Software and Systems Modeling manuscript No.
(will be inserted by the editor)

A Formal Approach to Model Refactoring and Model Refinement

Ragnhild Van Der Straeten[1], **Viviane Jonckers**[1], **Tom Mens**[2]

[1] System and Software Engineering Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
e-mail: rvdstrae@vub.ac.be|viviane@info.vub.ac.be

[2] Service de Génie Logiciel
Université de Mons-Hainaut
Av. du Champs de Mars 6, 7000 Mons, Belgium
e-mail: tom.mens@umh.ac.be

Received: date / Revised version: date

Abstract Model-driven engineering is an emerging software engineering approach that relies on model transformation. Typical kinds of model transformations are *model refinement* and *model refactoring*. Whenever such a transformation is applied to a consistent model, we would like to know whether the consistency is preserved by the transformation. Therefore, in this article, we formally define and explore the relation between behaviour inheritance consistency of a refined model with respect to the original model, and behaviour preservation of a refactored model with respect to the original model. As it turns out, there is a strong similarity between these notions of behaviour consistency and behaviour preservation. To illustrate this claim, we formalised the behaviour specified by UML 2.0 sequence and protocol state machine diagrams. We show how the reasoning capabilities of *description logics*, a decidable fragment of first-order logic, can be used in a natural way to detect behaviour inconsistencies. These reasoning capabilities can be used in exactly the same way to detect behaviour preservation violations during model refactoring. A prototype plug-in in a UML CASE tool has been developed to validate our claims.

Key words model-driven engineering, UML 2.0, description logics, model refinement, model refactoring, behaviour preservation.

1 Introduction

Model-driven engineering is an approach to software development where the primary focus is on models, as opposed to source code. Models are built representing different views on a software system. Models continually evolve into new versions, and can be used to generate executable code. The ultimate goal is to raise the level of abstraction, and to develop and evolve complex software systems by manipulating models only. The manipulation of models is achieved by means of *model transformation*, which is considered to be the heart and soul of model-driven engineering [33].

Because model-driven engineering is still in its infancy, there is a need for sophisticated formalisms, techniques and associated tools supporting model transformation. In this article, we focus on the activities of model refinement, model refactoring and model inconsistency management in particular. Model refactoring is a transformation used to improve the structure of a model while preserving its behaviour. Model refinement is a transformation that adds more detail to an existing model. Both kinds of model transformation activities are crucial during model evolution, but need to be complemented by the activity of inconsistency management, to deal with possible inconsistencies that may arise in a model after its transformation.

In this article, we introduce a formalism that relates all of the above activities. More specifically, we formally explore the relation between behaviour consistencies of model refinements and behaviour preservation of model refactorings. In addition, we show the practical use of this formalism through a plug-in we developed for a commercial UML CASE tool. This tool allows us to detect in an automatic way whether model refinements and model refactorings are behaviourally consistent.

As a motivating example of what can be achieved, consider the situation depicted in Figure 1. A class `ATM` (version 1.0) is *refined* into a subclass `CardChargingATM` (version 1.1) in a behaviourally consistent way. This means that the behaviour of the `CardChargingATM` class (expressed by means of a state machine or sequence diagram, for example) should specialise the behaviour of the `ATM` class in the way formally defined in [10].

Now suppose that the `CardChargingATM` class *evolves* into a new version 1.2, as illustrated in Figure 1. Then we would like to know whether or not the evolved behaviour of

Send offprint requests to:

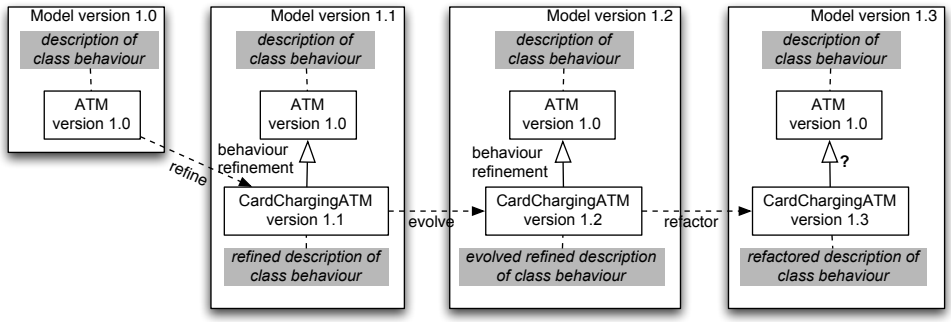

Fig. 1 Scenario of evolution of our motivating example.

the `CardChargingATM` class is still behaviourally consistent with the class `ATM` of which it is a refinement.

Similarly, suppose that the behaviour of the `CardChargingATM` class is *refactored* into a new version 1.3. Although the refactoring modifies the structure of the state machine of `CardChargingATM`, we would like to guarantee that it does not affect the existing behaviour (by definition of refactoring). In other words, we need to check that the refactored version of `CardChargingATM` is still behaviourally consistent with the original `ATM` class. Again, we can formally express this notion of behaviour preservation, and verify it in an automated way. A worked out example of non-trivial model refactorings at the level of state machine diagrams will be given in the next section. These model refactorings are similar to those that can be found in the research literature [40,3].

The remainder of the paper will be structured as follows. Section 2 presents the motivating example in detail. Section 3 formalises behaviour as defined in UML 2.0 protocol state machines and sequence diagrams. Section 4 defines behaviour inheritance consistencies in the context of state machines and sequence diagrams. Section 5 formalises behaviour preservation in the context of model refactoring and shows that the notions of consistency and preservation are closely related. Section 6 proves some additional properties about the behaviour of classes that already obey certain consistency and preservation properties. Section 7 introduces the formalism of Description Logics (DLs) [2] and shows how the different notions of consistency and preservation can be expressed as a set of logic facts in this formalism. Section 8 briefly discusses an implementation of our inconsistency detection approach in a description logics engine, and its integration in a commercial UML CASE tool using its built-in plug-in mechanism. Related work is discussed in Section 9. Section 10 concludes this paper.

2 Motivation

2.1 Motivating Example

The motivating example used throughout this paper, is based on the design of an automatic teller machine (ATM), originally developed by Prof. Russell Bjork for a computer science course at Gordon University[1]. The class diagrams of this design are presented in Figures 2 and 3.

The dynamic behaviour of the `ATM` class is partially represented by the protocol state machine in Figure 4. Other aspects of the dynamic behaviour are represented by usage scenarios of instances of `ATM`, such as the one shown in the sequence diagram in Figure 5. This sequence diagram shows part of an interaction between instances of the classes `ATM`, `CustomerConsole`, `CashDispenser`, `Session`, `Withdrawal` and `Message`, when a user decides to make a withdrawal. The messages sent in the diagram first retrieve the account number and the amount to withdraw. Next, messages are sent to verify if there is enough cash in the ATM and if the transaction is allowed by the bank. In this scenario, it is assumed that both conditions are fulfilled. Finally, the transaction is completed by dispensing the amount of cash to the user and printing a receipt.

2.2 Behaviour inconsistencies

Following the spirit of our motivating example, in this paper we assume that ***the behaviour of a class*** *is defined as a combination of its protocol state machine and all sequence diagrams in which instances of the considered class are involved*. This will enable us to formally define the behaviour of a class, and allow us to detect *behaviour inconsistencies*.

As an example, one of the possible behaviour inconsistencies that can arise is called *behaviour incompatibility*. This would happen if we can find a sequence in one of the sequence diagrams that is not contained in the set of call sequences of the protocol state machine of the class. For a more detailed classification and description of all possible inconsistencies, we refer to [43].

Coming back to the behaviour of the `ATM` class, as described by the protocol state machine of Figure 4 and the sequence diagram of Figure 5, we can verify that it does not have any *behaviour incompatibility*. Indeed, each sequence of the `ATM` sequence diagram of Figure 5 is contained in the set of call sequences of the `ATM` state machine diagram of Figure 4.

[1] `http://www.cs.gordon.edu/courses/cs211`

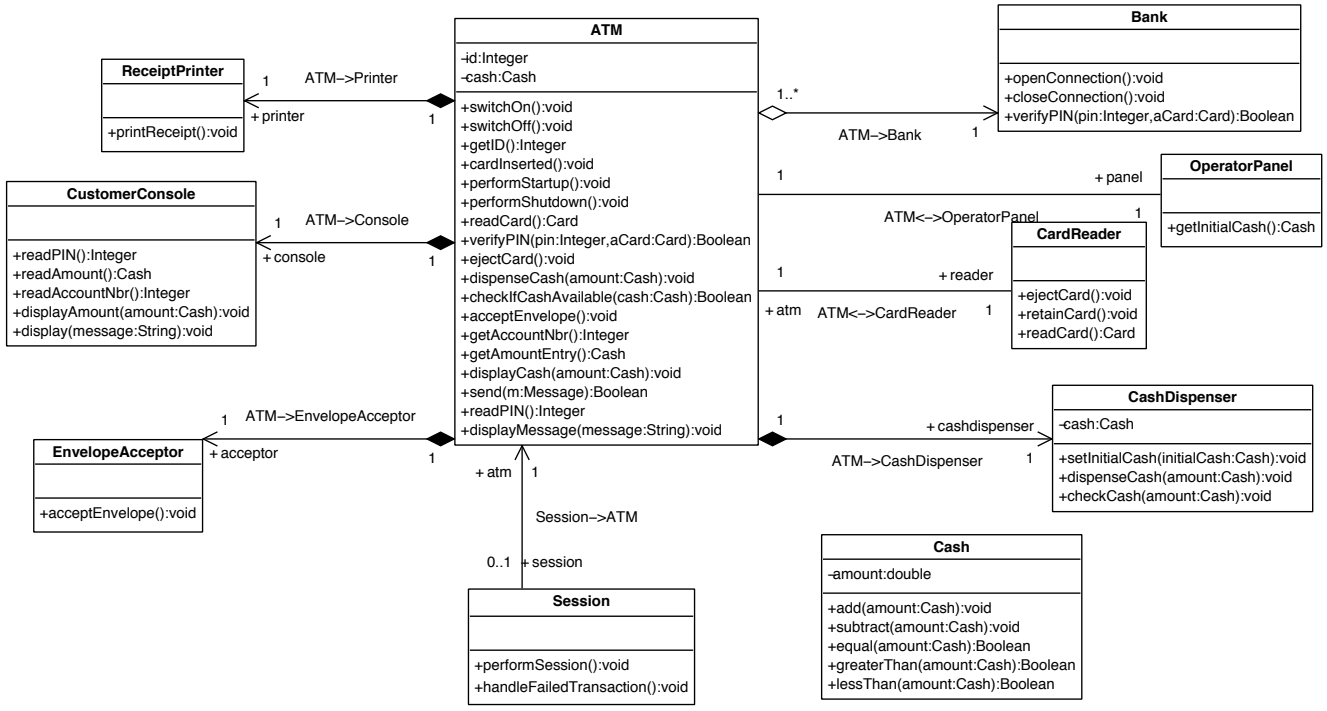

Fig. 2 Class diagram for the `ATM` design.

Analysing and detecting behaviour inconsistencies becomes essential when we start making changes to our design model, either by modifying the class diagram, sequence diagrams or protocol state machines, or by adding new diagrams or removing existing ones. In all of these cases, it is important to be able to determine whether the changes made give rise to new behaviour inconsistencies.

2.3 Model refinement

As a first concrete example of what can happen during model evolution, we will consider a refinement of a class. More specifically, we will add a subclass of an existing class with a "refined" behaviour w.r.t. the original class. This notion of behaviour refinement can be formalised by means of a specific notion of consistency, namely *behaviour inheritance consistency*. In this section we will only present the intuition behind it. For a formal definition, we refer to Section 4.

In version 1.1 of the design model of our running example, we add a new subclass `CardChargingATM` that is a refined version of class `ATM` that not only allows the customer to withdraw cash money, but also to charge "virtual" money to his bank card. This refined behaviour of `CardChargingATM` is represented by the protocol state machine in Figure 6. An orthogonal composite state `VerifyingTransaction` is added to the existing composite state `GettingCustomerSpecifics` (see Figure 4). Its substate `VerifyATMBalance` is moved one level deeper into the new composite state `VerifyingTransaction`. The customer still has to specify the account number and the amount of cash for withdrawal. The states `AccountEntry` and `AmountEntry` are still part of the `GettingCustomerSpecifics` state and not of the orthogonal `VerifyingTransaction` state. As a result, the same account number and amount will be used to charge the customer's bank card and to withdraw money. Verifying if the customer's account has sufficient funds and if the transactions are allowed by the bank is now done in parallel. Once these checks have been passed, the ATM dispenses the money and at the same time, the `CardChargingATM` class, unlike its parent, the `ATM` class, charges the card.

To be able to determine whether `CardChargingATM` is a formal refinement of `ATM`, we need to formalise the consistency relationship between both classes. Given that both classes are related via inheritance, we can base ourselves on the substitutability principle [20], and require that an instance of the subclass `CardChargingATM` must be usable in each situation where an instance of the superclass `ATM` is expected. We can state this more precisely in terms of sequence diagrams and protocol state machines as follows: *each sequence of the `ATM` sequence diagram of Figure 5 should be contained in the set of sequences of the `CardChargingATM` state machine diagram of Figure 6.*

In our case, `ATM` and `CardChargingATM` do not obey this consistency rule, because an instance of `CardChargingATM` will withdraw money and it will *always* charge a card. It is not possible to skip the charging of the card and immediately choose a new transaction, which is the original behaviour of the `ATM` class. Hence, according to this view, `CardChargingATM` cannot be regarded as a refinement of `ATM`.

