Supporting Model Refactorings through Behaviour Inheritance Consistencies

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

[1] System and Software Engineering Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
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
tom.mens@umh.ac.be

Abstract. This paper addresses the problem of consistency preservation in model-driven software development. Software models typically embody many different views that need to be kept consistent. In the context of *consistency within a model*, *behaviour inheritance consistencies* restrict the way the behaviour of a subclass can specialize the behaviour of a superclass. In the context of *model evolution*, *model refactorings* restructure a model while preserving its behavioural properties. It is still an open research question how to define behaviour preservation properties for model refactorings. We claim that behaviour inheritance consistencies correspond, in an evolution context, to the preservation of behavioural properties between model versions. To illustrate this claim, we implemented consistency rules and preservation behaviour rules in Racer, a reasoning engine for *description logics*. We show how the same logic rules can be used to detect behaviour inheritance inconsistencies in a model and to detect the preservation of call behaviour properties during model refactoring.

1 Introduction

During model-driven software development, models are built representing different views on a software system, or models can be evolved into a new version. Both situations may lead to inconsistencies. To address the first situation, so-called *behaviour inheritance consistencies* can be used to restrict the way the behaviour of a subclass should specialize the behaviour of a superclass in a class hierarchy (cf. Liskov's well-known substitutability principle). To address the second situation, so-called *model refactorings* can be used, because they have the important benefit that they restructure a model while preserving certain behavioural properties.

The aim of this paper is to explore the precise relation between behaviour inheritance consistencies and model refactorings. We claim that behaviour inheritance consistencies within a single model version correspond, in an evolution

context, to the preservation of certain behavioural properties between model versions. In the remainder of the paper, we validate our claim in four successive steps.

First, we investigate and formalise behaviour as defined in UML 2.0 state machines and sequence diagrams (Section 3)[13]. Second, we express different kinds of behaviour inheritance consistencies in the context of UML 2.0 state machines and sequence diagrams, and found in literature [19] with our formalism (Section 4). Third, we investigate and formalise different kinds of behaviour preservation properties for model refactoring (Section 5). Finally, we show that these notions of consistency and preservation are closely related (Section 6). To this extent, we implement a set of logic rules in a reasoning engine based on the formalism of Description Logic (DL) [2], and show that the logic queries that are used to detect behavioural inconsistencies in a model, can also be applied to guarantee the preservation of behavioural properties between different model versions.

2 Motivating Example

The motivating example used throughout this paper, is based on the design of an automatic teller machine (ATM), originally developed by Russell Bjork for a computer science course at Gordon University. A possible usage scenario of an instance of *ATM* is shown in the sequence diagram in Figure 1. This sequence diagram shows part of an interaction between instances of the classes *ATM*, *Session*, *CardReader* and *CashDispenser*, when a user decides to make a withdrawal. The messages sent in the diagram verify if there is enough cash in the ATM. If so, the amount of cash is dispensed and the card is returned to the user.

