Specification of Invariability in OCL

Piotr Kosiuczenko *
Department of Computer Science
University of Leicester
piotr AT mcs.le.ac.uk

Abstract. The paradigm of contractual specification provides a transparent way of specifying systems. It clearly distinguishes between client and implementer obligations. One of the best known languages used for this purpose is OCL. Nevertheless, OCL does not provide primitives for a compact specification of what remains unchanged when a method is executed. In this paper, problems with specifying invariability are listed and some weaknesses of existing solutions are pointed out. The question of specifying invariability in OCL is studied and a simple but expressive and flexible extension is proposed. It is shown that this extension has a simple OCL based semantics.

1 Introduction

Contracts are the prevailing way of specifying systems from the client point of view (see [10]). They clearly assign responsibilities to client/caller and to system implementer/callee. They allow one to trace back a contract violation to the corresponding party. Unfortunately, the current high-level object-oriented specification languages, such as OCL [16], do not provide primitives to specify what can and what must not be changed when a method is executed. OCL allows explicit comparison of object attributes before and after method execution. A method execution usually changes only a small part of a system and consequently most of the system remains unchanged. In the case of large systems, it is not feasible to specify what happens with all attributes and associations. This problem is not restricted to object-oriented specification languages (see [2] for an overview). In general there exist three approaches to this problem: axiom frames, modifies clauses and nonmonotonic logics.

The axiom frames are used in artificial intelligence (cf. [11, 17]). The idea is to specify modification of attributes using axiom schemata. It requires explicit listing of all attributes which remain unchanged. This results in large number of frame formulas. In principle, it is possible to specify invariable system parts correctly, but of course it is error prone and not feasible in the case of large systems.

* To appear in O. Nierstrasz et al. (Eds.): MoDELS'06, Springer, LNCS 4199, 2006.

The second approach dates back to Hoare logic [7]. In this logic all variables which are not mentioned in the formulas of a Hoare triple are assumed to be unchanged. This works fine for verification of procedural programs, since all variables used in a procedure are plainly specified. However it does not work well for object-oriented specifications because of the encapsulation principle, which allows hiding private attributes of objects, and because of the fact that a method execution can have very complex side effects. In particular, it may result in changes to objects different from method's parameters. Java Modelling Language (JML, see [3] and the references there) provides compact specifications of invariable parameters [12]. It allows one for static checking of invariability properties. On the other hand, it is possible to specify invariability requirements, which cannot be check statically and in general the problem of what remains unchanged is undecidable.

The third approach uses nonmonotonic logics (see [17, 9] and the references there). It provides compact specifications and allows one to deal with side effects, but it is not appropriate for large specifications due to complex fixed-point semantics. The problem is that one specification may result in several fixed points and the number of such points may be high in the case of a large specification [9].

There exist an approach which relies on a completion procedure [2]; basically the specifier must specify for every method and every predicate the circumstances under which the predicate changes its truth value. Unfortunately, in the case of large systems it is not feasible. Interestingly, there exists also an approach allowing extending graph rewriting rules with invariability constraints [1].

Basically, the above mentioned approaches fall into two categories. Either they specify the system parts which don't change (frame axioms) or they specify the islands of change (JML, Hoare logic, nonmonotonic logics, design by contract advocated by Meyer). The problems with invariability specification can be classified as follows:

- oversize - huge formulas
- non-scalability - inability to deal with large specifications
- inflexibility - the user cannot customize the approach to specific needs
- fragility - the resulting formulas must be modified after every system change
- over-specification - the specification exposes details, which should be hidden

The need of extending OCL with primitives for specifying invariability has been recognized long time ago. For example, a working group was set to deal with this problem at "The Constraint Language for UML 2.0" workshop (a satellite workshop of UML'01 conference in Toronto).

OCL is a very expressive, high-level language for specification of object oriented systems [15] (see also [18]). There are tools for monitoring the satisfaction of OCL constraints (cf. e.g. [5]). This language can be used directly to specify what cannot change, but such specifications are usually very extensive, fragile, hard to understand and modify. What we need is a compact way of localizing change, with simple and monotone semantics.

