the UML metamodel, and as a result, is not present in the *Poseidon*-generated XMI files. These new relationships and attributes are visible in the proposed UML extension presented in Chapter 3. An example is the successor - predecessor relationship between stimuli in a sequence diagram. This allows reasoning with respect to the ordering of the stimuli, but is not included in the UML metamodel. As a result, the user must add this information. A `Model View` window is opened for each model loaded.

## 8.2 Asserting Model Evolution Information

As mentioned in Section 1.1, current UML CASE tools do not provide tools adequate versioning capabilities[26, 4]. This is due to the lack of support for these ideas in the current specifications of UML[48]. The current UML metamodel provides poor support for consistency preservation and software evolution, e.g. versions are not supported. The idea of maintaining a link between different versions of a model is not present. As a result, models are saved in different files, and it is not possible to link the elements of these models. It is not possible to specify relationships between models, so this information is not available in the model data provided by these CASE tools.

In *Conan*, this information is specified by creating horizontal and evolution traces between currently loaded models. These relationships can be created through the `Add` menu of a `Design View` window, that shows the models and traces that belong to a composite model. These relationships are automatically integrated into the working knowledge base, allowing the immediate detection of horizontal and evolution-based inconsistencies. Figure 8.2 shows how primitive models are added to a composite model.

## 8.3 Asserting Detection and Solution Predicates

Once all the necessary model and model element information has been loaded, the user can start analysing his models. *Conan* currently supports inconsistency detection for the conflicts presented in this thesis, involving class, sequence and state diagrams. In this version, only detection predicates are available. Predicates are applied by choosing them from the `Predicate` menu. Figure 8.3 shows the application of the *Disconnected Model* detection predicate to composite model. The results are shown in the `Output View` window.

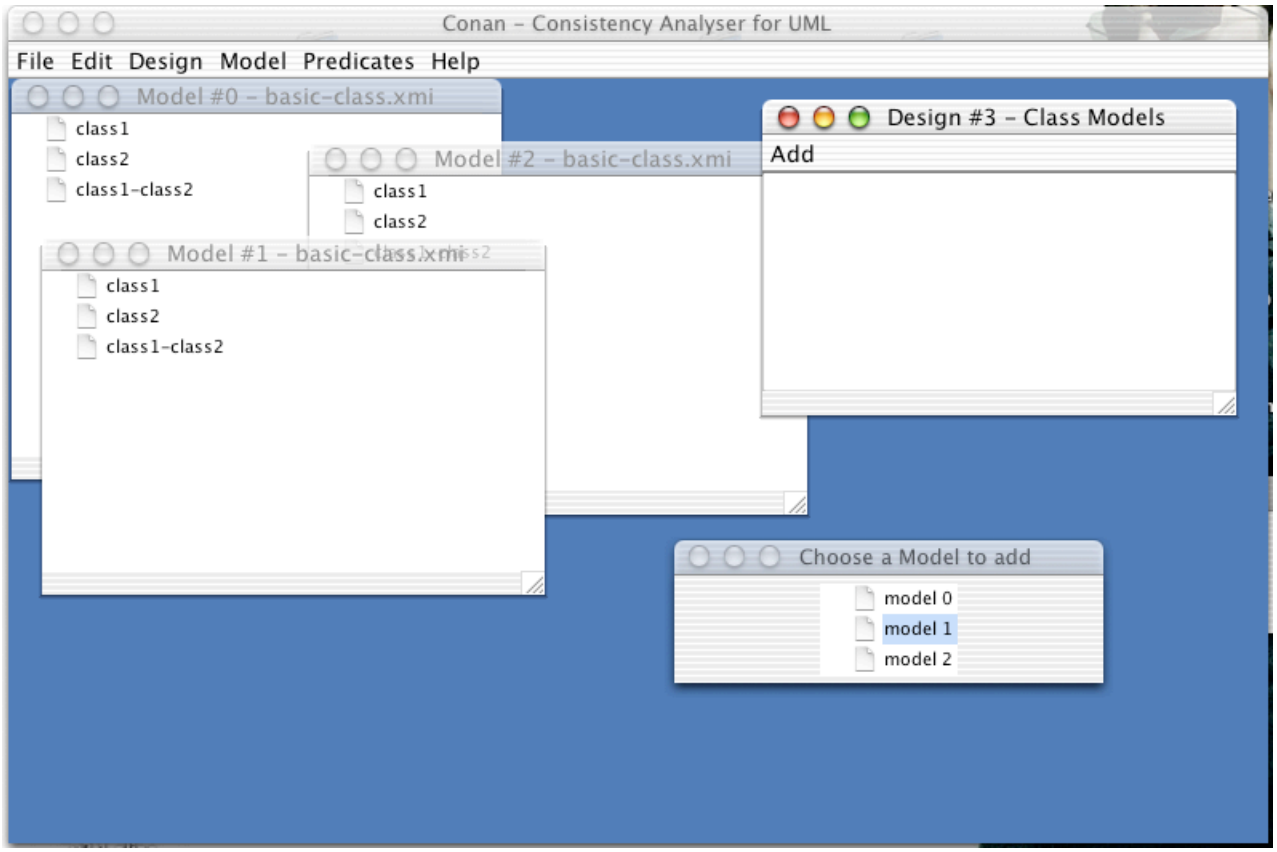


Figure 8.2: Screenshot - adding primitive models to a composite model

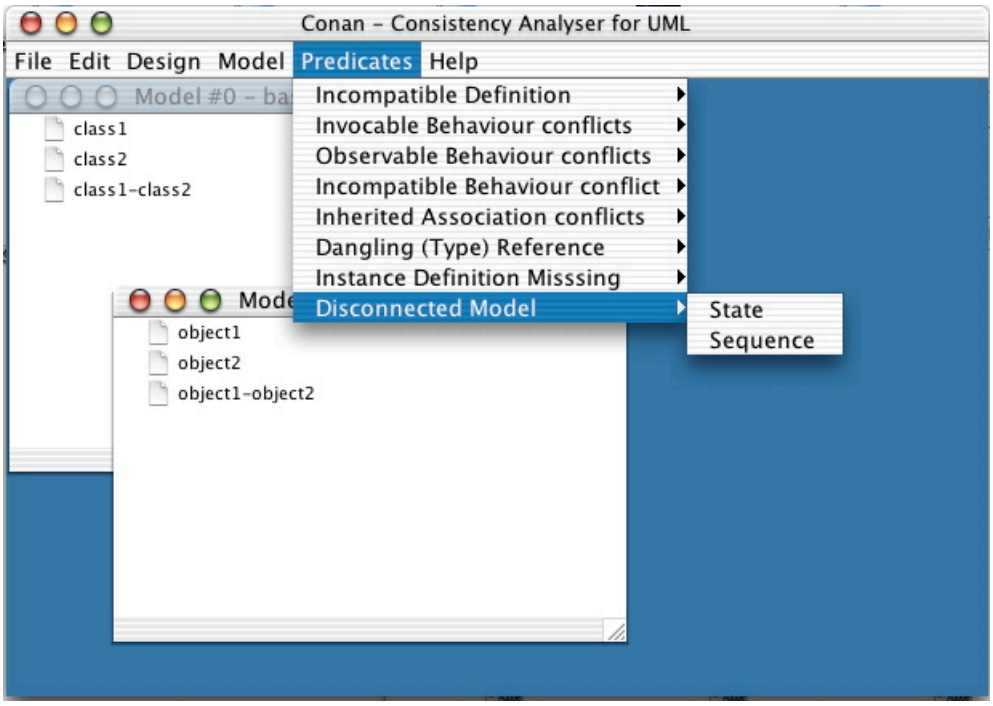


Figure 8.3: Screenshot - detecting incompatible behaviour

# Chapter 9

# Conclusion

In this chapter, an overview of this thesis and the related work is provided. The main research questions are revisited and the methodology used to explore them is discussed. This chapter also summarises the main findings, and discusses the significant issues and recommendations emerging from the study. Finally, limitations and directions for future research are presented.

## 9.1 Related Work

Current CASE tools do not provide any support for evolution of UML models. In this thesis, a UML profile that allows model management, has been studied. The underlying ideas are based on earlier work to extend the UML metamodel with support for evolution [45, 44].

Finkelstein et al. [21] explain that consistency between partial models is neither always possible nor is it always desirable. They suggest the use of temporal logic to identify and handle inconsistencies. This formalism is used to describe sequences of actions that lead to inconsistencies, unlike the approach taken in this thesis that uses logic to find conflicting instances.

Grundy et al. [28] 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. They describe their experience with building complex multiple-view software development tools supporting inconsistency management facilities. The DL approach is also easily configurable, by adding, removing or modifying logic rules in the knowledge base.

In addition, Finkelstein [20] has elaborated a list of the technical challenges that arise when trying to build a toolset that deals with evolution and consistency. Tools dealing with these two aspects should help establish, express and reason about the relationships between formal languages and check consistency with respect to these relationships and to provide diagnostic feedback. Where inconsistencies have been detected, these tools should help visualise the inconsistencies. The user should be able to specify policies with respect to when consistency should be checked and when resolution mechanisms should be applied. This approach has been used as a guideline for the tool development in this study (see Chapter 8).

A wide range of different approaches for checking consistency has been proposed in the literature. Engels et al. [18] motivate a general methodology to deal with consistency problems based on the problem of protocol statechart inheritance. In that example, statecharts as well as the corresponding class diagrams, are important. Communicating Sequential Processes (CSP)

are used as a mathematical model for describing the consistency requirements. This idea is further enhanced in [16, 19] with dynamic meta modelling rules as a notation for the consistency conditions because of their graphical, UML-like notation. Model transformation rules are used to represent evolution steps, and their effect on the overall model consistency is explored. Using DL, it was possible to detect similar consistency problems.

Ehrig and Tsiolakis [15] 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 [60] the information specified in class and statechart 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. [24] use systems of linear inequalities to check consistency for multiple view software architectures.

The problem of verifying whether the interactions expressed by a collaboration diagram can indeed be realised by a set of state machines, has been treated by Schfer et al.[58]. They have developed HUGO, a prototype tool that checks if state machines (compiled into PROMELA [32] models), and collaborations (translated into sets of Büchic automata) match up, using the SPIN model checker to verify the model against the automata. This problem has also been analysed by Litvak et al. [38], using an algorithmic approach, instead of using external model checkers. They have put their ideas into practice, by implementing the BVUML tool, that receives the state and sequence diagrams as XMI files produced by ArgoUML.