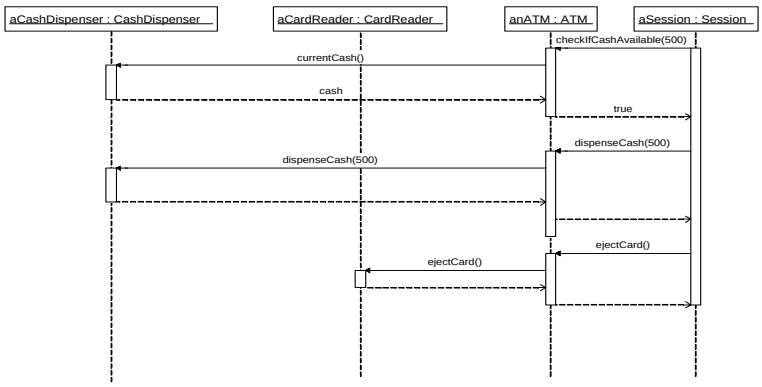

Fig. 1. Sequence diagram for withdrawal scenario on an *ATM*

The behaviour of the *PrintingATM* class, which is a subclass of *ATM* that has extra printing functionality, is represented by the state machine in Figure 2. When a customer wants to withdraw money from his account, he has to insert a bank card and enter the associated PIN number. If the PIN is not valid, the card is returned to the user. If a valid PIN has been entered, the *ATM* prompts the user to enter the amount to withdraw from his account. If the amount is less than *100*, the user is asked to re-enter an amount. In the other case, the *ATM* checks that the client's account has sufficient funds. If so, the *ATM* proceeds to check if it can dispense this amount. Once these checks have been passed, the *ATM* dispenses the money and at the same time, the *PrintingATM* class, unlike its parent, the *ATM* class, prints a receipt. Finally, the card is ejected.

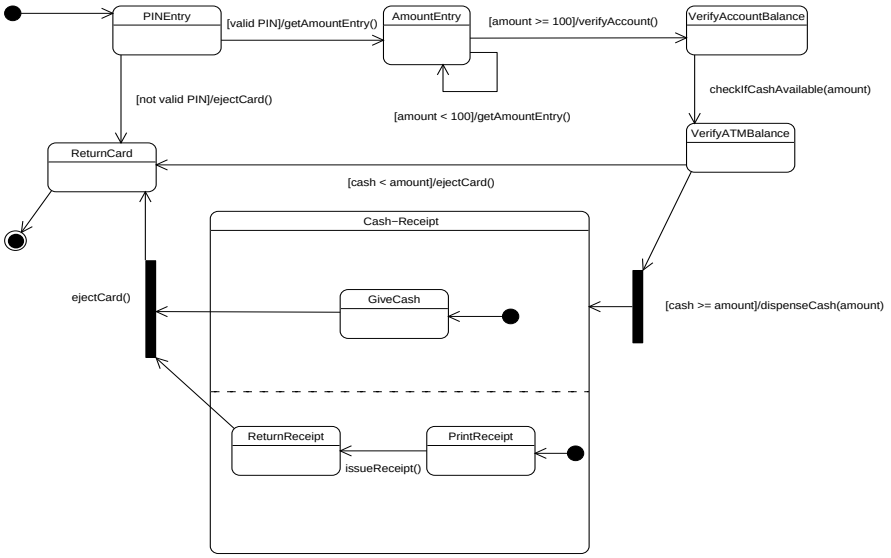

Fig. 2. UML protocol state machine for *PrintingATM* class

Consider the consistency relationship that an instance of *PrintingATM* must be usable in each situation where an instance of *ATM* is required (according to the substitutability principle). To guarantee this consistency relationship, *each sequence of the* ATM *sequence diagram of Figure 1 should be contained in the set of sequences of the* PrintingATM *state diagram of Figure 2.* In our case, *ATM* and *PrintingATM* do not obey this consistency rule, because an instance of *PrintingATM* will, after dispensing the cash, *always* print a receipt. It is not possible to skip this printing and immediately eject the card, which is the original behaviour of the *ATM* class.

Consider now an evolution of the *ATM* class obtained by extracting functionality contained in the *checkIfCashAvailable* method into a separate method. This

model refactoring is comparable to the source code *Extract Method* refactoring [8], but at the design level we do not have source code at our disposal.

In our example, a new method *validate* is created, which takes two arguments, *amount* indicating the amount to withdraw and *cash*, the amount of money available in the *ATM*. This method checks if there is sufficient cash available. The sequence diagram in Figure 3 is the refactored version of the sequence diagram in Figure 1 where in the body of the method *checkIfCashAvailable*, the extracted method *validate* is called with arguments *500* and *cash*, which is the return value of the invoked *currentCash* method.

