View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Springer - Publisher Connector

Softw Syst Model (2013) 12:415–434
DOI 10.1007/s10270-011-0215-y

REGULAR PAPER

Specification of invariability in OCL

Specifying invariable system parts and views

Piotr Kosiuczenko

Received: 21 May 2008 / Revised: 22 June 2011 / Accepted: 2 September 2011 / Published online: 5 October 2011
© The Author(s) 2011. This article is published with open access at Springerlink.com

Abstract The Object Constraint Language (OCL) is a high-level, object-oriented language for contractual system specifications. Despite its expressivity, OCL does not provide primitives for a compact specification of invariability. In this paper, problems with invariability specification are listed and some weaknesses of existing solutions are pointed out. The question of invariability specification is addressed and a simple but expressive extension of OCL is proposed. It allows a view-oriented specification of invariability constraints, whereby we restrict the notion of view to reducts based on order-sorted algebras. The semantics of this extension is defined in terms of standard OCL.

Keywords OCL · UML · Invariability · Frame problem · Views

1 Introduction

Contracts are the prevailing way of specifying systems from the caller point of view (see [33,34]). Object Constraint Language (OCL) [40,50] is a high-level language for writing contractual specifications of object-oriented systems. It is associated with the Unified Modeling Language (UML) [42,45] and supported by a variety of tools, e.g. [17,23] (see [5] for a tool overview). Specification of invariable system parts is a common problem in case of complex systems. OCL allows for the explicit comparison of object attributes before and after operation execution. An operation execution usually changes only a small part of a system and consequently most of the system remains unchanged. In case of large systems, it is not feasible to specify what happens with all attributes and associations. Unfortunately, OCL does not provide primitives to specify what can and what must not be changed when an operation is executed.

The problem of invariability specification is not restricted to contractual languages (see [8] for an overview). In general there exist three approaches to this problem: frame formulas, implicit specification and invariability clauses. The frame axioms are used in artificial intelligence (cf. [36,47]). The idea is to specify modification of attributes using axiom schemata. It requires explicit listing of all attributes which remain unchanged and results in large formulas. The second approach to invariability dates back to Hoare logic [28]. In this logic all variables which are not mentioned in a Hoare triple are assumed to be unchanged. However, it does not work well for contractual specifications because an operation execution can have very complex side-effects. The Java Modeling Language (JML, see [16,37]) and Spec# [10] use explicit invariability clauses and allow for a compact specification of invariable system parts. Invariability constraints are checked at compile-time. Thus, it is not possible to specify invariability requirements which cannot be checked statically.

A method for invariability specification has to address the following issues:

- complexity: huge formulas
- fragility: the resulting formulas must be modified after every system change
- over-specification: the specification exposes details which should be hidden

OCL can be used directly to specify what cannot be changed, but such specifications are usually very extensive, fragile, hard to understand and modify. The fact that an operation is side-effect-free is expressed in UML on the

Communicated by Prof. Martin Gogolla.

P. Kosiuczenko (✉)
Institute of Information Systems, WAT, Warsaw, Poland

meta-level by the attribute *isQuery*. However, its meaning cannot be specified in a compact way using OCL-constraints, but requires extensive formulas. Similarly, the problem of view-oriented specification with OCL has not been properly investigated. A mechanism allowing to hide specification details is lacking. What we need is a compact way of localizing change, with a simple and monotone semantics. It should be applicable to different system views.

In the paper [30], we proposed an extension of OCL for a compact and precise invariability specification. The idea is to use pairs consisting of a set of objects defined by an arbitrary OCL term and of a modifiable attribute. Thus, its expressive power matches the expressive power of OCL. It is a new concept, not just a simple extension of modifiable JML clause. This paper is a journal version of [30]. The main contribution of this paper is a new and more detailed semantics of invariability clauses and its investigation in the context of the notion of reduct as it is used in order-sorted algebras and database theory. This semantics is symmetric in the sense that the relaxation of one association-end implies the relaxation of the opposite end. It fits better to the idea that an association consists of tuples and that it owns its ends as specified in UML 2 (cf. [42]).

We define a formal semantics of invariability clauses and demonstrate that reducts preserve validity of specifications. We discuss so-called semantic variation points [42], i.e. different options in the semantics definition. We propose also a method for deriving invariability clauses from post-conditions. This method can be seen as a formalization of the implicit invariability assumption, or one of its possible variants. Derived clauses are a relatively good approximation of user intentions and as such can be used to assess soundness and completeness of user-specified clauses.

Abstraction and information hiding play crucial roles in software engineering. Complex systems cannot be designed without the use of abstraction. Similarly, information hiding facilitates software development [43]. A specification, in particular the specification of invariable system parts, should not disclose implementation details. In case of Spec#, invariability clauses do not disclose the structure of internal layers [10]. The possibility to specify and comprehend a system from different points of view is essential (cf., e.g. [15,38]). Invariability specification in OCL should also not force the specifier to disclose internal system details. In the paper [30], we proposed a notion of view based on UML 1.5 [41]. In the meantime, the standard has been upgraded. In this paper, we use the UML 2 metamodel [42] and OCL 2 [40]. The proposed notion is based on the concept of reduct and allows us to restrict specification of invariability to views. It allows a specifier to abstract away from the irrelevant system details. It can be treated as a semantic counterpart of package signatures and APIs. In general there are various notions of view for object-oriented systems (cf., e.g. [31]), but the concept of reduct is most fundamental and has regular properties. We demonstrate that it is possible to define views using OCL terms. In UML and OCL, the notion of query is defined in plain English. It turns out that in our framework it is easy to specify formally that an operation is a query, i.e. a side-effect free operation. We demonstrate the applicability of the proposed extension using a number of examples and explain how it addresses problems with the specification of invariability.

This paper is organized as follows: In Sect. 2, we use a simple example to explain problems with invariability specification; we also sketch a solution. In Sect. 3, we relate our extension to the UML metamodel and show how to define views. In Sect. 4, we present the formal syntax of the proposed extension. In Sect. 5, we present an OCL based semantics of the proposed extension, define a formal semantics and propose some semantic variation points; we show also how to derive invariability clauses from post-conditions. In Sect. 6, we perform a small case study and demonstrate the applicability of our approach. In Sect. 7, we discuss the related work. Section 8 concludes this paper.

2 Specification of invariability

In this section, we consider a simple example of a bank account, illustrate problems with invariability specification and explain our solution. In the first subsection, we demonstrate problems with invariability specification, in particular with the implicit invariability assumption. In the second subsection, we introduce informally invariability clauses and discuss their basic properties. In the third subsection, we show how to deal with model modification. In the fourth subsection, we show how to deal with operation's side-effects. Operations on lists are not easy to specify in OCL due to their side effects. In the fifth subsection, we show how to deal with this case.

2.1 Problems with invariability specification

Consider the class diagram in Fig. 1. It shows a bank account class and a credit card class. We can specify the operation *credit* in OCL in the following way:

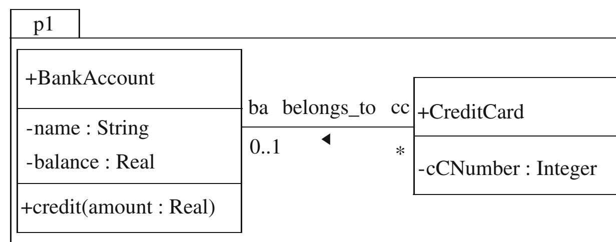

Fig. 1 Bank account model

context $p1::BankAccount::credit(amount : Real)$
post : $self.balance = self.balance@pre + amount$

This specification does not mention what happens to the attribute *name*, to the association-ends *cc* and *ba*, nor to the attribute *cCNumber*. Therefore, we have to extend the post-condition with the following frame formula:

$$\begin{aligned}&and\ self.name\ =\ self.name@pre\\&and\ self.cc\ =\ self.cc@pre\\&and\ self.cc.ba\ =\ self.cc@pre.ba@pre\\&and\ self.cc.cCNumber\ =\ self.cc@pre.cCNumber@pre\end{aligned}$$

Moreover, to make this specification complete, we need a formula guaranteeing that all objects of the class *BankAccount* different from *self* are not influenced by the execution, and a similar formula for the class *CreditCard*. This requires a separate equation for every attribute and association-end. Clearly, in case of larger systems, writing all such axioms results in huge formulas. Such formulas are fragile in respect to modifications. It is easy to omit something or to add an erroneous constraint.

A possible solution is to use an implicit invariability assumption. In the simplistic case, it says that all that is not specified to change does not change (see, for example, [15,32,33]). It allows one to write simple specifications. This approach is appealing, since it does not put any extra burden on the specifier. However, it is not always clear what this assumption means, especially when a high level specification language, such as OCL, is used.

For example, equation $self.cc.cCNumber = self.cc@pre.cCNumber@pre + 1$ does not say what may be changed. There are three possibilities: either the association-end *self.cc*, or the attribute *cCNumber*, or both. It is only clear that at least one of those properties is changed. It seems natural to assume that $self.cc = self.cc@pre$, but this constraint does not follow from the post-condition. It is rather our guess that the association should remain unchanged. Consider the following tautology:

$$\begin{aligned}&CreditCard.allInstances()\text{->}forAll(o \mid not\ o.oclIsNew()\ implies\\&o.cCNumber = o.cCNumber@pre\ or\ not(o.cCNumber\\&\qquad = o.cCNumber@pre))\end{aligned}$$

The implicit invariability assumption would allow arbitrary change of the attribute *cCNumber*, despite the fact that this formula is a tautology. Thus, it can hardly be used in combination with formal reasoning, since proving formulas requires application of tautological formulas (cf., e.g. [14]) and in logic, tautologically equivalent formulas are semantically equivalent.

There are also other problems. Changes to the underlying model require the specifier to rewrite the invariability specification, but in case of large formulas it is time consuming and error-prone. In case of subclassing, attributes of subclasses are usually not meant to be constrained by invariants, pre- and post-conditions concerning their superclasses. However, if they do not occur in those constraints, then they are assumed to be invariable. Another problem is the specification of side-effects, i.e. effects which are not meant to be visible to a client or concern objects different from actual parameters.

2.2 Solution in the simple case

In this subsection, we outline a solution for the case of single classes and packages. In the bank account example (see Fig. 1), we need to specify what can and what must not be changed. We restrict the specifications to packages and to sets of model elements in general; we call those sets views (see Sect. 3). We use the optional **in**-keyword to indicate the package, or view in general, to which the specification is restricted. The **modifies** clause specifies a list of variable object-properties such as attributes and association-ends. The variable system part can be specified either in respect to a specific view described by the in-part, or in respect to the whole class model if **in** does not occur.

We specify explicitly what changes in the package *p1*. The following formula puts the specification into perspective. More precisely, the specification is defined relatively to attributes and association-ends of classes contained in package *p1*. The keywords are indicated by the bold characters:

context $p1::BankAccount::credit(amount : Real)$
post : $self.balance = self.balance@pre + amount$
in $p1$ **modifies** : $self::balance$

We use the OCL primitive :: to indicate that the attribute *balance* of the implicit parameter *self* can be modified. In our case, this primitive has two arguments: a term defining an object or a set of objects, which is meant to be the scope of change, and an association-end or an attribute possessed by those objects. The clause **in** *p1* **modifies** : *self*::*balance* restricts the invariability specification to the view defined by the package *p1.credit* can change in this view only the attribute *balance* of the actual implicit parameter. This specification does not say anything about any other package.