The work presented in this thesis is related to design refactorings in two ways: with DL, refactorings to improve a given design can be suggested (see Chapter 7), or models can be checked so as to determine if a given design refactoring was performed correctly, in the sense that it preserves design consistency (see Chapter 6). Both uses go beyond current support for UML refactorings [57, 8] as for example that is being integrated into CASE tools such as Poseidon.

Another aspect related to evolving UML models, but not treated in this thesis, is co-evolution. Also known as vertical consistency, it deals with maintaining consistency at different levels of refinement. The situation that has been studied the most is the synchronisation of design and source code artefacts. Wuyts[64] uses logic meta-programming to support co-evolution of object-oriented design and source code, implementing a synchronisation framework, SOUL, that provides a layered library of logic rules that allows reasoning about Smalltalk code. Del Valle[14] uses this frame-work to provide general support for round-trip engineering between UML models and Java source code.

Finally, note that consistency of models should not be confused with consistency of a modelling language. UML has been formalised within rewriting logic and implemented in the Maude system by Ambrosio Toval and his students [1, 59]. Their objectives are to formalise UML and transformations between different UML models. They focus on using reflection to represent and support the evolution of the metamodel.

## 9.2 Summary

In this thesis, an approach for detection and resolution of inconsistencies between different versions of a UML model, specified as a collection of class, sequence and state diagrams, is proposed and validated. For research purposes, this study is restricted to a significant subset of the UML metamodel. This profile was extended so as to permit the representation of a diagram as a unit. This extended UML profile also allows the specification of the relationships between different diagrams, using horizontal and evolution traces.

The formalism used is description logic, a decidable fragment of first-order predicate logic. More specifically, the *Loom* knowledge representation tool is used to formally specify UML models as a collection of concepts and roles. Logic rules were used to specify consistency constraints, to suggest ways to resolve inconsistencies, and even to propose ways to improve the design by means of refactorings. Based on simple but illustrative examples, the feasibility of the approach is shown.

Based on a number of already performed small-scale experiments, it has been demonstrated that DL tools can be very helpful in detecting design inconsistencies and resolving these inconsistencies semi-automatically. Some initial experiments were carried out in the field of design restructurings, were it was demonstrated that it is possible to detect and suggest refactorings, on the UML level.

Introducing evolution steps at the UML level has the advantage that it captures and shows the system abstractions in a visual manner, whereas these same abstractions are scattered throughout the code. As a result, the author believes that giving the designer the opportunity to apply changes directly to UML diagrams will make it easier to maintain evolving systems. Even more, this should strengthen the design - source code link, as the designs will effectively be an accurate visual representation of the code.

Keeping design and source code synchronised will reinstate the role of the UML models as a documentation tool. What is increasingly frequent in projects today is that UML is used to provide an initial design, from which skeleton code is generated, but as the source code is modified, the design is not updated because the effort and hours necessary to carry this out is often seen as expendable. Project managers prefer to allocate these resources to the actual implementation effort, so as to meet deadlines.

Another advantage of working at the UML level, is that a generic, language independent approach can be offered, as UML maps to most object-oriented languages. Language-specific restructurings could also be included, as the approach used is easily configurable, through the assertion of facts in the knowledge base.

Concluding, this thesis offers a generic framework for working with evolving UML models, with functional consistency checking schemes. Unlike other studies, that are dedicated to the exhaustive study of a few consistency problems with a certain technique, this study has tried to group an exhaustive set of the possible consistency problems[1], and uses the same approach to deal with them all. The author believes that this approach will be considered useful by users, as it provides uniform support for the different consistency problems.

[1] Eighteen consistency problems were found when analysing class, state and sequence diagrams.

## 9.3 Future Work

The work done in this dissertation can be continued in several directions:

- Further develop the framework, incorporating support for all the UML diagram types where the DL approach is applicable. It will be necessary to study exactly how these diagrams are related to those already incorporated in the study - the links between the different model elements will have to be deduced and formalised.
- The expressiveness of the supported diagrams must be extended. This means that the supported UML metamodel must be extended, and more OCL well-formedness rules must be included. Changes introduced by the new UML standard, version 2.0, will have to be studied and introduced.
- Due to the amount of existing OCL well-formedness rules, it would be interesting to study the possibility of of automatically translating these into DL.
- This study demonstrated that it was possible to use the DL approach in order to detect design restructurings. The experiments concerning refactorings were limited to the study of two refactorings. Currently, more than 80 refactorings have been recognised, and Fowler[22] maintains an actualised Alphabetical List[23] on his Refactoring site. These have to be analysed, so as to determine if they can be applied to UML models and, if so, what information is necessary for their detection.
- A design decision taken with respect to the detection predicates is that these only inform the user about the inconsistency. This allows the separation of detection and solution, as some inconsistencies are introduced deliberately by designers and their solution is not always desired. Solution predicates have been developed for some of the detection predicates. This coverage has to be extended to the rest of the predicates.
- The case study analysed in this thesis is small, and the examples used to test the detection predicates were elaborated so as to produce the inconsistencies. The scalability of the DL approach should be studied, specially with legacy systems as case studies, as these would provide larger examples, have natural evolution steps, and the inconsistencies have not been artificially introduced.
- The concepts and relations that define the *Tbox* used should be studied, in order to see if it is possible to define additional concepts and relations (not from the UML metamodel) that will let the detection code be shorter, more understandable and efficient. Refining the *Tbox* should lead to a revision of the detection predicates, allowing for their optimisation and refinement.
- The current generation of DL tools use tableau calculus, that allows the implementation of more expressive DLs, while being more efficient. *Loom* is less expressive than these DLs, but it was chosen as the DL tool for this thesis because of the query language it provides. For example, *RACER*, one of these new-generation DLs, has a client-server architecture, which is ideal for communication between applications, but has no query language, which makes writing detection predicates more complex.
- The ideas presented in this thesis can be extended, so as to deal with co-evolution and consistency maintenance between different levels of abstraction, more specifically, source

code and UML models. This idea, which is also explored in [27] will allow the provision of better formal support for the round-trip engineering and model-driven architecture process.

- Finally, it would be nice to change from the distributed application approach, which was easier to manage while studying the problem, to the actual implementation of a UML CASE tool plugin, making the use of the detection predicates transparent for user. The other alternative is to using a DL tool as the underlying engine for a UML CASE tool,

# Bibliography


[1] J.L.F. Alemán, A. Toval, and J.R. Hoyos. Rigorously transforming UML class diagrams. In *Proc. 5th Workshop Models, Environments and Tools for Requirements Engineering (MENHIR)*, 2000. Universidad de Granada, Spain. 102

[2] Lori Alperin Resnick, Alex Borgida, Ronald J. Brachman, Deborah L. McGuinness, and Peter F. Patel-Schneider. *CLASSIC Description and Reference Manual for the COMMON LISP*. AI Principles Research Department, AT&T Bell Laboratories, version 2.3 edition, 1995. 9

[3] Carlos Areces. *Logic Engineering. The Case of Description and Hybrid Logics*. PhD thesis, ILLC, University of Amsterdam, 2000. 7

[4] argouml.tigris.org. ArgoUML, April 8 2003. http://argouml.tigris.org. 1, 36, 98

[5] Dave Astels. Refactoring with UML. In *Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, pages 67–70, 2002. Alghero, Sardinia, Italy. 87

[6] F. Baader, D. McGuinness, D. Nardi, and P.F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003. 6

[7] Russell C. Bjork. An example of Object-Oriented Design: An ATM Simulation, April 8 2003. http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample/. 38

[8] Marko Boger, Thorsten Sturm, and Per Fragemann. Refactoring Browser for UML. In *Proc. 3rd Int'l Conf. on eXtreme Programming and Flexible Processes in Software Engineering*, pages 77–81, 2002. Alghero, Sardinia, Italy. 87, 102

[9] Marko Boger, Thorsten Sturm, and Erich Schildhauer. *Poseidon for UML Users Guide*. Gentleware, June 2002. http://www.gentleware.com/products/documentation/PoseidonUsersGuide/book1.html. 1, 36

[10] Alex Borgida, Ronald J. Brachman, Deborah L. McGuinness, and Lori Alperin Resnick. CLASSIC: A Structural Data Model for Objects. In *ACM SIGMOD International Conference on Management of Data*, pages 59–67, 1989. 9

[11] R. Brachman. *A Structural Paradigm for Representing Knowledge*. PhD thesis, Harvard University, 1977. 9

[12] R. Brachman and H. Levesque. The tractability of nsubsumption in frame-based description languages. *AAAI-84*, pages 34–37, 1984. 7

[13] David Brill. *LOOM reference manual*. University of Southern California, Information Sciences Institute, version 2.0 edition, December 28 1993. 10, 22, 28

[14] João G. Del Valle. Towards round-trip engineering using logic metaprogramming. Master's thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium and Ecole des Mines de Nantes, France, 2003. 102

[15] H. Ehrig and A. Tsiolakis. Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In H. Ehrig and G. Taentzer, editors, *ETAPS 2000 workshop on graph transformation systems*, pages 77–86, March 2000. 102

[16] G. Engels, J.H. Hausmann, R. Heckel, and St. Sauer. Testing the consistency of dynamic UML diagrams. In *Proc. Sixth International Conference on Integrated Design and Process Technology (IDPT 2002)*, June 2002. Pasadena, CA, USA. 2, 20, 102

[17] G. Engels, J. M. Küster, L. Groenewegen, and R. Heckel. A methodology for specifying and analyzing consistency of object-oriented behavioral models. *In V. Gruhn (ed.): Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, pages 186–195, September 2001. 2, 20

[18] 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, *Proc. Int'l Conf. UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, number 2185 in Lecture Notes in Computer Science, pages 272–286. Springer-Verlag, October 2001. Toronto, Canada. 48, 55, 58, 60, 62, 65, 101

[19] 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, *Proc. Int'l Conf. UML 2002 - The Unified Modeling Language. Model Engineering, Concepts, and Tools*, number 2460 in Lecture Notes in Computer Science, pages 212–227. Springer-Verlag, October 2002. Dresden, Germany. 102

[20] A. Finkelstein. A Foolish Consistency: Technical Challenges in Consistency Management. In I. Ibrahim, J. Küng, and N. Revell, editors, *Database and Expert Systems Applications, 11th International Conference, DEXA 2000, London, UK, September 2000. Proceedings*, Lecture Notes in Computer Science 1873, pages 1–5. Springer Verlag, 2000. 101