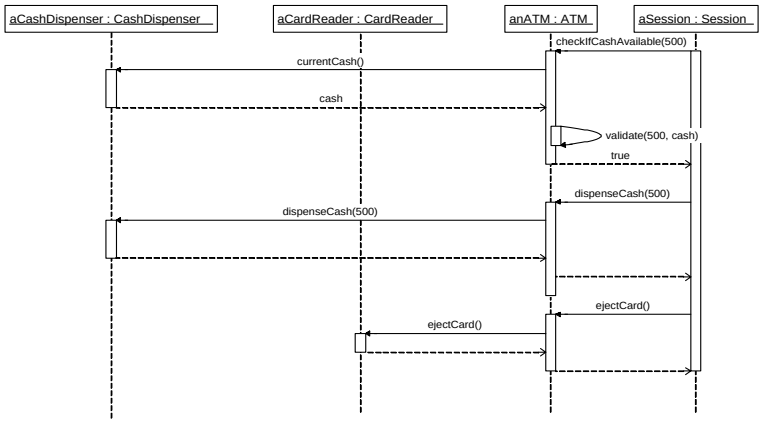

Fig. 3. Refactored sequence diagram for refactored *ATM* class

The behaviour specified in Figure 1 is not altered by this model refactoring. However, the call sequence in the method body of *checkIfCashAvailable* has been extended with a call to the new method *validate*.

3 Behaviour in UML 2.0

In this section, basic definitions of behaviour as defined in the UML 2.0 Superstructure and Infrastructure Specification [13] are given. These definitions enable a precise characterisation of behaviour inheritance consistencies and behaviour preservation in sections 4 and 5.

In UML 2.0, *Behaviour* is defined as a specification of how its context classifier changes state over time. *Behaviour* is an abstract metaclass and as such, the specification of a behaviour can take a number of forms, as described in its subclasses. A variety of specification mechanisms are provided by UML, such as *Statemachine*, *Activity*, *Usecase* and *Interaction*. To keep our experiments manageable, we deliberately confine ourselves to *Statemachine* and *Interaction* as specifications of behaviour.

3.1 Interaction

As described in chapter 14 of [13], " *Interactions are used in a number of different situations. They are used to get a better grip of an interaction situation for an individual designer or for a group that need to achieve a common understanding of the situation. Interactions are also used during the more detailed design phase where the precise inter-process communication must be set up according to formal protocols. When testing is performed, the traces of the system can be described as interactions and compared with those of the earlier phases.*"

The semantics of an *Interaction* is given by a pair of sets of traces [13] representing valid traces and invalid traces, respectively. Only the valid traces are described in [13].

