MODIFIERS IN OFL - AN APPROACH FOR ACCESS CONTROL CUSTOMIZATION

Philippe Lahire, Dan Pescaru

Projet OCL

Rapport de recherche
ISRN I3S/RR–2003-16–FR

Juin 2003
RÉSUMÉ :
Le modèle OFL est un métamodèle pour les langages de programmation à objets. Il fournit un paramétrage des principaux aspects de la sémantique du langage au travers d’actions et de paramètres. Le paramétrage fourni concerne uniquement des fonctionnalités suffisamment générales pour être appliquées à la plupart des langages de programmation à objets. Des expériences montrent qu’il est nécessaire de prendre en compte plus de sémantique. Dans ce but, nous proposons d’ajouter de nouveaux éléments au modèle OFL original. Dans ce contexte, un point important est de fournir un paramétrage pour les mécanismes de contrôle d’accès.

MOTS CLÉS :
Modificateurs, OFL, contrôle d’accès

ABSTRACT:
The OFL Model is a meta-model for object oriented programming languages. It provides customization of main aspects of the semantics of a language through actions and parameters. The provided customization can deal only with features than are general enough for being applicable to most existing object-oriented programming languages. Further experience points out the necessity to capture more of the semantics of these languages. This goal is achieved by adding new elements to the original OFL Model. In this framework one important issue is to provide customization of accessing control mechanism.

KEY WORDS :
Modifiers, OFL, Access Control
Abstract. The OFL Model is a meta-model for object oriented programming languages. It provides customization of main aspects of the semantics of a language through actions and parameters. The provided customization can deal only with features than are general enough for being applicable to most existing object-oriented programming languages. Further experience points out the necessity to capture more of the semantics of these languages. This goal is achieved by adding new elements to the original OFL Model. In this framework one important issue is to provide customization of accessing control mechanism.

1 Introduction

The importance of a systematic approach on access control mechanism represents an actual topic of research in the field of object oriented technology [1–3, 13]. Even the UML standard [10], which was planned to be language independent, lacks in defining protection mechanisms. Flower and Scott emphasize this aspect [8]:

“When you are using visibility, use the rules of the language in which you are working. When you are looking at UML model from elsewhere, be wary of the meaning of visibility markers, and be aware how those meanings can change from language to language.”

OFL Model [6] provides customization of main aspects of the semantics of an object-oriented language, but also lacks in customization of access control mechanisms.

In order to preserve simplicity, a large part of the language reification is not customizable in the OFL Model philosophy. However, in order to achieve acceptance in programmers’ community, some other customizations are needed. Generally, this additional semantics is handled by keywords (modifiers) in existing languages.

One main goal of introducing modifiers is to limit the number of components within an OFL-language. Using modifiers we avoid necessity to define one different component for any different combination of parameters. For instance, is
better not to have both *public java-class* and *package java-class* components differentiated by a parameter *visibility*. Instead, we can imagine just one java-class component and something else (like modifiers) allowing ensuring that access is *public*.

Another goal of modifiers is to improve the flexibility at the level of metaprogramming by providing a clean way to extend a language with new capabilities.

Following those goals we pay a special attention not to change the general aspect of the OFL model.

In order to take these issues into account we propose to introduce at the level of language components the ability to define different kinds of *modifiers* and to add reification elements according to that.

OFL *modifiers* are used together with other language entities in order to change protection or semantics feature aspects. Some of them have an equivalent in keywords that may be found in some programming languages, others could be added in order to simplify programming task.

The aim of this paper is to propose a way for adding customization of access control mechanism at the level of the OFL Model. In Sect.2 we present an overview of the OFL Model. In Sect.3 we describe the main aspects of modifiers definition. In Sect.4 we present discussion and examples of basic modifiers. In Sect.5 we briefly discuss about dealing with complex modifiers. Finally we present conclusion and perspectives of our approach.

