An Extension of OFL Model through Modifiers

Dan Pescaru, Pierre Crescenzo, Philippe Lahire
dan@cs.utt.ro
Pierre.Crescenzo@unice.fr
Philippe.Lahire@unice.fr

Faculty of Automatics and Computer Science,
"Politehnica" University of Timisoara,
Bd. V. Parvan no 2, 1900 Timisoara, ROMANIA,

Laboratoire I3S (UNSA/CNRS), Project OCL 2000,
Route de Lucioles, Les Algorithmes,
Btiment Euclide B BP121 F-06903,
Sophia-Antipolis CEDEX, FRANCE

November 10, 2003

Contents

Chapter 1

Introduction

OFL model provides a customization of main aspects of the semantics of a language through actions and parameters, but the customization provided can deal only with features that are enough general for being applicable to most existing object oriented programming languages[1]. Practical experience points out the necessity to capture more of the semantics of these languages. To achieve that it is necessary to add new elements to the original OFL Model [Cre01, CL02].

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 of main goal for introducing modifiers is to limit the number of components within an OFL-*language*. Using modifiers, it is not necessary anymore to define one different component for each different combination of parameters. For instance, instead of having both *public java-class* and *package java-class* components differentiated only by one parameter (*visibility*), we can design just one java-class component and something else (like modifiers) in order to ensure (when it is necessary) that the access is *public*.

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

According to that we propose a generic approach which allows to define rules for implementing access controls or additional semantics for language components. The general idea is to apply these rules to an application in order to provide for example metrics, error reporting, and design or debugging facilities. Thanks to these rules we can had constraints to language entities in order to enrich, when it is necessary, the expressiveness of a given language.

Comparing with other approaches found in [ACL03, Sch02], we focus on a generic technique independent from languages. Moreover, instead to define a

[1] For more information on the OFL Model, to read the thesis of Pierre Crescenzo [Cre01].

formalism which depicts access control mechanisms, we propose an approach that describes how to implement these mechanisms at a meta-programming level.

Following those goals we pay a special attention to the consistency of this approach with the OFL model philosophy.

Considering these issues we propose to add at the level of language components the ability to define different kinds of *modifiers* and to add the entities that are necessary to their reification.

OFL *modifiers* are used together with other language entities in order to change protection or other aspects of their semantics. Some of them correspond to keywords that may be found in existing programming languages, while others could be added in order to simplify some programming tasks.

Chapter 2

The OCL Language

Starting from the point that most of the OFL *modifiers* relay on constraints to be applied to program entities, we choose OCL as the language for specifying these constraints. OCL is a formal language which allows to express side effect-free constraints. The Object Management Group (OMG) defines OCL (Object Constraint Language) [OMG00] as a part of UML 1.3 standard specification. Main motivation regarding this choice are the independence of OCL from programming languages and its general acceptance within many communities.

OCL is designed to express side effect-free constraints. It was used by OMG in the *UML Semantics* document to specify the rules of the UML meta-model. Each rule in the static semantics sections in the UML Semantics document contains an OCL expression, which is an invariant for the involved class.

The usage of OCL is important because in object-oriented modeling a graphical model, like a class model, is not enough for a precise and unambiguous specification. There is a need to describe additional constraints about the objects in the model. Such constraints are often described in natural language. Practice has shown that this will always result in ambiguities. In order to write unambiguous constraints, so-called formal languages have been developed. The disadvantage of traditional formal languages is that they are usable to persons with a strong mathematical background, but difficult for the average business or system modeler to use.

OCL has been developed to fill this gap. It is a formal language that remains easy to read and write. It has been developed as a business modeling language within the IBM Insurance division, and has its roots in the Syntropy method.

OCL is a pure expression language. Therefore, an OCL expression is guaranteed to be without side effect; it cannot change anything in the model. This means that the state of the system will never change because of an OCL expression, even though an OCL expression can be used to specify a state change, for example in a post-condition. With other words, any fields of any objects, including links, cannot be modified. Whenever an OCL expression is evaluated, it simply delivers a return value.

OCL is not a programming language, so it is not possible to write program

logic or flow control in OCL.