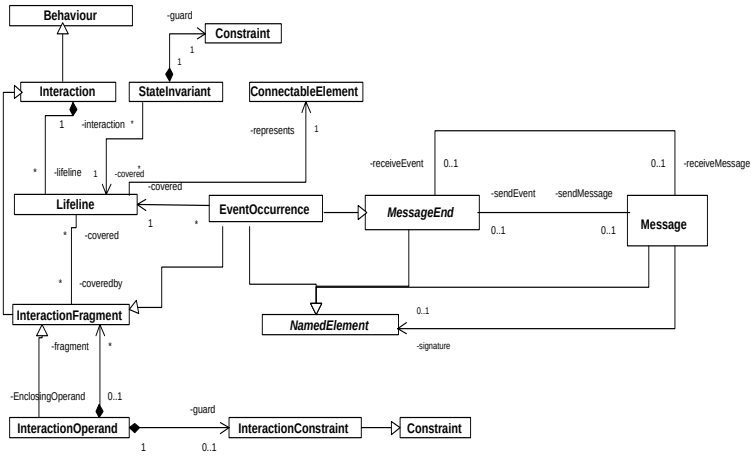

Fig. 4. UML meta model fragment for interactions

Figure 4 shows the relevant fragments of the UML meta model dealing with interactions. An *Interaction* consists of some *Lifeline*s which are covered by *EventOccurrence*s. *EventOccurrence*s are *MessageEnd*s representing either the receiving event of a *Message* or the sending event of a *Message*. A *Message* is a *NamedElement* that defines one specific kind of communication, represented by another *NamedElement*, e.g., an *Operation* in the case of an operation invocation. A *Lifeline* represents a *ConnectableElement*. A *ConnectableElement* represents a set of instances owned by a containing classifier instance. The UML meta class *InstanceSpecification* represents an instance in a modeled system. However, this meta class is not related to the *ConnectableElement* meta class in the UML 2.0 meta model. In this paper, a *Lifeline* is assumed to represent an *InstanceSpecification*. On a *Lifeline*, *StateInvariant*s can be specified. An *InteractionFragment* is a piece of an interaction, which is an interaction in its own right. An *InteractionOperand* is an *InteractionFragment* with an optional guard

expression. Only *InteractionOperand*s with a guard that evaluates to true at this point in the interaction will be considered for the production of the traces of the enclosing *Interaction*.

The guards are *InteractionConstraints*, which are *Constraints*. In this paper, we typically consider constraints that represent pre- and postconditions.

Notation 1 *The* **set of all preconditions of an operation** *op* **of a class** *c, i.e., the set of conditions specifying the state of the system when the operation op is invoked, is denoted by* $Pre_{op,c}$. *The* **set of all preconditions** *of all operations of a class c is denoted by* Pre_c.
The **set of all postconditions of an operation** *op* **of a class** *c, i.e., the set of conditions specifying the state of the system when the operation op is completed, is denoted by* $Post_{op,c}$. *The* **set of all postconditions** *of all operations of a class c, is denoted by* $Post_c$.

Depending on its purpose, an *Interaction* can be displayed with different types of 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 e. The elements of cons are instances of the meta classes* StateInvariant *and* InteractionConstraint.

Note that this definition of SD trace indicates a subset of what is meant by the term "trace" in the UML 2.0 [13] chapter 14 on *Interactions*. 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.*

For defining behaviour inheritance consistencies, 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 considered. 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 an operation.*

Example 1. A receiving SD trace of the instance *anATM* of class *ATM* in the sequence diagram Δ of Figure 1 is $< e_1, e_2, e_3 >$, where e_1 represents the receipt (by *anATM*) of the message $checkIfCashAvailable$, e_2 represents the receipt of the message $dispenseCash$ and e_3 represents the receipt of the message $ejectCard$.

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 $E_{\Delta,ATM} = \{e_1, e_2, e_3\}$

3.2 Statemachine

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 inheritance 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.[3]

A protocol state machine (PSM) can be defined as follows (based on the definition in [18] and in [19]) [4]:

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 is a triple (op, g, h) where op is the operation, $g \subseteq Pre_{op,c}$ specifies the precondition of the transition (which is part of the precondition of the operation op), and $h \subseteq Post_{op,c}$ specifies the postcondition of the transition (which is part of the postcondition of the operation op). ρ_c denotes the initial state and Λ_c denotes the set of final states of the state machine.*

Example 3. The state machine specified in Figure 2 is a protocol state machine which has six states of which one is a concurrent state. This concurrent state is treated (as specified by the definition of a PSM) as a set of simple states. The concurrent state is entered in the set $\{GiveCash, PrintReceipt\}$, if *issueReceipt()* is called, the new state configuration is $\{GiveCash, ReturnReceipt\}$. This brings us to the next definition.

[3] UML provides special kinds of states and transitions, such as history states, stubbed transitions, junction and choice transitions. These concepts are not considered in this paper.

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

The 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 *life cycle state configuration* of the object.

Definition 5. *A* **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.

Definition 6. *A* **PSM trace** γ *of an instance o of a class c in a PSM* Π_c *is a sequence of life cycle state configurations* $< \sigma_1, \ldots, \sigma_n >$ *such that* $\sigma_1 = \{\rho_o\}$ *and, for* $i \in \{1 \ldots n-1\}$, $\sigma_{i+1} = \sigma_i$ *or* $\exists(\sigma_i, \tau_i, \sigma_{i+1}) \in T_c$.