2 Overview of the OFL Model

OFL [4, 6, 7] is the acronym for Open Flexible Languages and the name of a meta-model for object-oriented programming languages based on classes. It relies on three essential concepts of them: the descriptions that are a generalization of the notion of class, the relationships such as inheritance or aggregation and the languages themselves. OFL provides a customization of these three concepts in order to adapt their operational semantics to the programmer’s needs. It is then possible to describe existing language entities and, in addition, to specify new kinds of relationship and description. Those new entities could be added to an existing programming language in order to improve its expressiveness, its readability and its capabilities to evolve.

Rather than allowing the redefinition of language behaviors thanks to intensive code writing, OFL propose a set of parameters and algorithms, already implemented, which take into account the values of these parameters to achieve the desired behavior. These algorithms are called actions and they define the operational semantics.

At first reading the OFL approach can be summed up as the search for a set of parameters whose value determines the operational semantics of an object-oriented language based on classes.
The model defines a set of parameters [6], which represents the main features of the three concepts behavior mentioned earlier: concept-relationship, concept-description and concept-language.

The operational semantics of each concept must be adapted to the value of its parameters. This is achieved due to a set of action's algorithms whose execution depends on these values. For example, the assignment of an object to an attribute, the dynamic binding of the features, the sending of messages and lots of other behaviors, are expressed according to parameters of concept-relationship and concept-description.

Several aspects of the language operational semantics are described by several sets of actions: primitive look-up actions (look-up...); primitive execution actions (attach-parameters, execute...); actions for managing descriptions instances (create-instance...); actions for managing descriptions extensions (calculate-extension...); control actions (verify-overloading,...) and basic actions (assign, copy...).

Figure 1 illustrates how to use the OFL Model to describe an application. The notation follows the UML convention. Three levels of modelling are shown:

1. the application level includes the program’s descriptions and objects (OFL-instances and OFL-data),
2. the language level describes the components of the programming language (OFL-components like ComponentJavaClass or ComponentJavaExtends), and
3. the OFL level represents the reification of those components (OFL-concepts and OFL-atoms).

The OFL atoms represent the reification of the non-customized entities of the model. The relationships, descriptions and languages have their own OFL atoms to describe the part of their structure and their behavior, which are not customized.

The OFL components inherit from atoms and represents reification of language entities (relationships and descriptions). Each component keeps a set of characteristics that represents meta-information for program entities (OFL instances) such as lists of attributes and methods for a description component or lists of redefined features for relationship components. The language itself is a component. It’s main function is to put together the relationships and descriptions which are supplied to the programmer.

In order to describe an application, the programmer uses the services supplied by the language level. He creates OFL-instances, which are the descriptions and the relationships of his application by instantiation of the OFL-components. At runtime, the application objects, called OFL-data, are instances of the OFL-instances representing the descriptions.

3 Main Aspects of OFL Modifiers

OFL-modifiers are designed to ensure better OFL customization for programming languages. Generally, modifiers imply constraints added to the application model in order to achieve a fine control.

Figure 2 illustrates the OFL Model extended with OFL-modifiers. We define three kind of modifiers for entities which support their semantics. These types are: description-modifier, method-modifier and attribute-modifier. The OFL modifiers components inherit from OFL-modifiers and represent reification of language modifiers.

Starting from the point that most of the modifiers relay on constraints to be applied to the program entities, we choose OCL as the language for specifying OFL modifier constraints. OCL is a formal language which allows to express side effect-free constraints. The Object Management Group (OMG) defines OCL (Object Constraint Language) [11] as a part of UML 1.3 standard specification. Main motivation regarding that choice is programming language independence of OCL and general acceptance of this language.

An OFL-modifier is defined by a modifier name, a context (an entity against it is defined), modifier assertions (OCL constraints) and a set of associated actions (modified OFL-actions).

Modifier Name. The name is used to identify the modifier. It should be a legal identifier related with OFL and the language binding.
**Modifier Context.** Type of entity that accepts the modifier is denoted by its context. Context could be description, relationship, attribute or method.

**Modifier Assertions.** We use OCL to specify the modifier constraints through assertions.

These constraints reside in *invariant* for OFL components or in *pre* and *post conditions* for OFL actions. Indeed, they will be attached to corresponding components and actions.

Implementation of access control implies assertions at the level of OFL entities reifying the corresponding control mechanisms.