OCL is a typed language, so each OCL expression has a type. OCL can be used for a number of different purposes:

- to specify invariant on classes and types in a class model,
- to specify type invariant for UML stereotypes,
- to describe pre/post conditions on operations and methods,
- to describe Guards,
- to specify constraints on operations,
- as a navigation language.

We use OCL to describe constraints introduced by modifiers. It can be also used to specify pre and post conditions for OFL-entities at the level of OFL-ML implementation.

As a notation convention for this document, the underlined word before an OCL expression determines the context for the expression and the OCL expression itself will be in italic.

In OCL, a number of basic types are predefined and available at any time: Boolean, Integer, Real, String and Enumeration. Several operations are also defined on these predefined types. In addition, all OFL-descriptions[1] coming from the OFL Model are types in OCL which are attached to the model.

The type Collection, which is predefined in OCL, plays an important role according to constraint definitions. It includes a large number of predefined operations for the handling of collections. Collection operations are consistent with the definition of OCL as an expression language, they never modify the contents of collections. They may return a collection, but rather than changing the original collection they put the result into a new one.

Collection is an abstract type, with the concrete collection types as its subtypes. OCL distinguishes three different collection types: Set, Sequence, and Bag. A Set is the mathematical set ; it does not contain duplicate elements. A Bag is like a set, which may contain duplicates, i.e. the same element may be in a bag twice or more. A Sequence is like a Bag in which the elements are ordered. Both Bags and Sets have no order defined on them. Sets, Sequences and Bags can be specified by a literal in OCL.

OCL defines a number of operators for collection manipulation:

- SELECT and REJECT - allows to specify a selection from a specific collection ;
- COLLECT - allows to specify a collection which is derived from some other collection, but which contains different objects from the original collection (i.e. it is not a sub-collection) ;

[1] An *OFL-Description* or *OFL-ComponentDescription* is the name chosen in OFL for *classifier*.

- FORALL - allows to specify a Boolean expression, which must be verified for all objects in a collection ;
- EXISTS - allows to specify a Boolean expression which must be verified for at least one object in a collection ;
- ITERATE - allows building one accumulation value by iterating over a collection. It is a very generic operation. *Reject*, *Select*, *forAll*, *Exists* and *Collect* can all be described in terms of *Iterate*.

Chapter 3

The OFL Modifiers

An intuitive definition of a modifier entity is the following: a *modifier* is a language keyword that is used in composition with other keywords in order to change their semantics. An important issue is that a modifier keyword have no stand-alone meaning.

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

Not all language modifiers are intended to be reified by OFL *modifiers*. Semantics changes induced by some of them are very deep and rely on several OFL components. We name them *component modifiers*. Following list presents modifiers for three well known object-oriented languages: Java [GJSB00], C++ [Str97] and Eiffel [Mey02].

3.1 Component Modifiers in Commercial Languages

The modifiers proposed in the following rely on several parameters or properties that are mentioned.

3.1.1 Java language.

abstract {class declaration} An *abstract* class is a class that is incomplete, or to be considered incomplete. The reification for a class declared *abstract* in Java results in several *OFL-ComponentDescription* for *abstract class*, *static abstract nested class*, *abstract inner class* and *abstract local class*. All these components have the parameters *generator* and *destructor* set to value *false*.

final {attribute declaration} A *final* attribute may only be assigned once. When a *final* attribute has been assigned, it always contains the same

value. To model this kind of attribute in OFL we use an *OFL-AtomAttribute* that has property *isConstant* set to *true*.

static {feature declaration} If a feature (attribute or method) is declared *static*, then it exists exactly one incarnation of the feature, no matter how many instances (possibly zero) of the class which may eventually be created. A static attribute, sometimes called "class variable", is incarnated when the class is initialized. A static method, called "class method", is always invoked without reference to a particular object. The *OFL-AtomAttribute* and *OFL-AtomMethod* that reifies these entities has the *isDescriptionFeature* property set to *false*.

3.1.2 C++ language.

static {member declaration} In C++ a variable that is part of a class, but is not part of an object of that class, is declared as *static* member. There is exactly one copy of a *static* member instead of one copy per object. Similarly, a function that needs access to members of a class, but which doesn't need to be invoked for a particular object, is called a *static* member function. The OFL reification resides in *OFL-AtomAttribute* and *OFL-AtomMethod* entities, which have the *isDescriptionFeature* property set to *false*.