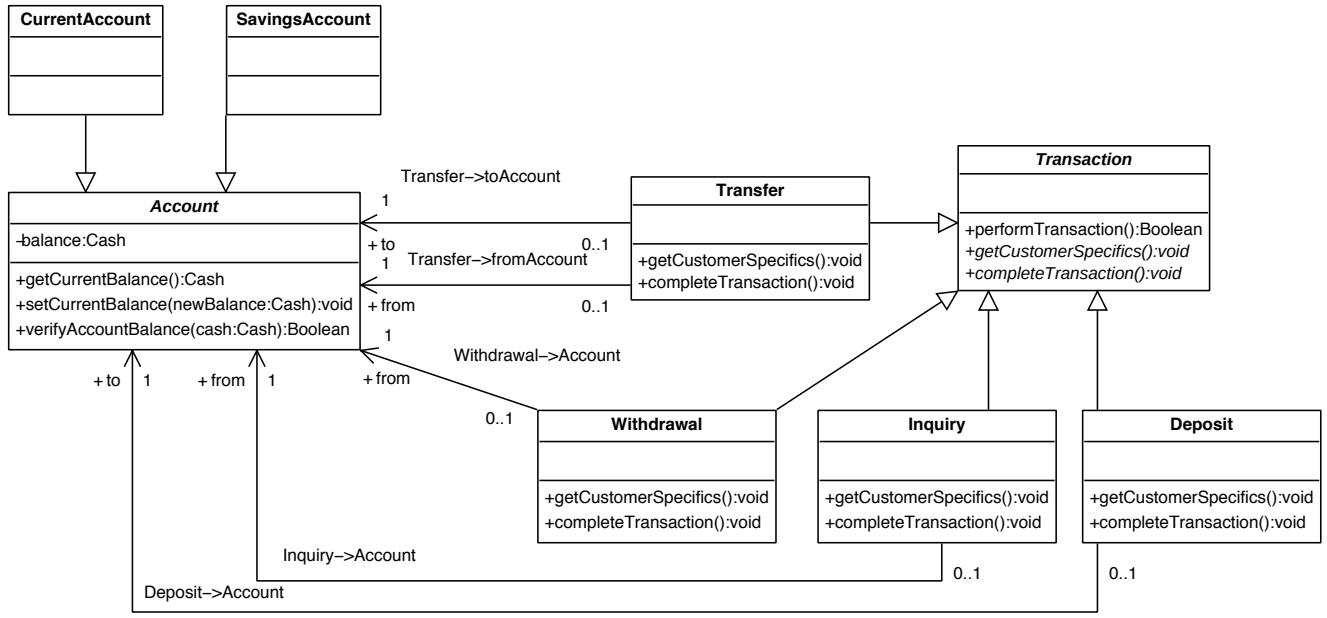

Fig. 3 Class diagram for the `Account` and `Transaction` class hierarchies.

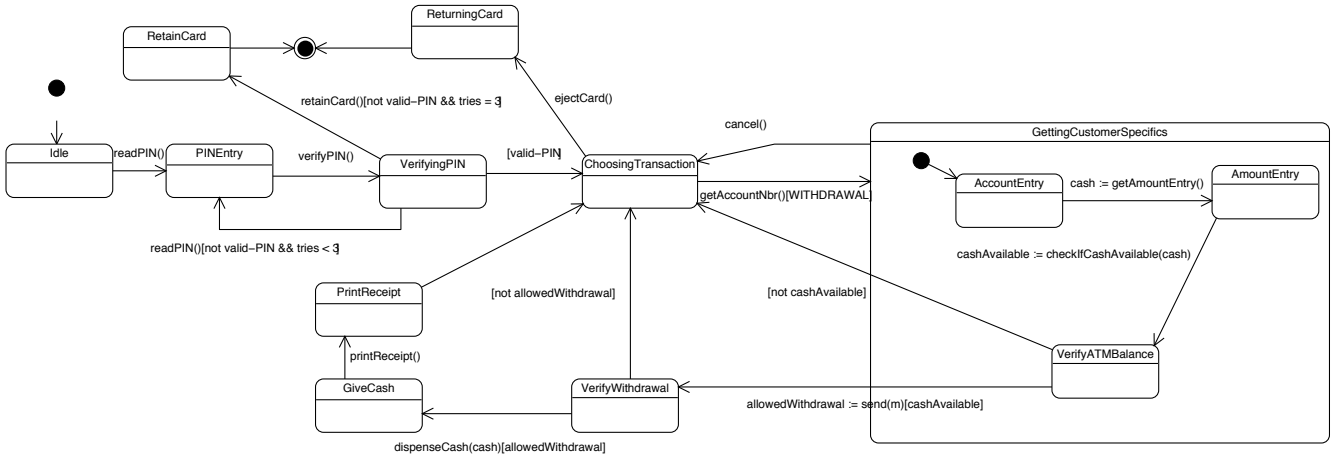

Fig. 4 UML protocol state machine diagram for `ATM` class.

2.4 Model evolution

Starting from version 1.1, one can make further changes to the design model. For example, we could continue to evolve the behaviour of `CardChargingATM` by introducing a new state (`AmountEntryCard`) and transitions in the composite state `GettingCustomersSpecifics` as shown in the state machine diagram in Figure 7 (representing part of version 1.2 of the design model). This is an example of an evolution step where we have added new functionality to the `CardChargingATM` class. The amount to be withdrawn and to be charged on the card can now be different. This was not the case for the previous version of the `CardChargingATM` (in version 1.1 of the design model).

2.5 Model refactoring

We can also consider more restrictive model evolutions, that do not add new functionality or remove existing functionality, but have the purpose of simplifying the design model without changing its behaviour. Such model evolutions are called *model refactorings*.

For model refactoring, the idea of behaviour consistency is very important. If we know that a given design model is behaviour consistent, and we perform a model refactoring, then we expect the evolved design model to be behaviour consistent too. In other words, the refactoring is assumed to preserve certain behaviour consistency properties. Being able to verify or guarantee preservation of these properties becomes crucial in this situation.

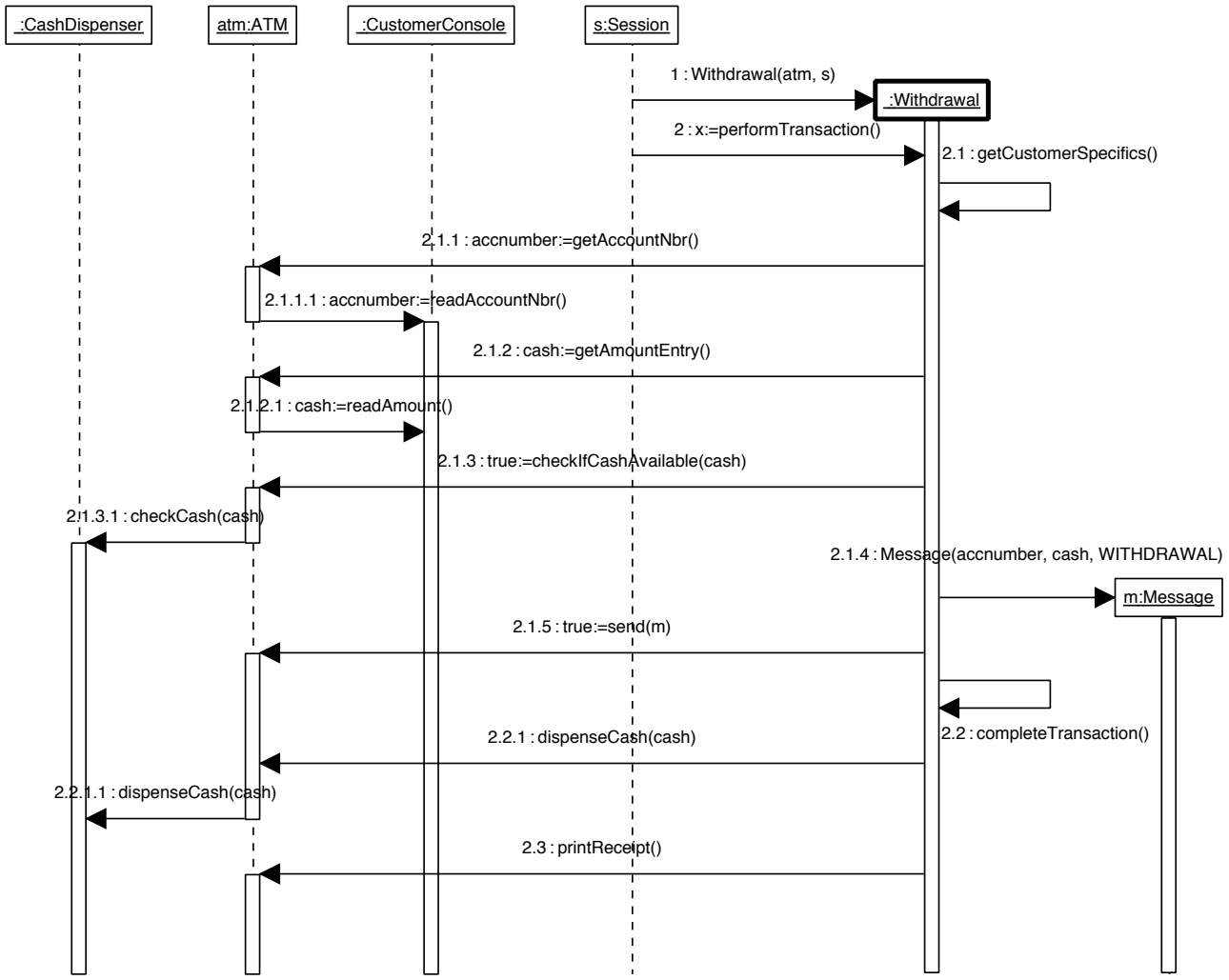

Fig. 5 Sequence diagram for withdrawal scenario on an `ATM`.

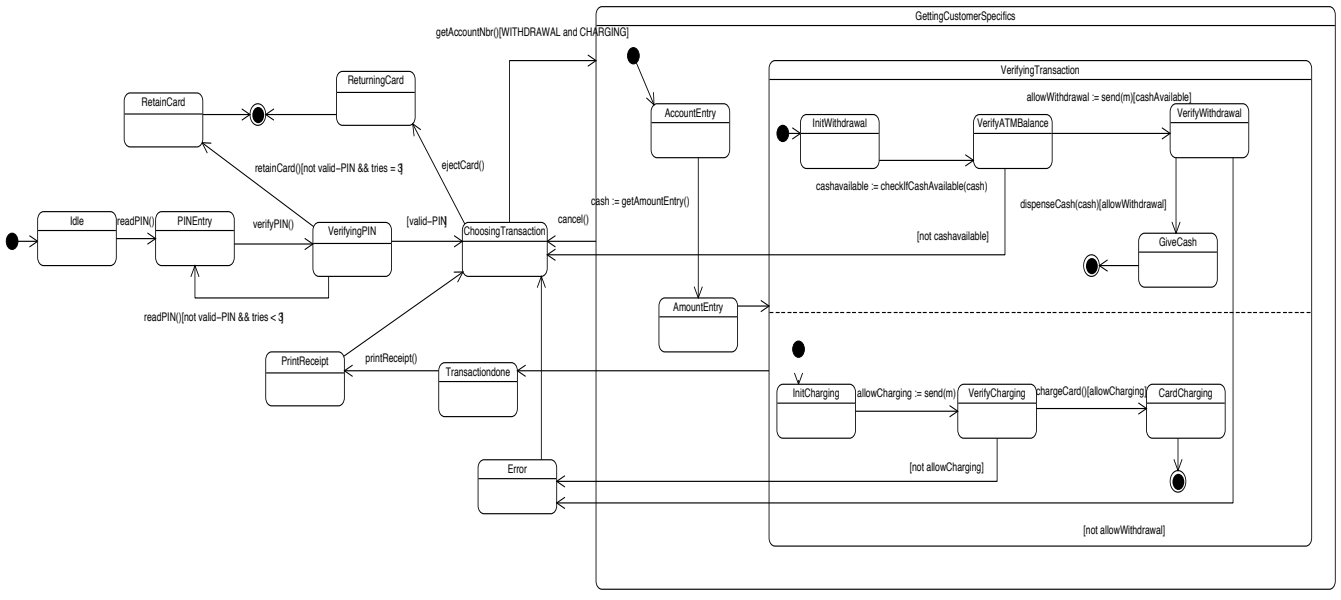

Fig. 6 UML protocol state machine for `CardChargingATM` class (version 1.1).

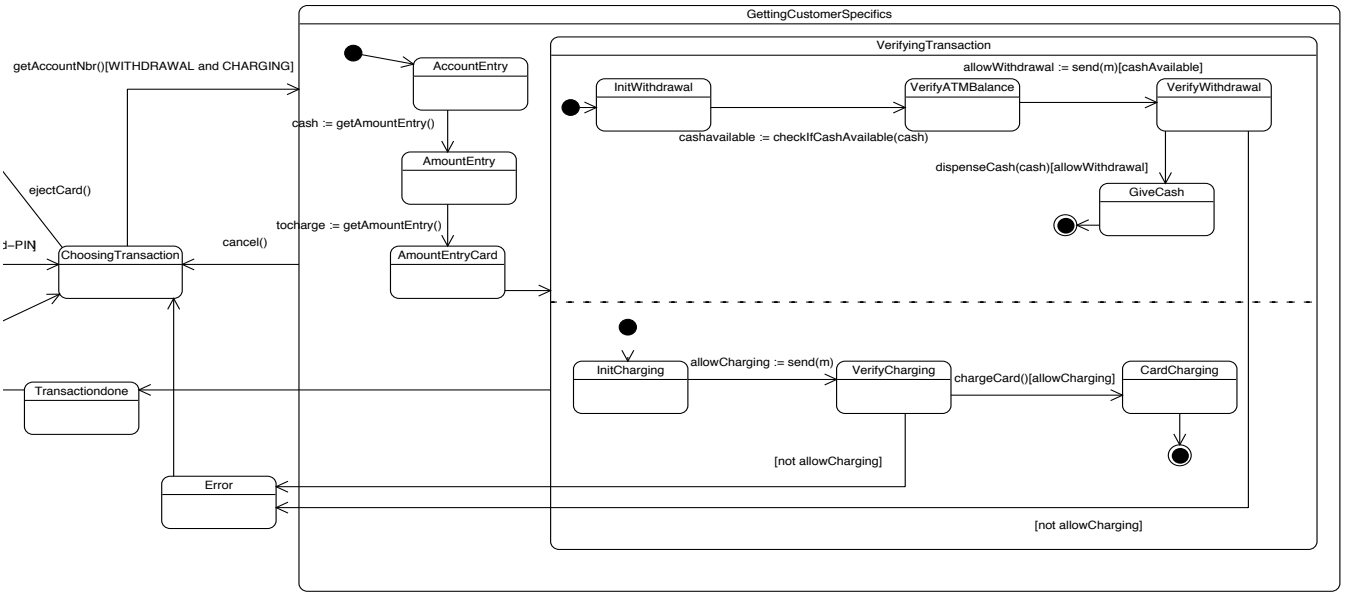

Fig. 7 Evolved part of the state machine for `CardChargingATM` class (version 1.2). Only the composite state `GettingCustomerSpecifics` and the states directly linked to it are shown here, since the other states and transitions have not been modified.