Figure 2 presents two object diagrams modelling states of the bank account system. They show how states may change when *credit* is executed. The first one (see Fig. 2, part (a)) consists of two objects, *ba1* and *ba2*, of class *BankAccount* and of the associated *CreditCard*-objects. The variable *self* points to the first object. The contract for *credit* allows changing of the corresponding attribute *balance*; this is indicated by italic font. All other attributes and associations must remain unchanged. When this operation is executed, *ba2* and the corresponding credit card *c2* are deleted and also a new bank

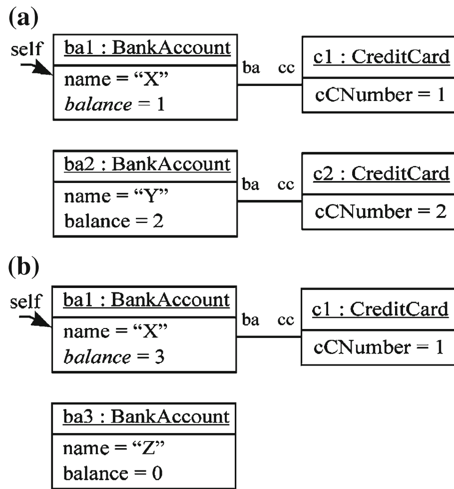

Fig. 2 A pre- and a post-state of *credit*

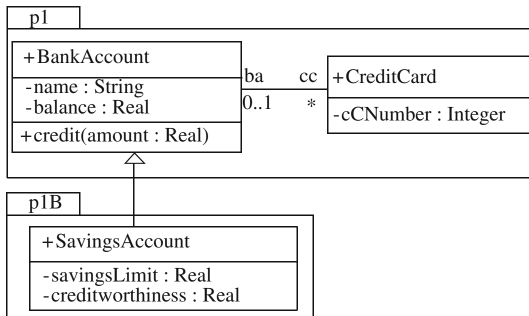

Fig. 3 Inter package extension

account *ba3* is created; this is in accordance with the contract (see part (b) of the figure).

2.3 Model modification

In this subsection, we deal with the problem of model modification. We investigate to what extent we need to change an invariability specification if a class is subclassed within the same package and within another package.

Let us consider Fig. 3. We subclass the class *BankAccount* using another package. The class *BankAccount* is extended by the class *SavingsAccount*. The attribute *savingsLimit* specifies the lower limit of the corresponding balance and the attribute *creditworthiness* specifies the creditworthiness of a client. We assume that the second attribute is correlated with the attribute *balance*; if for example the balance grows, creditworthiness grows as well. The previous specification does not specify the behaviour of the attributes *savingsLimit* and *creditworthiness*, since they belong to a different package. Consequently, they can be changed arbitrarily. To restrain changes in respect to the package *p1B* we have to specify them explicitly:

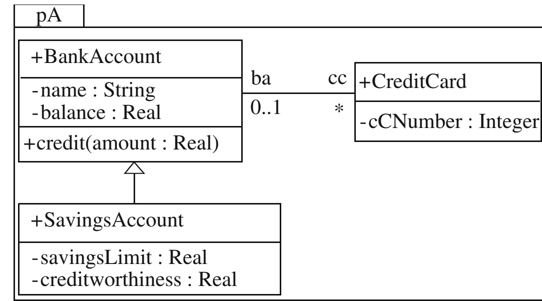

Fig. 4 Intra package extension

context $p1::BankAccount::credit(amount : Real)$
in $p1B$ **modifies** : $(if\ self.isKindOf(SavingsAccount)\ then$
$Set\{self.oclAsType(SavingsAccount)\}\ else\ Set\{\}\ endif)::creditworthiness$

The specification of invariability is stable in respect of extensions which do not change the corresponding view (the package *p1*, for example), since constraints concerning a superclass are inherited by subclasses and consequently do not need to be explicitly added. However, changes may be necessary if the view is modified. Fig. 4 shows another way of extending the *BankAccount* class. In this case, the view given by package *p1* is changed. Consequently, we have to change the specification of *credit*, since it was done relatively to the view defined by *p1*.

context $pA::BankAccount::credit(amount : Real)$
post : $self.balance = self.balance@pre + amount$
in pA **modifies** : $self::balance,\ (if\ self.isKindOf(SavingsAccount)\ then$
$self.oclAsType(SavingsAccount)\ else\ Set\{\}\ endif)::creditworthiness$

In this case, the execution of *credit* may change the attribute *balance* of the actual implicit parameter and if it is of class *SavingsAccount*, then also its attribute *creditworthiness*. We treat here single objects as singleton sets, e.g. *self* is treated as $Set\{self\}$.

Suppose that a class is meant to be subclassed and forwards operation calls to other classes. It is a good specification style to abstract in the superclass specification from the attributes in subclasses and in delegatee classes. In our case, it is possible to restrict a specification to a particular class. The following specification restricts the view to the class: *BankAccount*.

context $pA::BankAccount::credit(amount : Real)$
post : $self.balance = self.balance@pre + amount$
in $BankAccount$ **modifies** : $self::balance$

2.4 Side-effects

An operation execution may result in modification of objects different from its actual parameters. It may also modify attributes which are invisible in a certain view. This is usually

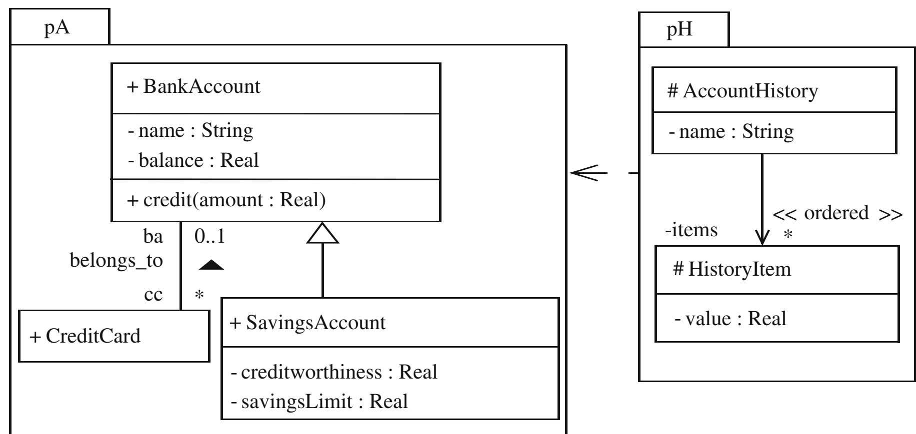

Fig. 5 Adding account history

the case of method logging. When aspect-oriented programming is used, it is possible to change attributes which are not navigable from parameters of executed methods. In this subsection, we show how to deal with such side-effects.

Figure 5 shows the class *AccountHistory*. An object of this class stores information about the history of a bank account object. If the operation *credit* is executed and if the values of the attribute *name* of a bank account and the value of the attribute *name* of a history object are equal, then the old balance of the bank account is stored in a newly created object of class *HistoryItem* and appended at the end of the list *items*. Apart of *pA*, we specify a bank-internal view including package *pH*:

context *pA::BankAccount::credit(amount : Real)*
post *post_credit : self.balance = self.balance@pre + amount and*
pH::AccountHistory.allInstances()->forAll(o | o.name = self.name
implies o.items->one(hi | hi.oclIsNew() and hi.value = self.balance@pre
and o.items = o.items@pre->including(hi)))
in *pA* **modifies** *mod_pA : self::balance, (if self.isKindOf(SavingsAccount)*
then self.oclAsType(SavingsAccount) else Set{} endif)::creditworthiness
in *pH* **modifies only** *mod_pH : pH::HistoryItem.allInstances(),*
pH::AccountHistory.allInstances()->select(o | o.name = self.name)::items

The OCL expression *one* means that there is exactly one object which satisfies the corresponding condition. The colon in modifies clauses is followed by a comma-separated list of modifiable properties and the corresponding terms defining the scope of change. The term *including*(*hi*) means that the object *hi* is appended to the end of the sequence *items*. The last clause is strict which is indicated by keyword **only**. It disallows the creation and deletion of *HistoryItem* objects. It restricts also the changes in package *pH* to the attribute *items* of the history objects which have the same name as the credited bank account and allows only the creation and deletion of *HistoryItem* objects.

2.5 Specification of operations on lists

In this subsection, we show how to specify operations on singly linked lists. In standard OCL, it is not easy to specify what remains unchanged when a list is sorted, an element is inserted or another list is appended. Consequently, the specification of invariable parts tends to be left out.

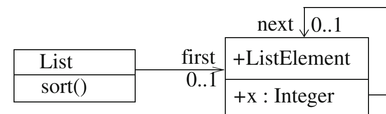

Fig. 6 Singly linked list with an anchor

The class diagram in Fig. 6 shows a list composed of an anchor object of class *List* and a number of elements instantiating the class *ListElement*. The collection of elements contained in a list *self.elements* is defined with the help of the auxiliary function *successorsOf* collecting all successors of a given list element *el* in the set *Acc*:

context *List* **def** :
elements : Set(ListElement) = if self.first->isEmpty() then Set{}
else self.first.successorsOf(Set{self.first}) endif
context *ListElement* **def** :
successorsOf(Acc : Set(ListElement)) : Set(ListElement) =
if self.next->isEmpty() or Acc->includes(self.next) then Acc
else self.next.successorsOf(Acc->union(Set{self.next})) endif

We consider here only finite acyclic lists. This constraint is expressed by an invariant saying that a nonempty list must contain an element that does not have a successor:

context *List* **inv** *no_loops* :
self.elements->notEmpty() implies
self.elements.exists(el | el.next->isEmpty())

The operation *sort* orders lists according to the value of attribute x. We use the term *elements@pre* to denote all list elements which exist in the pre-state.

context *List::sort()*
post *post_sort : self.elements = self.elements@pre and*
self.elements->forAll(el | el.next->notEmpty() implies el.x <= el.next.x)

We can make the specification of *sort* precise by adding the following two invariability clauses:

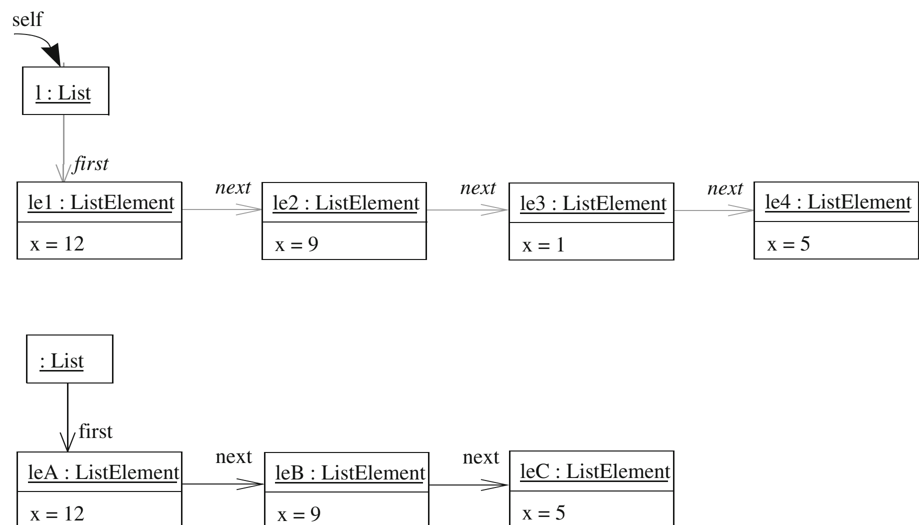

Fig. 7 Concrete lists with modifiable parts

in *List* **modifies** *mod_list* : *self*::*first*
in *ListElement* **modifies** : *self.elements*::*next*

The first clause says that the element associated with the list anchor can be replaced. The second one says that the *next* association-end can be modified. Those clauses in conjunction with the first part of the post-condition say that the elements of the list can be rearranged, but no element can be added or removed. They relativize invariability specification to classes *List* and *ListElement*. They do not specify what happens outside those classes. We can express the fact that nothing else changes using the absolute clause, i.e. clause which does not contain the in-part: **modifies** : *self*::*first*, *self.elements*::*next*.

Figure 7 shows an object diagram corresponding to the class diagram in Fig. 6. Names of modifiable links are indicated using italic. In this diagram, the variable *self* refers to the anchor object *l*. Let us observe that the set *self.elements* includes all elements of the list with anchor *l*, i.e. *self.elements* has the form $\{le1, le2, le3, le4\}$. The clause *self.elements*::*next* allows modification of the corresponding outgoing links. Similarly, *self*::*first* allows replacing the first element. All attributes and all other associations are not modifiable. Consequently, the operation *sort* can only rearrange the elements of this list, due to the constraint *elements* = *elements@pre*, but it cannot change attributes of the corresponding objects nor attributes or associations of other objects. On the other hand, the operation cannot modify the anonymous list in anyway; it can only delete it.

3 Views

In this section we define the notion of view and investigate its properties. We start with its brief discussion. In Sect. 3.1, we relate the notion of view to the UML 2 metamodel and to the notion of package. In Sect. 3.2, we show how to use the metamodel and OCL terms to define user specific views.