[21] Anthony Finkelstein, Dov M. Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multi-perspective specifications. In *European Software Engineering Conference*, LNCS, pages 84–99. SpringerVerlag, 1993. 101

[22] Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999. 3, 87, 88, 93, 104

[23] Martin Fowler. Alphabetical List of Refactorings, July 15 2003. http://www.refactoring.com/catalog/index.html. 104

[24] Pascal Fradet, Daniel Le Métayer, and Michaël Périn. Consistency checking for multiple view software architectures. In *Proc. Int'l Conf. ESEC/FSE'99*, volume 1687 of *Lecture Notes in Computer Science*, pages 410–428. Springer-Verlag, 1999. 102

[25] Free Software Foundation. GNU Emacs Lisp Reference Manual version 2.5, Nov 1998. http://gnu.digitaltrust.it/manual/elisp-manual-20-2.5/elisp.html. 23

[26] gentleware.com. Poseidon, April 8 2003. http://www.gentleware.com/products/poseidonPE.php3. 19, 36, 87, 97, 98

[27] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. A UML extension for automating source-consistent design improvements based on refactoring contracts. In *Proc. 6th International Conference on the Unified Modeling Language*, 2003. 105

[28] 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. 101

[29] Volker Haarslev and Ralf Möeller. RACE System Description. In *International Workshop on Description Logics - DL-99*, pages 133–132, 1999. 9, 12

[30] Volker Haarslev and Ralf Möller. *RACER UserÕs Guide and Reference Manual*. Concordia University and Univ. of Appl. Sciences in Wedel, version 1.7.6 edition, Dec 2002. http://www.fh-wedel.de/~mo/racer/download.html. 12

[31] Volker Haarslev and Ralf Möller. RACER, April 8 2003. http://www.fh-wedel.de/~mo/racer/. 8, 12

[32] G.J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice. 102

[33] Ian Horrocks. FaCT and iFaCT. In *International Workshop on Description Logics - DL-99*, pages 133–135, 1999. 9

[34] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for expressive description logics. In Harald Ganzinger, David McAllester, and Andrei Voronkov, editors, *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, pages 161–180. Springer-Verlag, 1999. 12

[35] ISI. Loom, April 2003. http://www.isi.edu/isd/LOOM/LOOM-HOME.html. 10, 28, 97

[36] Michael Kay. SAXON, April 2003. http://saxon.sourceforge.net. 37, 97

[37] B. Liskov. Data abstraction and hierarchy. In L. Power and Z. Weiss, editors, *Addendum to the Proc. OOPSLA-87: Object-Oriented Programming Systems, Languages and Applications*, pages 17–34. acm Press, New York, NY, 1988. 55, 58, 60

[38] Boris Litvak, Shmuel Tyszberowicz, and Amiram Yehudai. Consistency Validation of UML Diagrams. In *Correctness of Model-based Software Composition (CMC) Workshop, ECOOP*, 2003. 102

[39] Robert MacGregor. Inside the LOOM description classifier. *SIGART Bull.*, 2(3):88–92, 1991. 9, 10, 22

[40] Robert MacGregor. Retrospective on Loom, August 13 1999. 10, 22

[41] Eric Mays, Robert Dionne, and R. Weida. K-Rep System Overview. *SIGART Bull.*, 2(3):93–97, 1991. 9

[42] Kim Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, 2000. 6

[43] Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. Refactoring: Current research and future trends. In *Proc. 3rd Workshop on Language Descriptions, Tools and Applications*, volume 82(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003. 87

[44] Tom Mens and Theo D'Hondt. Automating support for software evolution in uml. *Automated Software Engineering Journal*, 7(1):39–59, February 2000. 101

[45] Tom Mens, Carine Lucas, and Patrick Steyaert. Supporting disciplined reuse and evolution of uml models. In J. Bezivin and P.-A. Muller, editors, *Proc. UML'98 - Beyond The Notation*, volume 1618 of *Lecture Notes in Computer Science*, pages 378–392. Springer-Verlag, 1999. Mulhouse, France. 101

[46] Joaquin Miller and Jishnu Mukerji. Model driven architecture (MDA). Draft ormsc/2001-07-01, Architecture Board ORMSC, July 2001. 46

[47] Object Management Group. Unified Modeling Language specification version 1.4, Sept 2001.
http://www.omg.org/uml. 19

[48] Object Management Group. Unified Modeling Language specification version 1.5. formal/2003-03-01, March 2003.
http://www.omg.org/uml. 14, 15, 16, 17, 19, 20, 48, 55, 60, 65, 98

[49] omg.org. Object management group, April 8 2003.
http://www.omg.org. 36

[50] William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992. 3, 87

[51] Peter Patel-Schneider. DLP System Description. In *International Workshop on Description Logics - DL-98*, pages 87–89, 1998. 9

[52] Peter F. Patel-Schneider. CLASSIC, April 8 2003.
http://www.bell-labs.com/project/classic/index.html. 9

[53] Chris Peltason. The BACK System - An Overview. *SIGART Bull.*, 2(3):114–119, 1991. 9

[54] Donald B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999. 3, 87

[55] M. Schmidt-Schauss and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991. 7

[56] Jean Louis Sourrouille. UML behavior: Inheritance and implementation in current object-oriented languages. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Fort Collins, CO, USA, Proceedings*, volume 1723 of *LNCS*, pages 457–472. Springer-Verlag, 1999. 56, 62, 65

[57] G. Sunyé, D. Pollet, Y. LeTraon, and J.-M. Jézéquel. Refactoring UML models. In *Proc. UML 2001*, volume 2185 of *Lecture Notes in Computer Science*, pages 134–138. Springer-Verlag, 2001. 102

[58] A. Knapp T. Schfer and S. Merz. Model checking uml state machines and collaborations. *In Electronic Notes in Theoretical Computer Science*, 47:1–13, 2001. 102

[59] A. Toval and J.L.F. Alemán. Formally modeling UML and its evolution: a holistic approach. In S.F. Smith and C.L. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems IV*, pages 183–206. Kluwer Academic Publishers, 2000. 102

[60] Aliki Tsiolakis. Semantic analysis and consistency checking of UML sequence diagrams. Master's thesis, Technische Universität Berlin, April 2001. Technical Report No. 2001-06. 102

[61] w3.org. World Wide Web Consortium, April 8 2003. `http://www.w3.org`. 37

[62] w3.org. The XPath Specification v 1.0, April 8 2003. `http://www.w3.org/TR/1999/REC-xpath-19991116`. 37

[63] w3.org. The XSLT Specification v 1.0, April 8 2003. `http://www.w3.org/TR/1999/REC-xslt-19991116`. 37