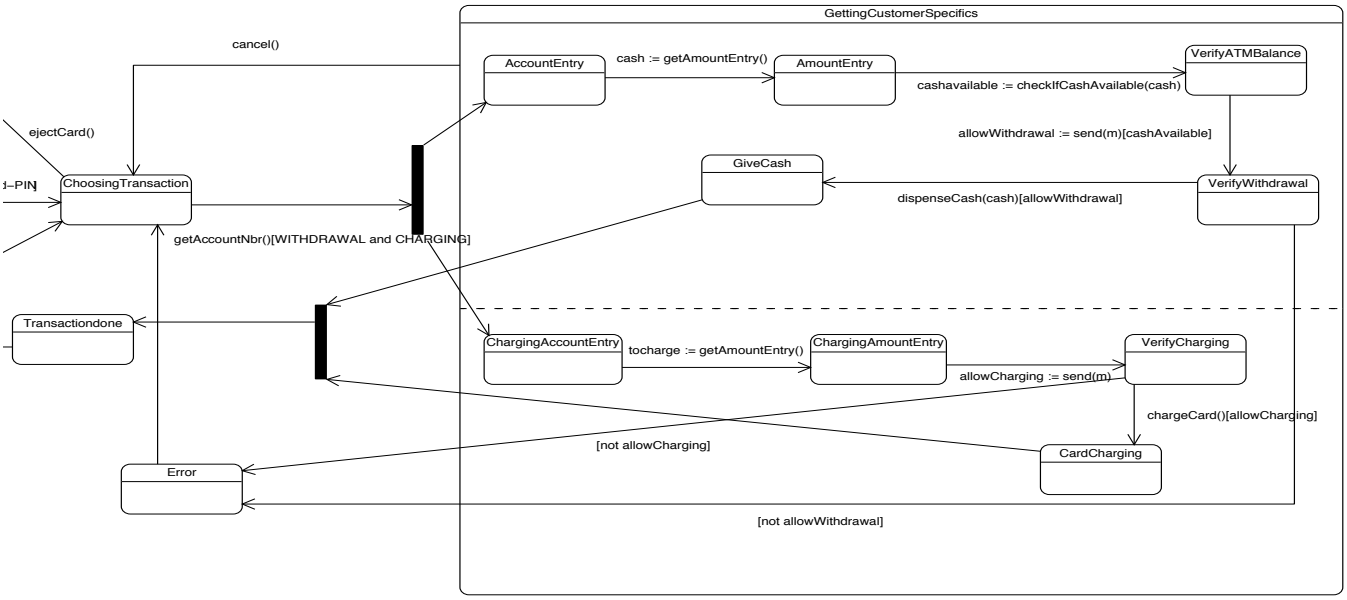

Fig. 8 Refactored part of the state machine for `CardChargingATM` class (version 1.3), after having performed a sequence of two refactorings *Move states into orthogonal composite state* and *Flatten states*.

An example of the result of a complex model refactoring is shown in Figure 8 (version 1.3). This protocol state machine represents a refactored version of the one shown in Figure 7 (version 1.2). To obtain the new state machine from the original one, a sequence of two refactorings has been applied, *Move states into orthogonal composite state* and *Flatten states*.

The first refactoring, *Move states into orthogonal composite state*, can be seen as the inverse of the *Sequentialize concurrent composite state* refactoring defined in [3]. States are moved into different regions of an orthogonal composite state. In our example the simple state `AmountEntry` is moved into one region of the `VerifyingTransaction` state and the state `AmountEntryCard` is moved into the other region and renamed into `ChargingAmountEntry`. The original state `AccountEntry` is split into two states (`AccountEntry` and `ChargingAccountEntry`). These states are the "initial states" of the two different regions. As

a result the previous initial states, `InitWithdrawal` and `InitCharging` become superfluous and are deleted. Moving a state into a certain region of an orthogonal composite state has some consequences. High-level transitions (i.e., transitions originating from a composite state) originating on the orthogonal composite state, are inherited by the moved states. By moving a state into one region of the orthogonal composite state, if the moved state becomes active, other states will be active too, one in each remaining region. As a result, if the `AmountEntry` state is active, the `Charging-AmountEntry` state can be active too. This situation is not possible in our example version 1.2.

The second refactoring, *Flatten States*, flattens the states `GettingCustomerSpecifics` and `VerifyingTransaction` into a new state also named `GettingCustomerSpecifics` that is an orthogonal composite state.

After applying both refactorings, it is important to know that the original behaviour has not been modified. For example, in this case, we can formally prove (or check with a tool) that the sequences of operations that can be invoked on the original class `CardChargingATM` (version 1.2) can also be invoked on the refactored class `CardChargingATM` (version 1.3).

3 Preliminary definitions

In this section, we introduce some preliminary definitions that are needed to define the dynamic behaviour of a class and to characterise precisely behaviour inheritance consistency and behaviour preservation in Sections 4 and 5.

Notation 1 *The* **set of all preconditions of an operation** *op, i.e., the set of conditions that must be true when the operation op is invoked, is denoted by* Pre_{op}.
The **set of all postconditions of an operation** *op, i.e., the set of conditions that must be true when the operation op is completed, is denoted by* $Post_{op}$.

We do not use a specific constraint language for the pre- and postconditions of operations or any other kinds of constraints that will be needed further in this paper. Instead, we assume that these constraints (such as Pre_{op} and $Post_{op}$) are sets of predicates, i.e., Boolean expressions.

3.1 Sequence Diagram

Depending on its purpose, an interaction between objects (i.e., instances of classes) can be modeled using different types of UML diagrams. For the sake of simplicity, we only consider sequence diagrams here. We formally define a *SD (sequence diagram) trace* as:

Definition 1 *A* ***SD trace*** ν_o *of an instance* o *of a class* c *is a sequence of event occurrences denoted* $< e_1, \ldots, e_n >$ *occurring on the lifeline of the instance* o*. An* ***event occurrence*** e *is defined as a couple* $(m, cons)$ *where* m *denotes the message that is associated to this event occurrence and* $cons$ *represents the constraints valid on the lifeline of the instance* o *before the execution of the event occurrence.*

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

Definition 2 *A* **sequence diagram** Δ *is a set of SD traces. This set typically contains SD traces for instances of different classes.*

Behaviour consistencies impose restrictions on the traces defined and more specifically on the order of invocations of the involved object's operations (see Section 4). As a consequence, we are only interested in the order of invocations of an object's operations. As such, the traces of event occurrences representing the *receipt* of a message are important and considered in the definitions of behaviour consistencies. Therefore, we define a *receiving SD trace* as follows:

Definition 3 *A* **receiving SD trace** $\nu_o/^{rec}$ *of an instance* o *of a class* c *is an SD trace* ν_o *for the instance* o *with only event occurrences representing the receipt of messages, which represents the invocation of operations.*

Example 1 A receiving SD trace of the instance `atm` of class `ATM` in the sequence diagram Δ of Figure 5 is $< e_1, e_2, e_3, e_4, e_5, e_6 >$, where e_1 represents the receipt (by `atm`) of the message `getAccountNbr`, e_2 represents the receipt of the message `getAmountEntry`, e_3 represents the receipt of the message `checkIfCashAvailable`, e_4 represents the receipt of the message `send` and e_5 represents the receipt of the message `dispenseCash`. Finally, e_6 represents the receipt of the message `printReceipt`.

Notation 2 *The* **set of all event occurrences denoting the receipt of a message** *for each instance* o *of a class* c *appearing in the sequence diagram* Δ *is denoted by* $\mathbf{E_{\Delta,c}}$.

Example 2 Let Δ and e_i be defined as in *Example 1*.
Then $E_{\Delta,ATM} = \{e_1, e_2, e_3, e_4, e_5, e_6\}$. If the sequence diagram Δ would have contained more than one instance of class ATM, then $E_{\Delta,ATM}$ would have contained all event occurrences corresponding to receipts of messages for each instance of ATM.

3.2 Protocol State Machine

UML 2.0 differentiates between two kinds of state machines, *behavioural state machines* and *protocol state machines*. *Behavioural state machines* are used to specify the behaviour of various model elements. *Protocol state machines* are used to express usage protocols and are always defined in the context of a classifier, which can have several protocol state machines. These state machines express the legal transitions that a classifier can trigger. As such they are a convenient way to define a *lifecycle of an object* or an *order of the invocation*

of its operations. Because in the context of behaviour consistencies, the order of invocation of operations is the most important, only protocol state machines are considered here.

A protocol transition specifies a legal transition for an operation. Transitions of protocol state machines have next to their trigger, which is an operation invocation, a pre- and a postcondition. We make some simplifying assumptions in this paper. UML provides special kinds of states and transitions, such as junction and choice states and submachine states, entry and exit points on transitions. These concepts are not considered in this paper.

A protocol state machine (PSM) can be defined as follows (based on the definition in [38] and in [39]) [2]:

Definition 4 *A* **protocol state machine** $\Pi_c = (S_c, T_c, L_c, \rho_c, \Lambda_c)$ *for a class c, consists of a set of states* S_c *and a labelled transition set* $T_c \subseteq \mathcal{P}(S_c) \times L_c \times \mathcal{P}(S_c)$ *containing labelled relations* (S_1, l, S_2) *such that* $l \in L_c$*, where* L_c *is a set of labels.*
A label l *is defined as a triple* (op, g, h) *where op is operation that is defined in c or in one of the ancestors of c,* $g \subseteq Pre_{op}$ *specifies the precondition of the transition (which is evaluated as part of the precondition of the operation op), and* $h \subseteq Post_{op}$ *specifies the postcondition of the transition (which is part of the postcondition of the operation op), or, as a triple* $l = (\epsilon, g, \{\})$*, where* ϵ *corresponds to a dummy operation and g specifies the guard.*

ρ_c denotes the top-most initial state, for which $\nexists\, S_1, S_2 \subseteq S_c : \rho_c \in S_2, (S_1, l, S_2) \in T_c$ *where* $l \in L_c$ *and transitions outgoing* ρ_c *only have labels of the form* $l = (\epsilon, g, \{\})$.

Λ_c denotes the set of final states of the state machine, for which $\nexists\, S_1 \subseteq \Lambda_c$ *and* $S_2 \subseteq S_c : (S_1, l, S_2) \in T_c$ *for any* $l \in L_c$.

Note that, in UML PSMs there can be transitions without operation calls and without any guard, as well as transitions that only have a guard specified but no operation call. Both kinds of transitions are supported by labels of the form $l = (\epsilon, g, \{\})$.

Due to the fact that a transition is specified as a relation between *sets of states*, simple, composite and orthogonal composite states are also supported (see Section 15.3.11 in [27]). Our definition also supports high-level, compound and completion protocol transitions (see Section 15.3.14 in [27]). The basic idea is that all these notions of states and protocol transitions can be transformed into a canonical form containing transitions between sets of states (which can be singletons).

We now define the notion of *(active) life cycle state configuration*.

Definition 5 *A* **(life cycle) state configuration** Σ_o *of an instance o of a class c in a PSM* Π_c *is a tree of states belonging to* S_c.

[2] Note that $\mathcal{P}(S)$ denotes the powerset of S.

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

Example 3 Consider as an example, the state machine shown in Figure 7. A possible life cycle state configuration is the tree (we represent a tree by nested sets) {`Getting-CustomerSpecifics`, {`AmountEntry`}}. Not only the simple state `AmountEntry` is considered as part of the life cycle state configuration but also all the directly or transitively composite states to which the simple state belongs. Another example of a life cycle state configuration is the tree {`GettingCustomerSpecifics`, {`Verifying-Transaction`, {`VerifyWithdrawal`, `CardCharging`}}}.

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

Definition 6 *An* **active (life cycle) state configuration** σ_o *of an instance o of a class c in a PSM* Π_c *is a subset of* S_c *and corresponds to the set of leafs of a life cycle state configuration.*

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

Definition 7 *A* **PSM trace** γ_o *of an instance o of a class c in a PSM* Π_c *is a sequence of active life cycle state configurations* $< \sigma_{o,1}, \ldots, \sigma_{o,n} >$ *such that* $\sigma_{o,1} = \{\rho_c\}$ *and, for* $i \in \{1 \ldots n-1\}$, $\sigma_{o,i+1} = \sigma_{o,i}$ *or* $\exists(\sigma_{o,i}, \tau_i, \sigma_{o,i+1}) \in T_c$.

Definition 8 *A* **call sequence** μ_o *of instance o of class c in a PSM* Π_c *is a sequence of labels* $< \tau_1, \ldots, \tau_n >$ *(*$n \geq 1$*), where* $\tau_i \in L_c$.

Definition 9 *A* **call sequence** $\mu =< \tau_k, \ldots, \tau_n >$ **is valid** *on an active state configuration* $\sigma_{o,k}$ *of instance o, if there is a PSM trace* $\gamma_o =< \sigma_{o,1} \ldots \sigma_{o,k} \ldots \sigma_{n+1} >$ *of o where for* $i \in \{k \ldots n\}$, $(\sigma_{o,i}, \tau_i, \sigma_{o,i+1}) \in T_c$.

Example 4 <`dispenseCash`, `printReceipt`, `ejectCard`> is a valid call sequence on the active state configuration {`VerifyWithdrawal`, `CardCharging`} of Figure 6.

4 Behaviour Inheritance Consistency of Class Refinements

When modelling the behaviour of classes in a class hierarchy, we are confronted with an important problem. How should the behaviour of the subclasses be related to the behaviour of the superclasses? This is a problem that does not only occur at the modelling level, but also at the source code level. For example, Meyer presents a taxonomy of 12 different valid kinds

of ways inheritance can be used in an object-oriented programming language [23]. Therefore, it is necessary to identify different notions of behaviour inheritance consistency between a class and its subclass. Depending on the precise relationship between a class and its subclass some notions of inheritance consistency will be valid, while others will not.

If we restrict ourselves to the definition of class behaviour that we have adopted in this paper, we can provide a precise characterisation of behaviour inheritance consistencies in terms of the relationship between the protocol state machines and sequence diagrams of a class and its subclass. Rather than inventing our own definition of behaviour inheritance consistency, we will rely on two variants that have already been defined by Ebert and Engels [10], namely *observation* and *invocation* inheritance consistency. Although it is very well possible that other useful definitions of inheritance consistency exist, we will restrict ourselves to these two in the current paper.