In the preceding sections we restricted our specifications to packages and classes. In general, it is possible to tune a specification to specific needs. A system specification can be written having a particular application in mind; it may focus on public or reachable model elements or on model elements named using special naming conventions. Cheesman and Daniels use class diagrams including selected classes, attributes and methods to specify system interfaces [15]. In general, a system specifier may define different views meant for different users. For example, the operation's monitoring and logging are usually not made visible for a client. We introduce an abstract concept of view which defines the focus of a specification. It can be treated as a semantic counterpart of the idea of package signatures and APIs. Our concept is based on the notion of reduct as it is known in model theory (cf., e.g. [14]) and database theory (cf., e.g. [31]). In our approach, the specification of the invariable part can be restricted to the appropriate view. This allows one to avoid a disclosure of internal details and a restriction of invisible system parts.

3.1 Relation to the UML metamodel

In this subsection, we define the notion of view in terms of the UML 2 metamodel [42] as opposed to the notion of view defined in [30] which was based on UML 1.5 [41]. We show how to define views corresponding to packages and discuss the relation between notions defined so far and the UML 2 metamodel.

Views, as defined in this paper, correspond to facades, i.e. groupings of arbitrary model elements, as they were defined in UML 1.5. Unlike its older version, UML 2 distinguishes between packageable elements and non-packageable ones. In UML 2, packages can include only elements such as types, in

particular classes and interfaces, and other packages. Other elements such as attributes, *allInstances* features and operations are only indirectly included in a package via the corresponding classes and interfaces. The OCL standard [40] refers to *C.allInstances* as a predefined feature of class *C* without precisely explaining what it means. We treat this feature simply as an attribute.

We define the notion of view using the UML metamodel (see [42], subsections 7.2 and 7.3). A view consists of a set of classes and interfaces, a set of properties, and a set of operations. The type of a view is specified by the following OCL expression:

$$\begin{array}{l} \mathit{Tuple}(\mathit{classesAndInterfaces} : \mathit{Set}(\mathit{Classifier}), \\ \mathit{properties} : \mathit{Set}(\mathit{Property}), \\ \mathit{operations} : \mathit{Set}(\mathit{Operation})) \end{array}$$

We say that v is a view if $v.\mathit{classesAndInterfaces}$ includes all classes and interfaces being types of properties belonging to $v.\mathit{properties}$ and of parameters of operations belonging to $v.\mathit{operations}$. We say that v includes a class or interface if that class or interface belongs to the set $v.\mathit{classesAndInterfaces}$. Similarly, we say that v includes a property or an operation if it belongs to $v.\mathit{properties}$ or $v.\mathit{operations}$, respectively. We say that a package includes a class, a property or an operation if it is included in the corresponding view (we define those notions precisely in Sect. 5.2).

Observe that the **in modifies** clause is defined on two levels of abstraction. The **in** part, specifying a view or a package, defines a number of properties. It is not fine enough to deal with runtime configurations. The **modifies** part defines a number of object sets and the corresponding modifiable properties. Since we are dealing with two different levels of abstraction, we have to distinguish between model elements and their names. For the sake of simplicity in the rest of this paper, we will make this distinction only when necessary.

Packages as defined in UML 2.2 [42] can contain only the so-called packageable elements like classes, interfaces and associations. As an example, we extract a view from a package. In order to do that, we identify the corresponding types, properties and operations. The first term presented below defines the set of types corresponding to operation's parameters and results. The second term defines the types of association-ends. Both terms are defined at the meta-level:

$$\begin{array}{l} \textbf{context}\ \mathit{Operation} \\ \textbf{def} : \mathit{relatedTypes} : \mathit{Set}(\mathit{Type}) = \\ \qquad \mathit{self.ownedParameter.type}{-}{>}\mathit{union}(\mathit{Set}\{\mathit{self.type}\}) \\ \textbf{context}\ \mathit{Association} \\ \textbf{def} : \mathit{types} : \mathit{Set}(\mathit{Type}) = \mathit{self.memberEnd.type} \end{array}$$

For a package p, the following formula defines a set of classes, interfaces and associations which are included in p (we skip analogous definitions). The set $p.\mathit{properties}$ contains properties of classes and interfaces included in p. The set $p.\mathit{operations}$ includes all operations owned by classes and interfaces included in p. Finally, $p.\mathit{classesAndInterfaces}$ is the set of all classes and interfaces which are either included in the package or form types of parameters of operations, association-ends and attributes of classes and interfaces belonging to p.

$$\begin{array}{l} \textbf{context}\ \mathit{Package} \\ \textbf{def} : \mathit{classes} : \mathit{Set}(\mathit{Classifier}) = \\ \qquad \mathit{self.ownedTypes}{-}{>}\mathit{select}(c \mid c.\mathit{oclIsKindOf}(\mathit{Class})).\mathit{oclAsType}(\mathit{Class}) \\ \textbf{def} : \mathit{interfaces} : \mathit{Set}(\mathit{Classifier}) = \ldots \\ \textbf{def} : \mathit{associations} : \mathit{Set}(\mathit{Classifier}) = \ldots \\ \textbf{def} : \mathit{properties} : \mathit{Set}(\mathit{Property}) = \\ \qquad (\mathit{self.classes}{-}{>}\mathit{union}(\mathit{self.interfaces}).\mathit{ownedAttribute} \\ \qquad\qquad {-}{>}\mathit{union}(\mathit{self.associations})).\mathit{memberEnd} \\ \textbf{def} : \mathit{operations} : \mathit{Set}(\mathit{Operation}) = \\ \qquad \mathit{self.classes.ownedOperation}{-}{>}\mathit{union}(\mathit{self.interfaces.ownedOperation}) \\ \textbf{def} : \mathit{classesAndInterfaces} : \mathit{Set}(\mathit{Classifier}) = \\ \qquad (\mathit{self.classes}{-}{>}\mathit{union}(\mathit{self.interfaces}){-}{>}\mathit{union}(\mathit{self.properties.type}) \\ \qquad\qquad {-}{>}\mathit{union}(\mathit{self.operations.relatedTypes})) \\ \qquad\qquad\qquad {-}{>}\mathit{select}(c \mid c.\mathit{oclIsKindOf}(\mathit{Class})\ \mathit{or}\ c.\mathit{oclIsKindOf}(\mathit{Interface})) \\ \qquad\qquad\qquad\quad .\mathit{oclAsType}(\mathit{Classifier}) \end{array}$$

The definition above allows us to associate views with packages. Let p be a package; v is the corresponding view if the following condition is satisfied:

$$\begin{array}{l} v.\mathit{classesAndInterfaces} = p.\mathit{classesAndInterfaces}\ \mathit{and} \\ v.\mathit{properties} = p.\mathit{properties}\ \mathit{and}\ v.\mathit{operations} \\ \qquad = p.\mathit{operations} \end{array}$$

Note that predefined OCL-types from the OCL-standard library (see [40], Section 11), such as *Integer*, *Boolean* and *OCLAny*, are not included in a view definition; similarly the collection types. They are the constant part of considered models and as such do not need to be explicitly listed in a definition of a particular view. This is due to the fact that they can be used to declare types of parameters and attributes independently of the visibility of the corresponding operations, classes and packages.

3.2 User-defined views

Views allow one to focus on relevant system aspects and hide irrelevant ones. One can define views using packages. However, their extensive use bloats models. In this section, we discuss how to define views using OCL terms.

Programming languages of different kinds provide the possibility to define units of programming without using explicit names. In functional languages such as ML (cf. [35]) there are anonymous functions. In object-oriented languages, like Java or C#, there are anonymous classes and methods. Unfortunately, UML is lacking a proper mechanism that would allow avoiding extensive use of packages when it is not necessary. Cheesman and Daniels use selected classes

with a choice of attributes to specify system interfaces [15]; those interfaces are the client visible parts of the system. They demonstrate how to define system models from the client perspective in terms of those selections. Our concept of view is similar to their idea of using class diagrams with arbitrarily selected elements of the underlying class-model. It allows one to define views corresponding to different perspectives. A specification can be restricted to public or protected model elements by selecting elements of proper visibility and hiding the private ones. For example, for an arbitrary view v we can define the corresponding public view v_{public} by selecting its public elements:

$$v_{public}.classesAndInterfaces =_{def} v.classesAndInterfaces{-}{>}select(c \mid c.visibility = \#public)$$

$$v_{public}.properties =_{def} v.properties{-}{>}select(p \mid p.visibility = \#public)$$

and similarly for $v_{public}.operations$.

We can define views based on naming conventions as well. For example, it is possible to select classes and interfaces with names having the suffix "Bean" as well as methods and attributes with names having the prefix "ejb". Since classes, interfaces, properties and operations are all named elements (cf. [42], Section 7.3.33), we can select those elements based on their attribute *name* instead of *visibility*:

$$v_{Bean}.classesAndInterfaces =_{def} v.classesAndInterfaces{-}{>}select(c \mid String.allInstances {-}{>}exists(s|c.name = s.concat(\text{“Bean”})))$$

and similarly for $v_{Bean}.properties$ and $v_{Bean}.operations$.

For each class one can also specify a view corresponding to all classes navigable from that class and restrict the invariability constraints to that view. In Sect. 2.3, we showed how to deal with the specification of subclasses in a package. Actually, it is rather inelegant to specify what happens to subclasses at the level of their superclass. Let *p but subclasses* denote a view including all classes and interfaces which occur in the package p, but do not subclass a context class C. The constraint specifying the operation *credit* can be then written in the following form:

context $pA::BankAccount::credit(amount : Real)$
post : $self.balance = self.balance@pre + amount$
in pA *but subclasses* **modifies** : $self::balance$

The above immutability clause is defined relatively to classes which do not subclass $BankAccount$. Every class subclassing this class requires its own contract, and in particular invariability specification, e.g. it needs to be specified what happens to the attribute *savingsLimit* of the class $SavingsAccount$, as the superclass contract does not restrain its behaviour.

In general, it is reasonable to restrict a specification to underived model elements, since the behaviour of derived elements can be deduced from the behaviour of underived ones, i.e. for a view v we can define a new view (v) *but derived* which differs from v in that (v) *but derived* contains only the underived properties included in $v.properties$. In some cases, it may be reasonable to restrict an operation specification to classes which are navigable from the operation parameters via association-ends and generalization relationships traversed bottom up, since normally only objects of those classes can be modified during an operation execution. Such a specification can have the following form:

context $C::Op(p_1 : C_1, \ldots, p_n : C_n) : D$
...
in $navigableFrom(typesOfParams(Op))$ **modifies** : ...

where $typesOfParams(Op)$ is the list containing parameter types of operation Op, i.e. $C, C_1, \ldots, C_n, D$. We assume that the term *navigableFrom* denotes a view including all properties of classes navigable from those types.

In fact, we can select an arbitrary set of model elements using an OCL term defined on the meta-level. We can express what is variable and what is not in a specific view. Let us observe that views based on visibility and naming conventions are defined on the meta-level without referring to any concrete class-model or any concrete package. In general, a view defined in terms of the UML metamodel can be applied to any class-model. This shows that views can be defined in a generic way using OCL terms. A general specification language should not restrict users to a particular view, for example to *navigableFrom*. On the contrary, users should be free to define their own views as suits them best.

4 Extension grammar

In this section, we redefine the syntax of the OCL extension proposed in [30]. The difference is that we use the string **only** as a keyword indicating that the predefined feature *allInstances* is taken into consideration. The grammar is presented using the EBNF notation: [ ] means optional occurrence, { } means arbitrary number of repetitions and | means option. We restrict this syntax with some constraints which cannot be expressed by a context free grammar. We use capital characters for nonterminals and small characters for terminals. An operation specification has the following form:

context $C :: Op$
pre : Pre
post : $Post$
{ **in** P **modifies** [Nm] : M | **in** P **modifies only** [Nm] : Mo} |
modifies [Nm] : M | **modifies only** [Nm] : Mo

C is the context of the specification, Op is an operation, Pre is a pre-condition and $Post$ is a post-condition as defined by OCL [40]. We call an invariability clause strict if it contains the keyword **only**; otherwise, we call it non-strict. We call it relative if it contains the in-part; in the other case we call

it absolute. The operation specification consists of a number of strict and non-strict relative invariability clauses, or one absolute, strict or non-strict, clause. The string *Nm* names the corresponding invariability constraint. Nonterminals M and *Mo* describe what can change. The difference between M and *Mo* is that the second one can contain the feature *allInstances*. The nonterminal P corresponds to a package or more generally a term specifying a view. Furthermore,

$$P = (Pn\,[\mathbf{r}] \mid Cn \mid Mt)\,O, \quad O = [+]\,[_]\,[\sim]\,[-]$$
$$M = [\,\mathbf{nothing} \mid Prs\,\{,\ Prs\}\,], \quad Prs = [T::]Pr \mid Cn{*}O$$
$$Mo = [\,\mathbf{nothing} \mid (Prs \mid Cn.allInstances())\,\{,\ (Prs \mid Cn.allInstances())\}\,]$$

Pn is a package or view name. The terminal **r** is optional; it specifies all sub-packages, like the $-r$ option of Unix tools. *Cn* is a class name. Nonterminal *Mt* corresponds to a tuple of the view type. The tuple *Mt* is defined on the class-model level. O specifies visibility of considered properties; the visibility can be public, package-public, protected and private. We allow the use of multiple visibility predicates meaning that all listed options are possible. The terminal **nothing** specifies that nothing can change; it is a syntactic sugar for an empty list. T is an OCL term defining a single object or a set of objects; it is defined at the object level. The nonterminal *Pr* corresponds to an attribute or an association-end. The expression $Cn{*}O$ denotes all properties of class *Cn* with visibility specified by O. Note that terms such as *p1 but subclasses* correspond to the nonterminal P or more precisely to *Mt* (cf. Sect. 3.2).