In this paper, we propose a simple extension of OCL allowing us to specify invariability in a compact way. We delimit the islands of changes using appropriate primitives and we translate those primitives into "standard" OCL. Views proved to be a very powerful mean of specification and presentation (cf. [4, 13]). There are different specification styles as there are different oo-programming styles. A specification can be written from the client or from the implementer point of view; it can be restricted to a single component or package. Proposed extension allows us to specify systems from different points of view. In our approach, the specification of invariable part can be restricted to the appropriate view. With the help of the UML metamodel [16], we define the notion of view in the UML framework and restrict specification of invariability to views. The OCL formulas defining the user views may be sophisticated, but it is possible to define them in a generic way and to reuse them. One can also define a view corresponding to the implicit invariability assumption as it is used for example in Eiffel [6].

We study the usefulness of this extension in a series of examples and explain in which way it addresses the above mentioned problems. We show how to translate expressions containing invariability primitives into OCL. Thanks to this translation, our proposed extension has well defined semantics.

The paper is organized as follows. In Section 2, we consider a simple example and use it to explain problems with invariability specification; we indicate also a possible solution. In Section 3, we relate our extension to the UML metamodel and show how to define views. In Section 4, we present the formal syntax of proposed extension. In Section 5, we present the OCL based semantics of the extension. Section 6 concludes this paper.

2 Specification of Invariability

In this section, we consider a simple example of a bank account and explain problems with specification of invariability. We show how to specify invariability using a rather basic OCL extension, how to deal with inheritance and side effects.

2.1 Problems with Invariability Specification

Design by contract is a very powerful method of specifying class and component behaviour (cf. e.g. [10]). Unfortunately this approach may cause problems when a high level specification language such as OCL [15] is used.

Let us consider the class diagram shown on Fig. 1. We can specify the method *credit* in OCL in the following way:

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 *os*, nor to the attribute x. Therefore we have to add the following frame formula:

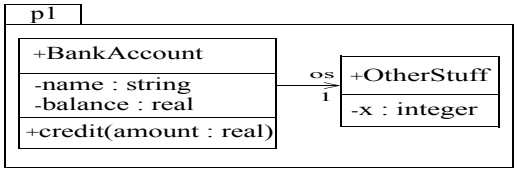

Fig. 1. Basic Class Diagram.

$and\ self.name = self.name@pre$
$and\ self.os = self.os@pre$
$and\ self.os.x = self.os@pre.x@pre$

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, i.e. all their attributes remain unchanged. This requires a separate equation for every attribute and association-end. Clearly in the case of larger systems, writing all such axioms results in large formulas. Such formulas are fragile in respect to modifications. It is easy to omit something or to add an erroneous constraint. Let us point out that this problem is not OCL specific and occurs in other object-oriented languages such as Eiffel (cf. e.g. [8]).

One of the possible solutions to the frame problem is to use the implicit invariability assumption. In simplistic case, this assumption says that all what is not specified to change does not change (see for example [10, 8]). It allows one to write simple specifications. The implicit approach to invariability is appealing, since it does not put an extra burden on the specifier. Nevertheless, it is not always clear what that assumption really means. In fact, the implicit invariability assumption seems to implicitly include some best practices used to specify object-oriented systems.

Literal interpretation of that assumption is problematic when a high level specification language such as OCL is used. Let us consider the OCL expression $self.os.x = self.os@pre.x@pre + 1$. It does not explicitly say whether $self.os$, x, or perhaps both have to change. It is only clear that at least one of those properties is supposed to change. The solution could be for example to say that all objects mentioned in a post-condition are allowed to change. However in such a case, logically equivalent formulas may have different meaning. In particular, adding a tautological expression to a constraint may change its meaning. Let us consider the following tautology:

$$OtherStuff.allInstances{-}{>}forAll(o \mid not\ o.oclIsNew()\ implies\ o.x = o.x@pre\ or\ not(o.x = o.x@pre))$$

That assumption would allow arbitrary change of x, despite the fact that this formula is a tautology. This disallows the use of logical deduction, since in logic tautologically equivalent formulas are semantically equivalent.

In the case of derived attributes, one does not specify what happens to them when a method is executed, since their values are derived from values of other attributes. But if the implicit approach is interpreted literally, then they should not change even if the values of the corresponding attributes change. Similarly, specification of subclasses causes problems, which can be hardly dealt with by the simplistic interpretation of the invariability assumption.