Observation inheritance consistency means that *each sequence of calls which is observable with respect to a subclass must result (under projection of the operations known) in an observable sequence of its corresponding superclass. If a subclass reacts to the invocation of an operation op, where op is also known to the superclass, this reaction must also be reflected in the superclass behaviour specification*. Observation consistency can be defined between state machines, between sequence diagrams, and between a state machine and sequence diagrams.

In order to define this kind of consistency, we need some auxiliary definitions.

Definition 10 *The* **restriction $\mu_{\mathbf{L}}$ of a sequence** $\mu =< \tau_1, \ldots, \tau_n >$ **to a set** L *is the sequence obtained from μ by removing all $\tau_i \notin L$.*

The **restriction** $U \mid L$ *of a set of sequences U to a set L is defined as $U \mid L = \{\phi \mid \exists \mu \in U : \phi = \mu_L\}$*

Definition 11 *Given a sequence diagram Δ and a PSM Π_c. The function* $\mathbf{label_c} : E_{\Delta,c} \rightarrow L_c : (m, cons) \rightarrow (op, g, h)$ *maps an event occurrence onto a label as follows:*

op is the operation corresponding to message[3] *m,*
$g = Pre_{op} \cup cons$,
$h = Post_{op}$.

Definition 12 Observation inheritance consistency. *Let c be a class, c' a subclass of c, and instances o of c and o' of c'.*

A PSM $\Pi_{c'} = (S', T', L', \rho', \Lambda')$ is observation inheritance consistent with a PSM $\Pi_c = (S, T, L, \rho, \Lambda)$ if, for every valid call sequence μ' of o', μ'_L is a valid call sequence of o.

A SD Δ' is observation inheritance consistent with a SD Δ with respect to c and c' if, for every instance o' of c', if $\nu' = \nu_{o'}/^{rec}$ is an SD trace in Δ', then $\nu'_{E_{\Delta,c}}$ is an SD trace in Δ.

[3] Remark that in this paper we only consider symbolic messages and transitions. See also Section 7.4.

A SD Δ' is observation inheritance consistent with a PSM $\Pi_c = (S, T, L, \rho, \Lambda)$ with respect to c' if, for every SD trace $\nu_{o'}/^{rec} = < e_1, \ldots, e_n >$ in Δ', there exists a valid call sequence $\mu_o =< \tau_1, \ldots, \tau_m >$ (with $m \geq n$), containing labels $\tau_j = (op, g, h) = label_c(e_i)$ (with $j \geq i$) and preserving the order of the e_i's in the trace (i.e., if $\tau_k = label_c(e_{i+1})$ than $k > j$).

Moreover, if $\tau_j = label_c(e_i)$ and $\tau_{j+u} = label_c(e_{i+1})$ (with $u \geq 1$) then all intermediate labels τ_r (with $r = 1 \ldots u-1$) are of the form $(\epsilon, g_r, \{\})$, and $\bigcup_{j \leq r \leq j+u} g_r$ is part of cons of e_i.

Remark that we do not define observation inheritance consistency between a PSM $\Pi_{c'}$ and an SD Δ with respect to the superclass c. Such a definition would imply that all possible scenarios are described by Δ, because every trace in the PSM $\Pi_{c'}$ must be observable in Δ under projection of the methods known. This demands completeness of the models which is seldom the case, especially not in early phases of the software development life cycle.

Also remark that we do not require $L \subseteq L'$ in our definition of observation inheritance consistency. Let $L = \{a, b\}$ and $L' = \{a\}$ and two state machines Π_c containing L and $\Pi_{c'}$ containing L'. Suppose that $\{< a >, < a, b >\}$ is a set of valid call sequences of Π_c and $\{< a >\}$ is a singleton containing the valid sequence of $\Pi_{c'}$. In this case, $\Pi_{c'}$ and Π_c are observation consistent while $L \nsubseteq L'$.

Example 5 Consider the protocol state machine $\Pi_{\texttt{ATM}}$ of Figure 4, and the protocol state machine $\Pi_{1.1}$ of the class `CardChargingATM` shown in Figure 6 that refines the behaviour specified by $\Pi_{\texttt{ATM}}$. $\Pi_{1.1}$ is observation consistent with $\Pi_{\texttt{ATM}}$. The objects of the class `CardChargingATM` do behave like the objects of the class `ATM` if viewed only to this class description. The extension of the state machine $\Pi_{1.1}$ representing the charging of the card only contains labels that are not known to $\Pi_{\texttt{ATM}}$.

Invocation inheritance consistency means that *any sequence of operations invocable on the superclass can also be invoked on the subclass*. This notion of behaviour inheritance consistency is based on the substitutability principle requiring that an object of subclass B of class A can be used where an object of class A is required.

Definition 13 Invocation inheritance consistency. *Let c be a class, c' a subclass of c, and instances o of c and o' of c'.*

A PSM $\Pi_{c'} = (S', T', L', \rho', \Lambda')$ is invocation inheritance consistent with a PSM $\Pi_c = (S, T, L, \rho, \Lambda)$ if every valid call sequence μ on $\{\rho\}$ in Π_c is also valid on $\{\rho'\}$ in $\Pi_{c'}$ and for all their respective PSM traces γ and γ' it holds that $\gamma = \gamma'_S$.

A SD Δ' is invocation inheritance consistent with a SD Δ with respect to c and c', if every SD trace $\nu_o/^{rec}$ in Δ is also a SD trace in Δ' for an instance o' of class c'.

A PSM $\Pi_{c'} = (S', T', L', \rho', \Lambda')$ is invocation inheritance consistent with a SD Δ with respect to c if, for every SD trace $\nu_o/^{rec} =< e_1 \ \ldots \ e_n >$ in Δ, there exists a valid

call sequence $\mu_{o'} =< \tau_1 \ldots \tau_m >$ (with $m \geq n$), containing labels $\tau_j = (op, g, h) = label_c(e_i)$ (with $j \geq i$) and preserving the order of the e_i's in the trace (i.e., if $\tau_k = label_c(e_{i+1})$ than $k > j$)

Moreover, if $\tau_j = label_c(e_i)$ and $\tau_{j+u} = label_c(e_{i+1})$ (with $u \geq 1$) then all intermediate labels τ_r (with $r = 1 \ldots u-1$) are of the form $(\epsilon, g_r, \{\})$, and $\bigcup_{j \leq r \leq j+u} g_r$ is part of cons of e_i.

Remark again that, in this case, we do not define invocation consistency between a PSM Π_c and a sequence diagram Δ' with respect to a subclass c' of c. Such a definition would imply completeness of the models involved, which is seldom the case.

Also remark that it is not necessary that S, respectively L, is a subset of S', respectively L'.

Example 6 The behaviour of the sequence diagram Δ of Figure 5 is **not** invocation consistent with behaviour of class `CardChargingATM` specified by the PSM $\Pi_{1.1}$ of Figure 6 with respect to class `ATM`. Indeed, the SD trace $< e_1, e_2, e_3, e_4, e_5, e_6 >$ of Example 1 does not correspond to a valid call sequence

```
<getAccountNbr, getAmountEntry,
checkIfCashAvailable, send, dispenseCash,
printReceipt>
```

in the PSM $\Pi_{1.1}$, that always requires the invocation of the operations concerning the charging of the card.

In the context of UML state diagrams, some inheritance policies are discussed in [28]. They are referred to as *Subtyping*, *Strict Inheritance* and *General refinement*. The subtyping policy corresponds to invocation consistency. The other policies correspond neither to observation nor to invocation consistency and are tailored towards implementation-level. The inverse of invocation consistency is known in literature as *Restriction Inheritance* [23]. This kind of inheritance occurs when the behaviour of the subclass is a subset of the behaviour of the parent class. It is an explicit violation of the principle of substitutability and should be avoided whenever possible, but sometimes it is inevitable. Restriction inheritance occurs for example, if a software modeler is modeling a class based on existing classes that should not, or cannot, be modified. This is the case if existing library classes are used.

Until now, we assumed that the PSMs expressed observable call sequences and invocable call sequences, while the sequence diagrams expressed observable traces and invocable traces. The following notations are introduced:

Notation 3 *The **set of invocable call sequences and SD traces** of a class c is denoted by* $\mathbf{IS}(c)$.
*The **set of observable call sequences and SD traces** of a class c is denoted by* $\mathbf{OS}(c)$.

In general, $IS(c)$ is a subset of $OS(c)$. This is stated in [10]. We will use these notations in Section 6 to prove behaviour inheritance consistencies or behaviour preservation properties between different classes. In the next section, behaviour preservation properties corresponding to the above presented inheritance consistencies are introduced and defined.

5 Behaviour Preservation of Model Refactorings

In the motivation of Section 2, we explained the need for guaranteeing behaviour consistency of UML models, and the need to preserve this consistency when the models evolve. When arbitrary changes are made to a model, it is quite likely that its behaviour consistency will not be preserved. However, there exists an important class of model evolutions, called *model refactorings*, that do preserve behaviour. The main goal of these model refactorings is to improve the structure (or other qualities) of the model, while preserving (most of) its behavioural properties.

In order to determine whether a given model refactoring preserves behaviour, we need to define precisely what this means. To achieve this, we will take an approach that is very similar to the one taken in Section 4: just like the behaviour of a subclass can be inheritance consistent with the behaviour of its superclass, the behaviour of a new version of a class can preserve the behaviour of the original version. Even more, the different flavours of behaviour inheritance consistency that were explored in Section 4 (namely observation and invocation inheritance consistency) also make sense in an evolution context. This will be formalised below.

Before doing so, however, we need to be clear about what it means to be "a new version of a class". We will adopt a very broad view here. It includes changes to the class itself (renaming, adding, removing or modifying operations or attributes), or to its associated behaviour (renaming, adding, removing or modifying state machine diagrams or sequence diagrams). But even more sophisticated changes can be envisioned, such as splitting a class into two or more classes (each of these new classes is then considered to be a new version of the original one), or combining two or more classes into a single merged version. Splitting or merging sequence diagrams or state machine diagrams can also be accommodated in this way.

The first notion of behaviour preservation between a class and its new version that we can formalise, is *observation call preservation*. Intuitively, it means that every call sequence observable with respect to a new version of a class (under projection of the operations known) must result in an observable call sequence of its corresponding original class.

Definition 14 Observation call preservation. *Let c be a class, c' a new version of c, and instances o of c and o' of c'.*

The behaviour specified by a PSM $\Pi_{c'} = (S', T', L', \rho', \Lambda')$ is observation call preserving with a PSM $\Pi_c = (S, T, L, \rho, \Lambda)$ if, for every valid call sequence μ' of o', μ'_L is a valid call sequence of o.

The behaviour specified by a SD Δ' is observation call preserving with a SD Δ with respect to c and c' if, for every instance o' of c', if $\nu' = \nu_{o'}/^{rec}$ is an SD trace in Δ', then $\nu'_{E_{\Delta,c}}$ is also an SD trace of Δ.

The behaviour specified by a SD Δ' is observation call preserving with a PSM $\Pi_c = (S, T, L, \rho, \Lambda)$ with respect to c' if, for every SD trace $\nu_{o'}/^{rec} = < e_1, \ldots, e_n >$ in Δ', there exists a valid call sequence $\mu_o =< \tau_1, \ldots, \tau_m >$ (with

$m \geq n$), containing labels $\tau_j = (op, g, h) = label_c(e_i)$ (with $j \geq i$) and preserving the order of the e_is in the trace (i.e., if $\tau_k = label_c(e_{i+1})$ than $k > j$)

Moreover, if $\tau_j = label_c(e_i)$ and $\tau_{j+u} = label_c(e_{i+1})$ (with $u \geq 1$) then all intermediate labels τ_r (with $r = 1 \ldots u-1$) are of the form $(\epsilon, g_r, \{\})$, and $\bigcup_{j \leq r \leq j+u} g_r$ is part of cons of e_i.

Observe that Definition 14 is almost identical to Definition 12. The main difference is that the words *observation inheritance consistent* are replaced by *observation call preserving*. Also, c' does not represent a subclass of c anymore, but a new version of c in the refactored model.

Example 7 The behaviour of the class `CardChargingATM` specified by the refactored PSM $\Pi_{1.3}$ of Figure 8 is not observation call preserving with respect to the PSM $\Pi_{1.2}$ of Figure 7. A first model refactoring used here, moves some simple states into a composite orthogonal state and a second model refactoring used, flattens two states. As a result of these refactorings, the amount to be charged on a card must not necessarily be entered after the amount to be dispensed (cf. $\Pi_{1.3}$). However, this precedence constraint is required in the description of the behaviour of the original version of `CardChargingATM` (cf. $\Pi_{1.2}$).

The second notion of behaviour preservation that we define is *invocation call preservation*. It guarantees that each call sequence invocable on a class, must also be invocable on the new version of the class in the refactored model.

Definition 15 Invocation call preservation. *Let c be a class, c' a new version of c, and instances o of c and o' of c'.*

A PSM $\Pi_{c'} = (S', T', L', \rho', \Lambda')$ is invocation call preserving with a PSM $\Pi_c = (S, T, L, \rho, \Lambda)$ if every valid call sequence μ on $\{\rho\}$ in Π_c is also valid on $\{\rho'\}$ in $\Pi_{c'}$ and for their respective PSM traces γ and γ' it holds that $\gamma = \gamma'_S$.

A SD Δ' is invocation call preserving with a SD Δ with respect to c and c', if every SD trace $\nu_o/^{rec}$ in Δ is also a SD trace in Δ' for an instance o' of class c'.

A PSM $\Pi_{c'} = (S', T', L', \rho', \Lambda')$ is invocation call preserving with a SD Δ with respect to c if, for every SD trace $\nu_o/^{rec} =< e_1 \ \ldots \ e_n >$ in Δ, there exists a valid call sequence $\mu_{o'} =< \tau_1 \ldots \tau_m >$ (with $m \geq n$), containing labels $\tau_j = (op, g, h) = label_c(e_i)$ (with $j \geq i$) and preserving the order of the e_i's in the trace (i.e., if $\tau_k = label_c(e_{i+1})$ than $k > j$)

Moreover, if $\tau_j = label_c(e_i)$ and $\tau_{j+u} = label_c(e_{i+1})$ (with $u \geq 1$) then all intermediate labels τ_r (with $r = 1 \ldots u-1$) are of the form $(\epsilon, g_r, \{\})$, and $\bigcup_{j \leq r \leq j+u} g_r$ is part of cons of e_i.