Context-free grammars are not expressive enough to deal with types. Therefore, in addition, we require that in case of a clause of the following form:

$$[\mathbf{in}\ p]\ \mathbf{modifies}\ :\ t_1::a_1, \ldots, t_m::a_m,\ a_{m+1}, \ldots, a_{m+k}$$

Property a_i must be an object-attribute or an association-end and the term t_i must be well defined in respect to the corresponding context. We assume that t_i must not contain the primitive *@pre*, for $i = 1, \ldots, m$, since those terms are evaluated in the pre-state. Moreover, all objects defined by t_i must have property a_i. One can equivalently require that $t_i.a_i$ is a subterm of a syntactically correct OCL pre-condition; for example, the pre-condition may have the trivial form $t_i.a_i = t_i.a_i$. Note that we use "::" in those contexts where "." can be used. To simplify the notation we allow t_i to have the form x instead of $Set\{x\}$ where x is a formal parameter of Op. In case of a relative clause **in** p **modifies** $t_1::a_1, \ldots, t_m::a_m,\ a_{m+1}, \ldots, a_{m+k}$, we assume that properties a_i belong to the view p. We call $t_i::a_i$ a 'scope of change term' or simply 'scope-term'.

If p is a package, then we treat it as a shorthand for the corresponding view (see Sect. 3.1). If p is a view, then the property a_i must belong to $p.properties$, for $i = 1, \ldots, m+k$. The properties $a_{m+1}, \ldots, a_{m+k}$ are class-attributes and consequently do not include the predefined feature *allInstances*; recall that we treat $C.allInstances()$ as a property. In case of **modifies only**, we relax the last requirement, but we demand that if an a_{m+i} has the form $C.allInstances()$, then it must belong to $p.properties$ and the corresponding class C must belong to $p.classesAndInterfaces$. p+ denotes all public types, properties and operations included in p. The clause **in** p+ **modifies** restricts the invariability specification to public types and features included in p.

5 The semantics

In this section, we define a semantics of invariability clauses and investigate its properties. In Sect. 5.1, we define the semantics in terms of standard OCL [40]. In Sect. 5.2, we discuss its formal counterpart. In Sect. 5.3, we define a procedure allowing one to extract invariability clauses from method's post-conditions. Finally, in Sect. 5.4, we discuss different options in the definition of invariability.

5.1 OCL-based semantics

In this subsection, we define the semantics of the proposed extension in terms of standard OCL [40]. First, we define the semantics of a restricted form of invariability clauses. Then we show that the semantics of all other clauses can be defined with the help of the restricted form. The semantics is illustrated using the bank account example. The advantage of this semantics is the fact that one can rely on existing formal semantics of OCL ([40], Annex A; see also, e.g. [13,20]) and the possibility of using standard OCL tools (cf., e.g. [5,17,23]).

In the paper [30], we used *allInstances@pre* to define the OCL-based semantics of our extension. In this paper, we use the predicate *oclIsNew* to define the interpretation of the frame formulas. This interpretation is logically equivalent to the previous one, but it is better suited for currently existing OCL tools, since most of them do not support *allInstances@pre*. In [30], we defined the semantics in such a way that the modification of one association-end does not imply the modification of the opposite end. In this paper we define the semantics in a symmetric way. It fits better to the idea that associations are tuples of objects or links [42]. Attributes and association-ends are called in UML properties. A class owns its attributes, whereas association-ends are not owned by the corresponding classes, but by the association itself (see [42]). Object-attributes and association-ends are called object-properties, as opposed to class-attributes. In this paper, we consider only binary associations, i.e. associations with two ends. Presented semantics can be extended to the case of n-ary associations for $2 \leq$ n; however, this would complicate the definition a lot.

The semantics is defined via frame formulas. For simplicity, we assume that packages, classes and properties have unique names; however, in general it is necessary to use fully qualified names to distinguish between different model elements. We define first the semantics of strict invariability clauses of the following form:

context $X::Op$
pre $: Pre$
post $: Post$
in p **modifies only** $: t_1::a_1, \ldots, t_m::a_m, a_{m+1}, \ldots, a_{m+k}$

We assume that properties $a_1, \ldots, a_{m+k}$ are included in p. For $i = 1, \ldots, m$, we assume that t_i is an OCL term of type $Set(A_i)$ that is well defined in the context of the operation Op, in particular it can include only the formal parameters of Op, and that it does not contain the primitive *@pre*. Since we can use the OCL operator *union*, we can assume without a loss of generality that properties a_i are pairwise different. We assume that there exists a number $g \leqslant m$ such that for $i = 1, \ldots, g$ property a_i is an object-attribute of class A_i and that for $i = g + 1, \ldots, m$ property a_i is an association-end of a binary association as_i. By $\overline{a}_i$ we denote the end of as_i opposite to a_i. We assume that a_i allows navigation from class A_i to $\overline{A}_i$, and consequently $\overline{a}_i$ allows navigation from class $\overline{A}_i$ to A_i. We present properties occurring in view p as a sequence. There exist a natural number h such that $g \leqslant h \leqslant m$ and properties contained in p form a sequence of the following form: $a_1, \ldots, a_g, a_{g+1}, \ldots, a_h, \overline{a}_{g+1}, \ldots, \overline{a}_h, a_{h+1}, \ldots, a_m,$ $a_{m+1}, \ldots, a_{m+k}, b_1, \ldots, b_n, c_1, \ldots, c_l$
where

1. $a_1, \ldots, a_g$ are object attributes listed in the invariability clause
2. $a_{g+1}, \ldots, a_h$ are association-ends listed in the invariability clause such that, for $i = g+1, \ldots, h$, the end $\overline{a}_i$, opposite to a_i, is not listed in the invariability clause; the opposite ends $\overline{a}_{g+1}, \ldots, \overline{a}_h$ may or may not be included in p
3. $a_{h+1}, \ldots, a_m$ are association-ends such that the opposite association-ends are listed in the clause
4. $b_1, \ldots, b_n$ are all object properties which occur in p, but are not listed in the invariability clause; B_i is the class corresponding to the object-property b_i
5. $a_{m+1}, \ldots, a_{m+k}$ are class attributes and the predefined *allInstances*-features occurring in the clause, i.e. a_{m+i} has the form $C.c$ or $C.allInstances()$, for some class C; $c_1, \ldots, c_l$ are all other class attributes and *allInstances* features occurring in p

For $i = h + 1, \ldots, m$, we define term $u_i = t_i{-}{>}union(t_j.\overline{a}_i)$, where j is the index at which the end opposite to a_i occurs in the clause. Below $t_i@pre$ denotes a term obtained from term t_i by suffixing all properties occurring in t_i by *@pre*. We translate the specification of Op above to standard OCL by considering cases 1–5:

context $X::Op$
pre $: Pre$
post $: Post \;\; and \;\; -for\, i = 1, \ldots, g, we\, generate:$
$A_i.allInstances(){-}{>}forAll(o\,|not(o.oclIsNew()) \; and \; t_i@pre{-}{>}excludes(o)$
$implies \; o.a_i = o.a_i@pre)$
$and \;\; -for\, i = g+1, \ldots, h, we\, generate:$
$A_i.allInstances(){-}{>}forAll(o\,|not(o.oclIsNew()) \; and \; t_i@pre{-}{>}excludes(o)$
$implies \; o.a_i = o.a_i@pre) \; and$
$\overline{A}_i.allInstances(){-}{>}forAll(o\,|not(o.oclIsNew()) \; and \; (t_i.a_i)@pre{-}{>}excludes(o)$
$implies \; o.\overline{a}_i = o.\overline{a}_i@pre)$
$and \;\; -for\, i = h+1, \ldots, m, we\, generate:$
$A_i.allInstances(){-}{>}forAll(o\,|not(o.oclIsNew()) \; and \; u_i@pre{-}{>}excludes(o)$
$implies \; o.a_i = o.a_i@pre)$
$and \;\; -for\, i = 1, \ldots, n, we\, generate:$
$B_i.allInstances(){-}{>}forAll(o\,|\,not(o.oclIsNew()) \; implies \; o.b_i = o.b_i@pre)$
$and \;\; -for\, i = 1, \ldots, l, \; we\, generate:$
$c_i = c_i@pre$

The non-strict clauses do not contain the keyword **only**; consequently the behaviour of *allInstances* is not restricted. They can be treated as a special case of strict ones. Thus the semantics of

in p **modifies** $: t_1::a_1, \ldots, t_m::a_m, a_{m+1}, \ldots, a_{m+k}$

is equivalent to the semantics of

in p' **modifies only** $: t_1::a_1, \ldots, t_m::a_m, a_{m+1}, \ldots, a_{m+k}$

where p' is obtained from p by removing all features of the form $C.allInstances()$. In this case no restrictions on object creation and deletion are made, apart from the post-condition and the frame formulas. In other words, the semantics of an absolute non-strict clause is defined as the semantics of an absolute strict-clause, but we remove the *allInstances*-features. As an example, let us consider the specification of method *credit* in Sect. 2.4. The strict-constraint *mod_pH* restricts object creation to the class *HistoryItem*. The modifies-clause *mod_pA* does not restrict object creation nor deletion.

The resulting post-condition is a conjunction of the original post-condition *Post* and a frame formula corresponding to the invariability clause. This formula has five parts. The first part deals with the change scope of object attributes. The corresponding clause means that for every object o of class A_i if o exists before and after execution of Op, and if o does not belong to the set defined by t_i in the pre-state, then the property a_i of o remains unchanged. In our example, the attribute *balance* is changed only for the implicit parameter *self*. For all other objects that exist before and after the method execution this attribute remains unchanged.

Since we use a non-strict invariability clause in case of package *pA*, object creation and initialization of the attribute *name* in the newly created objects is allowed. In case of class *SavingsAccount*, the attribute *balance* can be changed only for the implicit parameter *self*. This means that we have the following frame formula:

$$BankAccount.allInstances()\text{->}forAll(o \mid not(o.oclIsNew()) \\ and\ Set\{self\}\text{->}excludes(o)\ implies\ o.balance@pre = o.balance)$$

If the actual implicit parameter is of class *SavingsAccount*, then the operation *credit* may change its *creditworthiness* attribute:

$$SavingsAccount.allInstances()\text{->}forAll(o \mid not(o.oclIsNew())\ and \\ Set\{self\}\text{->}excludes(o)\ implies\ o.creditworthiness@pre = o.creditworthiness)$$

The second part of the resulting frame formula concerns association-ends a_i of binary associations such that the opposite end $\overline{a}_i$ is not listed in the invariability clause. In this case we allow the modification of association-end a_i for objects defined by term t_i, and, to make the definition symmetric, the modification of the opposite end $\overline{a}_i$ for objects belonging to the image of t_i in respect of a_i. As an example, we consider operation *drop* dropping a credit card assigned to a bank account (cf. Fig. 4):

context *BankAccount*::*drop*(*c* : *CreditCard*)
post : *self.cc* = *self.cc@pre*->*excluding*(*c*)
modifies : *self*::*cc*

In this case, *ba* and *cc* are the opposite ends of association *belongs_to*. The above clause allows the modification of *cc* for *self*. The opposite end, i.e. *ba*, is not listed in the clause. If a link between a bank account and a credit card is dropped, then the opposite link connecting the card and the account has to be dropped too. Thus, the modification of *ba* is allowed for objects included in the set *self.cc*.