3.1.3 Eiffel language.

expanded {class declaration} Declaring a class as *expanded* means that any of its instances which is addressed through a field of a given object, is expanded (included) in this object (by default a field contains only a reference to it). These classes will be reified by *OFL-componentDescription* corresponding to *expanding class* and *generic expanding class*. Those components could not be a target of neither *aggregation relationship* nor *generic derivation*. But, they could be a target of *inheritance relationship*, *expanded client relationship* and *expanded generically derivation*.

Figure 3.1 illustrates the OFL model extended with *OFL-Modifiers*. We define four kinds of modifiers (one for each type of entity which is concerned by modifier definition) : *description-modifier*, *relation-modifier*, *method-modifier* and *attribute-modifier*. The OFL *modifiers components* inherit from OFL-*modifiers* and represent the reification of language modifiers.

3.2 Definition of an OFL-Modifier

An OFL-*modifier* is defined by a *name*, a *context* (the entity on which it applies), a *keyword*, *assertions* (OCL constraints) and a set of associated *actions* (modified *OFL-actions*).

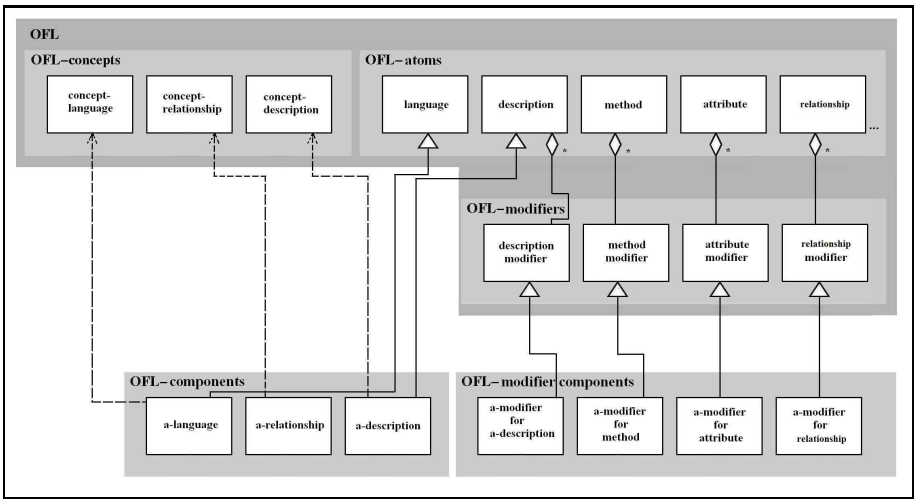

Figure 3.1: The extension of the OFL model through OFL-Modifiers

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. The context could be either one description, one relationship, one attribute or one method.

Modifier Keyword. The modifier keyword represents the string representation of the modifier in the language syntax.

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

A first solution is to define These constraints in *invariant* for OFL *components* or in *pre* and *post conditions* for OFL *actions*. Implementation of control implies assertions at the level of OFL entities reifying the corresponding mechanisms. Indeed, they will be attached to corresponding *OFL-Components* and *OFL-Actions*.

Another solution is to define the assertion within the *OFL-Modifier* itself but the drawback is that if assertion refers to other modifiers then it has to know about other modifiers and this decrease its reuse capabilities.

The role of an *OFL-Modifier* is to take into account those remarks in order to help the meta-programmer to manage and organize assertions.

For assertions we use notation that have the same meaning as in OCL definition [OMG00]. The *self* keyword refers the current instance of the associated component. The OCL modifier assertions are written in the context of the OFL

model definition; as a result of that, 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 on the complexity of the considered modifier. Most modifiers do not need action rewriting. They have just a set of assertions attached to them.

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 are specified within 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 composed. 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 extent 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 may 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 in the OFL entity which is the modifier context.

Following example considers the Java *public* modifier for attributes. For a better understanding we consider a 'package' modifier which replaces the default visibility rule for attributes. The OFL reification for an attribute is the *OFL-AtomAttribute*. At the time the modifier is defined we attach an invariant to this entity which means : *incompatible modifiers set* for *public* is {*protected*, *private*, *package*}.

```
context AtomAttribute
inv: self.modifiers->includes('public')
    implies
     NOT (
    self.modifiers->includes('private')
        OR
    self.modifiers->includes('package')
```

```
    OR
self.modifiers->includes('protected')
)
```

In order to cover all situations a new definition of invariant should be made for each newly added modifier.