The definition of invocation call preservation is identical to Definition 13 by substituting *invocation call preserving* for *invocation inheritance consistency*.

Referring to Example 7 above, the behaviour specified by the refactored PSM $\Pi_{1.3}$ is invocation call preserving with the PSM $\Pi_{1.2}$.

6 Combining Behaviour Preservation and Behaviour Inheritance Consistencies

In this section, we will explore the relationship between inheritance consistencies and the notion of behaviour preservation. Assuming that there is a behaviour inheritance consistency relationship between a superclass and its subclass, we would like to find out if this consistency relationship is preserved when either the subclass or the superclass evolves. This question is schematically illustrated in Figures 9 and 10.

As a concrete example, reconsider the class `ATM` version 1.0 and its subclass `CardChargingATM` version 1.1 (see Figure 1). Consider now the PSM Π_1 of the class `ATM` as shown in Figure 4. Assume that the PSM $\Pi_{1.1}$ (shown in Figure 6) is extended by the composite state of the PSM Π_1. The PSMs are invocation inheritance consistent. The PSM $\Pi_{1.3}$ (partly shown in Figure 8) is also invocation call preserving with respect to the (extended) PSM $\Pi_{1.1}$. The question arises if, based on this information, we can prove that version 1.3 of class `CardChargingATM`, obtained by performing a model refactoring, is still behaviour inheritance consistent with version 1.0 of class `ATM`.

Abstracting away from SD traces and PSM call sequences, we can define invocation inheritance consistency or invocation call preservation as $IS(c) \subseteq IS(c')$, where c' is a subclass of c or a new version of c. Observation inheritance consistency or observation invocation call preservation can be defined by $OS(c' \mid V) \subseteq OS(c)$, where V denotes the set that is the union of the set of labels L_c and the set of event occurrences $\mathbf{E} = \bigcup_{\forall \Delta} \mathbf{E}_{\Delta,c}$, and c is a superclass of c' or a previous version of c.

In general, the following properties can be proven:

Proposition 1 *Let c_1 be a class, c'_1 a subclass of c_1, c_2 an identical copy of c_1 that is contained in a different model version (e.g., `ATM` class 1.0 in model version 1.2 and `ATM` class 1.0 in model version 1.3), and c'_2 a subclass of c_2 such that c'_2 is a new (modified) version of c'_1. (see also Figure 9)*

1. *If c'_1 and c_1 are* **invocation inheritance consistent** *and c'_2 and c'_1 are* **invocation call preserving**
   then c'_2 and c_2 are **invocation inheritance consistent**.
2. *If c'_1 and c_1 are* **observation inheritance consistent** *and c'_2 and c_2 are* **invocation inheritance consistent** *and $IS(c_1) = OS(c_1)$*
   then $OS(c'_1 \mid V_1) \subseteq OS(c'_2)$.
3. *If c'_1 and c_1 are* **invocation inheritance consistent** *and c'_2 and c_2 are* **observation inheritance consistent** *and $IS(c_1) = OS(c_1)$*
   then $OS(c'_2 \mid V_2) \subseteq OS(c'_1)$.

Proof 1. c'_1 and c_1 are invocation inheritance consistent, hence $IS(c_1) \subseteq IS(c'_1)$. c'_2 and c'_1 are invocation call preserving, hence $IS(c'_1) \subseteq IS(c'_2)$. Because c_1 and c_2 are identical, we conclude that
$IS(c_2) = IS(c_1) \subseteq IS(c'_1) \subseteq IS(c'_2)$
This implies that c_2 is invocation inheritance consistent with c'_2.

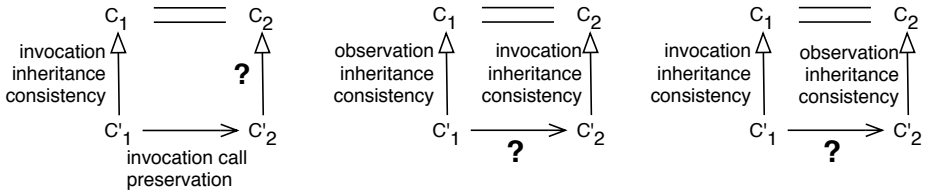

Fig. 9 Examples illustrating Proposition 1.

2. c'_1 and c_1 are observation inheritance consistent, hence $OS(c'_1 \mid V_1) \subseteq OS(c_1)$. c'_2 and c_2 are invocation inheritance consistent, hence $IS(c_2) \subseteq IS(c'_2)$. Because c_1 and c_2 are identical, and given that $IS(c_1) = OS(c_1)$, and, in general, $IS(c'_2) \subseteq OS(c'_2)$ we can conclude that $OS(c'_1 \mid V_1) \subseteq OS(c_1) = IS(c_1) = IS(c_2) \subseteq IS(c'_2) \subseteq OS(c'_2)$
   This results in: $OS(c'_1 \mid V_1) \subseteq OS(c'_2)$.
3. c'_1 and c_1 are invocation inheritance consistent, hence $IS(c_1) \subseteq IS(c'_1)$. c'_2 and c_2 are observation inheritance consistent, hence $OS(c'_2 \mid V_2) \subseteq OS(c_2)$. Because c_1 and c_2 are identical, and given that $IS(c_1) = OS(c_1)$, and, in general, $IS(c'_1) \subseteq OS(c'_1)$, we can conclude that $OS(c'_2 \mid V_2) \subseteq OS(c_2) = OS(c_1) = IS(c_1) \subseteq IS(c'_1) \subseteq OS(c'_1)$
   This results in: $OS(c'_2 \mid V_2) \subseteq OS(c'_1)$. □

From the first item of the proposition, we can conclude that the behaviour of version 1.3 of class `CardChargingATM`, specified by the union of the PSM Π_1 (shown in Figure 4) and the PSM $\Pi_{1.3}$ (shown in Figure 8), is invocation inheritance consistent with the behaviour of the `ATM` class as specified by the PSM Π_1.

The second item in the proposition means that the set of valid call sequences or traces of the class c'_1 under the projection of the methods known by c_1, must be included in the set of the call sequences or traces of the class c'_2, which is a new version of the class c'_1. Depending on how the behaviour of the different classes is specified, we conclude that:

1. Every valid call sequence of the PSM Π' of c'_1, restricted to the operations known by c_1, is also a valid call sequence of the PSM of c'_2;
2. Assume the existence of sequence diagrams Δ for c_1 and Δ' for c'_1. Then for each trace ν' in Δ', $\nu'_{E_{\Delta,c_1}}$ is an SD trace for c'_2;
3. Assume the existence of a PSM Π_{c_1} and sequence diagram Δ for c'_1. Then each call sequence $\mu =< \tau_1 \dots \tau_n >$ such that, for each $i \in \{1 \dots n\}$, $\tau_i = label_{c'_1}(e_i)$, is also a call sequence of instances of c'_2.

The third item in the proposition means that the set of valid call sequences or traces of the class c'_2 under the projection of the methods known by c_2, must be included in the set of the call sequences or traces of the class c'_1. As a consequence, the behaviour of the new version c'_2 is smaller than the behaviour of the original class. Depending on how the behaviour of the different classes is specified, we conclude that:

1. Every valid call sequence of the PSM Π' of c'_2, restricted to the operations known by c_2, is also a valid call sequence of the PSM of c'_1;
2. Assume the existence of sequence diagrams Δ for c_2 and Δ' for c'_2. Then for each trace ν' in Δ', $\nu'_{E_{\Delta,c_2}}$ is an SD trace in a sequence diagram $\Delta_{c'_1}$ with respect to c'_1;
3. Assume the existence of a PSM Π_{c_2} and sequence diagram Δ for c'_2. Then each call sequence $\mu =< \tau_1 \dots \tau_n >$ such that, for each $i \in \{1 \dots n\}$, $\tau_i = label_{c'_2}(e_i)$, is also a call sequence of instances of c'_1.

In the scenario described in Proposition 1 a certain subclass in a hierarchy is evolved. We can prove similar properties when a superclass in a hierarchy is evolved.

Proposition 2 *Let c_1 be a class, c'_1 a subclass of c_1, c_2 a new (modified) version of c_1, and c'_2 an identical copy of c'_1 that is contained in a different model version. (see also Figure 10)*

1. *If c'_2 and c_2 are* **invocation inheritance consistent** *and c_2 and c_1 are* **invocation call preserving**
   then c'_1 and c_1 are **invocation inheritance consistent**.
2. *If c'_1 and c_1 are* **observation inheritance consistent** *and c_2 and c_1 are* **invocation call preserving** *and*
   $IS(c_1) = OS(c_1)$
   then $OS(c'_2 \mid V_1) \subseteq OS(c_2)$.
3. *If c'_1 and c_1 are* **invocation inheritance consistent** *and c_2 and c_1 are* **observation call preserving** *and*
   $IS(c_1) = OS(c_1)$
   then $OS(c_2 \mid V_1) \subseteq OS(c'_2)$.

Proof 1. c'_2 and c_2 are invocation inheritance consistent, hence $IS(c_2) \subseteq IS(c'_2)$. c_2 and c_1 are invocation call preserving, hence $IS(c_1) \subseteq IS(c_2)$. Because c'_1 and c'_2 are identical, we conclude that
   $IS(c_1) \subseteq IS(c_2) \subseteq IS(c'_2) = IS(c'_1)$
   This implies that c_1 is invocation inheritance consistent with c_2.
2. c'_1 and c_1 are observation inheritance consistent, hence $OS(c'_1 \mid V_1) \subseteq OS(c_1)$. c_2 and c_1 are invocation call preserving, hence $IS(c_1) \subseteq IS(c_2)$. Because c'_1 and c'_2 are identical, and given that $IS(c_1) = OS(c_1)$, and, in general, $IS(c_2) \subseteq OS(c_2)$, we can conclude that $OS(c'_2 \mid V_1) = OS(c'_1 \mid V_1) \subseteq OS(c_1) = IS(c_1) \subseteq IS(c_2) \subseteq OS(c_2)$
   This results in: $OS(c'_2 \mid V_1) \subseteq OS(c_2)$.
3. c'_1 and c_1 are invocation inheritance consistent, hence $IS(c_1) \subseteq IS(c'_1)$. c_2 and c_1 are observation call

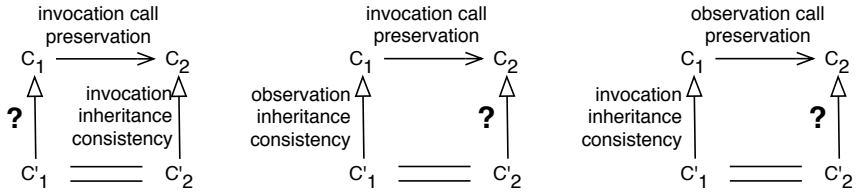

Fig. 10 Examples illustrating Proposition 2.

preserving, hence $OS(c_2 \mid V_1) \subseteq OS(c_1)$. Because c'_1 and c'_2 are identical, and given that $IS(c_1) = OS(c_1)$, and, in general, $IS(c'_1) \subseteq OS(c'_1)$, we can conclude that $OS(c_2 \mid V_1) \subseteq OS(c_1) = IS(c_1) \subseteq IS(c'_1) \subseteq OS(c'_1) = OS(c'_2)$
This results in: $OS(c_2 \mid V_1) \subseteq OS(c'_2)$. □

The second item in the proposition means that the set of valid observable call sequences or traces of the class c'_2 under the projection of the methods known by c_1, must be included in the set of the observable call sequences or traces of the class c_2, which is the new modified version of the class c_1. As a consequence, the behaviour of the subclass c'_2 is smaller than the behaviour of the modified class viewed from the original class.

The third item in the proposition means that the set of valid observable call sequences or traces of the class c_2 under the projection of the methods known by c_1, must be included in the set of the call sequence or traces of the class c'_2.

7 Specification in Description Logics

To be able to verify consistency and preservation, we need a formal specification and a formal reasoning engine relying on this specification. A formal approach is also beneficial for CASE tools, resulting in a precise and unambiguous approach. In this section, we present Description Logics (DLs) as a formal approach for checking behaviour inheritance consistencies and, as such, also call preservation properties based on the definitions given in Sections 4 and 5.

To detect behaviour preservation violations or inconsistencies between different UML elements, we adopted a logic-based approach for the following reasons: (1) The declarative nature of logic is well suited to express the design models which are also specified in a declarative way; (2) The logic reasoning algorithms are well understood due to their extensively studied, well-defined and sound semantics. First-order logic and theorem proving have been proposed by several authors for expressing software models and the derivation of inconsistencies from these models (e.g., [14,25,24,11]). Most of these techniques operationalise the consequence relation ($\vdash$) by using theorem proving based on the standard inference rules of classical logic; (3) Some logic reasoning engines can deduce implicitly represented knowledge from the explicit knowledge allowing an adequate treatment of incomplete, subjective, or time-dependent knowledge.

Spanoudakis *et al.* [37] identified two inherent limitations of logic-based approaches: (i) first-order logic is semi-decidable, hence it is impossible to provide for semantically adequate inference procedures, and (ii) theorem proving is computationally inefficient. DLs are an attempt to overcome both problems by restricting the expressive power.

Remark that the use of the Object Constraint Language (OCL) [26] is not really an option for detecting behaviour inheritance inconsistencies or call preservation violations. OCL is a query language and it only addresses static UML diagrams. It is not possible to formalise UML models using OCL and to reason about those models in the way DLs can. It would be possible to use OCL to check some of the behaviour preservation properties. However, in the case where traces are to be compared or enumerated, it can only be used in a limited way (due to, e.g., the infinity of the set of traces).

7.1 Introduction to DLs

DLs are a family of formalisms that are less expressive than first-order logic but have more specific reasoning abilities and are decidable. DLs represent the knowledge of the world by defining the *concepts* of the application domain and then using these concepts to specify properties of individuals occurring in the domain. The basic syntactic building blocks are *atomic concepts* (unary predicates), *atomic roles* (binary predicates) and *individuals* (constants). A DL is a two-variable fragment of first-order predicate logic. This implies that DLs only use a small set of constructors to construct *complex concepts and roles*.

Table 1 presents the syntax and semantics of $\mathcal{SHIQ}$, one of the most expressive DLs [17]. The constructors characterising $\mathcal{SHIQ}$ are: $\sqcap$ (conjunction of two concepts), $\sqcup$ (disjunction of concepts), $\neg$ (complement), $\exists$ (existential qualification), $\forall$ (universal quantification), inverse roles and number restrictions on qualified roles. $\mathcal{SHIQ}$ also allows for the definition of transitive roles, which are to be interpreted as transitive relations.