Definition 7. *A* **call sequence** μ *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 8. *A* **call sequence** $\mu =< \tau_k, \ldots, \tau_n >$ **is valid** *on a state configuration* σ_k *of instance o, if there is a PSM trace* $\gamma =< \sigma_1 \ldots \sigma_k \ldots \sigma_{n+1} >$ *of o where for* $i \in \{k \ldots n\}$, $(\sigma_i, \tau_i, \sigma_{i+1}) \in T_c$.

Example 4. $< issueReceipt, ejectCard >$ is a valid call sequence on the state configuration $\{GiveCash, PrintReceipt\}$ of Figure 2.

4 Behaviour Inheritance Consistencies

The basic definitions given in the previous section enable the precise definition of behaviour inheritance consistencies in and between protocol state machines and sequence diagrams. It is expected that the behaviour specification of classes described by sequence diagrams and/or state machines is consistent with the behaviour specification of their superclasses. This kind of consistency is not defined in most object-oriented modeling languages and also not in UML 2.0.

In Ebert and Engels [5] two kinds of consistencies between state machines are defined, *observation* and *invocation* inheritance consistency.

Observation inheritance consistency means that *each sequence of calls which is observable with respect to a subclass must result (under projection of the methods known) in an observable sequence of its corresponding superclass. If a subclass reacts to the invocation of an operation op, where op is also known to the superclass, this reaction must also be reflected in the superclass behaviour specification.* 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 9. *The* **restriction** $\mu_{\mathbf{L}}$ **of a sequence** $\mu =< \tau_1, ..., \tau_n >$ **to a set** L *is the sequence obtained from* μ *by removing all* $\tau_i \notin L$.

Definition 10. *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 m

g is a set of preconditions of op met by the set of constraints cons

h is a set of postconditions of op met by the set of constraints cons

Definition 11. Observation inheritance consistency. *Given a class c and a subclass c' of c and instances o of c and o' of c'.*

A PSM $\Pi_{c'} = (S', T', L', \rho', \Lambda')$ is observation 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 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.

A SD Δ' is observation 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 $\mu_o =< \tau_1, \ldots, \tau_n >$, containing only labels τ_i for which $\tau_i = label_c(e_i)$

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 not always the case, especially in early phases of the software development life cycle.

Example 5. Consider the protocol state machine $\Pi_{PrintingATM}$ of Figure 2, and the protocol state machine Π_{ATM} that is the same as $\Pi_{PrintingATM}$ except for the absence of substates *PrintReceipt* and *ReturnReceipt*. $\Pi_{PrintingATM}$ is observation consistent with Π_{ATM}. After hiding the message call $\{issueReceipt\}$, the behaviour of the subclass *PrintingATM* is identical to the behaviour of the superclass *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 12. Invocation inheritance consistency. *Given a class c and a subclass c' of c and instances o of c and o' of c'.*

A PSM $\Pi_{c'} = (S', T', L', \rho', \Lambda')$ is invocation 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 their respective PSM traces γ and γ' it holds that $\gamma = \gamma'_S$.

A SD Δ' is invocation 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 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 $\mu_{o'} =< \tau_1 \ldots \tau_n >$ such that, for each $i \in \{1 \ldots n\}$, $\tau_i = label_c(e_i)$.

Remark 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 implies completeness of the models involved.

Example 6. The behaviour of the sequence diagram Δ of Figure 1 is **not** invocation consistent with the PSM $\Pi_{PrintingATM}$ of Figure 2 with respect to class *ATM*. Indeed, the SD trace $< e_1, e_2, e_3 >$ of Example 1 does not correspond to a valid call sequence <*checkIfCashAvailable, dispenseCash, ejectCard*> in the PSM $\Pi_{PrintingATM}$, that always requires the message invocation *issueReceipt* between *dispenseCash* and *ejectCard*.