For assertions we use notation that have the same meaning as in OCL definition [11]. The *self* keyword refers the current instance of the associated component.

The OCL modifier assertions are written in context of the OFL model definition; all types defined by the OFL model could be used in assertions.

Some component features correspond to OCL collection type and support OCL collection operators. For instance,

```
component.modifiers -> includes('modifier name')
```

that tests if the component has modifier 'modifier name' attached to it or not.

**Modifier’s Actions.** Modifier’s actions are OFL-actions rewritten to consider new semantics. The modifier keeps references to all rewritten action, helping meta-programmer to manage them. Actions play different roles depending of the complexity of the considered modifier. Most modifiers do not need action rewriting. They have just a set of assertions attached to them.

![Image](image.png)

*Fig. 2.* The extension of the OFL Model
In order to build a complex semantics from simpler ones and to extend modifiers, we define a modifier composition operator. This operator specifies how to combine assertions and actions that belong to composed modifiers. In the context of composition operation we state the definition of "compatible modifiers" and "incompatible modifiers". Two modifiers defined in the same context are compatible if they can be the parts of a composition. They are incompatible if their actions and assertions are not disjunctive. Actions and assertions are not disjunctive if their semantics interfere. According to that we extend the definition of OFL modifier by adding a characteristic named incompatible modifier set. One modifier keeps in this set information about all modifiers that are incompatible with it.

In the composition process, two aspects of modifiers are addressed: the assertions and the actions associated with it. For compatible modifiers all interactions will be just cumulative. For the assertions, which are OCL expressions, other constraints can be composed using the AND logical operator. Because OCL avoids side effects, composition of assertions is commutative. Actions will be called in a random order. Indeed, if there are some interactions at the level of action semantics, the modifiers are incompatible and the composition operator cannot be applied.

To deal with incompatible modifiers we define an invariant at the level of OFL entity representing the modifier context.

As an example we consider the Java public modifier for attributes. The OFL reification for an attribute is the AtomAttribute. We attach an invariant to this entity.

\[
\text{context AtomAttribute}
\]

\[
\text{inv: self.modifiers->includes('public')}
\]

\[
\quad \text{implies}
\]

\[
\quad \text{NOT (}
\]

\[
\quad \quad \text{self.modifiers->includes('private')}
\]

\[
\quad \quad \text{OR}
\]

\[
\quad \quad \text{self.modifiers->includes('package')}
\]

\[
\quad \quad \text{OR}
\]

\[
\quad \quad \text{self.modifiers->includes('protected')}
\]

\[
\quad )
\]

In order to cover all situations an invariant should be added for each modifier considered.

In the context of a language extension made by a meta-programmer we can distinguish two kind of modifiers. An OFL-modifier could represent the reification of a modifier that belongs to the language binding - we name it native modifier - or could be a custom modifier added by the meta-programmer in order to enrich language semantic.

The native modifiers will have the same meaning, related with the language binding components, like in the original language. The meta-programming task
will consist in describing the meaning and the behavior of modifiers according with their definition. When a meta-programmer adds new extension for the language (new components) he has the responsibility to extend the definition of the modifiers according to the new entities.

In the following sections we try to provide an orthogonal approach in order to define both native and custom modifiers.

**Basic Modifiers**

Some modifiers imply only constraints related with visibility of an entity (ex. public, private, protected in Java and C++). They do not address any special rights like writing/reading attributes, calling/redefining methods or extending/instantiating descriptions. Their implementation resides only in assertions at the level of components and look-up action. We name them basic modifiers.

**Complex Modifiers**

More complex modifiers provide access control at the level of operation performed (ex. final - for classes or methods in Java, frozen - for features in Eiffel etc.). All the time complex modifiers implies protection and some time they implies also visibility (ex. protected-write [5]) Usually they imply action writing.

4 Basic Modifiers

4.1 Examples of Native Basic Modifiers

For Java language [9] there are several modifiers used for access control: public, protected, private, and default(package). All of them do not refer any specific operation, do not make distinction between reading/writing an attribute, calling/redeclaring a method, extending/instantiating a class.