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. Often, clients access component functionality via so called facades, i.e. a number of selected classes and methods, but don't have any knowledge about other classes. For example let us assume that we want to save the old value of attribute *balance* of the class *BankAcount* whenever it is changed and that this operation should be invisible to the client. The values of the attribute *balance* can be saved in a class which is not navigable from the class *BankAccount* (see Section 2.4). The assumption that all objects mentioned in a clause can be modified would disallow that kind of logging unless the changes were specified explicitly. However this would force exposition of information, which should be hidden. All those issues are dealt with using best practices which emerged over years of experience in specification and implementation of object-oriented systems. Unfortunately their solution can not be simply derived from the simplistic assumption.

2.2 Solution in the Simple Case

In this subsection, we propose a solution for the case of single classes and packages. In UML, packages are used to group model elements. They can be used to define system views, in particular so called facades [16], which play the role of client window on the system. It is natural to restrict a client side specification to the corresponding facade.

In the case of the bank account (see Fig. 1, Subsection 2.1) we need to specify what can and what must not change. In our approach we restrict the specifications to packages and to sets of model elements in general (see below). We use the **in** keyword to indicate the package. The **modifies** clause specifies variable object attributes.

Let us specify explicitly what changes in the package $p1$. The following formula relativizes the specification to $p1$, more precisely to all properties contained in this package. 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 object *self* can be modified. The clause **in** $p1$ **modifies** : $self$::$balance$ says that if we restrict our view to the package $p1$, then an execution of the method *credit* can change only the value of the attribute *balance* of the actual implicit parameter. This specification focuses entirely on package $p1$ and does not say anything about any other package.

2.3 Inheritance

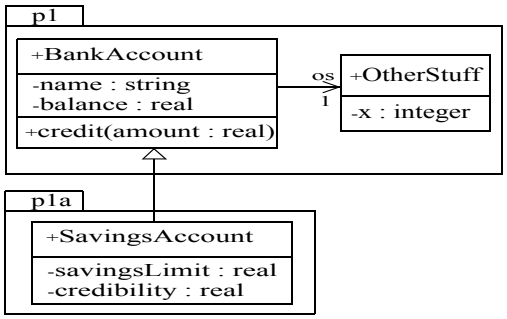

Fig. 2. Extra Package Extension.

In this subsection we deal with the problem of specifying invariability in the presence of inheritance. We investigate to what extent we need to change a specification, if a class is sub-classed.

Let us consider Fig. 2. 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 *credibility* specifies the credibility of a client. We assume that the second attribute is correlated with the balance; if for example the balance grows, credibility grows as well. The previous specification does not say anything about the behavior of the attributes *savingsLimit* and *credibility* when the method *credit* is executed. Consequently, they can change arbitrarily. To restrain changes in respect to the package $p1a$, we have to specify them explicitly:

context $p1$::$BankAccount$::$credit(amount : real)$
in $p1a$ **modifies** : $(if\ self.isKindOf(SavingsAccount)\ then$
$self.oclAsType(SavingsAccount)\ else\ Set\{\}\ endif)$::$credibility$

Let us point out that unlike Java, OCL requires that every *if* keyword has to be followed by *else* and end up with *endif*. In this case, the *else* part is just an empty set.

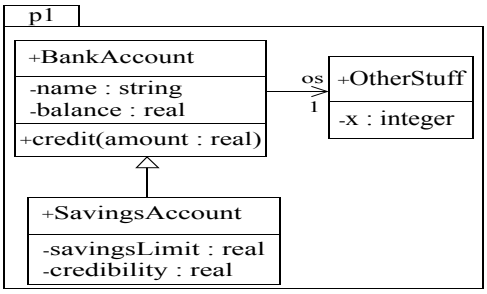

Fig. 3. Intra Package Extension.

The specification of invariability is stable in respect to extensions, which do not change the corresponding view (the package $p1$, for example), but changes may be necessary, if the view is modified. Indeed, Fig. 3 shows another way of extending the *BankAccount* class. In this case, the view given by package $p1$ is changed. We have to change the specification of *credit*, since it was done relatively to the view defined by $p1$.

context $p1::BankAccount::credit(amount : real)$
post : $self.balance = self.balance@pre + amount$
in $p1$ **modifies** : $self::balance$, $(if\ self.isKindOf(SavingsAccount)$
$then\ self.oclAsType(SavingsAccount)\ else\ Set\{\}\ endif)::credibility$

When specifying a method in a class, which is meant to be subclassed and which forwards method calls to other classes, it is a good specification style to abstract from changes the method has on attributes in subclasses and in the delegatee classes. In the case of our notation, it is possible to restrict a specification to a particular class. The following specification restricts the view to the class *BankAccount* only.