The third part concerns associations with both ends listed in the clause. Hence, for an association-end a_i, $i = g + 1, \ldots, h$, the opposite end $\overline{a}_i$ is in the list as well, i.e. $\overline{a}_i = a_j$ for some j, and consequently $g < j \leqslant h$. In this case, a_i can be modified for objects in the set defined by t_i and by $t_j.\overline{a}_i = t_j.a_j$ being the image of t_j in respect to a_j. Hence, $u_i = t_i\text{->}union(t_j.a_j)$ defines the scope of change for a_i. Similarly, we relax the association-end a_j for the set $t_j\text{->}union(t_i.a_j)$. Figure 8 shows a conceptual visualization of this case. The sets of all objects of classes A_i and A_j are indicated by rectangles. Sets of objects defined by terms t_i and t_j, and by their images in respect of a_i and a_j, are indicated by ellipses. The two scopes of change are indicated by the colour gray. As an example, we consider an operation assigning a credit card to a bank account:

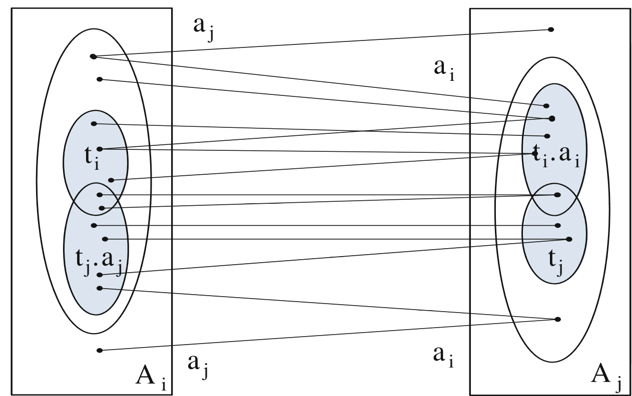

Fig. 8 Visualization of interdependent scope-terms

context *BankAccount*::*assign*(*c* : *CreditCard*)
post : *self.cc* = *self.cc@pre*->*including*(*c*)
modifies : *self*::*cc*, *c*::*ba*

In this case, both ends of *belongs_to* are listed in the clause. The association-end *cc* can be modified for *self* and for objects in *c.ba*. Vice versa, *ba* can be modified for *c* and *self.cc*.

The fourth part of the post-condition concerns all other object-properties included in p. For every b_i and every object o of the corresponding class B_i if o exists before and after execution of Op, then its property b_i must not change. For example in case of *credit*, attribute *name* and association-end *cc* are not listed in the modifies list. Consequently, they cannot be modified.

The fifth part forbids the modification of class-attributes and *allInstances* features included in p which are not listed in the clause. Note that there is no class-attribute in the class diagram in question. The modification of properties $a_{m+1}, \ldots, a_{m+k}$ is not restricted. For example in case of *credit*, the strict clause allows only the modification of *HistoryItem.allInstances*(), but *AccountHistory.allInstances*() cannot be modified, since it is not listed there. Thus objects of the second class can be neither created nor deleted. It is interesting to consider the question of object existence at the programming level. Some objects may be referenced by program variables occurring on the program stack. Those variables can be manipulated in an arbitrary way as long as the post-condition and the resulting frame formula are satisfied. In general, an object cannot be deleted during an operation execution if it is reachable from variables on the execution-stack via links, in particular the unmodified ones.

The grammar defined in Sect. 4 allows for using the implicit parameter *self* instead of the set *Set*{*self*}. In the examples presented in Sect. 2, we use *self* in that way. In general, if a term t_i occurring in a modifies clause defines

a single object instead of a set of objects, then to apply the semantics definition we replace the term by $Set\{t_i\}$. It should be noted that we do not deal with qualified associations separately. This is due to the fact that a qualified association defines an unqualified one. If qualifiers in a qualified association are missing, then one obtains the set of all associated objects (cf. [40,50]).

One OCL-constraint may contain several **in modifies only** clauses (see Sect. 4). In such a case, we define a conjunction of the corresponding frame formulas. Other kinds of invariability clauses can be treated as abbreviations of the above one. In case of the absolute invariability clause

modifies only : $t_1::a_1, \ldots, t_m::a_m, a_{m+1}, \ldots, a_{m+k}$

the specification of changes concerns all properties. This kind of constraint is an abbreviation of the following formula:

in *ap* **modifies only** : $t_1::a_1, \ldots, t_m::a_m, a_{m+1}, \ldots, a_{m+k}$

where *ap* includes all classes, interfaces, properties and operations contained in a model. The relative clause **in** p **modifies only** : **nothing** can be equivalently expressed by the formula **in** p **modifies only** :, which includes an empty list of properties. It means that no property included in p is modified.

5.2 Formal semantics

In this subsection, we formalize the notion of view. The OCL standard defines a formal semantics of OCL (see [40], Appendix A), but there exist other ones defined for different purposes (see [13,20] and the references there). The semantics we use here is a variation of the semantics defined in [26]. It is based on the notion of order-sorted algebra [25]. The difference between our semantics and the standard OCL semantics is that we treat system states as first-order beings and the state space as a sort like in [7]. We need this kind of approach to apply results from the standard model theory, in particular that reducts preserve validity of formulas (cf., e.g. [14]).

The concept of view presented here corresponds to the notion of reduct as it is used in model theory (cf., e.g. [14,51]) and on the other hand to the concept of hidden algebra developed in the realm of algebraic specification (cf. [51]). Hidden algebras are also a very powerful means of system specification (cf. [6]). A hidden algebra provides only one external interface to a given model, whereas there can be several views of the same class model.

An order-sorted signature has the form $\Sigma = (S, F, \leqslant)$, where S is a set of sorts, F is a set of function symbols with sorts in the set S and $\leqslant$ is a partial order on S. Given a view v, we define the corresponding signature $\Sigma = (S, F, \leqslant)$. The set of sorts S contains sorts corresponding to predefined OCL-types and to the elements of $v.classesAndInterfaces$. In particular, S contains a sort symbol C for every class and interface C belonging to $v.classesAndInterfaces$ and a sort symbol for every predefined OCL-type such as $Real$. Moreover, for every type T in $v.classesAndInterfaces$, S contains the corresponding collection types such as $Set(T)$. Finally, S includes the sort $OclAny$ and also the sort $State$ modelling global system states (cf. [7]). The set S is partially ordered by the relation $\leqslant$, i.e. if C subclasses B, then $C \leqslant B$ holds. We also assume that for every sort T different from $State$, $T \leqslant OclAny$ holds and that $State$ is not comparable with any other sort. F is a set of typed function symbols corresponding to attributes, association-ends, queries, state changing operations and predefined OCL functions. For example, if a is an attribute owned by class C with values of class D, then S contains the sorts C and D; moreover, F contains the function symbol $a : State \times C \to T$. The additional argument of sort $State$ is due to the fact that the value of an attribute depends on the current system state (cf. [7]). The predefined feature $C.allInstances()$ is formalized by the function symbol $C.allInstances : State \to Set(C)$.

The set of terms is defined as the smallest set containing variables, constants, such as 0, and closed on composition, i.e. for function symbol f of type $s_1 \times \cdots \times s_n \to s$, terms t_i of sort s'_i and $s'_i \leqslant s_i$, $f(t_1, \ldots, t_n)$ is a term of sort s. We formalize OCL terms using terms of the corresponding order-sorted signature. For example, the OCL term *self.balance >= 0* is formalized by the algebraic term $(s : State, self).balance >= 0$. Note that the function formalizing the attribute *balance* has an argument of type $State$, since the value of attributes depend on system states.

A model $\mathbb{M}$ of signature $\Sigma = (S, F, \leqslant)$ is a triple consisting of the following elements: a set of sets $\{s^{\mathbb{M}} \mid s \in S\}$ interpreting the sort symbols, a set of functions $\{f^{\mathbb{M}} \mid f \in F\}$ interpreting the function symbols, and the subset relation $\subseteq$ formalizing the partial order relation on sorts. If C is a class, then $C^{\mathbb{M}}$ can be interpreted as the corresponding location/address space. For a function symbol f, $f^{\mathbb{M}}$ is the corresponding function. If T_1 subclasses T_2, then we assume that the corresponding sorts are ordered $T_1 \leqslant T_2$ and consequently the inclusion $T_1^{\mathbb{M}} \subseteq T_2^{\mathbb{M}}$ holds. We assume that for every sort T, the set $T^{\mathbb{M}}$ contains the undefined symbol $\perp$ (cf. [26]). We assume also that for every state σ and every type T, the function $T.allInstances^{\mathbb{M}}$ returns the set of instances existing in σ, i.e. $T.allInstances^{\mathbb{M}}(\sigma) \subseteq T^{\mathbb{M}}$. In case of predefined OCL types such as *Real*, we assume that *Real.allInstances* is invariable, i.e. the equation $Real.allInstances^{\mathbb{M}}(\sigma) = Real^{\mathbb{M}}$ holds for every state σ (cf. [26]).

Let $Op(x_1 : T_1, \ldots, x_n : T_n)$ be an operation specified by a post-condition. We formalize the post-condition by a term of the form $t(s, s' : State, x_1 : T_1, \ldots, x_n : T_n)$, where s and s' are state variables corresponding to the pre- and post-state, respectively; the properties occurring in the

post-condition are formalized by the above-defined functions. We say that operation Op is deterministically defined if for every two valuations of the form $\mathbf{v}_1 = [s \mapsto \sigma,\ s' \mapsto \sigma_1,\ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ and $\mathbf{v}_2 = [s \mapsto \sigma,\ s' \mapsto \sigma_2 x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ such that $t^{\mathbb{M}}\mathbf{v}_1 = t^{\mathbb{M}}\mathbf{v}_2 = true$, the equation $\sigma_1 = \sigma_2$ holds. The proposed invariability clauses allow us to reduce the determinism in operation specifications. We say that a signature $\Sigma_0 = (S_0,\ F_0,\ \leqslant_0)$ is an order-sorted reduct of a signature $\Sigma = (S,\ F,\ \leqslant)$ if the following conditions are satisfied:

- $S_0 \subseteq S$, $\leqslant_0 \subseteq \leqslant$ and $F_0 \subseteq F$
- $\leqslant_0 = \leqslant \cap (S_0 \times S_0)$

In other words, an order-sorted reduct is a reduct in the sense of algebraic specification [51], with the restriction that the restricted sort-ordering relation is a restriction of the initial sort-ordering relation.

Let $\Sigma_0 = (S_0,\ F_0,\ \leqslant_0)$ be an order-sorted reduct of $\Sigma = (S,\ F,\ \leqslant)$. A model $\mathbb{M}_0$ of signature Σ_0 is an order-sorted reduct of a model $\mathbb{M}$ of signature Σ if it is a reduct in the sense of algebraic specification, i.e. for every sort T in S_0, $T^{\mathbb{M}_0} = T^{\mathbb{M}}$ and for every function symbol $f \in F_0$, $f^{\mathbb{M}_0} = f^{\mathbb{M}}$. Let us observe that in particular if the sorts *OclAny* and *State* belong to S_0, then $OclAny^{\mathbb{M}} = OclAny^{\mathbb{M}_0}$ and $State^{\mathbb{M}} = State^{\mathbb{M}_0}$. If $\mathbb{M}_0$ is an order-sorted reduct of $\mathbb{M}$, then for every term t of the signature Σ_0, the value of t is the same in both models. Consequently, the satisfaction relation for formulas of the signature Σ_0 is preserved by reducts. Note that this result does not always hold (cf. [24]).

Statement

Let $\mathbb{M}_0$ be an order-sorted reduct of $\mathbb{M}$ as described above. Let $t(x_1 : T_1, \ldots, x_n : T_n)$ be a term of the restricted signature Σ_0 and let $\mathbf{v} = [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ be a valuation such that $v_i \in T_i^{\mathbb{M}_0}$, for $i = 1, \ldots, n$. Then $t^{\mathbb{M}}\mathbf{v} = t^{\mathbb{M}_0}\mathbf{v}$.