5 Behaviour Preservation

Model refactorings restructure a model while preserving its behavioural properties. On the source code level, refactorings of an object-oriented program are restructurings that preserve program behaviour. Despite the available tool support for source-code refactorings and also model refactorings, it is still an open research question how to define behaviour preserving properties for (model) refactorings. In [12], *call preservation* was defined for source code refactorings. This preservation property can be redefined for model refactorings as follows:

Definition 13. *A model refactoring is* **call preserving** *if each operation still invokes at least the same operations after the model refactoring as it did before the model refactoring.*

However, there are different variants of call preservation. Assume that we have a model M_1 and a refactored version M_2 of this model. The most restricted form of call preservation specifies that, if an operation m can invoke an operation n in M_1, the operation m can still invoke operation n in M_2, following exactly the same chain of messages as in M_1. A less restricted form of call preservation specifies that this message chain can be completely arbitrary in M_2, as long as n remains reachable from m. Another notion of call preservation we can define formally, is *observation call preservation*. In this case, every call sequence observable with respect to a class in M_2 must result in an observable call sequence of its corresponding class in M_1.

Definition 14. Observation call preservation. *Let c be a class, c' a refactored 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 $\mu_o =< \tau_1, \ldots, \tau_n >$, containing only labels τ_i for which $\tau_i = label_c(e_i)$.

Remark that Definition 14 is almost identical to Definition 11. The main difference is that the words *observation consistent* are replaced by *observation*

call preserving. Also, c' does not represent a subclass of c anymore, but a new version of c in the refactored model.

Example 7. The behaviour specified by the refactored sequence diagram Δ' of Figure 3 is observation call preserving with the original sequence diagram Δ of Figure 1. The model refactoring presented here, abstracts existing behaviour into a new operation and as such, boils down to the addition of a message *validate* in Δ'. However, this does not affect the behaviour. All traces of the ATM class in Δ' are also traces in Δ if we exclude the message *validate*.

Another kind of call preservation, *invocation call preservation* guarantees that each call sequence invocable on the original version of a class, must also be invocable on the corresponding class in the refactored model. The definition of invocation call preserving is identical to Definition 12 by substituting *invocation call preserving* for *invocation consistency*.

Referring to Example 7 above, the behaviour specified by the refactored sequence diagram Δ' is *not* invocation call preserving with sequence diagram Δ. However, if we would ignore the message *validate* (because it can be considered as an auxiliary method that is of no interest to the user), the property would hold. This lead us to refine the two above notions of behaviour preservation into weaker, more specialised, variants, where the traces and call sequences can be restricted to a specific set of messages of interest to the user. As an example, we give the formal definition of *weak invocation call preservation* below:

Definition 15. Weak invocation call preservation. *Assume a class c, a refactored version c' of c and instances o of c and o' of c', and user-defined sets of labels $L_{user} \subseteq L'$ and $E_{user} \subseteq E_{\Delta',c'}$.*

A PSM $\Pi_{c'} = (S', T', L', \rho', \Lambda')$ is weak 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 Π'' where $\Pi'' = (S', T'', L_{user}, \rho', \Lambda')$, where T'' is the restriction of transition relations of T' to the ones with only labels contained in L_{user}, and for their respective PSM traces γ and γ'' it holds that $\gamma = \gamma''_S$.

A SD Δ' is weak 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' restrictred to E_{user} for an instance o' of class c'.

A PSM $\Pi_{c'} = (S', T', L', \rho', \Lambda')$ is weak 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 $\mu_{o'} =< \tau_1, \ldots, \tau_n >$ such that, for each $i \in \{1 \ldots n\}$, $\tau_i = label'_c(e_i)$, where $label'_c(e_i)$ is similar to $label_c(e_i)$, except that $label'_c(e_i)$ maps $E_{\Delta,c}$ on L_{user}.