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

2.4 Side Effects

A method execution may result in modification of objects different from method parameters and their immediate neighbors. It may also modify attributes, which are invisible in a certain view. For example, this is usually the case of method logging. When aspect-oriented programming is used, it is possible to change attributes, which are not navigable from methods parameters. In this subsection we show how to deal with side effects.

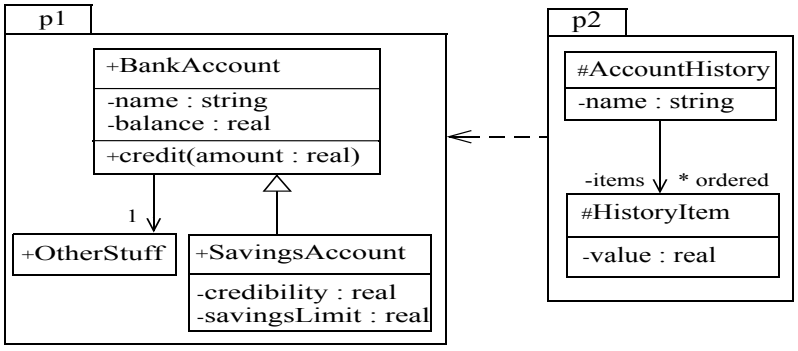

Fig. 4. Dependent Packages.

Fig. 4 shows the class *AccountHistory*. An object of this class stores information about the history of a bank account object. When the method *credit* is executed and when 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*. In the previous subsection, we have shown how to specify changes in respect to the package $p1$. However we may also need to specify a system internal view, which includes package $p2$:

$$
\begin{aligned}
&\textbf{context}\ p1{::}BankAccount{::}credit(amount : real)\\
&\textbf{post}\ : self.balance = self.balance@pre + amount\ and\\
&\qquad AccountHistory.allInstances{->}forAll(o \mid o.name = self.name\\
&\quad implies\ o.items{->}one(hi \mid hi.oclIsNew()\ and\ hi.value = self.balance@pre\\
&\qquad\qquad and\ o.items = o.items@pre{->}including(hi)))\\
&\textbf{in}\ p1\ \textbf{modifies}\ : self{::}balance,\ (if\ self.isKindOf(SavingsAccount)\ then\\
&\qquad self.oclAsType(SavingsAccount)\ else\ Set\{\}\ endif){::}credibility\\
&\textbf{in}\ p2\ \textbf{modifies}\ : AccountHistory.allInstances\\
&\qquad\qquad {->}select(o \mid o.name = self.name){::}items
\end{aligned}
$$

The OCL expression *one* means that there is exactly one object satisfying the corresponding condition. $including(hi)$ means that the object hi is appended to the end of the sequence *items*. The last clause restricts the changes in package $p2$ to the attribute *items* of the history objects, which have the same name as the credited bank account.

We may want to make sure that the method does not change anything more than specified above. To achieve this, we use the construct **modifies only**. The expression **modifies only** : $p1{::}*$, $p2{::}*$ specifies that the changes are restricted to packages $p1$ and $p2$. That sentence seals the specification of variable parts. It uses the absolute **modifies only** clause which concerns all properties of a model.

2.5 Specification of Operations on Lists

In this subsection, we show how to specify operations on 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 invariability specification tends to be left out.

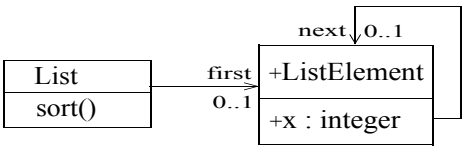

Fig. 5. List with an Anchor.

The class diagram in Fig. 5 shows a list composed of an anchor object of class *List* and a number of elements instantiating the class *ListElement*. The method *sort* is meant to sort lists according to the value of attribute x. We assume that $self.elements$ denotes the set of all elements of the list *self*. (We skip the definition of *elements*.) We consider here only finite acyclic lists. This constraint is expressed by an invariant saying that a nonempty list must contain an element, which does not have a successor. We use the term $elements@pre$ to denote all list elements, which exist in the pre-state.

context *List* **inv** :
$elements{->}notEmpty()\, implies\, elements.exists(el \mid el.next{->}isEmpty())$
context $List::sort()$
post : $self.elements = self.elements@pre\, and$
$self.elements{->}forAll(el \mid el.next{->}notEmpty()\, implies\, el.x <= el.next.x)$

We can make that specification precise by adding the following two invariability clauses:

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

The first clause says that the element associated to the list anchor can be replaced. 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.