If we consider the extension of a given language extension, we can distinguish two kind of modifiers. An OFL-*modifier* can represent the reification of a modifier that belongs to the language binding - we name it *native modifier* - or it can be a *custom modifier* added by the meta-programmer in order to enrich the language semantics.

The native modifiers will have the same meaning (related to the language binding components), as in the original language. The meta-programming task will consist in describing the meaning and the behavior of modifiers according to 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.

Next we present a classification based on the semantics which is behind modifiers. The meaning of semantics in this context deals with the aspect of entity semantics that is changed by the modifier. To evaluate semantic changes, we consider all the *OFL-Actions* that are involved.

3.3 Modifiers Classification Regarding OFL Implementation Issues

All the kinds of modifiers presented in this section will be addressed in detail within the next chapters.

3.3.1 Access Control Modifiers

The importance of a systematic approach on access control mechanism represents an actual topic of research in the field of object oriented technology [Aba98, Ard02, CNP89, Sny86]. Even the UML standard [OMG03], which was planned to be language independent, lacks in defining protection mechanisms. Flower and Scott emphasize this aspect [FS01]:

"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 also lacks in customization of access control mechanisms [PL03]. Modifiers represent a way to add this customization. Considering the OFL-*Actions* involved by the semantics we can split these modifiers into two subcategories: *basic modifiers* and *complex modifiers*.

Basic Access Control Modifiers. Some modifiers add constraints to some facets of the language which are customizable in OFL by setting values to some of the parameters and characteristics built in the *OFL Model*. To implement these modifiers, meta-programmer has to write assertions at the level of one or several *OFL-Components* only. They do not imply any action rewriting. We call them *basic modifiers*.

Complex Access Control Modifiers. Some other modifiers address mechanisms that are implemented in OFL through pieces of code wrote by a meta-programmer. To implement these modifiers, he has to rewrite some of the *OFL-Actions* and/or to extend their assertions. Because writing actions is a more complicated job, we call them *complex modifiers*.

complex modifiers implies always protection and some time they implies also visibility (ex. protected-write [CKMR99]).

3.3.2 Optimization Modifiers

These modifiers have no impact at the level of application model semantics. They are used only to establish optimization strategies for compilers or, more generally, translators (ex. in line, volatile, register etc.). The corresponding OFL-modifiers are used only to allow the programmer to specify optimization for OFL Parser. This is particularly important if he plans to run the resulting OFL-application.

3.3.3 Service Modifiers

Service modifiers are used to introduce new kind of services like custom look-up, persistency or concurrency; They could have an impact at the level of model semantic or only at the level of code generation. (ex. persistent, synchronised etc.)

3.3.4 Additional Modifiers

In addition to previous considered modifiers, languages propose also other keywords used to influence the semantics of program entities. The meaning of these additional modifier is to force compiler to treat in a special way the entity that declare the modifier. This category does not include modifiers that modify the reification of considered entity (this subject was discussed in sec. 3). The modified semantics is handled by the native compiler (ex. explicit, agent etc.).

Chapter 4

Basic Access Control Modifiers

Most of access-control modifiers add constraints regarding the way features could be reached by other entities that are connected through different kinds of relationships. They imply only constraints related with mechanisms reified by OFL relationships (dynamic relationships like the one that links an instance to its class could also be considered). According to that they could be considered as *basic modifiers*. Their implementation relies only on assertions in OFL-componentDescriptions which involve those relationships.

4.1 Examples of Native Basic Access Control Modifiers

4.1.1 Java Language

Java [GJSB00] has several modifiers used for basic access control: *public*, *protected*, *private*, and *default* (to be more expressive we name it *package*).

Java class members (attributes and methods) that are declared *public* can be accessed from any class which can access to the class where they are declared.

Members that are declared as *protected* can be accessed from any class of the package, and also from any subclasses, of the class where they are declared.

Members that are declared as *private* are only accessible from the class in which they are defined (it means that subclasses are not allowed to).