[64] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, 2001. 102

# Appendix A

# Code Translations

## A.1 UML Profile

```
(load "outliner.lisp")
(defpackage "UML" (:use "COMMON-LISP"))
(in-package "UML")
(use-loom "UML")
(set-features :display-match-changes :emit-match-stars :emit-classify-dots)
(unset-features :ask-for-concept-names)

(creation-policy :classified-instance)
(change-context 'UML-THEORY)
(initialize-network)

(defrelation name :is (:and String) :characteristics :single-valued)

(defrelation version :characteristics :single-valued)

(defrelation the-prev-ver :characteristics :single-valued)

(defrelation Instance-of-subclass-of :characteristics :single-valued)

(defrelation operations-received :characteristics :single-valued)

(defrelation kind :characteristics :single-valued)

(defrelation added :characteristics :single-valued)

;role definition
;R action-state
(defrelation Entry-action-of :domain Action :range State)
(defrelation Has-entry-action :is (:inverse Entry-action-of))
;
;R action-state
(defrelation Exit-action-of :domain Action  :range State)
(defrelation Has-exit-action :is (:inverse Exit-action-of))
;
;R action-state
(defrelation Do-activity-of :domain Action :range State)
```

```
(defrelation Has-do-activity :is (:inverse Do-activity-of))
;
;R action-actionsequence
(defrelation Has-action :domain ActionSequence :range Action)
(defrelation Action-of :is (:inverse Has-action))
;
;R action-argument
(defrelation Has-argument :domain Action  :range Argument)
(defrelation Argument-of :is (:inverse Has-argument))
;
;R action-stimulus
(defrelation Initiated-by :domain Action :range Stimulus)
(defrelation Initiates :is (:inverse Initiated-by))
;
;R action-transition
(defrelation Action-transition :domain Action :range Transition)
(defrelation Transition-action :is (:inverse Action-transition)
 :characteristics :single-valued)
;
;R object-class
(defrelation Instance-of-class :domain Object :range Class)
(defrelation Instantiated :is (:inverse Instance-of-class))
;
;R stimulus-object
(defrelation Sender-of :domain Object :range Stimulus)
(defrelation Sent-by  :is (:inverse Sender-of))
;
;R stimulus-object
(defrelation Receiver-of :domain Object :range Stimulus)
(defrelation Received-by :is (:inverse Receiver-of))
;
;R operation-object (NOT FROM METAMODEL)
(defrelation Receives-call :domain Object :range Operation)
(defrelation Received-by-object :is (:inverse Receives-call))
;
;R operation-object (NOT FROM METAMODEL)
(defrelation Sends-call :domain Object :range Operation)
(defrelation Sent-by-object :is (:inverse Sends-call))
;
;R stimulus-operation (NOT FROM METAMODEL)
(defrelation Sends-operation :domain Stimulus :range Operation)
(defrelation Sent-by-stimulus :is (:inverse Sends-operation))
;
;R stimulus-association (NOT FROM METAMODEL)
(defrelation Related-to :domain Stimulus :range Association)
(defrelation association-stimulus :is (:inverse Related-to))
;
;R class-operation (NOT FROM METAMODEL)
(defrelation Inherited-feature :domain Class :range Operation)
(defrelation Inherited-by :is (:inverse Inherited-feature))
;
;R stimulus-object
(defrelation Argument-of-stimulus :domain Object :range Stimulus)
(defrelation Has-object-argument :is (:inverse Argument-of-stimulus))
```

```
;
;R stimulus-stimulus
(defrelation Stimulus-Predecessor :domain Stimulus :range Stimulus)
(defrelation Stimulus-Successor :is (:inverse Stimulus-Predecessor))
;
;R stimulus-stimulus
(defrelation Stimulus-Activator :domain Stimulus :range Stimulus)
(defrelation Stimulus-Activee :is (:inverse Stimulus-Activator))
;
;R attributeLink-updateAction
(defrelation Updates-attribute :domain AttributeLink :range UpdateAction)
(defrelation Is-updated-by :is (:inverse Updates-attribute))
;
;R attributeLink-accessAction
(defrelation Accesses-attr :domain AttributeLink :range AccessAction)
(defrelation Is-accessed-by :is (:inverse Accesses-attr))
;
;R object-attributeLink
(defrelation Has-slots :domain AttributeLink :range Object)
(defrelation Is-slot-of :is (:inverse Has-slots))
;
;R attributeLink-attribute
(defrelation AttributeLink-attribute :domain AttributeLink :range Attribute)
(defrelation Attribute-attributeLink :is (:inverse AttributeLink-attribute))
;
;R compositeState-stateVertex
(defrelation Is-container-of :domain CompositeState :range StateVertex)
(defrelation Is-subvertex-of :is (:inverse Is-container-of) :characteristics
 :single-valued)
;
;R stateMachine-state
(defrelation StateMachine-state :domain StateMachine :range State)
(defrelation State-stateMachine :is (:inverse StateMachine-state))
;
;R stateVertex-transition
(defrelation Is-outgoing-of :domain Transition :range StateVertex)
(defrelation Is-source-of :is (:inverse Is-outgoing-of))
;
;R stateVertex-transition
(defrelation Is-incoming-of :domain Transition :range StateVertex)
(defrelation Is-target-of :is (:inverse Is-incoming-of))
;
;R state-transition
(defrelation Is-internalTransition-of :domain Transition :range State)
(defrelation Has-internalTransition :is (:inverse Is-internalTransition-of))
;
;R operation-callAction
(defrelation Operation-callAction :domain Operation :range CallAction)
(defrelation CallAction-operation :is (:inverse Operation-callAction))
;
;R stateMachine-transition
(defrelation StateMachine-transition :domain StateMachine :range Transition)
(defrelation Transition-stateMachine :is (:inverse stateMachine-transition))
;
```

```
;R transition-guard
(defrelation Transition-guard :domain Transition :range Guard)
(defrelation Guard-Transition :is (:inverse Transition-guard))
;
;R action-method
(defrelation Action-method :domain Action :range Method)
(defrelation Method-action :is (:inverse Action-method))
;
;R transition-callEvent
(defrelation Triggered-by :domain Transition :range CallEvent)
(defrelation Is-trigger-of :is (:inverse Triggered-by))
;
;R operation-callEvent
(defrelation Is-occurence-of :domain CallEvent :range Operation)
(defrelation Operation :is (:inverse Is-occurence-of))
;
;R operation-method
(defrelation Is-specification-of :domain Operation :range Method)
(defrelation Has-specification :is (:inverse Is-specification-of))
;
;R parameter-operation
(defrelation Has-parameter :domain Operation :range Parameter)
(defrelation Is-parameter-of :is (:inverse Has-parameter))
;
;R stateMachine-class
(defrelation Has-stateMachine :domain Class :range StateMachine)
(defrelation Is-stateMachine-of :is (:inverse Has-stateMachine))
;
;R class-createAction
(defrelation Created-by :domain Class :range CreateAction)
(defrelation Creates :is (:inverse Created-by))
;
;R class-parameter
(defrelation Parameter-class :domain Parameter :range Class)
(defrelation Class-parameter  :is (:inverse Parameter-class))
;
;R class-attribute
(defrelation Is-of-type :domain Attribute :range Class)
(defrelation Has-typed-feature :is (:inverse Is-of-type))
;
;R class-feature
(defrelation Has-feature :domain Class :range Feature)
(defrelation Is-owned-by  :is (:inverse Has-feature) :characteristics :single-valued)
;
;R class-generalization
(defrelation Is-generalization-of :domain Generalization :range Class)
(defrelation Is-child-of :is (:inverse Is-generalization-of))
;
;R class-generalization
(defrelation Is-specialization-of :domain Generalization :range Class)
(defrelation Is-parent-of :is (:inverse Is-specialization-of))
;
;R class-associationEnd
(defrelation Is-participant-of :domain Class :range AssociationEnd)
```

```
(defrelation Has-participant :is (:inverse Is-participant-of))
;
;R association-associationEnd
(defrelation Association-associationEnd :domain Association :range AssociationEnd)
(defrelation AssociationEnd-association :is (:inverse Association-associationEnd))
;
;R link-linkEnd
(defrelation Link-linkEnd :domain Link :range LinkEnd)
(defrelation LinkEnd-link :is (:inverse Link-linkEnd))
;
;R link-object
(defrelation Owned-by :domain Link :range Object)
(defrelation Owns-link :is (:inverse Owned-by))
;
;R link-stimulus
(defrelation Link-stimulus :domain Link :range Stimulus)
(defrelation Stimulus-link :is (:inverse Link-stimulus))
;
;R link-association
(defrelation Link-association :domain Link :range Association)
(defrelation Association-link :is (:inverse Link-association))
;
;R linkEnd-associationEnd
(defrelation LinkEnd-associationEnd :domain LinkEnd :range AssociationEnd)
(defrelation AssociationEnd-linkEnd :is (:inverse LinkEnd-associationEnd))
;
;R modelElement-model
(defrelation In-namespace :domain ModelElement :range Model :characteristics
 :single-valued)
(defrelation Owned-element :is (:inverse In-namespace))
;
;R compositeModel-versionedModel
(defrelation Contains-model :domain CompositeModel :range VersionedModel)
(defrelation Contained-by :is (:inverse Contains-model))
;
;R model-abstraction
(defrelation Supplied-by :domain Model :range Abstraction)
(defrelation Supplier :is (:inverse Supplied-by))
;
;R model-abstraction
(defrelation Client :domain Abstraction :range Model)
(defrelation Has-client :is (:inverse Client))
;
;R multiplicity-multiplicityRange
(defrelation Has-range :domain Multiplicity :range MultiplicityRange)
(defrelation Range-of :is (:inverse Has-range))
;
;R lower
(defrelation lower :is (:and Integer) :characteristics :single-valued)
;
;R upper
(defrelation upper :is (:and String) :characteristics :single-valued)
;
;R multiplicity-modelElement
```

```
(defrelation Has-multiplicity :domain ModelElement :range Multiplicity)
;
;the activee set
(defrelation activees :characteristics :single-valued)
;
;Concept MODELELEMENT
(defconcept ModelElement
 :roles (name the-prev-ver))
(tell (:about ModelElement (:at-most 1 In-namespace)))
;
;Concept MULTIPLICITYRANGE
(defconcept MultiplicityRange
 :is-primitive (:and
  (:at-most 1 Range-of)
  (:at-least 1 Range-of))
 :roles (lower upper))
;
;Concept MULTIPLICITY
(defconcept Multiplicity
 :is-primitive (:and
  (:at-least 1 Has-range)
  (:all Has-range MultiplicityRange)))
;
;Concept ACTION
(defconcept Action
 :is (:and ModelElement
  (:at-most 1 Entry-action-of)
  (:at-most 1 Exit-action-of)
  (:at-most 1 Do-activity-of)
  (:at-most 1 Action-transition)
  (:at-most 1 Action-of))
 :partitions (($Action$ (CallAction CreateAction DestroyAction ActionSequence
  UninterpretedAction)))
 :roles (isAsynchronous recurrence target))
;
;Concept CALLACTION
(defconcept CallAction
 :is (:and Action
  (:at-most 1 CallAction-operation)
  (:at-least 1 CallAction-operation))
 :in-partition $Action$
 :roles (isAsynchronous))
;
;Concept CREATEACTION
(defconcept CreateAction
 :is (:and Action
  (:at-most 1 Creates)
  (:at-least 1 Creates))
 :in-partition $Action$)
;
;Concept DESTROYACTION
(defconcept DestroyAction
 :is (:and Action)
 :in-partition $Action$)
```

```
;
;Concept ACTIONSEQUENCE
(defconcept ActionSequence
 :is (:and Action)
 :in-partition $Action$)
;
;(defconcept ActionSequence
; :is Action :in-partition $Action$)
;
;Concept UNINTERPRETEDACTION
(defconcept UninterpretedAction
 :is (:and Action)
 :in-partition $Action$
 :partitions (($UninterpretedAction$ (UpdateAction AccessAction))))
;
;Concept UPDATEACTION
(defconcept UpdateAction
 :is (:and UninterpretedAction
  (:at-most 1 Updates-attribute)
  (:at-least 1 Updates-attribute))
 :in-partition $UninterpretedAction$)
;
;Concept ACCESSACTION
(defconcept AccessAction
 :is (:and UninterpretedAction
  (:at-most 1 Accesses-attr)
  (:at-least 1 Accesses-attr))
 :in-partition $UninterpretedAction$)
;
;Concept OBJECT
(defconcept Object
 :is (:and ModelElement
  (:at-least 1 Instance-of-class)
  (:at-most 1 Instance-of-class))
 :roles (Instance-of-subclass-of operations-received))
;
;Concept STIMULUS
(defconcept Stimulus
 :is (:and ModelElement
  (:at-least 1 Sent-by)
  (:at-most 1 Sent-by)
  (:at-least 1 Received-by)
  (:at-most 1 Received-by)
  (:all Initiates Action)
  (:at-least 1 Initiates)
  (:at-most 1 Initiates)
  (:all Belongs-to InteractionInstanceSet)
  (:at-most 1 Stimulus-Activator)
  (:all Stimulus-link Link)
  (:at-most 1 Stimulus-link))
 :roles (activees added))
;
;Concept INTERACTIONINSTANCESET
(defconcept InteractionInstanceSet
```

```
 :is (:and ModelElement
  (:at-least 1 Consists-of)
  (:at-least 1 Exists-in-context)
  (:at-most 1 Exists-in-context)))
;
;Concept COLLABORATIONINSTANCESET
(defconcept CollaborationInstanceSet
 :is (:and ModelElement
  (:all Is-context-for InteractionInstanceSet)
  (:all Participating-instance Object)
  (:at-least 1 Participating-instance)))
;
;Concept CLASS
(defconcept class
 :is (:and ModelElement
  (:all Instantiated Object)
  (:all Created-by CreateAction)
  (:all Has-stateMachine StateMachine)
  (:all Is-child-of Generalization)
  (:all Is-parent-of Generalization)
  (:all Is-participant-of AssociationEnd))
 :roles (isAbstract)
)
;
;Concept ATTRIBUTELINK
(defconcept AttributeLink
 :is (:and ModelElement
  (:all Is-slot-of Object)
  (:at-most 1 Is-slot-of)
  (:at-least 1 Is-slot-of)
  (:all Is-updated-by UpdateAction)
  (:all Is-accessed-by AccessAction)
  (:all AttributeLink-attribute Attribute)
  (:at-most 1 AttributeLink-attribute)
  (:at-least 1 AttributeLink-attribute)))
;
;Concept STATEVERTEX
(defconcept StateVertex
 :is (:and ModelElement
  (:at-most 1 Is-subvertex-of)
  (:all Is-source-of Transition)
  (:all Is-target-of Transition))
 :roles (OK)
 :partitions $StateVertex$)
;
;Concept PSEUDOSTATE
(defconcept PseudoState
 :is (:and StateVertex)
 :in-partition $StateVertex$
 :roles (kind))
;
;Concept STATE
(defconcept State
 :is (:and StateVertex
```

```
  (:at-most 1 State-stateMachine)
  (:all Has-internalTransition Transition)
  (:all Has-entry-action Action)
  (:at-most 1 Has-entry-action)
  (:all Has-exit-action Action)
  (:at-most 1 Has-exit-action)
  (:all Has-do-activity Action)
  (:at-most 1 Has-do-activity))
 :in-partition $StateVertex$
 :partitions $State$)
;
;Concept FINALSTATE
(defconcept FinalState
 :is (:and State)
 :in-partition $State$)
;
;Concept SIMPLESTATE
(defconcept SimpleState
 :is (:and State)
 :in-partition $State$)
;
;Concept COMPOSITESTATE
(defconcept CompositeState
 :is (:and State
  (:all Is-container-of StateVertex))
 :in-partition $State$
 :roles (isConcurrent isRegion))
;
;Concept STATEMACHINE
(defconcept StateMachine
 :is (:and ModelElement
  (:at-most 1 Is-stateMachine-of)
  (:all StateMachine-state State)
  (:at-most 1 StateMachine-state)
  (:at-least 1 StateMachine-state)))
;
;Concept TRANSITION
(defconcept Transition
 :is (:and ModelElement
  (:exactly 1 Transition-stateMachine)
;  (:at-most 1 Transition-stateMachine)
  (:all Transition-guard Guard)
  (:at-most 1 Transition-guard)
  (:at-most 1 Is-outgoing-of)
  (:at-least 1 Is-outgoing-of)
  (:at-most 1 Is-incoming-of)
  (:at-least 1 Is-incoming-of)
  (:at-most 1 Is-internalTransition-of)
  (:all Triggered-by CallEvent)
  (:at-most 1 Triggered-by)
  (:all Transition-action Action)
  (:at-most 1 Transition-action))
 :roles (OK))
;
```

```
;Concept GUARD
(defconcept Guard
 :is (:and ModelElement
  (:at-most 1 Guard-transition)
  (:at-least 1 Guard-transition))
 :roles (expression))
;
;Concept CALLEVENT
(defconcept CallEvent
 :is (:and ModelElement
  (:at-most 1 Is-occurence-of)
  (:at-least 1 Is-occurence-of)))
;
;Concept FEATURE
(defconcept Feature
 :is (:and ModelElement
  (:at-most 1 Is-owned-by))
 :partitions $Feature$
 :roles (ownerScope visibility))
;
;Concept OPERATION
(defconcept Operation
 :is (:and Feature
  (:all Operation CallEvent)
  (:all Has-specification Method)
  (:all Has-parameter Parameter)
  (:all Operation-callAction CallAction))
 :in-partition $Feature$
 :roles (concurrency isAbstract isLeaf isRoot isQuery))
;
;Concept METHOD
(defconcept Method
 :is (:and Feature
  (:at-most 1 Is-specification-of)
  (:at-least 1 Is-specification-of))
 :in-partition $Feature$
 :roles (body isQuery)) ;last 1 from behavioral feature
;
;Concept ATTRIBUTE
(defconcept Attribute
 :is (:and Feature
  (:all Is-of-type Class)
  (:at-most 1 Is-of-type)
  (:at-least 1 Is-of-type)
  (:exactly 1 Has-multiplicity))
 :in-partition $Feature$
 :roles (initialValue changeability ordering targetScope))
 ;
;Concept PARAMETER
(defconcept Parameter
 :is (:and ModelElement
  (:at-most 1 Is-parameter-of)
  (:all Parameter-class Class)
  (:at-most 1 Parameter-class)
```

```
  (:at-least 1 Parameter-class))
 :roles (defaultValue kind))
;
;Concept GENERALIZATION
(defconcept Generalization
 :is (:and ModelElement
  (:at-most 1 Is-generalization-of)
  (:at-least 1 Is-generalization-of)
  (:at-most 1 Is-specialization-of)
  (:at-least 1 Is-specialization-of))
 :roles (discriminator))
;
;Concept ASSOCIATIONEND
(defconcept AssociationEnd
 :is (:and ModelElement
  (:at-most 1 Has-participant)
  (:at-least 1 Has-participant)
  (:at-most 1 AssociationEnd-association)
  (:at-least 1 AssociationEnd-association)
  (:exactly 1 Has-multiplicity)
  (:all AssociationEnd-linkEnd LinkEnd))
 :roles (aggregation ordering isNavigable visibility))
;
;Concept ASSOCIATION
(defconcept Association
 :is (:and ModelElement
  (:all Association-associationEnd AssociationEnd)
  (:at-least 2 Association-associationEnd)
  (:all Association-link Link)))
;
;Concept ABSTRACTION
(defconcept Abstraction
 :is (:and ModelElement
  (:all Supplier Model)
  (:at-least 1 Supplier)
  (:all Client Model)
  (:at-least 1 Client))
 :roles (mapping))
;
;Concept LINK
(defconcept Link
 :is (:and ModelElement
  (:all Link-linkEnd LinkEnd)
  (:at-least 2 Link-linkEnd)
  (:all Owned-by Object)
  (:at-most 1 Owned-by)
  (:at-most 1 Link-association)
  (:at-least 1 Link-association)))
;
;Concept LINKEND
(defconcept LinkEnd
 :is (:and ModelElement
  (:at-most 1 LinkEnd-link)
  (:at-least 1 LinkEnd-link)
```

```
  (:at-most 1 LinkEnd-associationEnd)
  (:at-least 1 LinkEnd-associationEnd)))
;
;Concept MODEL
(defconcept Model
 :is (:and ModelElement)
 :partitions $Model$)
;
;Concept ARGUMENT
(defconcept Argument
 :is-primitive (:and
  (:all Argument-of Action)
  (:at-most 1 Argument-of))
 :roles (value))
;
;Concept VERSIONEDMODEL
(defconcept VersionedModel
 :is (:and Model
  (:at-most 1 Contained-by))
 :roles (version)
 :in-partition $Model$
 :partitions $VersionedModel$)
;
;Concept COMPOSITEMODEL
(defconcept CompositeModel
 :is (:and VersionedModel
  (:all Contains-model VersionedModel))
 :in-partition $VersionedModel$)
;
;Concept PRIMITIVEMODEL
(defconcept PrimitiveModel
 :is (:and VersionedModel)
 :in-partition $VersionedModel$
 :partitions $PrimitiveModel$)
;
;Concept CLASSMODEL
(defconcept ClassModel
 :is (:and PrimitiveModel)
 :in-partition $PrimitiveModel$)
;
;Concept SEQUENCEMODEL
(defconcept SequenceModel
 :is (:and PrimitiveModel)
 :in-partition $PrimitiveModel$)
;
;Concept STATEMODEL
(defconcept StateModel
 :is (:and PrimitiveModel)
 :in-partition $PrimitiveModel$)
;
;Concept TRACE
(defconcept Trace
 :is (:and Abstraction)
 :partitions $Trace$)
```

```
;
;Concept HORIZONTALTRACE
(defconcept HorizontalTrace
 :is (:and Trace)
 :in-partition $Trace$)
;
;Concept EVOLUTIONTRACE
(defconcept EvolutionTrace
 :is (:and Trace)
 :in-partition $Trace$)
```