3 Views

There are different specification styles as there are different oo-programming styles. In the preceding sections we have restricted our specifications to packages and classes. In general, it is possible to tune a specification to specific needs. A specification can be written from the client or from the implementer point of view. It may focus for example on public or reachable model elements. In general, a user may construct his/her own view. We introduce an abstract concept of view, which defines the focus of a specification (cf. [4]). In our approach, the specification of invariable part can be restricted to the appropriate view. The first subsection relates the OCL extension to the UML metamodel. The second subsection investigates in which way users may define their own views.

3.1 Relation to the UML Metamodel

UML metamodel [16] allows us for a precise definition of a view. The basic views are defined by packages. A package is a grouping of model elements. It owns and imports classes, other packages and model elements such as properties. Client's view of a system is often defined by a facade. In UML a facade is just a package [16].

Let us observe that the **in modifies** clause is defined on two levels of abstraction. The **in** part is defined on the level of class diagrams and the **modifies** part is defined on the level of objects. The **in** part refers to class diagrams and it is not fine enough to deal with run-time configuration. The **modifies** part on the other hand is defined in terms of the **in** part but concerns objects.

The **in** p **modifies** clause refers to a number of model elements grouped in a package p. According to the UML metamodel, a class and more generally a classifier is composed of behavioral features (in particular methods and attributes). It is also associated to association-ends. The following OCL expression defines in the context of the UML metamodel all OCL-properties contained in a package. It selects all properties owned or imported ($ownedElemens$, $importedElements$, respectively) by the package p.

$$
\begin{aligned}
&p.ownedElemens{->}select(pr \mid pr.isKindOf(StructuralFeature)\ and\\
&\qquad (pr.isKindOf(Operation)\ and\ pr.oclAsType(Operation).isQuery\ or\\
&\qquad\quad pr.isKindOf(Attribute)\ or\ pr.isKindOf(AssociationEnd)))\\
&{->}union(\\
&p.importedElements{->}select(pr \mid pr.isKindOf(StructuralFeature)\ and\\
&\qquad (pr.isKindOf(Operation)\ and\ pr.oclAsType(Operation).isQuery\ or\\
&\qquad\quad pr.isKindOf(Attribute)\ or\ pr.isKindOf(AssociationEnd))))
\end{aligned}
$$

This OCL formula demonstrates that the content of packages can be defined in the metamodel by OCL terms. Similarly, one can defined all properties corresponding to a class (cf. subsection 2.3).

3.2 User defined Views

The notion of view is fundamental for this approach. One can use predefined views provided by packages, however one may want to define own views corresponding to different perspectives. For example, a specification can be restricted to public or protected model elements. In fact, we can select an arbitrary set of model elements using an OCL term defined on the meta-level. It allows us to specify different system views and to express what is mutable and what is not. For example, for each class one can specify a view corresponding to all classes which are navigable from that class and restrict the invariability constraints only to that view. One can also explicitly define a view corresponding to the implicit invariability assumption including the best practices used in this approach.

Let us consider Fig. 4 again. We can define different views depending on the visibility of model elements. It is possible to restrict views to public or to protected model elements. Let us assume that for every attribute a there is a corresponding query method $getA$ returning the value of the attribute a and that this method has the same visibility as its class. If we focus on the behavior of public and protected properties, then the corresponding view contains the following queries: $getBalance$, $getSavingsLimit$, $getCredability$, $getName$, $getValue$ and so on. A restriction to public properties would remove $getValue$ since it is a method of the protected class $HistoryItem$.

In Subsection 2.3, we have shown how to deal with the specification of subclasses in a package. Actually, it is inelegant to specify what happens to subclasses at the level of their superclass. Let $p\,but\,subclasses$ mean all model elements which occur in package p, but are not a part of a subclass of the context class. This set can be defined by an OCL term. Due to lack of space, we skip the formal definition of this construct. The constraint specifying the method *credit* can be then written in the form:

context $p1::BankAccount::credit(amount : real)$
post : $self.balance = self.balance@pre + amount$
in $p1\,but\,subclasses$ **modifies** : $self::balance$

This clause relativizes the immutability clause to classes, which do not subclass the class $BankAccount$. In this case, every class subclassing that class requires its own contract.