Using these concept and role constructors, complex concepts and roles can be formed. The following concept represents a model consisting of classes that have only abstract operations.[4]

[4] In the remainder of this section, we will use $\exists R$ as an abbreviation for $\exists R.\top$

Constructor	Syntax	Semantics	
atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$	
universal concept	$\top$	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$	
atomic role	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$	
transitive role	$R \in R_+$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^+$	
conjunction	$C_1 \sqcap C_2$	$C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$	
disjunction	$C_1 \sqcup C_2$	$C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$	$\mathcal{S}$
negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$	
value restriction	$\forall R.C$	$\{d_1 \mid \forall d_2 \in \Delta^{\mathcal{I}}.((d_1, d_2) \in R^{\mathcal{I}} \rightarrow d_2 \in C^{\mathcal{I}})\}$	
exists restriction	$\exists R.C$	$\{d_1 \mid \exists d_2 \in \Delta^{\mathcal{I}}.((d_1, d_2) \in R^{\mathcal{I}} \wedge d_2 \in C^{\mathcal{I}})\}$	
role hierarchy	$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$	$\mathcal{H}$
inverse role	R^-	$\{(x, y) \mid (y, x) \in R^{\mathcal{I}}$	$\mathcal{I}$
qualified number	$(\geq n\ R.C)$	$\{d_1 \mid \lvert\{d_2 \mid (d_1, d_2) \in R^{\mathcal{I}} \wedge d_2 \in C^{\mathcal{I}}\}\rvert \geq n\}$	$\mathcal{Q}$
restriction	$(\leq n\ R.C)$	$\{d_1 \mid \lvert\{d_2 \mid (d_1, d_2) \in R^{\mathcal{I}} \wedge d_2 \in C^{\mathcal{I}}\}\rvert \leq n\}$	

Table 1 Syntax and Semantics of $\mathcal{SHIQ}$

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

DLs come with a knowledge base formalism. A knowledge base in a DL is a pair consisting of a *Tbox* and *Abox*. A *Tbox* is used to introduce names for complex concepts. For example, for the concept defined above, we can introduce the name $ModelwithAbstractClasses$. In a DL *Tbox* the introduction of a concept name (CN) for a complex concept (C), is denoted by $CN \doteq C$.

More expressive *Tbox* formalisms allow the expression of so-called *general concept inclusion axioms* (GCI's). The left-hand side as well as the right-hand side of a GCI are complex concepts. The following GCI specifies that only classes with abstract operations can be abstract.

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

Concepts are classified in a *Tbox* according to subconcept-superconcept relationships using the subsumption ($\sqsubseteq$) relationship. An *Abox* is a finite, possibly empty set of individuals. These individuals are instances of concepts or form pairs representing the population of a certain role. If the corresponding concepts and roles are defined in the corresponding *Tbox*, then the individuals appearing in the *Abox* must obey the restrictions specified on these concepts and roles in the *Tbox*.

The most important feature of DLs is their classification ability. This reasoning task allows them to infer knowledge that is implicitly present in the knowledge base. The other basic reasoning capabilities are: instance checking (is a certain individual an instance of a certain concept?), relation checking (does a pair of individuals belong to the population of a certain role?), concept satisfiability (is there an interpretation for the concept?). These reasoning tasks are implemented by sound and complete reasoning algorithms. For several DLs, there exist optimal automata-based algorithms that decide satisfiability of concepts with respect to a *Tbox*, and subsumption of concepts. The satisfiability problem is reduced to the emptiness problem of automata. To get an idea of the technique the interested user is referred to [21] and [6].

Several implemented DL reasoning engines exist from which we have selected the state-of-the-art *RACER* [16] system. Other qualities of DLs and DL systems that make them suitable for our work, are:

- DL systems have an open world semantics, which allows the specification of incomplete knowledge. This is useful, e.g., for modeling sequence diagrams which typically specify incomplete information about the dynamic behaviour of the system.
- Due to their semantics, DLs are suited to express the static structure of the software application. For example, Calí *et al.* [5] translate UML class diagrams to a DL.
- Due to the close relationship between modal logics and DLs (e.g., there is a one-to-one mapping between the DL $\mathcal{ALCI}_{reg}$ and *converse*-Propositional Dynamic Logic), DLs are also suited to express, to a certain extent, the behaviour of a software application.

In [43, 44, 36], we reported on how a fragment of the UML metamodel is translated into such an expressive DL *Tbox*. User-defined models are translated into a corresponding *Abox*. This translation guarantees the consistency of the user-defined models with respect to the UML metamodel. We also reported on how different queries can be executed on those instances representing the user-defined models, e.g., different consistency checks that use meta information of the user-defined models.

The question that we will attempt to answer in the remainder of this section is to what extent DLs can be used as a semantic domain to express PSMs and call sequences/traces. If it is possible to use DLs for this purpose, we can also use them to check the different behaviour inheritance consistencies and call preservation properties defined in Sections 4 and 5.

7.2 Call Sequence and SD Trace Encoding in $\mathcal{SHIQ}(\mathcal{D}^-)$

The logic $\mathcal{SHIQ}(\mathcal{D}^-)$ is supported by RACER. This logic is equal to the logic $\mathcal{SHIQ}$ augmented with support for so-called *concrete domains* ($\mathcal{D}^-$). In many applications it is necessary to refer to concrete domains and predefined predicates on these domains when defining concepts. In our context, this is necessary to express pre- and postconditions. A concrete domain consists of a domain and a set of predicate names. The concrete domain $\mathcal{N}$ has as its domain the set of non-negative integers $\mathbb{N}$ and its set of predicate names $pred(\mathcal{N})$ consists of the binary predicate names, $<, \leqslant, \geqslant, >$ and the unary predicate names $<_n, \leqslant_n, \geqslant_n, >_n$ for $n \in \mathbb{N}$ and those predicate names are all interpreted by predicates on $\mathbb{N}$. The DL $\mathcal{SHIQ}(\mathcal{D}^-)$ supports the concrete domains representing integers, reals, complex numbers and strings.

If sequence diagrams and PSMs are compared to check for behaviour inheritance consistencies or preservation properties, SD traces and call sequences are compared.

Recall Definition 8, defining a call sequence of a PSM Π. A call sequence $\mu =< \tau_1, \ldots, \tau_n >$ $(n \geq 1)$, where $\tau_i \in L$, can be encoded in $\mathcal{SHIQ}(\mathcal{D}^-)$ in different ways depending on the format of the label τ_i.

- If $\tau_i = (operation, g, h)$, the label represents the call of an operation together with possible pre- and postconditions. The invocation of the operation $operation$ is encoded by the GCI: $\forall op.op' \sqcap (= 1\ op)$ where $operation$ is translated into an atomic concept op'. The invocation of the different operations is represented by the definition of a role op. This role is connected to the concept op' representing the operation $operation$. The pre- and postconditions, g and h are translated into concepts. How pre- and postconditions are translated into $\mathcal{SHIQ}(\mathcal{D}^-)$ and how expressive these constraints can be, is explained in [42].
- If $\tau_i = (\epsilon, g, \{\})$, only g is translated into a concept.
- The question remains how to express that the different operations have to be called in sequence. For this purpose, a binary relation r is used and also the subsumption relation is exploited. Different GCI's of the form $guard_1 \sqcap call_1 \sqsubseteq \exists\ r.(call_2 \sqcap postcondition_1 \sqcap guard_2)$ are defined in a *Tbox* representing the different labels of a call sequence. $call_1$ and $call_2$ represent the *invocation* of a certain operation. For example, a call sequence $<$ (op_1,g_1,h_1), (op_2,g_2,h_2), (ϵ, g_3, {}), (op_4,g_4,h_4) $>$, will be represented by the GCI's:
  $g'_1 \sqcap \forall\ op.op'_1 \sqcap \exists\ op \sqcap (\leq 1\ op) \sqsubseteq \exists\ r.(\forall\ op.op'_2 \sqcap \exists\ op \sqcap (\leq 1\ op) \sqcap h'_1 \sqcap g'_2)$,
  $g'_2 \sqcap \forall\ op.op'_2 \sqcap \exists\ op \sqcap (\leq 1\ op) \sqsubseteq \exists\ r.(\forall\ op.op'_4 \sqcap \exists\ op \sqcap (\leq 1\ op) \sqcap h'_2 \sqcap g'_4 \sqcap g'_3)$,
  $g'_4 \sqcap g'_3 \sqcap \forall\ op.op'_4 \sqcap \exists\ op \sqcap (\leq 1\ op) \sqsubseteq \exists\ r.h'_4$,
  where g'_i are concepts representing the preconditions g_i, and h'_i are concepts representing the postconditions h_i. The different concepts op'_i represent the corresponding operations op_i.

In case a call sequence represents the complete set of possible labels, the different $\exists r.X$ concepts, where X is a concept variable are replaced by $\forall r.X$. The binary relation r is similar to the accessibility relation in modal logic. The first GCI specified above, expresses that it is possible to reach the world where op'_2 is triggered and h'_1 and g'_2 hold, starting from the world where g'_1 holds and op'_1 is triggered.

The receiving SD traces of Definition 3, containing event occurrences denoting the receipt of a message, are translated similarly. Consider a receiving SD trace $\nu_o/^{rec} =<(m_1, cons_1), (m_2, cons_2), (m_3, cons_3) >$, this trace is translated into the different GCI's:
$cons'_1 \sqcap m'_1 \sqsubseteq \exists\ r.(m'_2 \sqcap cons'_2)$,
$cons'_2 \sqcap m'_2 \sqsubseteq \exists\ r.(m'_3 \sqcap cons'_3)$,
where m'_i is a concept representing the message m_i, and $cons'_i$ is a complex concept representing the set of constraints $cons_i$ which must be valid before the execution of the owning event occurrence. In our formalisation, the different messages m_i only contain the operation invoked. Each message m_i is defined as $m_i \doteq \forall\ op.op'_i \sqcap \exists\ op \sqcap (\leq 1\ op)$, where op'_i represents the operation op_i. This translation of a message is similar to the translation of the invocation of a call of an operation. This similarity allows straightforward verification of properties between call sequences and SD traces.

The SD trace denoting the receipt of messages by the object `atm` shown in the sequence diagram in Figure 5, is represented in $\mathcal{SHIQ}(\mathcal{D}^-)$ by the following GCI's:

$$\text{m1} \doteq \forall \text{op.getAccountNbr} \sqcap (\leq 1\ \text{op}) \sqcap \exists \text{op}$$
$$\text{m2} \doteq \forall \text{op.getAmountEntry} \sqcap (\leq 1\ \text{op}) \sqcap \exists \text{op}$$
$$\text{m3} \doteq \forall \text{op.checkIfCashAvailable} \sqcap (\leq 1\ \text{op}) \sqcap \exists \text{op}$$
$$\text{m4} \doteq \forall \text{op.send} \sqcap (\leq 1\ \text{op}) \sqcap \exists \text{op}$$
$$\text{m5} \doteq \forall \text{op.dispenseCash} \sqcap (\leq 1\ \text{op}) \sqcap \exists \text{op}$$
$$\text{m6} \doteq \forall \text{op.printReceipt} \sqcap (\leq 1\ \text{op}) \sqcap \exists \text{op}$$
$$\text{m1} \sqsubseteq \exists \text{r.m2}$$
$$\text{m2} \sqsubseteq \exists \text{r.m3}$$
$$\text{m3} \sqsubseteq \exists \text{r.m4}$$
$$\text{m4} \sqsubseteq \exists \text{r.m5}$$
$$\text{m5} \sqsubseteq \exists \text{r.m6}$$

The corresponding call sequence in the PSM shown in Figure 4 is represented by the following GCI's:

$$\text{t1} \doteq \forall \text{op.getAccountNbr} \sqcap (\leq 1\ \text{op}) \sqcap \exists \text{op}$$
$$\text{t2} \doteq \forall \text{op.getAmountEntry} \sqcap (\leq 1\ \text{op}) \sqcap \exists \text{op}$$
$$\text{t3} \doteq \forall \text{op.checkIfCashAvailable} \sqcap (\leq 1\ \text{op}) \sqcap \exists \text{op}$$
$$\text{t4} \doteq \forall \text{op.send} \sqcap (\leq 1\ \text{op}) \sqcap \exists \text{op}$$
$$\text{t5} \doteq \forall \text{op.dispenseCash} \sqcap (\leq 1\ \text{op}) \sqcap \exists \text{op}$$
$$\text{t6} \doteq \forall \text{op.printReceipt} \sqcap (\leq 1\ \text{op}) \sqcap \exists \text{op}$$

$$\exists \text{WITHDRAWAL}.\top \sqcap \text{t1} \sqsubseteq \forall \text{r.t2}$$
$$\text{t2} \sqsubseteq \forall \text{r.t3}$$
$$\text{t3} \sqsubseteq \forall \text{r.(t4} \sqcap \exists \text{cashAvailable}.\top)$$

$\exists$cashAvailable.$\top$ $\sqcap$ t4 $\sqsubseteq$ $\forall$r.(t5 $\sqcap$ $\exists$allowedWithdrawal.$\top$)

$\exists$allowedWithdrawal.$\top$ $\sqcap$ t5 $\sqsubseteq$ $\forall$r.t6

Remark that in this case, the call sequence of the PSM is considered to be complete as opposed to the SD trace of the sequence diagram. This is why $\forall$r.X is used in the translation of the call sequence.

If all the call sequences of a certain PSM must be encoded, a label can be followed by several other labels. Consider as an example, the PSM of Figure 4 and the transition calling the operation $printReceipt$. After this transition, two transitions are possible, or the card is ejected or the withdrawal transaction is chosen. To express this, the *or* constructor is used. The GCI $(\forall\ op.printReceipt \sqcap (=\ 1\ op)) \sqsubseteq \exists\ r.(\forall\ op.ejectCard \sqcap (=\ 1\ op)) \sqcup (\forall\ op.\ getAccountNbr \sqcap (=\ 1\ op) \sqcap (\exists\ withdrawal))$ expresses that the invocation of the operation $printReceipt$ is followed by a call to $ejectCard$ *or* by an invocation of the $getAccountNbr$ operation.

By translating the call sequences of the whole PSM shown in Figure 6, we can check if this is, e.g., invocation inheritance consistent with the sequence diagram of Figure 5.