## A.2 User-defined Models

### A.2.1 Class Diagram Example

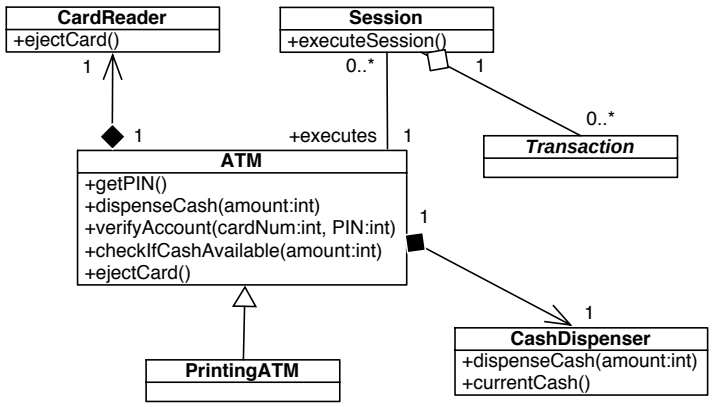


Figure A.1: Class diagram example for UML to Loom translation

- Instances of the *MultiplicityRange* metaclass

```
(tellm (:about mr1-1.0 (lower 1) (upper 1)))
(tellm (:about mr2-1.0 (lower 1) (upper 1)))
(tellm (:about mr3-1.0 (lower 1) (upper 1)))
(tellm (:about mr4-1.0 (lower 0) (upper "*")))
(tellm (:about mr5-1.0 (lower 1) (upper 1)))
(tellm (:about mr6-1.0 (lower 0) (upper "*")))
(tellm (:about mr7-1.0 (lower 1) (upper 1)))
(tellm (:about mr8-1.0 (lower 1) (upper 1)))
```

- Instances of the *Multiplicity* metaclass

```
(tellm (:about m1-1.0 (Has-range mr1-1.0)))
(tellm (:about m2-1.0 (Has-range mr2-1.0)))
(tellm (:about m3-1.0 (Has-range mr3-1.0)))
(tellm (:about m4-1.0 (Has-range mr4-1.0)))
(tellm (:about m5-1.0 (Has-range mr5-1.0)))
(tellm (:about m6-1.0 (Has-range mr6-1.0)))
```

```
(tellm (:about m7-1.0 (Has-range mr7-1.0)))
(tellm (:about m8-1.0 (Has-range mr8-1.0)))
```