For C++ language [14] the public, protected and private modifiers has slightly different meaning as in Java [2]. It has no "package" resolution but has instead a special class of visibility denoted by friend. Using the friend keyword, a class can grant access to non-member functions or to another class. These friend functions and friend classes are permitted to access private and protected class members. The public and protected keywords do not apply to friend functions, as the class has no control over the scope of friends.

In Eiffel [12] there are two constructions that can deal with access modifiers; these are feature and export. In this language some of the protection semantics are hidden in the language philosophy. For instance, the writing protection has no direct meaning for an attribute because access to an attribute from outside class is considered as a query and it is not possible to write into a result of a query).

4.2 Basic Modifiers for Features

**Modifier Assertions.** The assertions of basic modifiers for features (attributes and methods) are defined at the level of OFL relationship components that
manage export of those features. They should be tested each time a relationship involving that feature is created. An invariant at the level of description that own the feature is not necessary. Basic modifiers do not protect features against the description itself. Independently of the language syntax we can consider three possibilities: the feature belongs to current class or is inherited through an inheritance relationship from a direct or indirect ancestor or it is accessed through a use relationship (current class is a client of description that owns the feature). In the last situation we consider that the current description could access supplier description. Indeed, this problem is covered by description’s access control. By current class we mean the class that accesses the feature.

If we consider the Java syntax, features belonging to a class or inherited by the class, are accessed using this keyword as qualifier. This keyword could be explicit or implicit (non-qualified features). Features accessed through a use relationship are explicit qualified with the supplier name. To consider all situations, an invariant is needed for every component of import relationship type and use relationship type defined for that language. The following example presents invariants for extends Java inter-class relationship and Java aggregation relationship.

*Java features basic modifiers: { 'public', 'protected', 'private', 'package' }*

```plaintext
context ComponentJavaExtends
inv: self.showedFeatures->forall(f:Feature | f.modifiers->includes('public') OR f.modifiers->include('protected'))
inv: self.redefinedFeatures->forall(f:Feature | f.modifiers->includes('public') OR f.modifiers->include('protected'))
inv: self.hiddenFeatures->forall(f:Feature | f.modifiers->includes('private'))
```

The invariant says that all showed and redefined features through an extend relationship should have modifiers public or protected attached. All hidden features have private modifier.

```plaintext
context ComponentJavaAggregation
inv: self.showedFeatures->forall(f:Feature | f.modifiers->includes('public') OR ((f.modifiers->include('package') OR f.modifiers->include('protected')) AND self.source.package = self.target.package))
inv: self.hiddenFeatures->forall(f:Feature | f.modifiers->includes('private'))
```

8
In addition to previous assertion, this one tests also information about description’s packages. Descriptions are accessed through ‘source’ and ‘target’ members of the relationship component instance (self).

**Modifier Actions** Interference with model actions is minimal. Assertions are added to control features access through relationships and no action rewriting is necessary. Indeed, modifiers do not redefine any actions.

4.3 Basic Modifiers for Descriptions

**Modifier Assertions.** The assertions of basic modifiers for descriptions are defined at the level of relationship components and at the level of description component itself. They should be tested each time a relationship involving that description is created and each time an instance of description is created. The last situation deals with relationships that enable polymorphism. Considering this, the invariant becomes a post-condition for look-up OFL actions.

The following example refers the Java language semantics for class access control. The example does not take into account primitive types, interfaces, abstract and inner classes.

*Java class modifiers:* { public, package }

context ComponentJavaExtends
inv: self.source.package = self.target.package
  OR
  ( self.source.package <> self.target.package
    implies
    self.source.modifiers->includes('public'))

A class can extend another class from the same package and a class can extend a public class from other package.

context ComponentJavaAggregation
inv: self.source.package = self.target.package
  OR
  ( self.source.package <> self.target.package
    implies
    self.source.modifiers->includes('public'))

The following assertion address dependencies between classes, which are not covered by OFL customization.

context Description::
  lookup(accessed: Description):Description
post: self.package = result.package
    OR
    self.package <> result.package
    implies
    result.modifiers->includes('public')

Modifier Actions  For those modifiers, assertions are also added to control features access through relationships. Post-conditions are used to filter the lookup action result. Modifiers do not redefine any actions.