In some cases it may be reasonable to restrict method specification to classes, which are navigable from the method parameters via association-ends and generalization relationships traversed bottom up, since only objects of those classes can be modified during a method execution. It is possible to define the set of navigable properties, though the corresponding OCL formula would be quite large. Such a specification can have the form:

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

where $typesOfParams(Op)$ is the list containing parameter types of method Op, i.e. $C, C_1, ..., C_n, D$. We assume that the term $navigableFrom$ denotes all properties owned by classes navigable from those types; as in the previous case we skip the definition.

In our opinion, a general specification language should not restrict users to a particular view, such as for example $navigableFrom$. In contrary, a user should be free to define own views as suits him/her best. The OCL formulas defining on the meta-level the user view may be sophisticated and therefore hard to write and hard to understand, but it is possible to define them in a generic and reusable way.

4 Extension's Grammar

In this section we define the syntax of proposed OCL extension. We restrict this syntax with some constraints, which cannot be expressed by a context free grammar. The grammar is presented using the EBNF notation: [ ] means optional occurrence, { } means arbitrary number of repetitions and | means option. We use capital characters for nonterminals and small characters for terminals. The invariability constraints have the following form:

context $C :: OP$
pre $: Pre$
post $: Post$
{ **in** P **modifies** $: M$ {, M }}
[**modifies only** : [P::] M{, [P::]M}]

C is a context specification, Op is a method signature, Pre is a pre-condition and $Post$ is a post-condition as defined by OCL [15]. M describes what can change and P is a package or more generally a term specifying a view. Furthermore:

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

Pn is a package name. The terminal **r** is optional; it specifies all sub-packages, like $-r$ in Unix. Cn is a class name. Mt is an OCL term defining a set of OCL-properties; Mt is defined on the class diagram level. O specifies visibility of considered properties; the visibility can be private, public, package public and protected respectively. We allow the use of multiple visibility predicates meaning that all listed options are possible. **nothing** is a terminal specifying that nothing can change. T is an OCL term defining a collection of objects; it is defined at the object level. Pr is an attribute or an association-end. $*$ denotes all OCL-properties. Let us point out that terms such as $p1\,but\,subclasses$ correspond to the nonterminal P (cf. Subsection 3.2).

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

in p **modifies** : $t_1::a_1, ..., t_m::a_m$

the term t_i, for $i = 1, ..., m$, must be valid in the corresponding context, that it does not contain the primitive $@pre$, that all objects defined by t_i must have property a_i and that a_i is a property of a class belonging to p, if p is a package, and that a_i is defined by p, if p is a term.

To facilitate the localization of changes we use the symbol $*$. $C::*$ means all properties of class C. Similarly, $p+::*$ means all public properties contained in the package p. We write **modifies only** : $p_1::*, ..., p_n::*$ to specify that only properties contained in packages $p_1, ..., p_n$ can be modified. Similarly, **modifies only** : $C::*$ specifies that only properties of class C can be modified.

5 The Semantics

In this section we define the semantics of invariability clauses. We discuss the OCL primitive $allInstances$ and its role in the semantics. This semantics allows us to translate invariability primitives to standard OCL. However translating even a medium size class diagram may result in a huge OCL formula. A language can have several semantics; one can modify the semantics proposed below by a proper tuning of the OCL translation. The advantage of this semantics is that one can rely on existing formal semantics of OCL and use standard OCL tools (cf. eg. [5]).

In our semantics, we need to relate sets of objects, which exist before method execution to sets of objects, which exist after method execution. There are two OCL primitives, which can be used for that purpose: $allInstances$ and $@pre$. $allInstances$ is a predefined feature of each type, which results in the set of all instances of the type in existence at the time when the expression is evaluated (c.f. [15], Subsection 7.5.10). In the case of program execution, $C.allInstances$ can be interpreted as the set of all objects of class C, which can be navigated from variables present in the program stack at a given moment of time.