Example 8. If the message *validate* is not considered in the sequence diagram Δ' in Figure 3, the sets $E_{\Delta,ATM}$, where Δ is the sequence diagram in Figure 1, and $E_{user} = \{< e_1, e_2, e_3 >\}$, where e_i is defined as in *Example 1*, are equal. This trivially implies invocation call preservation.

6 Tool Support

Consistency maintenance requires a decidable formalism to detect inconsistencies and also a generic framework to facilitate the addition, removal and modification of consistency specifications. In earlier work [22], we already proposed and used description logics (DL) [2] to detect and resolve inconsistencies; also the translation of the UML meta model and user defined models were described. A crucial property of DL is that it allows us to guarantee that the consistency rules that we can specify are *decidable*.

Given the similarity between the definitions for behaviour consistency and behaviour preservation, identified in the previous section, it is possible to use the same formalism in the context of checking the preservation of behavioural properties between a model and its refactored version.

To achieve this, we set up the following tool chain. UML design models are expressed in a UML CASE tool (Poseidon [9]) which exports UML models in XMI format. Using an XML parser (Saxon [17]), the models are translated into description logic statements. These are asserted into a knowledge base maintained by a description logic reasoning engine. We chose Racer [10] for this purpose as it is a state-of-the-art logic reasoning engine for DL.

To check behaviour inheritance consistency within a model and behaviour preservation between a model and its refactored version, rules are specified. These rules can be immediately translated into our logic framework using the query language of Racer (nRQL) [11]. For example, to check invocation consistency or preservation between the behaviour specified by a SD Δ with respect to a class c, and a PSM $\Pi_{c'} = (S', T', L', \rho', \Lambda')$ with c' subclass or refactored version of class c, the following rule (written in pseudo-code) needs to be checked:
consistent($\Pi_{c'}$, Δ, c) $\leftarrow$
$query_1(events, c, \Delta)$,
for each $e \in events$
$query_2(e, op, startstate, target, \Pi_{c'})$
if $op =$ NIL then "consistency error at state" $startstate$

Consider, first of all, the $query_1$ that generates the SD traces for the sequence diagram Δ:

```
(retrieve (?events ?c ?seqinteraction)
 (and
  (?objectid ?c instance-of) ;retrieve the instances of class c
  (?objectid ?lifeline (inv represents));using the instance ?objectid the
                                        ;representing lifeline ?lifeline is retrieved
  (?seqinteraction ?lifeline ownedlifeline);only lifelines from the involved sequence diagram
  (?lifeline ?events eventoccurrences) ;the event occurrences ?events occurring on ?lifeline
  (?events (some receivemessage message)))) ;only receiving event occurrences
```

Using these events, the state machine $\Pi_{c'}$ is traversed. The next query $query_2$ checks if the operation belonging to the event $?e$ is also the operation referred to by the corresponding transition in the statemachine.

```
(retrieve (?e ?op ?startstate ?targetstate ?statemach)
  (and
   (?e ?msg receivemessage)   ;retrieve the ?msg related to event ?e
```

```
(?msg ?op signature)       ;retrieve the invoked operation ?op
(?transition ?op referred) ;?transition referring to ?op
(?statemach ?transition owningtransitions) ;?transition owned by statemachine
(?transition ?startstate source) ;?startstate must be the startstate of ?transition
(?transition ?targetstate target))) ;the target state of the ?transitions
```

We applied the above mentioned rules to check the consistency of the sequence diagram in Figure 1 and the protocol state machine in Figure 2. Finally, we applied the rules to check observation and invocation consistencies and preservation, to the examples of Section 2. These experiments let us conclude that our framework for inconsistencies as presented in [22] is also suitable to check behaviour preservation for model refactorings.

7 Discussion and Related Work

Many notions of behaviour inheritance consistency can be found in literature. Compared to Schrefl *et al.* [19], our 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.* [5] define observable and invocation consistency using homomorphisms on state diagrams. Criteria for inheritance of object life cycles based on Petri nets are discussed in [19], [16] and [21]. Approaches based on CSP are discussed in [6] and [15]. 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.