The simple proof follows by structural induction on the term complexity: we prove the property for variables first and then from the assumption that it holds for all subterms of a term t we infer that it holds for t. If t is a variable, then the statement holds trivially. Let it hold for all terms of complexity smaller than the complexity of the term t; let t have the form $f(t_1, \ldots, t_n)$ and let $\mathbf{v}$ be a valuation as defined above. Then $f^{\mathbb{M}} = f^{\mathbb{M}_0}$ and $t_i^{\mathbb{M}}\mathbf{v} = t_i^{\mathbb{M}_0}\mathbf{v}$, for $i = 1, \ldots, n$. Therefore, the equation $t^{\mathbb{M}}\mathbf{v} = t^{\mathbb{M}_0}\mathbf{v}$ holds too.

The statement implies that the validity of OCL-constraints is preserved by reducts and consequently by views. Another consequence is that for arbitrary formulas Ψ and Φ of a restricted signature, if Φ follows semantically from Ψ in respect to models of the restricted signature, then Φ follows semantically from Ψ in respect to models of the full signature. Thus, reasoning in terms of the restricted signature is sound in respect to the full one.

5.3 Extraction of invariability clauses

In this subsection we present a method for extracting invariability clauses from post-conditions. It can be seen as a formalization of the heuristic called "nothing else changes" defined in the paper [12]. This heuristics is a liberal form of the implicit invariability assumption. It restricts the scope of variability to properties traversed during the evaluation of a post-condition. Extracted clauses can be used as a touchstone to assess the soundness and completeness of user specified clauses.

We define an extraction function that returns a set of scope-terms. The idea is to treat subterms of a post-condition as scope terms. In cases of simple post-conditions which do not contain recursively defined function symbols nor iterators the corresponding invariability clauses are rather straightforward. We have to treat post-conditions containing iterators in a special way because of bound variables. Extracted scopes terms correspond to subterms of the iterated term. They are obtained by composing those subterms with the term we iterate over. It should be mentioned that quantifiers and the select-operation can be expressed using iterators (see [40]); thus we do not have to treat them separately. Formally, a variable x is called free in a term t if it is not bound by an iterator, i.e. t does not contain a subterm of the form t_1->*iterate*$(x; acc = v \mid t_2(acc, x))$. Let t be an OCL term that does not contain @*pre*; *extract*(t) is the smallest set of scope-terms such that the following conditions are satisfied:

1. if a is a property, $r.a$ is a subterm of t and the free variables of r are not bound in t, then the scope-term r::a belongs to *extract*(t)
2. if t has a subterm of the form t_1->*iterate*$(x\,;\ acc = v \mid t_2(acc, x))$, the scope-term r::a belongs to *extract*(t_2) and variable x is free in r, then the scope-term t_1->*collect*$(x \mid r)$::a belongs to *extract*(t)

Note that in the second case x is bound in t and ranges over the collection of values defined by t_1. We treat OCL constraints as terms. For a post-condition *Post*, *extract*(*Post*) is defined as *extract*(t) where t is obtained from *Post* by removing all primitives @*pre*.

As an example we consider operation *credit* specified in Sect. 2.2. In this case, the extraction results in a singleton set containing the scope-term *self*::*balance*. This term corresponds to the scope-term specified in that subsection. In case of *credit* specification in Sect. 2.3, we get the same scope-term despite the fact that *credit* is allowed to modify the attribute *creditworthiness*. Thus, the scope of change is too narrow, because this specification does not say how this attribute has to be modified. In the other case, the extraction function would also result in a scope-term for this attribute.

A post-condition can contain recursively defined function symbols. To handle this case, we define function $extract_R$.

In case of recursive definitions, one has to synthesize invariability clauses from a number of terms. For a term t we have to consider all its unfoldings obtained by replacing those symbols by their definitions. More precisely, an one-step unfolding u is obtained from t by replacing a recursively defined function symbol $f(y_1, \ldots, y_m)$ by its definition $F(y_1, \ldots, y_m)$. We assume that an unfolding u' of an unfolding u of t is an unfolding of t. For a term t that does not contain *@pre*, $extract_R(t)$ is the smallest set of scope-terms such that if u is an unfolding of t and $r{::}a \in extract(u)$, then $r{::}a \in extract_R(t)$. It should be noted that $extract_R(t)$ can be infinite.

As an example we consider the operation *sort* defined in Sect. 2.5. Its post-condition contains the forall-quantifier and the recursively defined symbol *elements*. The following term

$$self.elements{-}{>}forAll(el \mid el.next{-}{>}notEmpty()\ implies\ el.x\ <=\ el.next.x)$$

can be expressed using an iterator as follows:

$$self.elements{-}{>}iterate(el\,;\ acc = true \mid acc\ and\ el.next{-}{>}notEmpty()\ implies\ el.x\ <=\ el.next.x)$$

The application of (1) and the unfolding of *elements* result in term *self*::*first* and also *self.first*::*next*, *self.first.next*::*next*,... and so on. There are infinitely many terms of the form $self.first(.next)^n{::}next$. Nevertheless, the set of objects defined by terms $self.first(.next)^n$ is equal to *self.elements* and consequently we can synthesize *self.elements*::*next*. The application of (2) results in scope-terms *self.elements*—>*collect* (*el* | *el*)::*next* and *self.elements*.—>*collect*(*el* | *el.next*)::*x*. The operation *collect*() occurring in above terms is redundant. Note that scope-terms *el*::*next*, *el*::*x* and *el.next*::*x* are not extracted, since the variable *el* is bound in *post_sort*. Thus, we can express the invariability clause as follows:

modifies : *self*::*first*, *self.elements*::*next*, *self.elements.next*::*x*

The resulting clause is a bit too loose, since the last term allows undesired changes, but this is a general phenomenon in case of the above definition. Clauses synthesized in this way can be treated as the first approximation of the intended scope. The specifier can select scope-terms which are really intended and skip those which are not (for example *self.elements.next*::*x*). On the other hand, if needed one can adjust the extraction function so that it would result in a more restricted set of scope-terms. It should be noted that the extraction result depends on the actual form of post-conditions and can be different for logically equivalent forms.

5.4 Semantic variation points

It is possible to define the semantics of invariability clauses in different ways by modifying the translation procedure defined in Sect. 5.1. UML calls such options "semantic variation points" [42]. In general, it would be possible to tune the proposed semantics according to different needs using appropriate disambiguation heuristics (cf. [12]) and specification patterns (cf. [3]).

Invariability can be specified relatively to all OCL-properties including queries, not only attributes and association-ends. It is possible to extend accordingly the grammar and the semantics defined in previous sections. However, such specifications tend to be very complex. On the other hand, queries are usually specified in terms of attributes and associations anyway. Therefore, we have decided to exclude queries.

Another semantic variation point can be identified in case of terms defining the scope of change. In our semantics we assume that if term t defines the scope of change, then t is evaluated in the pre-state, or equivalently that *t@pre* is evaluated in the post-state. It would be possible to allow in the invariability clauses general terms containing at some positions the primitive *@pre* and to evaluate such terms in the post-state. However, we were not able to find any reasonable application of such a general definition, and for the sake of simplicity we decided to avoid this complication.

It is possible to define the semantics of invariability clauses in such a way that the modification of one association-end does not imply the modification of the other end; in fact we followed this idea in the paper [30]. That approach fits better to the implicit invariability assumption. Nevertheless, it has somewhat unexpected consequences for object existence when association-ends are modified; we demonstrate this in Sect. 6.1. In general, it is more natural to allow change of an association-end if the other end is modified.

One can treat extracted invariability clauses as implicit parts of post-conditions. More precisely, if a post-condition does not have an invariability clause, then the extractable invariability clause may be assumed to hold. In this way one can restrict scope of change as it was meant by the implicit invariability assumption. However, this procedure does not always yield what the specifier intends (see examples in Sect. 5.3). Therefore, in case of this semantic variation point, one has to check that extracted invariability clauses are correct and if not, then appropriately modify them.

One can also adjust the extraction function. For example, in its definition it is possible to consider only maximal subterms, instead of all subterms. It corresponds to the heuristic called "change only the last navigation in a navigation chain" (COtlN, see [12]). It is also possible to liberalize the heuristic "nothing else changes", by allowing the change of all properties mentioned in a post-condition and, in case of associations, their opposite ends for all objects traversed during the evaluation of a post-condition. In Sect. 6.1, we present an example where this kind of liberalization is necessary. Furthermore, it is possible to combine the liberalized heuristic with the COtlN heuristic.

6 Case study

In this section, we consider a specification of a more complex system and show how it can be simplified and decomposed using the notion of view. This case study is based on examples defined so far. In Sect. 6.1, we present a global view of a bank account management system. In Sect. 6.2, we define different views as needed by different system users.

Specification of complex systems resembles programming in the large. A large "spaghetti specification" is hard to understand and manipulate. It is easy to overlook an existing error or make a new one when modifying the specification. It also makes validation of the system implementation hard. As in case of programming, a system specification may serve different purposes. All these specification kinds correspond to different system views. We present an example of how views can be used to hide internal details and how to specify invariability relatively to an appropriate view.

6.1 Global specification

In this subsection, we specify a bank account management system. The specification is divided into three packages. We show that standard OCL mechanisms such as naming of conditions can be used in combination with invariability clauses. We demonstrate how to specify queries using invariability clauses and how to deal with qualified associations.

Package *pE* concerns employees and managers (see Fig. 9). An employee is characterized by their number *eNumber*. Each employee supervises a number of bank accounts related by the association-end *ba*. It is qualified by the attribute *n* corresponding to names of customers. The class *Employee* contains the attribute *eNumber* specifying the number of an employee. A manager supervises a number of employees. The *Employee*-objects are stored in a list similar to the one in Sect. 2.5. However, here, the list stores objects of class *Employee*. The association-end *staff* is a derived model element (cf. [42]) representing the list of supervised employees in the form of a sequence.

The sequence of *Employee* objects stored in a list can be defined analogously to the set of list elements (see Sect. 2.5). The difference is that the list stores employee objects instead of integers and that in this case we have to use the datatype *Sequence* instead of *Set*. Let $l.el_seq$ denote the sequence of elements occurring in a list l; we assume that the ordering of the sequence corresponds to the ordering of the list. We skip the corresponding definition. The following invariant specifies the *staff* association-end:

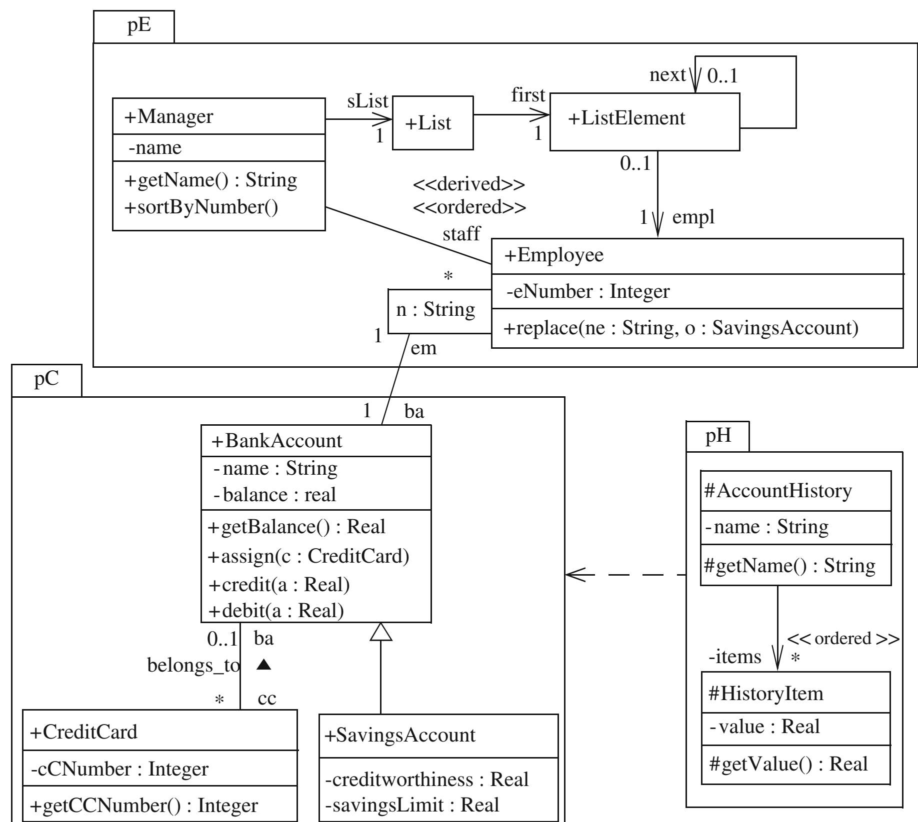

Fig. 9 Global view of the bank account

```
context Manager inv inv_staff :
Set{1.. self.staff->size()}->forAll(i |
                         self.staff->at(i) = self.sList.el_seq->at(i))
```

Note that this invariant relates the derived association-end *staff* to model elements which determine its value. It can be seen as the relation between a specific view and the global one.