Class members that have no access control modifier associated are considered to have default visibility. These members can be accessed only from classes of the package where they are declared.

A Java class, an abstract class or an interface which is declared as *public* can be referenced from outside its package. If a class is not declared as *public*, it can be referenced only within its package.

Classes and members that are not explicitly associated to a modifier have the *default* Java visibility, that it is to say that they are visible only within the package.

4.1.2 C++ Language

For C++ language [Str97] the *public*, *protected* and *private* modifiers have a meaning which is slightly different than in Java [Ard02]. There is no "package" resolution but another kind 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.

If a member of a C++ class is *private*, its name can be used only by member functions and friends of the class in which it is declared. A *protected* member can be used only by member functions and friends of the class in which it is declared and by member functions and friends of classes derived from this class. A *public* member can be used by any function. The default access for C++ class members is *private*.

These modifiers could be used to change access control through inheritance between classes. When preceding the name of a base class, the *public* keyword specifies that the public and protected members of the base class are public and protected members, respectively, of the derived class. The *protected* keyword used for inheritance specifies that the public and protected members of the base class are protected members of its derived classes. Finally, when preceding the name of a base class, the *private* keyword specifies that the *public* and *protected* members of the base class are *private* members of the derived class.

4.1.3 Eiffel Language

In Eiffel [Mey02] 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 Access Control Modifiers for Features

4.2.1 Modifier Assertions

The assertions of basic access control modifiers for features (attributes and methods) are defined in *OFL-Relationship* components that manage export of those features. They should be tested each time a relationship involving that feature is created. An invariant in the description to which belongs 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: *i)* the feature belongs to current class *ii)* it is inherited through an inheritance relationship from a direct or indirect ancestor or *iii)* it is accessed through an use relationship (current class is a client of the description which owns the feature). In the last situation we consider that the *current description* can access to the supplier one. Indeed, this aspect is covered by access control handled at the level of descriptions. By *current description* we mean the one that accesses to the feature.

If we consider the Java syntax, features belonging to a class or inherited by the class, are accessed using the keyword *this* as qualifier. This keyword could be explicit or implicit (non-qualified features). Features accessed through an use relationship are explicitly qualified with the supplier name. To consider all situations, an invariant is needed for all OFL-components, dealing with both *import relationship* and *use relationship*, which are defined for a given language.

The following example presents invariants for the *extends* Java inter-class relationship and the Java *aggregation* relationship. In Java, basic modifiers related to features are *public*, *protected*, *private*, *package*.

```
context ComponentJavaClassExtends
inv: self.shownFeatures->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 *shown* and *redefined* features through an *extend relationship* should have modifiers *public* or *protected* attached. All *hidden* features have *private* modifier. It has to be noted that in OFL, *shownFeatures*, *redefinedFeatures* and *hiddenFeatures* are collections of features which participates to the reification of a relationship.

```
context ComponentJavaAggregation
inv: self.shownFeatures->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')
```