Due to the above translations of call sequences and SD traces, the set of constraints $cons$ of an event occurrence is logically consistent with the preconditions of the operation if these preconditions are specified in the corresponding label. No relation is imposed on the set of constraints of an event occurrence and the postconditions of the corresponding label by this translation.

7.3 Taking State Information into account

The states specified in a PSM $\Pi_c = (S_c, T_c, L_c, \rho_c, \Lambda_c)$ can be taken into account and translated into $\mathcal{SHIQ}$ statements as follows:

- $s \in S_c$ is translated into an atomic $\mathcal{SHIQ}(\mathcal{D}^-)$ concept.
- Initial states and final states are also translated into atomic $\mathcal{SHIQ}(\mathcal{D}^-)$ concepts. The restrictions on these kinds of states as specified in the definition of a PSM, are implemented on the UML metamodel representation in $\mathcal{SHIQ}(\mathcal{D}^-)$.
- A composite state is also explicitly translated into a complex $\mathcal{SHIQ}(\mathcal{D}^-)$ concept consisting of the intersection of the different substates. As an example, consider the composite state `GettingCustomerSpecifics` in Figure 4 consisting of three different substates. This composite state is represented by: $GettingCustomerSpecifics \equiv AccountEntry \sqcup AmountEntry \sqcup VerifyATMBalance$, $AccountEntry \sqsubseteq GettingCustomerSpecifics$, $AmountEntry \sqsubseteq GettingCustomerSpecifics$ and $VerifyATMBalance \sqsubseteq GettingCustomerSpecifics$.
- A labelled relation (S_1, l, S_2) such that l is a triple (op, g, h) and $S_1 = \{s_{1,1}, \ldots, s_{1,n}\}$ and $S_2 = \{s_{2,1}, \ldots, s_{2,m}\}$, is translated into a GCI $s'_{1,1} \sqcap \ldots \sqcap s'_{1,n} \sqcap g' \sqcap (\forall op.op' \sqcap (=\ 1\ op)) \sqsubseteq \exists r.(s'_{2,1} \sqcap \ldots \sqcap s'_{2,m} \sqcap h')$. $s'_{i,j}$ are $\mathcal{SHIQ}(\mathcal{D}^-)$ concepts representing the state $s_{i,j}$. The operation op is translated into an atomic concept op'. h and g are translated into complex concepts h' and g'.
- A labelled relation (S_1, l, S_2) such that l is a triple $(\epsilon, g, \{\})$ and $S_1 = \{s_{1,1}, \ldots, s_{1,n}\}$ and $S_2 = \{s_{2,1}, \ldots, s_{2,m}\}$, is translated into a GCI $s'_{1,1} \sqcap \ldots \sqcap s'_{1,n} \sqcap g' \sqsubseteq \exists r.(s'_{2,1} \sqcap \ldots \sqcap s'_{2,m})$. The different states and the precondition g are translated in the same way as described earlier.

In certain contexts, additional restrictions must be integrated in the *Tbox* representing call sequences, SD traces or PSMs. For example, completeness of the set of states can be enforced. Another restriction is the disjointness of the different top-level states of a PSM, i.e., states directly belonging to the PSM. This disjointness restriction guarantees that two states cannot be active at the same time.

Depending on which preservation property or inheritance consistency needs to be checked, only relevant parts of the PSMs are translated. Suppose we want to check observation inheritance consistency between the PSMs as specified in Figure 4 and Figure 6, then both state machines will be translated into the DL $\mathcal{SHIQ}(\mathcal{D}^-)$. To be able to check observation inheritance consistency, the PSM of the original class is assumed to be complete and the PSM of the refined class is translated by only taking into account the operations known to the original class. The reasoning task *Tbox coherence*, i.e., checking whether each concept in the *Tbox* is satisfiable, is used to check observation inheritance consistency or observation call preservation. This reasoning task is used to check all consistency and preservation properties defined in this paper.

7.4 Discussion

By translating a PSM and its call sequences into a DL and by translating SD traces into DL concepts, it is not only possible to check the consistency and preservation properties of Section 6. We can also prove that, e.g, a certain message m always occurs, by checking the satisfiability of the concept $\forall r.m$. Similarly we can prove that a certain message m occurs at least once, by checking the satisfiability of the concept $\exists r.m$.

By translating constraints into a DL *Tbox* together with a class diagram (as specified in [43]) or with a PSM or sequences and traces, some other properties can be checked. The constraints can be checked for consistency with respect to a given class diagram. A constraint is consistent with respect to the class diagram if it can be satisfied without contradicting the conditions imposed by the classes. The set of constraints defined on, e.g., a state machine, can be checked for internal consistency. Furthermore, it would be possible to check constraint equivalence, this boils down to equivalence of logical formula's.

The current disadvantage of our above introduced translations, is the lack of feedback given to the user. If a *Tbox* is not

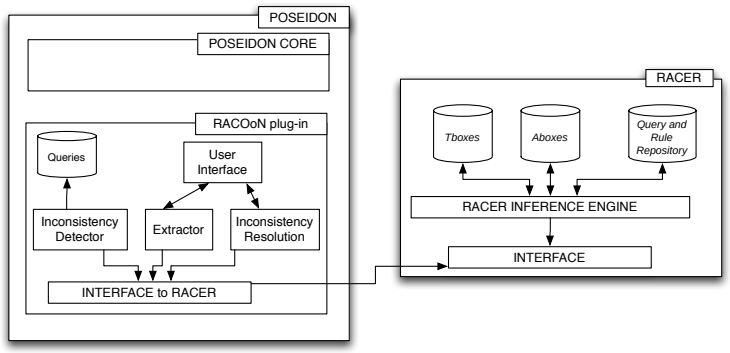

Fig. 11 Architecture of *RACOoN*.

satisfiable, the DL reasoning engine returns the set of unsatisfiable concepts. From this information only, we are not able to deduce which SD traces occur in the sequence diagram. As a result, we do not have access to the corresponding call sequences in the PSM. To be able to inform the user correctly, i.e., to back-annotate the UML model with information concerning the cause of the inconsistency or preservation violation, two items must be further investigated. First, current DL tools, such as RACER, should give more and proper feedback on the cause of the satisfiability problem in case we check for *Tbox* satisfiability. Second, the necessary information for reconstructing a UML model from a DL translation must be stored.

Remark that in the definitions of Sections 3, 4 and 5 and, as a consequence, also in the DL formalisation, only symbolic labelled messages and transitions, however, with constraints are considered. In our examples, we used parametrised messages and transitions. Consider the transition from the state `VerifyATMBalance` to the state `VerifyWithdrawal` in the protocol state machine shown in Figure 4. In our translation we only take into account the constraint, i.e., `cashAvailable` and the operation called, i.e., `send`. The parameter `m` and the return parameter `allowedWithdrawal` of the operation are not considered. Nevertheless, we are confident that our ideas can be extended and mounted to a more detailed level of messages and transitions. Remark that `allowedWithdrawal` is used in a constraint on the transitions outgoing the state `VerifyWithdrawal` and translated into a DL concept.

8 Implementation in a CASE tool

In [36], we set up a preliminary tool chain for the purpose of checking inconsistencies between UML models. This tool chain has evolved into a Poseidon [15] plug-in which was initially developed by Jocelyn Simmonds [35]. We extended this plug-in with various inconsistency detection queries, the ability to generate different *Tboxes* and an inconsistency resolution approach.

The architecture of our environment is depicted in Figure 11. This figure shows Poseidon and RACER as starting elements. *RACOoN* is plugged into the Poseidon tool and contains several components. In this paper, we focus on the *Extractor*, *Inconsistency Detector* and *User Interface* components. Each of these components will be discussed below.

- The *User Interface* allows for choosing and executing the detection of particular inconsistencies on UML models. It also allows the end-user to configure the tool and to load the DL translation of the UML metamodel into RACER. It makes a conceptual difference between checking consistencies and verifying behaviour preservation properties. This resulted in the creation of a pane *Preservation Manager*. A screenshot of the plug-in and this pane is shown in Figure 12. The end-user can select the preservation property to be checked and also the UML diagram specifying the behaviour of the original class and the refactored class respectively.
- The *Extractor* has a double functionality. It translates the user-defined models into *Abox* assertions and loads this *Abox* into RACER. It also allows for the translation of PSMs, call sequences and SD traces as specified in Section 7.
- Due to the correspondence between inheritance behaviour consistencies and call preservation properties, the component *Inconsistency Detector* can be used in the same way for the detection of inheritance inconsistencies as for the detection of violations of corresponding preservation properties.
- The *Interface to RACER* handles the communication between the different components (such as *Extractor* and *Inconsistency Detector*) and the RACER engine.

The plug-in is written in a modular way. Translating models from Poseidon to RACER and checking properties on these models are independent activities. The checks are executed on user-demand. The catalogue of inconsistencies and preservation properties is not hard-coded. It can be selected by the user and dynamically changed.

9 Discussion and Related Work

Many notions of behaviour inheritance consistency have been presented in literature. Compared to Schrefl *et al.* [39], our

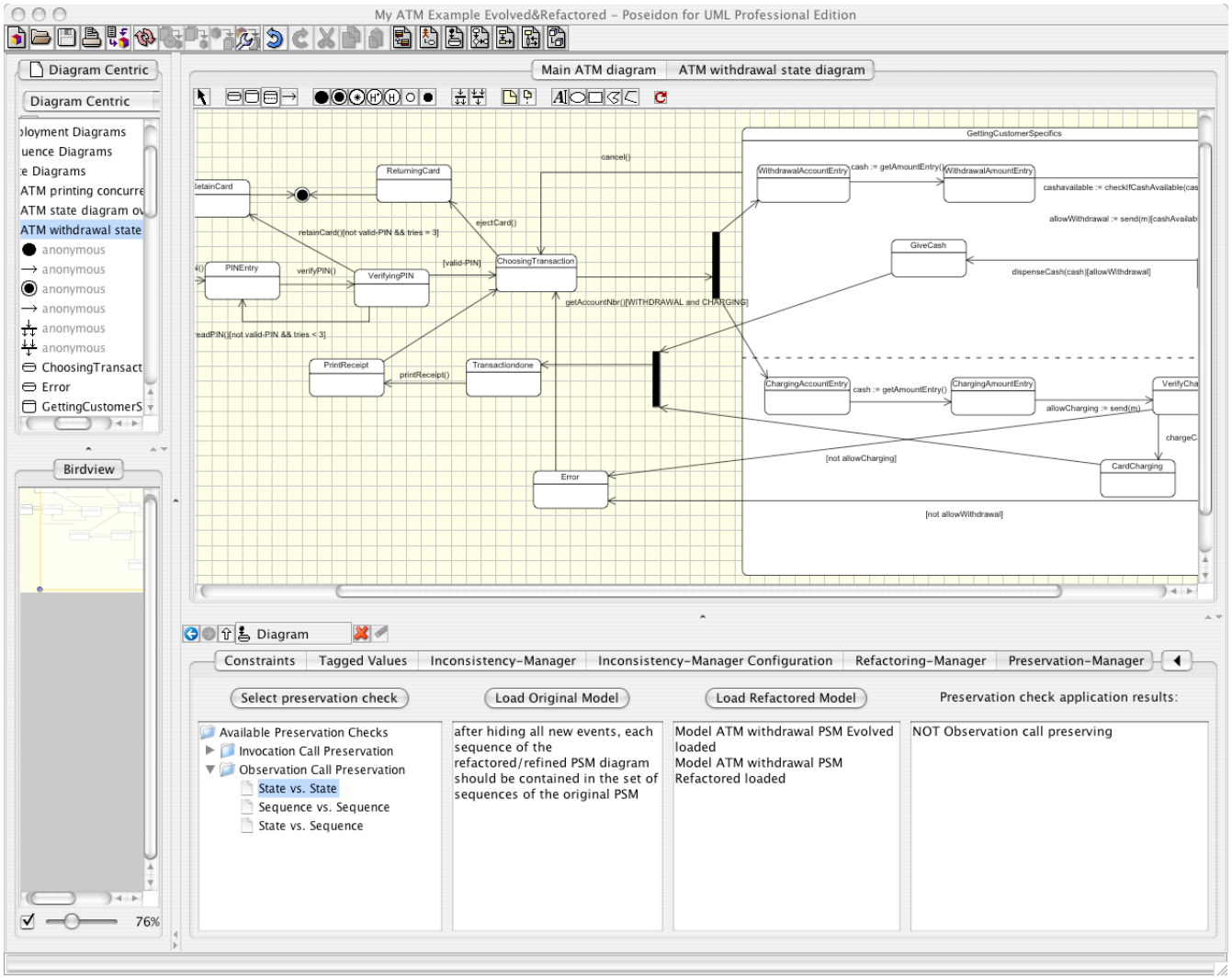

Fig. 12 Screenshot of *RACOoN* plugged into Poseidon.

notions of behaviour inheritance consistency are more general, since they are defined independent of the kind of subtype relation between the superclasses and their subclasses. Engels *et al.* [10] define observable and invocation consistency using homomorphisms on state diagrams. Criteria for inheritance of object life cycles based on Petri nets are discussed in [39, 32,41]. Approaches based on CSP are discussed in [12,30]. CSP is used as a medium to check consistency, i.e., the UML model remains consistent if its CSP translation remains consistent. Moreover, CSP refinement relations are used to check and define several inheritance approaches and subtyping relations. In this approach, it is necessary to understand the effects that CSP refinement relations induce on UML models.

There is an abundant amount of work on model checking for different statechart variants. Recent work concerns the translation of UML state machines into the model checker SPIN [31], [19]. Model checking is an automatic, model-based, property-verification approach. It is intended to be used for concurrent, reactive systems. Model checking focuses explicitly on temporal properties and the temporal evolution of systems. It starts with a model described in a description language, and it discovers whether properties described in a specification language, are valid on the model. If they are not valid, it can produce counterexamples, consisting of execution traces. The specification language can be a temporal logic or a labelled transition system. The specification language used by SPIN is PROMELA. SPIN can be used as a full LTL (linear-time temporal logic) model checking system, i.e., the properties to be verified are LTL formulas. Temporal logics have a dynamic aspect, since truth of a formula is not fixed in a model, as it is in predicate or propositional logic. The models of temporal logic contain several states and a formula can be true in some states and false in others. Other specification languages are also supported by SPIN. In [31] for example, UML state machines are translated into PROMELA, while Büchi automata are used to describe the properties to be verified. In this particular case the properties are UML collaboration diagrams.