The operation *sortByNumber* in class *Manager* sorts employees according to their numbers. Term *Set{1.. self.sList.el_seq->size()}* specifies the set of natural numbers corresponding to positions in sequence *self.sList.el_seq*:

```
context Manager::sortByNumber()
post post_sortByNumber :
self.sList.el_seq->asSet() = self.sList.el_seq@pre->asSet() and
Set{1.. self.sList.el_seq->size()}->forAll(i, j | i <= j implies
    self.sList.el_seq->at(i).eNumber <= self.sList.el_seq->at(j).eNumber)
in (pE, pC, pH) but derived modifies mod_sort : self.sList::first,
                                               self.sList.el_seq::next
```

The expression (*pE*, *pC*, *pH*) *but derived* denotes here a view consisting all underived properties defined in packages *pE*, *pC*, *pH* (cf. Sect. 3.2). Term *self.sList.el_seq::next* defines the collection of links corresponding to the attribute *next* for objects in the sequence. The invariability clause implies that *self.sList* remains unchanged and that the links between objects of class *ListElement* and objects of class *Employee* do not change, since the corresponding attributes do not occur in the modifies-part. The application of the extraction function defined in Sect. 5.3 would extend the list of scope-terms by *self::sList*. However, if the extraction function was defined in respect to maximal subterms, then *self::sList* would not be extracted (see Sect. 5.4).

The following constraint specifies the operation *replace*. This operation replaces an object of class *BankAccount* with an object of class *SavingsAccount* at the qualifier *ne*. The credit cards related to the replaced objects are reassigned to the *SavingsAccount*-object. Note that qualified associations can be used as unqualified ones; *self.ba* denotes the set of all objects associated to *self* via *ba*. The association-end *ba* is qualified with customers' names.

```
context Employee::replace(ne : String, o : SavingsAccount)
pre pre_repl : not(self.ba[ne].oclIsUndefined())
post post_repl : self.ba[ne] = o and o.cc = self.ba@pre[ne].cc@pre and
String.allInstances()->forAll(s | s <> ne and not(self.ba[s].oclIsUndefined())
                                  implies self.ba[s] = self.ba@pre[s])
in (pE, pC, pH) but derived modifies : o::cc, self::ba, self.ba[ne].cc::ba
```

If one end of a bidirectional association is relaxed, then the opposite end has to be relaxed as well (see Sect. 5.1, cases 2 and 3). In particular, the term *o::cc* implies that also the end *ba* is relaxed for objects defined by the term *o.cc*. It should be noted that if the relaxation of association-ends was not symmetric, then the bank account related to *self* would depend existentially on the corresponding credit cards (cf. 5.4). More precisely, if only the *ba* association-end of *belongs_to* was relaxed, then the replaced *BankAccount*-object would have to be deleted. Otherwise, it would point to the same credit cards as the savings account and vice versa those cards would have to point to two different accounts, the old one and the new one; this would contradict the multiplicity constraint imposed on the association-end *ba* (see Fig. 9).

The following operation assigns a credit card to a bank account:

```
context BankAccount::assign(c : CreditCard)
post : self.cc = self.cc@pre->including(c)
in (pE, pC, pH) but derived modifies : self::cc, c::ba
```

The semantics of invariability clauses proposed in this paper means that occurrence of *self::cc* in the modifies list implies relaxation of *ba* for objects defined in *self.cc*, i.e. *self.cc::ba*. This does not mean that all credit cards associated with *self* can be assigned to new bank accounts, since we have the condition *self.cc = self.cc@pre->including(c)*. The class diagram in Fig. 9 assures that every credit card is associated with at most one bank account. Note that in this case the extraction function does not produce the scope-term *c::ba*. However, if we applied its relaxed form (see Sect. 5.4), then this scope-term would be generated, since the parameter *c* occurs in the post-condition and *ba* is opposite to *cc*.

The operation *debit* charges a bank account only if there is enough money.

```
context BankAccount::debit(a : Real)
pre : 0 <= a and self.balance − a >= 0
post : self.balance = self.balance@pre − a and ...
in (pE, pC, pH) but derived modifies : mod_pA, mod_pH
```

mod_pA and *mod_pH* are defined in Sect. 2.4. We skip the part corresponding to handling of an account's history, since it is identical with the specification presented in that subsection.

The query *getBalance* returns the balance of a bank account. By definition, a query cannot modify anything, i.e. it is side-effect-free in the terminology of UML. In our notation, we can express this fact formally as follows:

```
context BankAccount::getBalance() : Real
post post_getBalance : result = self.balance
modifies : nothing
```

The operation *debit* can be executed only if there is enough money in the bank account. The balance of a bank account can be checked by executing the operation *getBalance*. Validity preservation in case of invariants requires stronger assumptions. Let us consider the following invariant:

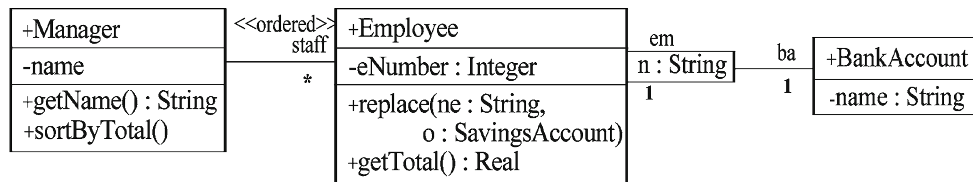

Fig. 10 Personnel department's view

context *SavingsAccount inv* :
self.savingsLimit <= *self.balance*

Its validity after an operation execution depends not only on the already existing objects, but also on the newly created ones. According to the semantics of non-strict invariability clauses, the invariability constraint for *getBalance* does not prohibit creation of new objects nor deletion of already existing ones.

Using a strict invariability constraint, we can specify that no object is created or deleted during a query execution. The following constraint guarantees preservation of this invariant:

context *BankAccount*::*getBalance*() : *Real*
post *post_getBalance* : *result* = *self.balance*
modifies only : **nothing**

6.2 User specific views

In this subsection, we define two different views of the bank account management system corresponding to different system users. We show that those views as well as the corresponding pre- and post-conditions and invariability clauses can be extracted from the global view. We define those views by using class diagrams rather than OCL terms and demonstrate that the abstraction from inessential details can be used in combination with invariability clauses.

The first view corresponds to a personnel department (see Fig. 10). It abstracts from the list implementation as well as operations and attributes which are irrelevant for the department. A manager has a list *staff* of their supervisees. We do not allow members of the department to check the balance of a particular bank account. The operation *replace* is specified in a similar way as in the global view, but here we do not refer to credit cards.

context *Employee*::*replace*(*ne* : *String*, *o* : *SavingsAccount*)
pre : *pre_repl*
post : *self.ba*[*ne*] = *o and*
String.allInstances()->*forAll*(*s* | *s* <> *ne and not*(*self.ba*[*s*].*oclIsUndefined*())
implies self.ba[*s*] = *self.ba@pre*[*s*])
in *personnel's_view* **modifies** : *self*::*ba*

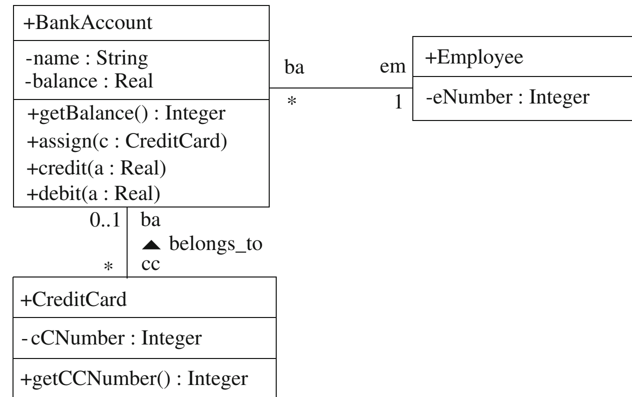

Fig. 11 Credit card company's view

The operation *sortByNumber* is specified in terms of the personnel's view:

context *Manager*::*sortByNumber*()
post *post_sort_local* : *Set*{*1*..*self.staff*->*size*()}->*forAll*(*i*, *j* | *i* <= *j*
implies self.staff->*at*(*i*).*eNumber* <= *sel.staff*->*at*(*j*).*eNumber*)
and self.staff->*asSet*() = *self.staff@pre*->*asSet*()
in *personnel's_view* **modifies** *mod_sort_local* : *self*::*staff*

Actually, the post-condition *post_sort_local* follows from invariant *inv_staff* and post-condition *post_sortByNumber* defined in Sect. 6.1. Results concerning reducts mentioned in Sect. 5.2 imply that this post-condition holds also for the reduct and therefore the post-condition is valid in the client view if it is valid in the global one. It should be noted that the post-condition *post_sortByNumber* and the modifies term *mod_sort* are not compatible with the personnel's view, since they contain properties not visible for a personnel department. However, the post-condition *post_sort_local* and the corresponding invariability clause are expressed in terms of the department's view.

The next view corresponds to a credit card company (see Fig. 11); we name it *cc_view*. It crosscuts packages *pC*, *pE* and abstracts from their irrelevant details. The company may credit and debit accounts corresponding to credit cards. It can also access the employee who supervises an account. However, it cannot see the bank organization, the way how employees are related to managers, how managers store the employee lists and so on. The operations *debit* and *assign* are in this case specified relatively to the corresponding view.

The specification abstracts away from effects which are not visible.

context *BankAccount*::*debit*(*a* : *Real*)
pre : *0* <= *a* and *self.balance* − *a* >= *0*
post : *self.balance* = *self.balance*@*pre* − *a*
in *cc_view* **modifies** : *self*::*balance*
context *BankAccount*::*assign*(*c* : *CreditCard*)
post : *self.cc* = *self.cc*@*pre*->*including*(*c*)
in *cc_view* **modifies** : *self*::*cc*, *c*::*ba*

7 Related work

The specification of invariability is crucial in case of state-oriented specification languages, such as OCL, VDM (cf., e.g. [22]) and Z (cf., e.g. [49]), as opposed to event-oriented ones, such as process algebras. There exist various approaches to invariability specification. The so-called frame axioms (cf. [36,47]) result in large specifications. There exists an approach which relies on a completion procedure [8]; basically for every method and every predicate one has to specify the circumstances under which the corresponding atomic formulas change their truth values. Unfortunately, for large systems it is not feasible, even if the set of underlying objects is finite. In general, if abstract datatypes such as integers are used, then the set of the corresponding atomic formulas is infinite.

Z provides the Ξ-operator and its dual Δ to facilitate specification of invariable parts. However, this operator does not have the flexibility and expressive power of the invariability clause proposed in this paper. In the realm of Z, frame formulas are used to restrict the scope of changes. In the paper [29] it is investigated as to when the scope of change, or its dual, determines invariability of certain program variables. Interestingly, there exists also an approach allowing the extension of graph rewriting rules with invariability constraints [4]. It combines OCL-constraints with imperative extensions. In the area of object-oriented systems, an analogous approach to ours was proposed in the paper [46]; the idea was to define a set of objects which may be modified. This approach does not detail which attributes may be changed. A similar approach was proposed for OCL [9]. It uses the primitive *modifiedOnly* to define the set of objects with modifiable attributes. As in the paper [30], it advocates the use of *allInstances@pre*() in the definition of invariability clauses. In our approach we define a pair consisting of a set of objects and a modifiable attribute. Thus, our approach was more expressive. In the paper [11], the approach defined in [9] was augmented so that one can specify modifiable attributes.

The paper [3] introduces a library of reusable OCL specification patterns. They facilitate the building of trusted components. There is a tool for automatic constraint generation. In our case, the scope-term extraction function can be used to generate invariability constraints; however, those constraints often need to be tuned. Nevertheless, like in case of specification patterns, the function facilitates the specification and helps inexperienced users. In order to execute specifications, declarative specifications have to be made imperative. The paper [12] provides a pattern-based method to translate declarative specifications to imperative ones. They are based on heuristics disambiguating post-conditions. The scope-term extraction function defined in Sect. 5.3 formalizes one of the proposed patterns. However, we do not aim here at executing OCL-specifications. The PhD thesis [44] introduces a number of fundamental concepts concerned with OCL.