5 Complex Modifiers

5.1 Examples of Native Complex Modifiers

Complex modifiers define protection at the level of special rights like writing/reading an attribute, calling/redefining a method or extending/instantiating a description.

Java language does not include complex access control modifiers for attributes. It includes one modifier for methods, final that disallow method redefinition. A modifier with same name in context of classes and interfaces is used to avoid extension. Other language mechanisms (like making all constructors private) could be used to control instantiation.

C++ does not provide any specific modifiers to control rights for using an entity. Changing access to constructor does control at the level of class instantiation like in Java.

Frozen modifier from Eiffel could be considered in this category. Frozen, appearing before the feature name, express that the declaration is not subject to redefinition in descendants.

5.2 Method Complex Modifiers

Rights concerning method usage address mechanisms like calling or redefining. Modifiers presented in the previous section do not make distinction between these mechanisms.

Modifier Assertions. Implementation of control implies assertions at the level of OFL entities reifying corresponding mechanisms. Redefinition mechanism is reified in OFL by redefinedFeatures characteristic of relationship components. Access control is done by invariant for these components. Calling mechanism is reified in execute action. Assertion concerning calling rights is implemented in a post-condition for this action.

The following example is an implementation of final modifier for Java methods.
context ComponentJavaExtends
inv: self.redefinedFeatures->forall(f:Feature |
  f.typeOfFeature = method
  implies
  NOT f.modifiers->includes('final'))

Final modifier is compatible with public, protected, package and private modifiers
and can be present in a composition. Its invariant will be added to the component
invariant.

Modifier Actions. Complex method modifiers some times require rewriting of
the execute OFL action.

5.3 Complex Modifiers for Attributes

Rights concerning attribute usage address control against reading or writing.
Protection on writing is achieved by a pre-condition at the level of assign action.
We can consider here a proposal of Cook and Rumpe [5] for defining a read-only
modifier for attributes. They conclude that is useful to constraint the visibility
of an attribute to be readable, but not changeable. The concept of a read-only-
modifier is introduced in combination with private and protected modifiers.

Modifier Assertions  Assertions for attribute complex modifiers resides in pre
and post conditions at the level of assign OFL action.

Modifier Actions  Necessity for action writing resides in complexity of considered
semantic.

As an example we consider a modifier that implements a heavy writing pro-
tection for an attribute. By heavy protection we mean to protect not only the
reference of the object against writing but also the internal state of the referred
object.

A solution that lacks in efficiency is to give access to a clone of the object
that contains attribute and to look after that if any changes appear. To ensure
this control, attribute access action should be embedded in the following code:

```java
// cloning the original object
aux = deep_clone(f)
// original action
// ( any kind of action that may imply changing
// of attribute’s internal state )
*action(aux)
// test if the object preserve same state
if (not deep_compare(f, aux) )
  generate_error("Could not write attribute")
end_if
destroy_object(aux)
```
Actions that permit changing of attribute’s internal state are considered the following OFL-actions: evaluate-parameters, attach-parameters, detach-parameters, assign, execute etc.

5.4 Complex Modifiers Descriptions

Description may be extended, used or instantiated.

**Modifier Assertions** Extension is controlled through invariant on inheritance relationship components. To control client-supplier relationship, invariant is attached to use relationship components.

As an example we consider the Java final modifier in context of a description. The invariant for Java extends relationship will check absence of this modifier at the level of target description of relationship.

```context ComponentJavaExtends
inv: NOT self.target.modifiers->includes(‘final’)
```

**Modifier Actions** For description modifiers, actions are necessary to control instantiation. Instead, most of the times a precondition at the level of create-instance action is enough to ensure all semantics.

6 Conclusion and perspectives

In this paper we proposed to extend the OFL Model. The main goal of this extension was to add customization of the access control mechanism. We introduced the notion of OFL modifier to provide a clean way for control implementation. For better understanding of the concept we present in sections 4 and 5 examples of several native modifiers reification.

As future work we proposed to add support for OFL modifiers and to integrate them in all OFL tools. We also plan to extend the modifiers with high level actions. The OFL modelling tool will execute these actions to ensure automatic model correction.

References