DLs can also be seen as an automatic, model-based, property-verification approach. In DLs the specification and de-

scription languages are the same. The built-in reasoning capabilities are used to verify whether hypotheses asserted by the user are valid on the model. This does not result in counterexamples which consist of execution traces. Model checking starts with a logic model and discovers whether properties asserted by the user are valid. In a model checking approach the verification relies on an exhaustive search of all states that the system will encounter. This gives rise to the known problem of state explosion. DLs construct models in which the property is valid. Model checking focuses explicitly on temporal properties unlike DLs. However, some DLs correspond to certain modal logics of which temporal logics are special cases. Which properties can be checked by a model checker depends on the expressiveness of the specification language.

Research on model refactoring is also emerging. A set of basic UML refactorings is provided in [40] to improve the software design in a stepwise fashion. Boger *et al.* show how model refactorings can be integrated in the Poseidon UML refactoring browser [3]. Astels uses a UML tool to perform refactorings more easily, and also to aid in code smell detection [1]. Model refactorings are defined in [29] as a sequence of transformation rules. Surprisingly, none of the above approaches towards model refactoring takes behaviour preservation into account. One of the reasons is that there is no generally accepted behavioural interpretation of UML models. Therefore, we consider this as an important contribution of our paper.

The approach presented in our paper does not explicitly specify model refactorings as model transformations. In order to do this, the UML metamodel first needs to be extended with a model transformation language (e.g., based on the ideas of graph transformation [22]). We also need a formal means to prove that a transformation preserves precisely those behavioural properties that we want to reason about (e.g., observation and invocation call preservation). Using such a formalism, we can guarantee that the refactored model is still consistent, without needing to recheck all consistency rules. This is precisely the approach taken by [13] in the context of UML-RT. Transformation rules specify local modifications that preserve a local consistency property (e.g., absence of deadlocks) that can be checked locally. This enables an incremental approach to consistency checking. Such an incremental approach provides a promising alternative to traditional model checking approaches [7], where the entire model needs to be verified again, even when small and local changes have been made to a model.

A lot of research has been done in the context of program refinements. Specifically, in the context of model refinement, two methodologies are well-known, namely Catalysis [9] and KobrA [4]. Catalysis [9] provides complete support for component-based development with objects and frameworks, building on emergent standards including the UML. Component interactions can be described on different levels of detail. More detailed descriptions can be built in a systematic way from abstract ones. High-level process guidelines are provided for applying refinement techniques. The KobrA method [4] represents a synthesis of several advanced software technologies. The basic goal is to provide a systematic approach to the development of component-based application frameworks. UML models are instantiated and mapped through a set of well-defined refinement and translation steps into an executable representation.

Recently, refinement techniques have been applied in the context of UML. Davies and Crichton [8] use CSP refinement relations to induce a notion of refinement for UML. The proposed approach necessitates a thorough understanding of the effects and restrictions of CSP refinement relations on UML. Jürjens [18] uses refinement for security-critical systems in UML. He introduces a formal semantics for UML and defines two kinds of refinements which preserve some domain-dependent safety properties. Shen *et al.* [34] propose a set of rules based on UML class diagram refinement, more specifically refinement of relationships specified between classes, to support model checking for software refinement. These rules are embedded in the UML metamodel using stereotypes and keep the class models consistent. Only limited rules considering basic elements of class diagrams are taken into account. Whittle [45] investigates the role of refinement in UML class diagrams with OCL constraints. In particular, he provided an extended example of the automation of parts of the design process using transformations.

10 Conclusion and Future Work

In this paper, we formalised behaviour expressed by UML protocol state machines and sequence diagrams. Based on this formalisation, different kinds of behaviour inheritance consistencies were defined between state machines and sequence diagrams of a class and its refined subclass. Using those consistency definitions, we derived similar definitions of behaviour preservation properties between a class and its refactored version. We also proved some interesting additional properties regarding inheritance consistency and behaviour preservation between a refined class and its refactored version.

Based on our formalisation, description logics were proposed and applied as a reasoning formalism for the detection of inconsistencies and call preservation violations. As a proof of concept of our approach, we implemented our ideas as an extension of a Poseidon plug-in in which the RACER DL system is used to formally specify and reason about UML models as a collection of DL concepts and roles. We used this tool to detect the inconsistencies we encountered in the evolving design of an ATM simulation. The DL reasoning abilities were successfully used to detect inconsistencies between the behaviour of a class and its refinements. We also used the tool to check the preservation of behaviour between subsequent versions of a class.

Until now, we only carried out experiments on small examples. Experiments on real industrial models remain to be done. Based on these experiments, we hope to find out which kinds of model refactorings are most useful in practice, and what are the kind of behavioural properties that should be preserved by these refactorings.

We also plan to explore if other consistency specifications such as the ones defined in [43] correspond to the preservation of certain behavioural properties. We need to extend our formalisation for parametrised messages and transitions. As a consequence, we must investigate to which extent DLs can be used in this case or whether another reasoning formalism will be needed.

From a tool perspective, we need to supply the user with more detailed feedback about the detected inconsistencies or preservation violations. This can be achieved by storing extra information on the translation of UML model elements into DL concepts. The tool also needs to be enhanced to support *resolution* of detected inconsistencies or preservation violations. Finally, we plan to extend our ideas to deal with consistency maintenance and behaviour preservation across different levels of abstraction (e.g., requirements specifications, analysis and design models, and programs). This will allow us to provide better formal support throughout the model-driven engineering process.

Biographies

Ragnhild Van Der Straeten received the degrees of Licentiate in Applied Mathematics at the Universiteit Gent (Belgium), and Licentiate in Applied Computer Science at the Vrije Universiteit Brussel (Belgium). She is a teaching assistant and Ph.D. student at the System and Software Engineering Lab at the Vrije Universiteit Brussel. She has published many peer-reviewed international articles on the topic of consistency maintenance between UML models.

Tom Mens received the degrees of Licentiate in Mathematics, Advanced Master in Computer Science, and PhD in Science at the Vrije Universiteit Brussel. He has been a post-doctoral fellow of the Fund for Scientific Research Flanders (FWO). He currently lectures on software engineering and programming languages at the Université de Mons-Hainaut. He has published numerous peer-reviewed articles on the topic of software evolution, and has been co-organiser, program committee member and referee of many international workshops and conferences. He is involved in various national and international projects and networks on software evolution. He is a member of both the ACM and the IEEE Computer Society.

Viviane Jonckers received her Ph.D. in Computer Science from the VUB (1987). Since 1987, she has been a professor both at the Department of Computer Science of the Faculty of Science and at the Department of Informatics of the Faculty of Engineering. Currently, she is head of the Department of Computer Science and director of the System and Software Engineering Lab. Her research interests are knowledge-based programming, the use of (semi-) formal methods in software engineering and knowledge engineering, and more recently, component-based and service-based software development.

References

1. D. Astels. Refactoring with UML. In *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, pages 67–70, 2002.
2. F. Baader, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.
3. M. Boger, T. Sturm, and P. Fragemann. Refactoring browser for UML. In *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, pages 77–81, 2002.
4. C. Bunse and C. Atkinson. The normal object form: Bridging the gap from models to code. In *Proc. Int'l Conf. UML'99*, volume 1723, pages 691–705. Springer-Verlag, 1999.
5. A. Calí, D. Calvanese, G. De Giacomo, and M. Lenzerini. A formal framework for reasoning on UML class diagrams. In *Foundations of Intelligent Systems: 13th International Symposium*, volume 2366 of *LNCS*, pages 503–. Springer-Verlag, 2002.
6. D. Calvanese, G. De Giacomo, and M. Lenzerini. 2atas make dls easy. In *Proc. Int'l Workshop on Description Logics*, volume 53 of *CEUR Electronic Workshop Proceedings*, pages 107–118, 2002.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
8. J. Davies and C. Crichton. Concurrency and refinement in the UML. In J. Derrick, E. Boiten, J. Woodcock, and J. von Wright, editors, *Electronic Notes in Theoretical Computer Science*, volume 70. Elsevier, 2002.
9. D. D'Souza and A. Wills. *Objects, Components and Frameworks with UML: the Catalysis Approach*. Addison-Wesley, 1998.
10. J. Ebert and G. Engels. Specialization of object life cycle definitions. Fachbericht Informatik 19/95, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, 1995.
11. W. Emmerich, A. Finkelstein, S. Antonelli, S. Armitage, and R. Stevens. Managing standards compliance. *IEEE Transactions on Software Engineering*, 25(6):836–851, 1999.
12. G. Engels, J. Hausmann, R. Heckel, and S. Sauer. Testing the consistency of dynamic UML diagrams. In *Proc. Sixth Int'l Conf. Integrated Design and Process Technology (IDPT 2002)*, June 2002.
13. G. Engels, R. Heckel, J. M. Küster, and L. Groenewegen. Consistency-preserving model evolution through transformations. In *Proc. 5th Int'l Conf. UML 2002*, volume 2460 of *LNCS*, pages 212–226. Springer-Verlag, 2002.
14. A. Finkelstein, D. M. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. In *European Software Engineering Conference*, volume 717 of *LNCS*, pages 84–99. Springer-Verlag, 1993.
15. Gentleware. Poseidon, http://www.gentleware.com/index.php?id=poseidon, March 18 2004.
16. V. Haarslev and R. Möller. RACER system description. In *Proc. Int'l Joint Conf. Automated Reasoning (IJCAR 2001)*, volume 2083 of *LNCS*, pages 701–706. Springer-Verlag, 2001.
17. I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for expressive description logics. In *Proc. 6th Int'l Conf. Logic for Programming and Automated Reasoning (LPAR'99)*, volume 1705 of *LNAI*, pages 161–180. Springer-Verlag, 1999.
18. J. Jürjens. Formal Semantics for Interacting UML subsystems. In *Proc. 5th Int'l Conf. Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002)*, pages 29–44. Kluwer Academic Publishers, 2002.

19. D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.
20. B. Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, 23(5):17–34, 1987. OOPSLA '87 Keynote Speech.
21. C. Lutz and U. Sattler. Mary likes all cats. In F. Baader and U. Sattler, editors, *Proc. Int'l Workshop on Description Logics*, volume 33 of *CEUR Electronic Workshop Proceedings*, pages 213–225, 2000.
22. T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In *Proc. First Int'l Conf. Graph Transformation*, volume 2505 of *LNCS*, pages 286–301. Springer-Verlag, 2002.
23. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
24. H. Nissen, M. Jeusfeld, M. Jarke, G. Zemanek, and H. Guber. Managing multiple requirements perspectives with metamodels. *IEEE Software*, pages 37–47, March 1996.
25. B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationship between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, 1994.
26. Object Management Group. UML 2.0 Object Constraint Language Final Adopted Specification. ptc/03-10-14, January 2005.
27. Object Management Group. Unified Modeling Language 2.0 Superstructure Draft Adopted Specification. ptc/03-08-02, January 2005.
28. Object Management Group. Unified Modeling Language Version 1.5. formal/2003-03-01, January 2005.
29. I. Porres. Model refactorings as rule-based update transformations. In *Proc. Int'l Conf. UML 2003*, volume 2863 of *LNCS*, pages 159–. Springer-Verlag, 2003.
30. G. Rasch and H. Wehrheim. Checking consistency in UML diagrams: Classes and state machines. In *Formal Methods for Open Object-based Distributed Systems*, volume 2884 of *LNCS*, pages 229–243. Springer-Verlag, 2003.
31. T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3), 2001.
32. M. Schrefl and M. Stumptner. Behavior consistent specialization of object life cycles. *ACM Trans. Software Engineering and Methodology*, 11(1):92–148, 2002.
33. S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003. Special Issue on Model-Driven Software Development.
34. W. Shen, Y. Lu, and W. L. Low. Extending the UML metamodel to support software refinement. In *Consistency Problems in UML-based software development II: Workshop Materials*, number 2003:06, 2003. Available at http://www.ipd.bth.se/consistencyUML/ Consistency_Problems_in_UML_II.pdf, October 2003.
35. J. Simmonds and M. C. Bastarrica. Description logics for consistency checking of architectural features in UML 2.0 models. DCC Technical Report TR/DCC-2005-1, Departamento de Ciencias de la Computacion, Santiago, Chile, 2005.
36. J. Simmonds, R. Van Der Straeten, and V. Jonckers. Maintaining consistency between uml models using description logic. *Série L'objet - logiciel, base de données, réseaux*, 10(2-3):231–244, 2004.
37. G. Spanoudakis and A. Zisman. *Inconsistency Management in software engineering: Survey and open research issues*, volume 1, pages 329–380. World Scientific Pub. Co., 2001.
38. P. Stevens and J. Tenzer. Modelling recursive calls with UML state diagrams. In M. Pezzé, editor, *Proc. Fundamental Approaches to Software Engineering (FASE 2003)*, volume 2621 of *LNCS*, pages 135–149. Springer-Verlag, 2003.
39. M. Stumptner and M. Schrefl. Behavior consistent inheritance in UML. In *Proc. 19th Int'l Conf. Conceptual Modeling (ER 2000)*, volume 1920 of *LNCS*, pages 527–542. Springer-Verlag, 2000.
40. G. Sunyé, D. Pollet, Y. LeTraon, and J.-M. Jézéquel. Refactoring UML models. In *Proc. Int'l Conf. UML 2001*, volume 2185 of *LNCS*, pages 134–138. Springer-Verlag, 2001.
41. W. van der Aalst. Inheritance of dynamic behaviour in UML. In *Proc. 2nd Int'l Workshop on Modelling of Objects, Components and Agents (MOCA'02)*, pages 105–120, August 2002.
42. R. Van Der Straeten. *Inconsistency Management in Model-driven Engineering. An Approach using Description Logics.* PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium, August 2005.
43. R. Van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between UML models. In *Proc. Int'l Conf. UML 2003*, volume 2863 of *LNCS*, pages 326–340. Springer-Verlag, 2003.
44. R. Van Der Straeten, J. Simmonds, and T. Mens. Detecting inconsistencies between UML models using description logic. In D. Calvanese, G. D. Giacomo, and E. Franconi, editors, *Description Logics*, volume 81 of *CEUR Workshop Proceedings*, 2003.
45. J. Whittle. Transformations and software modeling languages: Automating transformations in UML. In *Proc. 5th Int'l Conf. UML 2002*, volume 2460 of *LNCS*, pages 227–242. Springer-Verlag, 2002.