- Instance of the *ClassModel* metaclass

```
(tellm (:about Class-Diagram-1.0 (version 1.0) (name Class-Diagram)))
```

- Instances of the *Class* metaclass

```
(tellm (:about ATM-1.0
 (name ATM)
 (isAbstract false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about PrintingATM-1.0
 (name PrintingATM)
 (isAbstract false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about CardReader-1.0
 (name CardReader)
 (isAbstract false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about CashDispenser-1.0
 (name CashDispenser)
 (isAbstract false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about Session-1.0
 (name Session)
 (isAbstract false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about Transaction-1.0
 (name Transaction)
 (isAbstract true)
 (In-namespace Class-Diagram-1.0)))
```

- Instances of the *Parameter* metaclass

```
(tellm (:about amount-dispenseCash-ATM-1.0
 (name amount)
 (Is-parameter-of dispenseCash-ATM-1.0)
 (Parameter-class int)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about cardNum-verifyAccount-ATM-1.0
 (name cardNum)
 (Is-parameter-of verifyAccount-ATM-1.0)
 (Parameter-class int)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about PIN-verifyAccount-ATM-1.0
 (name PIN)
 (Is-parameter-of verifyAccount-ATM-1.0)
 (Parameter-class int)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about amount-checkIfCashAvailable-ATM-1.0
```

```
 (name amount)
 (Is-parameter-of checkIfCashAvailable-ATM-1.0)
 (Parameter-class int)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about amount-dispenseCash-CashDispenser-1.0
 (name amount)
 (Is-parameter-of dispenseCash-CashDispenser-1.0)
 (Parameter-class int)
 (In-namespace Class-Diagram-1.0)))
```

- Instances of the *Operation* metaclass

```
(tellm (:about getPIN-ATM-1.0
 (name getPIN)
 (Is-owned-by ATM-1.0)
 (IsAbstract false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about dispenseCash-ATM-1.0
 (name dispenseCash)
 (Is-owned-by ATM-1.0)
 (IsAbstract false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about verifyAmount-ATM-1.0
 (name verifyAmount)
 (Is-owned-by ATM-1.0)
 (IsAbstract false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about checkIfCashAvailable-ATM-1.0
 (name checkIfCashAvailable)
 (Is-owned-by ATM-1.0)
 (IsAbstract false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about ejectCard-ATM-1.0
 (name ejectCard)
 (Is-owned-by ATM-1.0)
 (IsAbstract false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about ejectCard-CardReader-1.0
 (name ejectCard)
 (Is-owned-by CardReader-1.0)
 (IsAbstract false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about dispenseCash-CashDispenser-1.0
 (name dispenseCash)
 (Is-owned-by CashDispenser-1.0)
 (IsAbstract false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about currentCash-CashDispenser-1.0
 (name currentCash)
 (Is-owned-by CashDispenser-1.0)
```

```
 (IsAbstract false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about executeSession-Session-1.0
 (name executeSession)
 (Is-owned-by Session-1.0)
 (IsAbstract false)
 (In-namespace Class-Diagram-1.0)))
```

- Instance of the *Generalization* metaclass

```
(create 'ATM-PrintingATM-1.0 'Generalization)
(tellm (:about ATM-PrintingATM-1.0
 (name ATM-PrintingATM)
 (Is-generalization-of PrintingATM-1.0)
 (Is-specialization-of ATM-1.0)
 (In-namespace Class-Diagram-1.0)))
```

- Instances of the *AssociationEnd* metaclass

```
(tellm (:about ATM-end1-1.0
 (name ATM-end1)
 (Has-multiplicity m1-1.0)
 (Has-participant ATM-1.0)
 (aggregation composite)
 (isNavigable false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about ATM-end2-1.0
 (name executes)
 (Has-multiplicity m3-1.0)
 (Has-participant ATM-1.0)
 (aggregation none)
 (isNavigable true)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about ATM-end3-1.0
 (name ATM-end3)
 (Has-multiplicity m7-1.0)
 (Has-participant ATM-1.0)
 (aggregation composite)
 (isNavigable false)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about Session-end1-1.0
 (name Session-end1)
 (Has-multiplicity m4-1.0)
 (Has-participant Session-1.0)
 (aggregation none)
 (isNavigable true)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about Session-end2-1.0
 (name Session-end2)
 (Has-multiplicity m5-1.0)
 (Has-participant Session-1.0)
```

```
 (aggregation aggregate)
 (isNavigable true)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about CashDispenser-end1-1.0
 (name CashDispenser-end1)
 (Has-multiplicity m2-1.0)
 (Has-participant CashDispenser-1.0)
 (aggregation none)
 (isNavigable true)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about CardReader-end1-1.0
 (name CardReader-end1)
 (Has-multiplicity m8-1.0)
 (Has-participant CardReader-1.0)
 (aggregation none)
 (isNavigable true)
 (In-namespace Class-Diagram-1.0)))
(tellm (:about Transaction-end1-1.0
 (name Transaction-end1)
 (Has-multiplicity m6-1.0)
 (Has-participant Transaction-1.0)
 (aggregation none)
 (isNavigable true)
 (In-namespace Class-Diagram-1.0)))
```

- Instances of the *Association* metaclass

```
(tellm (:about Session-ATM-1.0
 (name Session-ATM)
 (In-namespace Class-Diagram-1.0)
 (Association-associationEnd Session-end1-1.0)
 (Association-associationEnd ATM-end2-1.0)))
(tellm (:about CashDispenser-ATM-1.0
 (name CashDispenser-ATM)
 (In-namespace Class-Diagram-1.0)
 (Association-associationEnd CashDispenser-end1-1.0)
 (Association-associationEnd ATM-end1-1.0)))
(tellm (:about CardReader-ATM-1.0
 (name CardReader-ATM)
 (In-namespace Class-Diagram-1.0)
 (Association-associationEnd CardReader-end1-1.0)
 (Association-associationEnd ATM-end3-1.0)))
(tellm (:about Session-Transaction-1.0
 (name Session-Transaction)
 (In-namespace Class-Diagram-1.0)
 (Association-associationEnd Session-end2-1.0)
 (Association-associationEnd Transaction-end1-1.0)))
```

### A.2.2 Sequence Diagram Example

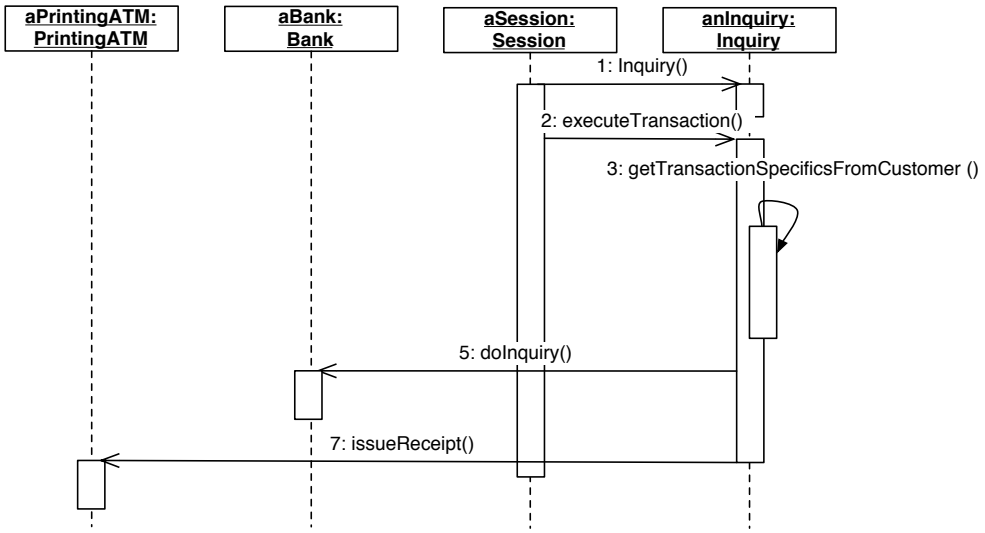


Figure A.2: Sequence diagram example for UML to Loom translation

- Instance of the *SequenceModel* metaclass

```
(tellm (:about Sequence-Diagram-1.0 (version 1.0) (name Sequence-Diagram)))
```

- Instances of the *Object* metaclass

```
(tellm (:about aPrintingATM-1.0
 (name aPrintingATM)
 (In-namespace Sequence-Diagram-1.0)
 (Instance-of-class PrintingATM-1.0)))
(tellm (:about aBank-1.0
 (name aBank)
 (In-namespace Sequence-Diagram-1.0)
 (Instance-of-class Bank-1.0)))
(tellm (:about aSession-1.0
 (name aSession)
 (In-namespace Sequence-Diagram-1.0)
 (Instance-of-class Session-1.0)))
(tellm (:about anInquiry-1.0
 (name anInquiry)
 (In-namespace Sequence-Diagram-1.0)
 (Instance-of-class Inquiry-1.0)))
```

- Instances of the *Link* metaclass

```
(tellm (:about link1-1.0
 (name link1)
 (In-namespace Sequence-Diagram-1.0)
 (Link-stimulus stim1-1.0)))
(tellm (:about link2-1.0
```

```
 (name link2)
 (In-namespace Sequence-Diagram-1.0)
 (Link-stimulus stim2-1.0)))
(tellm (:about link3-1.0
 (name link3)
 (In-namespace Sequence-Diagram-1.0)
 (Link-stimulus stim3-1.0)))
(tellm (:about link4-1.0
 (name link4)
 (In-namespace Sequence-Diagram-1.0)
 (Link-stimulus stim4-1.0)))
(tellm (:about link5-1.0
 (name link5)
 (In-namespace Sequence-Diagram-1.0)
 (Link-stimulus stim5-1.0)))
```