Below we will use $C.allInstances@pre$ in post-conditions to refer to all instances of class C, which exist at the moment when the underlying method is invoked. Interestingly, $allInstances@pre$ is rarely used in specifications, though its meaning is as clear as the meaning of $allInstances$ itself. In general, OCL allows us to use properties in invariants, pre- and post-conditions. A feature is a property, like operation or attribute, which is encapsulated within a classifier. Actually, the OCL standard (c.f. [15], Subsection 7.5) restricts the notion of property to queries, attributes and association-ends "for the purpose of this document". We refer to the restricted notion of property as OCL-property. Interestingly, the OCL grammar doesn't restrict the use of $@pre$ to OCL-properties. On the other hand, it is common to use the feature $allInstances$ in invariants and post-conditions.

The semantics is defined via frame formulas. Initially we define the semantics of constraints of the form:

context X::Op
pre : Pre
post : $Post$
in p **modifies** : t_1::$a_1, ..., t_m$::a_m

We assume that $a_1, ..., a_m$ are attributes and association-ends, but not queries. Moreover for simplicity we assume that packages, classes and properties have unique names.

The term p is obtained from the nonterminal P and defines a number of OCL-properties (see section 4). We define an invariability formula for every attribute and every association-end belonging to p. There are two cases. Such a property may belong to the sequence $a_1, ..., a_m$ (i.e. it may have the form a_i); in this case the term t_i defines the scope of change of property a_i during execution of Op. In the other case, the attribute or the association-end cannot change. Let us notice that comparing the value of a property before and after method execution makes sense only for objects, which exist before and after operation execution.

More precisely for $i = 1, ..., m$, let t_i be an OCL term defined in the context X::Op, which defines a set of objects of a class C_i. We assume that t_i does not contain @pre. Let a_i be an attribute or association-end of the class C_i. We assume also that the properties a_i are pairwise different; because if a_i is equal to a_j, then we can consider $(t_i$->$union(t_j))$::a_i. Let $b_1, ..., b_n$ be all attributes and association-ends defined by p, which are different from properties $a_1, ..., a_m$. For $j = 1, ..., n$, let B_j be the class corresponding to the property b_j. Let t@pre denote a term, which is obtained from the term t by suffixing all OCL-properties by @pre. We translate the above constraint to standard OCL as follows:

context X::Op
pre : Pre
post : $Post$ *and*
$C_i.allInstances$@pre->$intersection(C_i.allInstances)$->$forAll(o \mid$
t_i@pre->$excludes(o)$ *implies* $o.a_i$@$pre = o.a_i)$, for $i = 1, ..., m$, *and*,
$B_i.allInstances$@pre->$intersection(B_i.allInstances)$->$forAll(o \mid$
$o.b_j$@$pre = o.b_j)$, for $j = 1, ..., n$

The resulting post-condition is a conjunction of the original post-condition $Post$ and a frame formula. The frame formula has two parts. The first one identifies OCL-properties, which may change. For $i = 1, ..., m$, the term t_i defines the scope of change of property a_i. The corresponding clause means that for every object o of class C_i, which exist before and after execution of Op, if o is not defined by t_i in the pre-state, then the property a_i of o remains unchanged. The second part concerns all other OCL-properties defined by p; it says that for every such property b_j and every object o of the corresponding class B_j, if o exists before and after execution of Op, then its property b_j cannot change. Let us point out that the term t_i can include the implicit parameter *self* and other parameters of Op.

Let us observe that the resulting post-condition does not exclude creation or deletion of new objects, as far as properties of objects existing before and after method execution conform to above mentioned constraints.

For example, let us consider the specification of method *credit* in subsection 2.4. There is no pre-condition in this case. In the case of package $p2$, the change is restricted to association-end *items* of those account histories, which correspond to *self*. More precisely, it is restricted to those objects o of class $AccountHistory$, which exist before and after operation execution and which have the same name as *self* in the pre-state: $o.name@pre = self.name@pre$. According to the first part of the frame formula, $o.items = o.items@pre$ must hold for every object o of class $AccountHistory$, such that o exists before and after method execution and o's name is different from the name of *self*. The post-condition says that the method appends a new object to the end of the associated sequence of items. The attribute *value* does not occur in the modifies-clause. Therefore according to the second part of the frame formula, for every object o of class $HistoryItem$, which exists before and after operation execution, it must be true that $o.value = o.value@pre$. However as stated above, this does not disallow proper initialization of the attribute *value* in the newly created objects. The case of attribute *name* is similar to the case of *value*.