Java Modeling Language [16] and Spec# [10] define invariability clauses which allow one to specify what happens with actual method parameters. The change of those parameters is specified in a way similar to our semantics. The validity of invariability clauses can be checked at compile-time. This of course restricts their expressiveness. It is not possible to specify how the set of all existing objects of a given class is influenced by a method execution. JML and Spec# specifications are closely related to the corresponding Java code, whereas OCL is a high-level specification language. It suits the purpose of JML and Spec#, but is not well suited for a high-level modelling.

Another point is that JML, like the implicit invariability assumption, requires the exposition of all system details, which contradicts the information-hiding principle [43]. It does not allow definition of views; consequently, it is not possible to relativize invariability definitions to views. It uses predefined abstract data types to cope for example with lists. However, in general, there are infinitely many datatypes. JML and Spec# do not have the expressiveness and flexibility of OCL. OCL combines the expressive power of the first-order logic and the relational algebra. Spec#, like JML, requires the specifier to define layers of abstraction in order to define invariants via an object ownership relation. Unlike JML, in Spec# invariability constraints do not refer to the internal layer structure, which can be changed arbitrarily (cf. [10]). This is similar to the view oriented invariability specification. However, views can be defined in an arbitrary way.

Information hiding is in general one of the most important principles in programming (cf. [43]). The purpose of information hiding is to obtain a system modularization and to hide implementation details. This concept also plays a crucial role in system specification. Component-based software development is a realization of this principle. It has been advocated by many authors (cf., e.g. [48] and the references there). There are different specification styles as there are different object-oriented programming styles. In the context of UML, the view-oriented approach was advocated in the Catalysis method [18]. The concept of view has been also defined in the realm of MDA [39]. Cheesman and Daniels

defined informally the notion of view as a selection of interfaces and classes with a choice of methods and attributes [15].

The viewpoints framework [21] provides a temporal logic-based approach to inconsistency handling. Although this framework was in the first place meant for inconsistency handling, it was one of the first formal approaches using a very restricted form of views. Packages, in particular facades, as defined in UML 1.5 (see [41], subsection 2.15.2.4), can be used to define system views; they allow the selection of relevant system properties and the hiding of irrelevant ones. However, an extensive use of named packages bloats models. At present, UML is lacking a proper mechanism which would allow one to avoid an extensive use of packages. Formal specification languages such as Z [49], its new form B (cf. [1,2]), its object-oriented version Object-Z (cf. [19]) as well as Temporal Logic of Actions (TLA) (cf. [52]) provide mechanisms for a clear separation of different abstraction levels. All these formalisms define the notion of refinement to relate different levels of abstraction.

8 Conclusion

Specification of invariability poses a real problem in OCL. Specification of complex systems resembles programming in the large. In order to be understandable and manageable it has to be done in a modular way and abstract from irrelevant details. Abstraction and modularization are the two fundamental mechanisms for handling complexity.

In this paper we presented new OCL primitives for the specification of invariability. We defined their semantics in terms of standard OCL and demonstrated their applicability. We showed that these primitives can be used to define precisely the notion of query. We investigated also how the concept of view can be used to hide irrelevant details. One can define views which suit different purposes without bloating the corresponding models. We showed that views are compatible with invariability constraints. We defined also a procedure allowing one for an automatic derivation of invariability clauses from post-conditions. Derived clauses can be easily customized by the specifier if needed.

Open Access This article is distributed under the terms of the Creative Commons Attribution License which permits any use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.

References

1. Abrial, J.: B#: toward a synthesis between Z and B. In: Bert, D., et al. (eds.) ZB'03—Formal Specification and Development in Z and B. International Conference of B and Z Users, Turku, Finland. LNCS, vol. 2651, pp. 168–177. Springer (2003)
2. Abrial, J., Cansell, D., Méry, D.: Refinement and reachability in Event B. In: Treharne, H., King, S., Henson, M., Schneider, S. (eds.) Formal Specification and Development in Z and B. 4th International Conference of B and Z Users. LNCS, vol. 3455, pp. 222–241. Springer (2005)
3. Ackermann, J., Turowski, K.: A library of OCL specification patterns for behavioral specification of software components. In: CAiSE'06, LNCS, vol. 4001, pp. 255–269. Springer (2006)
4. Baar, T.: OCL and graph-transformations—a symbiotic alliance to alleviate the frame problem. In: Proc. of MoDELS'05 Satellite Workshop on Tool Support for OCL and Related Formalisms, Montego Bay, Jamaica, October 4, pp. 83–99 (2005)
5. Baar, T., et. al.: Tool support for OCL and related formalisms needs and trends. In: Bruel, J.M. (ed.) Satellite Events at the MoDELS'05 Conference. LNCS, vol. 3844 (2006)
6. Bergstra, J., Tucker, J.: Algebraic specifications of computable and semicomputable data types. Theor. Comput. Sci. **50**, 137–181 (1987)
7. Bidoit, M., Hennicker, R., Tort, F., Wirsing, M.: Correct realizations of interface constraints with OCL. In: France, R., Rumpe, B. (eds.) The UML—Beyond the Standard, UML'99. LNCS, vol. 1723, pp. 399–415. Springer (1999)
8. Borgida, A., Reiter, R., Mylopoulos, J.: On the frame problem in procedure specifications. In: 15'th Int. Conf. on Software Engineering, Baltimore. IEEE Computer Society Press (1993)
9. Brucker, A.D.: An interactive proof environment for object-oriented specifications. PhD thesis, Dissertation No. 17097, ETH Zurich (2007)
10. Barnett, M., DeLine, R., Fhndrich, M., Leino, K.R., Schulte, W.: Verification of object-oriented programs with invariants. J. Object Technol. **3**(6), 27–56 (2004)
11. Brucker, A., Krieger, M., Wolff, B.: Extending OCL with null-references. In: Models in Software Engineering. LNCS, vol. 6002, pp. 261–275. Springer (2009)
12. Cabot, J.: From declarative to imperative UML/OCL operation specifications. In: Conceptual Modeling—ER 2007. LNCS, vol. 4801, pp. 198–213. Springer (2008)
13. Cengarle, M., Knapp, A.: OCL 1.4/1.5 vs. OCL 2.0 expressions: formal semantics and expressiveness. Softw. Syst. Model. **3**(1), 9–30 (2004)
14. Chang, C., Keisler, J.: Model Theory. North-Holland, New York (1990)
15. Cheesman, J., Daniels, J.: UML Components. Addison-Wesley, Boston (2000)
16. Darvas, A., Müller, P.: Reasoning about method calls in JML specifications. In: Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP'05), Glasgow, Scotland (2005)
17. DOT.: Dresdener OCL Toolkit. http://dresden-ocl.sourceforge.net/
18. D'Souza, D., Wills, A.: Object, Components, Frameworks with UML: The Catalysis Approach. Addison-Wesley, Reading (1998)
19. Dunne, S.: Understanding object-Z operations as generalised substitutions. In: International Conference on Integrated Formal Methods (IFM 2004). LNCS, vol. 2999. Springer (2004)
20. Evans, A., France, R., Lano, K. Rumpe, B.: Meta-modelling semantics of UML. In: Kilov, H. (ed.) Behavioural Specifications for Businesses and Systems, Chap. 4. Kluwer, Dordrecht (1999)
21. Finkelstein A., Kramer J., Nuseibeh B., Finkelstein L., Goedicke M.: Viewpoints: a framework for integrating multiple perspectives in system development. Int. J. Softw. Eng. Knowl. Eng. **2**, 31–58 (1991)
22. Fitzgerald, J.S., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object-Oriented Systems. Springer, Berlin (2005)
23. Gogolla, M., Richters, M.: Use: A UML-Based Specification Environment. http://www.db.informatik.uni-bremen.de/projects/USE/

24. Goguen, J., Burstall, R.: Institutions: abstract model theory for specification and programming. J. Assoc. Comput. Mach. **39**(1), 95–146 (1992)
25. Goguen, J., Meseguer, J.: Order sorted algebra. Theor. Comput. Sci. **105**(2), 167–215. Elsevier, Amsterdam (1992)
26. Hennicker, R., Knapp, A., Baumeister, H.: Semantics of OCL operation specifications. ENTCS **102**(2), 111–132 (2004)
27. Hitz, M., Kappel, G.: UML@Work. Dpunkt Verlag, Heidelberg (1999)
28. Hoare, T.: An axiomatic basis for computer programming. CACM **12**(10), 576–580 (1969)
29. Kassios, I.T.: Dynamic frames: support for framing. Dependencies and sharing without restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) Formal Methods'06. LNCS, vol. 4085, pp. 268–283. Springer (2006)
30. Kosiuczenko, P.: Specification of invariability in OCL. In: Nierstrasz, O., et al. (eds.) MoDELS'06, LNCS, vol. 4199, pp. 676–691. Springer, Berlin (2006)
31. Kozankiewicz, H., Stencel, K., Subieta, K.: Optimization of queries invoking views by query tail absorption. In: ADVIS'06, LNCS, vol. 4243, pp. 129–138 (2006)
32. Mitchell, R., McKim, J.: Design by Contract by Example. Addison-Wesley, Boston (2001)
33. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, Upper Saddle River (1998)
34. Meyer, B.: Applying design by contract. Computer **25**(10), 40–51 (1992)
35. Milner, R., Tofte, M., Harper, R.: The Definition of Standard ML. MIT Press, New York (1990)
36. Minsky, M.: A framework for representing knowledge. Technical Report 306, Artificial Intelligence Laboratory, MIT (1974)
37. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Sci. Comput. Program. **62**(3), 253–286 (2006)
38. O'Hearn, P., Yang, H., Reynolds, J.C.: Separation and information hiding. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 268–280. ACM, Berlin (2004)
39. OMG.: MDA Guide, Version 1.0.1, Jun 2003
40. OMG.: OCL Specification, Version 2.2. Formal/2010-02-01 (2010)
41. OMG.: Unified Modeling Language Specification, Version 1.5. Formal/03-03-01 (2003)
42. OMG.: Unified Modeling Language Specification, Version 2.2. Formal/2009-02-02 (2009)
43. Parnas, D.: On the criteria to be used in decomposing systems into modules. Commun. ACM **5**(12), 1053–1058 (1972)
44. Richters, M.: A Precise approach to validating UML models and OCL constraints. PhD thesis, University Bremen (2002)
45. Rumbaugh, J., Jacobson, J., Booch, G.: The Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley, Boston (2004)
46. Schoeller, B.: Eiffel0: An Object-Oriented Language with Dynamic Frame Contracts. Technical Report Nr. 542, ETH Zurich (2006)
47. Schubert, L.: Monotonic solution of the frame problem in the situation calculus. In: Kyburg, H., Loui, R., Carlson, G. (eds.) Knowledge Representation and Defeasible Reasoning, pp. 23–67. Kluwer, Dordrecht (1990)
48. Szyperski, C.: Component Software, 2nd edn. Addison-Wesley, Harlow (2002)
49. Spivey, J.M.: The Z Notation: A Reference Manual, 2nd edn. Prentice-Hall, Upper Saddle River (1992)
50. Warmer, J., Kleppe, A.: Object Constraint Language: Getting Your Models Ready for MDA. Addison Wesley Professional, Boston (2003)
51. Wirsing, M.: Algebraic specification. In: Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B. Formal Models and Semantics, pp. 675–788. Elsevier (1991)
52. Yu, Y., Manolios, P., Lamport, L.: Model Checking TLA+ Specifications. In: Pierre, L., Kropf, T. (eds.) Correct Hardware Design and Verification Methods (CHARME'99). LNCS, vol. 1703, pp. 54–66. Springer, Berlin (1999)

Author biography

Piotr Kosiuczenko studied mathematics and obtained PhD in this subject from the University of Warsaw in 1995. Afterwards, he joined the University of Munich. In Munich, Dr. Kosiuczenko was working also as a research collaborator at the FAST software company. Since October 2003, he worked as a lecturer at the Department of Computer Science, University of Leicester, UK. In 2008, he joined the Institute of Information Systems at WAT in Warsaw. Currently he is an associate professor there. His research interests include software engineering, graphical modelling languages, contractual specification, formal methods and foundations of object-oriented programming.