- Instances of the *Stimulus* metaclass

```
(tellm (:about stim1-1.0
 (name 1)
 (In-namespace Sequence-Diagram-1.0)
 (Sent-by aSession-1.0)
 (Received-by anInquiry-1.0)
 (Initiates create1-1.0)))
(tellm (:about stim2-1.0
 (name 2)
 (In-namespace Sequence-Diagram-1.0)
 (Sent-by aSession-1.0)
 (Received-by anInquiry-1.0)
 (Initiates call1-1.0)
 (Stimulus-Activee stim3-1.0)
 (Stimulus-Activee stim4-1.0)
 (Stimulus-Activee stim5-1.0)
 (Stimulus-Predecessor stim3-1.0)))
(tellm (:about stim3-1.0
 (name 3)
 (In-namespace Sequence-Diagram-1.0)
 (Sent-by anInquiry-1.0)
 (Received-by anInquiry-1.0)
 (Initiates call2-1.0)
 (Stimulus-Predecessor stim4-1.0)))
(tellm (:about stim4-1.0
 (name 4)
 (In-namespace Sequence-Diagram-1.0)
 (Sent-by anInquiry-1.0)
 (Received-by aBank-1.0)
 (Initiates call3-1.0)
 (Stimulus-Predecessor stim5-1.0)))
(tellm (:about stim5-1.0
 (name 5)
```

```
(In-namespace Sequence-Diagram-1.0)
(Sent-by anInquiry-1.0)
(Received-by aPrintingATM-1.0)
(Initiates call4-1.0)))
```

- Instances of the *CallAction* metaclass

```
(tellm (:about call1-1.0
 (name executeTransaction)
 (In-namespace Sequence-Diagram-1.0)
 (CallAction-operation executeTransaction-Inquiry-1.0)))
(tellm (:about call2-1.0
 (name getTransactionSpecificsFromCustomer)
 (In-namespace Sequence-Diagram-1.0)
 (CallAction-operation getTransactionSpecificsFromCustomer-Inquiry-1.0)))
(tellm (:about call3-1.0
 (name doInquiry)
 (In-namespace Sequence-Diagram-1.0)
 (CallAction-operation doInquiry-Bank-1.0)))
(tellm (:about call4-1.0
 (name issueReceipt)
 (In-namespace Sequence-Diagram-1.0)
 (CallAction-operation issueReceipt-ATM-1.0)))
```

- Instance of the *CreateAction* metaclass

```
(tellm (:about create1-1.0
 (name Inquiry)
 (In-namespace Sequence-Diagram-1.0)
 (Creates Inquiry-1.0)))
```

### A.2.3 State Diagram Example

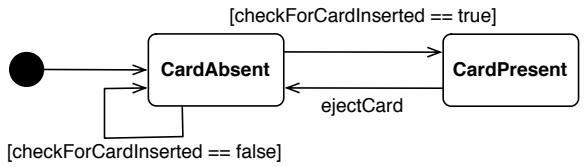


Figure A.3: State diagram example for UML to Loom translation

- Instance of the *StateModel* metaclass

```
(tellm (:about CR-State-Diagram-1.1 (version 1.0) (name CR-State-Diagram)))
```

- Instance of the *StateMachine* metaclass

```
(tellm (:about CR-StateMachine-1.0
 (name CR-StateMachine)
 (In-namespace CR-State-Diagram-1.0)
```

```
(Is-stateMachine-of CR-1.0)
(StateMachine-state top-CRSM-1.0)))
```

- Instance of the *CompositeState* metaclass

```
(tellm (:about top-CRSM-1.0
 (name top)
 (In-namespace CR-State-Diagram-1.0)
 (Is-container-of initial-CRSM-1.0)
 (Is-container-of CardAbsent-CRSM-1.0)
 (Is-container-of CardPresent-CRSM-1.0)))
```

- Instance of the *PseudoState* metaclass

```
(tellm (:about initial-CRSM-1.0
(name initial)
(In-namespace CR-State-Diagram-1.0)
(kind initial)))
```

- Instances of the *SimpleState* metaclass

```
(tellm (:about CardAbsent-CRSM-1.0
(In-namespace CR-State-Diagram-1.0)
(name CardAbsent)))
(tellm (:about CardPresent-CRSM-1.0
(In-namespace CR-State-Diagram-1.0)
(name CardPresent)))
```

- Instances of the *Transition* metaclass

```
(tellm (about trans1-1.0
 (name trans1)
 (In-namespace CR-State-Diagram-1.0)
 (Transition-stateMachine CR-StateMachine-1.0)
 (Is-outgoing-of initial-CRSM-1.0)
 (Is-incoming-of CardAbsent-CRSM-1.0)))
(tellm (about trans2-1.0
 (name trans2)
 (In-namespace CR-State-Diagram-1.0)
 (Transition-stateMachine CR-StateMachine-1.0)
 (Is-outgoing-of CardAbsent-CRSM-1.0)
 (Is-incoming-of CardPresent-CRSM-1.0)))
(tellm (about trans3-1.0
 (name trans3)
 (In-namespace CR-State-Diagram-1.0)
 (Transition-stateMachine CR-StateMachine-1.0)
 (Is-outgoing-of CardPresent-CRSM-1.0)
 (Is-incoming-of CardAbsent-CRSM-1.0)))
(tellm (about trans4-1.0
 (name trans4)
 (In-namespace CR-State-Diagram-1.0)
 (Transition-stateMachine CR-StateMachine-1.0)
 (Is-outgoing-of CardAbsent-CRSM-1.0)
 (Is-incoming-of CardAbsent-CRSM-1.0)))
```

- Instances of the *CallEvent* metaclass

```
(tellm (:about callevent1-1.0
 (name ejectCard)
 (Is-trigger-of trans3-1.0)
 (Is-occurence-of ejectCard-CardReader-1.0)
 (In-namespace CR-State-Diagram-1.0)))
(tellm (:about callevent2-1.0
 (name none)
 (Is-trigger-of trans1-1.0)
 (Is-trigger-of trans2-1.0)
 (Is-trigger-of trans4-1.0)
 (In-namespace CR-State-Diagram-1.0)))
```

- Instances of the *Guard* metaclass

```
(tellm (:about guard1-1.0
 (name guard1)
 (expression "checkForCardInserted==true")
 (Guard-transition tran2-1.0)
 (In-namespace CR-State-Diagram-1.0)))
(tellm (:about guard2-1.0
 (name guard2)
 (expression "checkForCardInserted==false")
 (Guard-transition tran4-1.0)
 (In-namespace CR-State-Diagram-1.0)))
```

# Appendix B

# Case Study - Static Structure

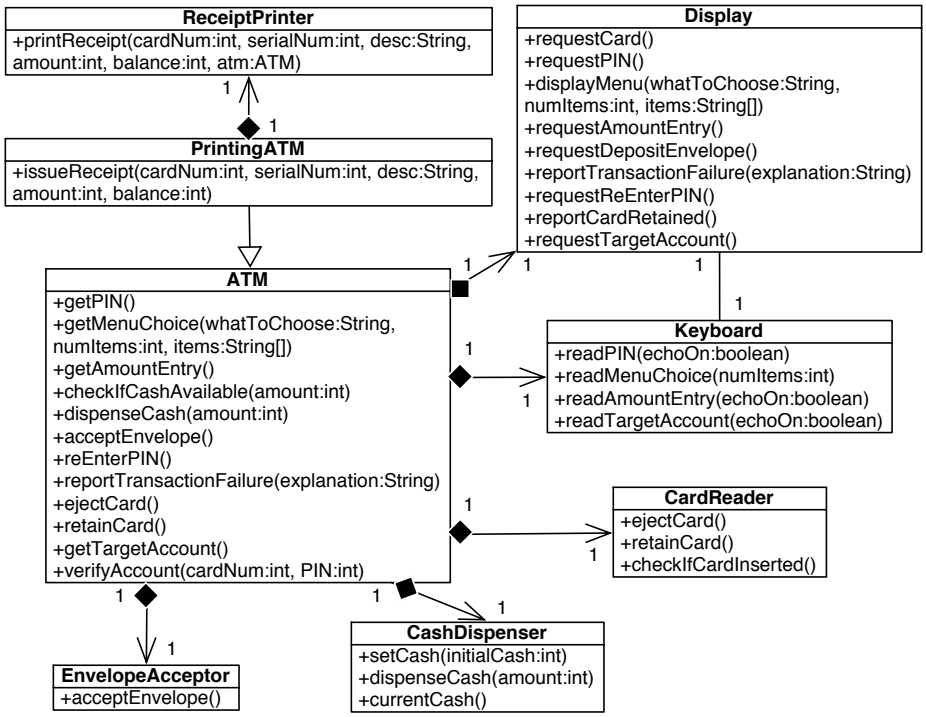


Figure B.1: Class Diagram - Part I, version 1

The class diagrams shown in figures B.1, B.2 and B.3, combined, give the overall structure of the Banking Simulation case study. In this appendix, the class interfaces have been detailed to the level of what each operation present in the class diagrams does. This system is in no way complete, as various constructors, accessors and mutators have been omitted from this simulation, so as not to sacrifice clarity.

- *ATM* class: manages the ATM and its component parts. Handles withdraw, deposit, inquiry and transfer transactions. It cannot issue receipts.
  - *getPIN()*: gets the PIN number from the user.