Other kinds of invariability clauses can be treated as abbreviations. In the case of the absolute invariability clause **modifies only** : $t_1::a_1, ..., t_m::a_m$, the localization of changes is not relativized, but concerns all properties. This kind of constraint can be seen as an abbreviation of **in** ap **modifies** : $t_1::a_1, ..., t_m::a_m$, where ap defines all OCL-properties in a model.

We have mentioned that it is possible to use $*$ as an abbreviation for any property. Formally, the clause **modifies only** : $p::*$ means that for any OCL-property a, which is not defined by p and for the corresponding class C the following holds:

$$C.allInstances@pre->intersection(C.allInstances)\\ ->forAll(o \mid o.a@pre = o.a)$$

The relative expression **in** p **modifies** : **nothing** means that no property contained in p is modified. It can be equivalently expressed by the formula **in** p **modifies** :, which uses an empty list of terms.

6 Conclusion

Specification of invariability in OCL has been a long standing problem. OCL extension proposed in this paper provides a solution to that problem. The UML metamodel and OCL allow us for an elegant definition of the notion of view; this notion proved to be essential for specification of invariability. Interestingly, OCL turned out to be proper language to define the semantics of proposed extension. There are only few invariability primitives with simple semantics expressed in terms of OCL itself; so that the invariability clauses can be understood as merely OCL macros. Consequently the existing OCL tools can be used.

In the future we are going to perform a realistic case study to demonstrate scalability of our extension. We are going to develop methodology for specification of invariability. On the other hand, we are going to implement a tool for automatic generation of OCL constraints from the invariability clauses and to integrate this tool with existing OCL tools. The notion of view proved to be very flexible and powerful; we are going to study its applicability for layered modeling of complex systems.

Acknowledgement. We would like to thank the anonymous referees for their helpful comments, which helped us to improve this paper.

References

1. Baar, T., *OCL and Graph-Transformations - A Symbiotic Alliance to Alleviate the Frame Problem*. Proc. of MoDELS'05 Satelite Workshop on Tool Support for OCL and Related Formalisms, Montego Bay, Jamaica, October 4, 2005, pp. 83-99, 2005.
2. Borgida, A., Reiter, R. and Mylopoulos, J., *On the Frame Problem in Procedure Specifications*. 15'th Int. Conf. on Software Engineering, Baltimore, IEEE Computer Society Press, 1993.
3. Darvas, A., Mueller, P., *Reasoning About Method Calls in JML Specifications*. Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP'05), Glasgow, Scotland, July, 2005.
4. Finkelstein A., Kramer J., Nuseibeh B., Finkelstein L., and Goedicke M., *Viewpoints: A Framework for Integrating Multiple Perspectives in System Development*. International Journal on Software Engineering and Knowledge Engineering, 1991, pp. 31 – 58.
5. Gogolla, M, Richters, M. *Use: A UML-based Specification Environment*. http://www.db.informatik.uni-bremen.de/projects/USE/.
6. Jezequel, J. M., *Object-Oriented Software Engineering with Eiffel*. Addison-Wesley, (Eiffel in Practice Series), 1996.
7. Hoare, T., *An Axiomatic Basis for Computer Programming*. CACM, 12(10), 1969.
8. Mitchell, R., McKim, J. *Design by contract by example*. Addison-Wesley, 2001.
9. Marek, W., Truszczynski, M., *Nonmonotonic Logic, Context-Dependent Reasoning*. Series: Artificial Intelligence, Springer, 1993.
10. Meyer, B., *Object-Oriented Software Construction*. Prentice, Hall, N.J., 1998.
11. Minsky, M., *A framework for representing knowledge*. Technical Report 306, Artificial Intelligence Laboratory, MIT, 1974.
12. Mueller, P., Poetzsch-Heffter, A., Leavens, G. T., *Modular Specification of Frame Properties in JML*. Concurrency and Computation: Practice and Experience, Volume 15, pp. 117–154, Wiley, 2003.
13. OMG, *MDA Guide*, Version 1.0.1, Jun 2003.
14. OMG, *Meta-Object Facility Specification*, Version 1.4, April 2003.
15. OMG, *OCL Specification, Version 2.0*. October 2004.
16. OMG, *Unified Modeling Language Specification*, Version 2.0, October 2004.
17. 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, Kluwer, 1990, pp. 23–67.
18. Warmer, J., Kleppe, A., *Object Constraint Language: Getting Your Models Ready for MDA*. Addison Wesley Professional, 2003.