Research on model refactoring is less abundant. A set of basic UML refactorings is provided in [20] 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 [14] 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 [12]). 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 [7] 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.

The approach explained above also provides a promising alternative to traditional model checking approaches [4] that need to recheck the entire model whenever small and local changes have been made to a model.

8 Conclusion

In this paper, we have defined different kinds of behaviour inheritance consistencies between UML 2.0 state machines and sequence diagrams. We also have shown that those consistency specifications correspond to behaviour preservation properties between a UML model and its refactored version. Based on those consistency definitions, definitions of behaviour preservation are given. We also showed that our tool chain for detecting behavioural inconsistencies can be used to check the preservation of behaviour between different model versions.

We only carried out experiments on small examples, experiments on larger models must be done. As future work, we want to explore if other consistency specifications such as the ones defined in [22] correspond to the preservation of certain behavioural properties. We also need to extend our ideas to deal with consistency maintenance and behaviour preservation between different levels of abstraction. This will allow us to provide better formal support for the model-driven architecture process.

References

1. D. Astels. Refactoring with UML. In *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, pages 67–70, Alghero, Sardinia, Italy, 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, Alghero, Sardinia, Italy, 2002.
4. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.
5. J. Ebert and G. Engels. Specialization of object life cycle definitions. Fachbericht Informatik 19/95, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, 1995.
6. 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. Pasadena, CA, USA.
7. G. Engels, R. Heckel, J. M. Küster, and L. Groenewegen. Consistency-preserving model evolution through transformations. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings*, volume 2460 of *LNCS*, pages 212–226. Springer, 2002.

8. M. Fowler. *Refactoring: Improving the Design of Existing Programs.* Addison-Wesley, 1999.
9. Gentleware. Poseidon, http://www.gentleware.com/products/poseidonpe.php3, March 18 2004.
10. V. Haarslev and R. Möller. RACER system description. In *Int'l Joint Conf. Automated Reasoning (IJCAR 2001)*, 2001.
11. V. Haarslev, R. Möller, R. Van Der Straeten, and M. Wessel. Extended query facilities for RACER and an application to software-engineering problems. In V. Haarslev and R. Möller, editors, *Proc. Int'l Workshop on Description Logic*, 2004.
12. T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In *Proc. First Int'l Conf. Graph Transformation*, pages 286–301. Springer-Verlag, 2002.
13. Object Management Group. Unified Modeling Language 2.0 Superstructure Draft Adopted Specification. ptc/03-08-02, February 2004.
14. I. Porres. Model refactorings as rule-based update transformations. In P. Stevens, J. Whittle, and G. Booch, editors, *Proc. Int'l Conf. UML 2003*, volume 2863 of *LNCS*, pages 159–. Springer-Verlag, 2003.
15. 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.
16. M. Schrefl and M. Stumptner. Behavior consistent specialization of object life cycles. *ACM Trans. Software Engineering and Methodology*, 11(1):92–148, January 2002.
17. Sourceforge. Saxon, http://saxon.sourceforge.net/, March 18 2004.
18. 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.
19. M. Stumptner and M. Schrefl. Behavior consistent inheritance in UML. In A. H. F. L. et al. editor, *Proc. 19th Int'l Conf. Conceptual Modeling (ER 2000)*, volume 1920 of *LNCS*, pages 527–542. Springer-Verlag, 2000.
20. 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.
21. W. van der Aalst. Inheritance of dynamic behaviour in UML. In D. Moldt, editor, *Proc. of the Second International Workshop on Modelling of Objects, Components and Agents (MOCA'02)*, pages 105–120, August 2002.
22. R. Van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between UML models. In P. Stevens, J. Whittle, and G. Booch, editors, *Proc. Int'l Conf. UML 2003*, volume 2863 of *LNCS*, pages 326–340. Springer-Verlag, 2003.