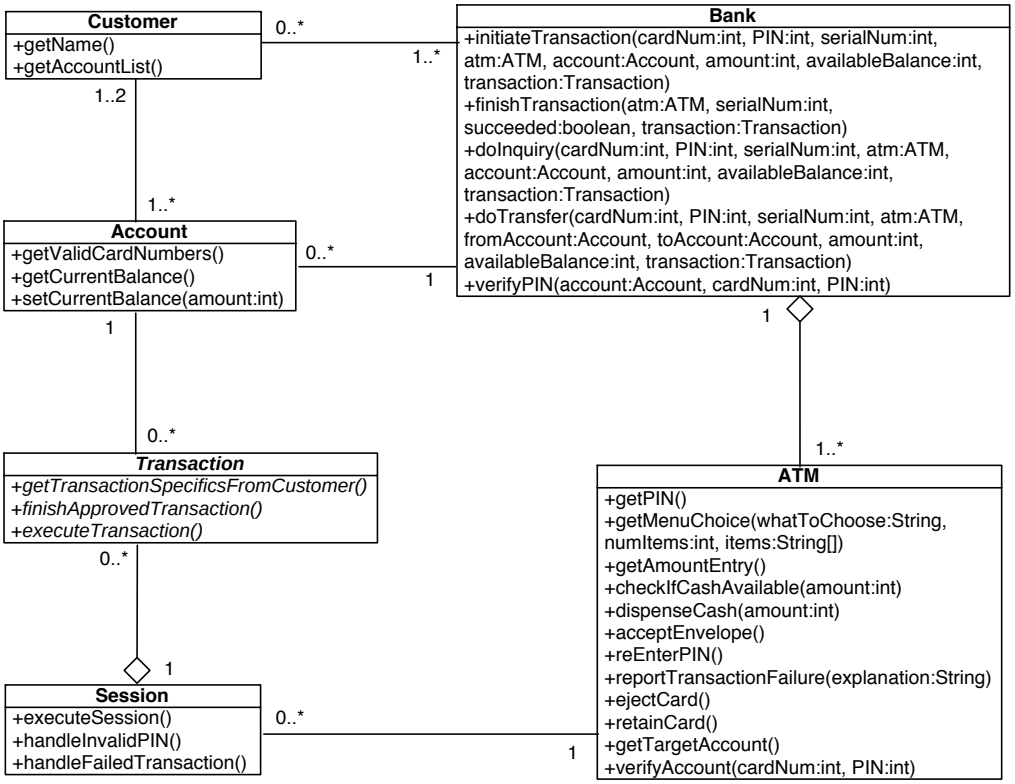


Figure B.2: Class Diagram - Part II, version 1

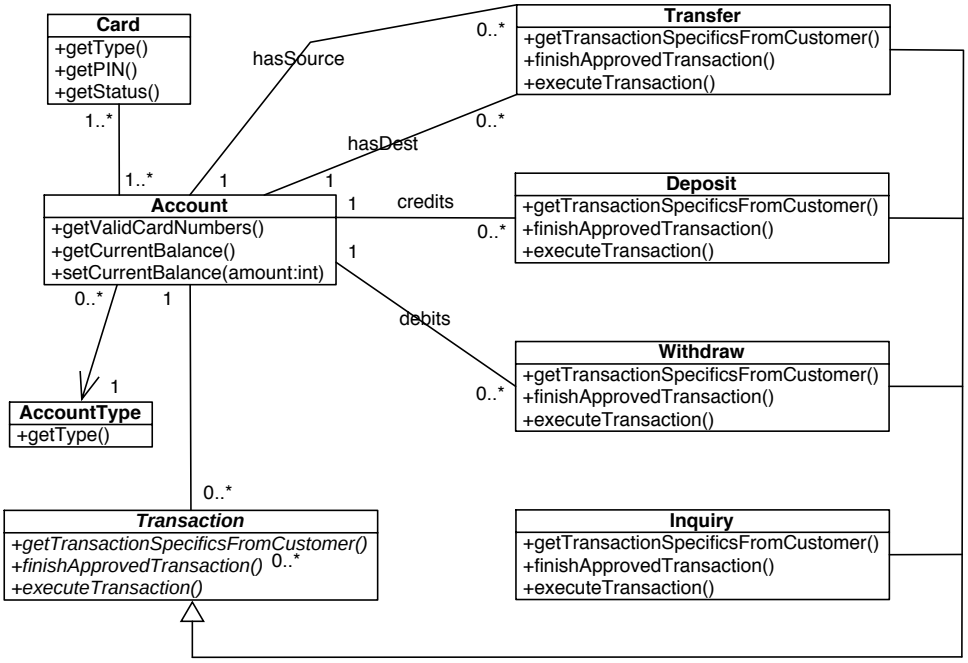


Figure B.3: Class Diagram - Part III, version 1

  - *getMenuChoice(whatToChoose:String, numItems:int, items:String[])*: gets choice from a menu of options from the user.
  - *getAmountEntry()*: asks user to enter an amount of money.
  - *checkIfCashAvailable(amount:int)*: checks if the cash dispenser has a certain amount of cash.
  - *dispenseCash(amount:int)*: dispenses cash to the user.
  - *acceptEnvelope()*: accept a deposit envelope from the user.
  - *reEnterPIN()*: asks the user to re-enter the PIN.
  - *reportTransactionFailure(explanation:String)*: informs the user that a transaction failed, and why.
  - *ejectCard()*: returns the card to the user at the end of a session.
  - *retainCard()*: retains the card because the user did not enter a valid PIN.
  - *getTargetAccount()*: asks the user to enter an account identification, when making transfers.
  - *verifyAccount(cardNum:int, PIN:int)*: verifies the validity of the card - PIN number pair.

- *PrintingATM* class: inherits from the *ATM* class, adding printing capabilities to the functionalilties offered by its parent.
  - *issueReceipt(cardNum:int, serialNum:int, desc:String, amount:int, balance:int)*: print a receipt for a transaction.

- *CardReader* class: manages the magnetic strip card reader.
  - *ejectCard()*: physically ejects the user's card.
  - *retainCard()*: physically retains the user's card.
  - *checkForCardInserted()*: reports if a readable card has been inserted in the card reader.

- *Display* class: manages the video display unit.
  - *requestCard()*: asks the user to insert a valid card in the card reader.
  - *requestPIN()*: asks the user to enter his PIN through the keyboard.
  - *displayMenu(whatToChoose:String, numItems:int, items:String[])*: displays a menu of options for the user to chose from.
  - *requestAmountEntry()*: asks the user to enter an amount of money, through the keyboard.
  - *requestDepositEnvelope()*: asks the user insert a deposit envelope in the envelope acceptor.
  - *reportTransactionFailure(explanation:String)*: informs the user that a transaction failed, and why.
  - *requestReEnterPIN()*: asks the user to re-enter his PIN, as the previously entered one was invalid.
  - *reportCardRetained()*: informs the user that his card has been retained, because of too many incorrect PIN entries.

  - *requestTargetAccount()*: asks the user to enter the identification of the account to which he wants to make a transfer to, through the keyboard.

- *Keyboard* class: manages the keyboard used for customer input.
  - *readPIN(echoOn:boolean)*: reads the PIN typed by the user.
  - *readMenuChoice(numItems:int)*: reads the menu item chosen by the user.
  - *readAmountEntry(echoOn:boolean)*: reads the amount entered by the user.
  - *readTargetAccount(echoOn:boolean)*: reads the account identification entered by the user.

- *CashDispenser* class: manages the cash dispenser.
  - *setCash(initialCash)*: sets the initial amount of money available in the ATM's cash dispenser.
  - *dispenseCash(amount:int)*: physically dispenses money to the user.
  - *currentCash()*: reports the amount of cash currently available in the cash dispenser.

- *EnvelopeAcceptor* class: manages the slot that accepts a deposit envelope from the customer.
  - *acceptEnvelope()*: physically accepts a deposit envelope from the user.

- *ReceiptPrinter* class: manages the printer that produces customer receipts.
  - *printReceipt(cardNum:int, serialNum:int, desc:String, amount:int, balance:int, atm:ATM)*: physically prints a receipt for a transaction.

- *Session* class: logs and controls a user session, which starts when the user inserts a card, and ends when this card is either ejected or retained. Controls the amount of invalid PIN numbers entered by the user.
  - *executeSession()*: starts the log and control of a user session.
  - *handleInvalidPIN()*: performs the actions needed to control the entry of an invalid PIN.
  - *handleFailedTransaction()*: performs the actions needed to control the failure of a transaction.

- *Bank* class: manages communications with a bank.
  - *initiateTransaction(cardNum:int, PIN:int, serialNum:int, atm:ATM, account:Account, amount:int, availableBalance:int, transaction:Transaction)*: initiates a *Withdraw* or *Deposit* transaction.
  - *finishTransaction(atm:ATM, serialNum:int, succeeded:boolean, transaction:Transaction)*: informs the bank whether or not a *Withdraw* or *Deposit* transaction was physically completed.
  - *doInquiry(cardNum:int, PIN:int, serialNum:int, atm:ATM, account:Account, amount:int, availableBalance:int, transaction:Transaction)*: performs an inquiry transaction.
  - *doTransfer(cardNum:int, PIN:int, serialNum:int, atm:ATM, fromAccount:Account, toAccount:Account, amount:int)*: performs a transfer transaction.

  - *verifyPIN(account:Account, cardNum:int, PIN:int)*: verifies the validity of the account - card number - PIN number trio.

- *Customer* class: manages bank customers.
  - *getName()*: gets the customer's name.
  - *getAccountList()*: gets the list of accounts associated to the customer.

- *Account* class: manages customer accounts.
  - *getValidCardNumbers()*: gets the list of valid card numbers associated to the account.
  - *getCurrentBalance()*: gets the current balance of the account.
  - *setCurrentBalance(amount:int)*: sets the current balance of the account to a certain amount.

- *Transaction* class: abstract class, encompasses the common features of the various types of transaction available in this simulation.
  - *(abstract) getTransactionSpecificsFromCustomer()*: get specific information relevant to the transaction, from the user.
  - *(abstract) finishApprovedTransaction()*: finishes a transaction once it has been approved by the bank.
  - *(abstract) executeTransaction()*: executes the steps necessary to carry out a transaction.

- *Withdraw* class: inherits from the *Transaction* class, implements the withdrawal-specific functionalities.
  - *getTransactionSpecificsFromCustomer()*: asks the user the amount of cash he wishes to withdraw, etc.
  - *finishApprovedTransaction()*: finishes a transaction once it has been approved by the bank.
  - *executeTransaction()*: executes the steps necessary to carry out a transaction.

- *Deposit* class: inherits from the *Transaction* class, implements the deposit-specific functionalities.
  - *getTransactionSpecificsFromCustomer()*: asks the user to insert the deposit envelope, etc.
  - *finishApprovedTransaction()*: finishes a transaction once it has been approved by the bank.
  - *executeTransaction()*: executes the steps necessary to carry out a transaction.

- *Transfer* class: inherits from the *Transaction* class, implements the transfer-specific functionalities.
  - *getTransactionSpecificsFromCustomer()*: asks the user to enter the account identification to which he wants to transfer money, etc.
  - *finishApprovedTransaction()*: finishes a transaction once it has been approved by the bank.

  - *executeTransaction()*: executes the steps necessary to carry out a transaction.

- *Inquiry* class: inherits from the *Transaction* class, implements the inquiry-specific functionalities.
  - *getTransactionSpecificsFromCustomer()*: asks what type of information should appear on the receipt, etc.
  - *finishApprovedTransaction()*: finishes a transaction once it has been approved by the bank.
  - *executeTransaction()*: executes the steps necessary to carry out a transaction.

- *AccountType* class: manages the different types of account, like debit and savings accounts.
  - *getType()*: gets the account type.

- *Card* class: manages the customer magnetic cards.
  - *getType()*: gets the card type.
  - *getPIN()*: gets the PIN number associated to the card.
  - *getStatus()*: gets the status of the card (active, stolen, etc.).