```
OR
(( f.modifiers->include('package') OR
           f.modifiers->include('protected'))
   AND
   self.source.package <> self.target.package)))
```

In addition to previous assertion, this one tests also information about the packages to which belong the descriptions; it considers the descriptions which are *source*[1] and *target*[2] of the instance of the OFL-relationship component (*self*).

All these modifiers are incompatible. When the feature is a method, the set of incompatible modifiers contains also the modifier *abstract*.

4.2.2 Modifier Actions

Interference with OFL-actions (actions which are defined in the OFL model) is minimal. Assertions are added (see above), in order to control the access to features through relationships and no action rewriting is necessary. Indeed, modifiers for basic access control generally do not redefine any actions.

But there are some exception; for example let us consider the modifier *protected* applied to Java features. Action is needed in this case to express a particular semantic presented in Figure 4.1. Method m of class C have access to protected member f of B. This happens because class A, which declares the member f, and class C, belongs to the same package. To express this semantics we need to rewrite the action *lookup* for features. This action must allow the access to protected members for any feature that is declared by an ancestor belonging to same package as the class which accesses to the feature.

4.3 Basic Access Control Modifiers for Descriptions

4.3.1 Modifier Assertions

The assertions of basic modifiers dealing with the access control of descriptions are defined in OFL-relationship components and also in OFL-description component. They should be tested each time a relationship involving that description is created and each time an instance of it is created. The last situation deals with relationships that enable polymorphism. According to these assumptions, the assertion associated to such modifier should be a post-condition of the OFL-action *lookup*.

The following example refers to the Java language semantics related to class access control. Please note that this example does not consider interfaces, ab-

[1] The *source* is the class which declares the relationship. In Java, for an *extends relationship* this is the class which declare the keyword *extends*.

[2] the target is the class which is addressed by the relationship. In Java, for an *extends relationship* this is the class whose name is mentioned after the keyword *extends*.

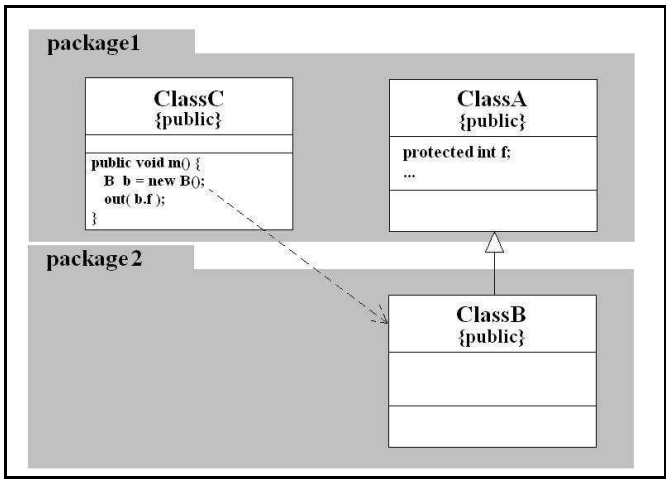

Figure 4.1: Java *protected* modifier semantics

stract classes and inner classes. the modifiers associated to classes that have to be considered are *public* and *package*.

```
context ComponentJavaClassExtends
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 any 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 addresses 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')
```

Then, we consider the Java language semantics for the access control of interfaces. The example does not take into account inner interfaces. The modifiers *public* and *package* are considered for Java interfaces.

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

An interface can extend another interface from the same package and it may also extend a *public* interface from any other package.

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

A class can implements any interface from the same package but also a *public* interface from any other package.

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

A class can declare an attribute whose type is represented by interface from the same package and also by a *public* interface from any other package.

To handle dependencies between classes and interfaces we use the same post-condition as for the action *lookup* previously defined for class modifiers. Action *lookup* is defined in components related to description and relationships.

4.3.2 Modifier Actions

For the modifiers mentioned above, assertions are also added in OFL-relationship components in order to control the access to features. Post-conditions are used to filter the result of the action *look-up*. Those modifiers do not redefine any actions.

Chapter 5

Complex Access Control Modifiers

Complex access control modifiers define protection dealing with special rights such as writing or reading an attribute, calling or redefining a method and extending or instantiating a description.

5.1 Examples of Native Complex Access Control Modifiers

5.1.1 Java Language.

Java language does not include complex access control modifiers for attributes. It includes *final* modifier for methods and classes and interfaces. Modifier *final* associated to a method disallows the ability to redefine it. When Modifier *final* is applied on classes or interfaces it prevent from extending them. Other language mechanisms (like making all constructors *private*) could be used to control instantiation of classes.

5.1.2 C++ Language.

C++ does not provide any specific modifiers to control the use of an entity (like *final* in Java). Changing access rights of constructors does also control the ability to create or not an instance of a given class, like in Java.

5.1.3 Eiffel Language.

Eiffel modifiers *Frozen* and *deferred* could be considered as belonging to this category. *Frozen*, put before a feature name express that the declaration is not subject to redefinition in descendants. The modifier *Deferred* also put before a

feature allows the feature to not have any body or implementation. This transfers to descendants the responsibility for providing an implementation through a new declaration. This is called "effecting" the feature.

5.2 Complex Access Control Modifiers for Methods

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

5.2.1 Modifier Assertions.

Implementation of control implies assertions in 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 dealing with the right to call a feature is implemented as a post-condition of this action.

The following example is an implementation of the modifier *final* applied to Java methods.

```
context ComponentJavaClassExtends
inv: self.redefinedFeatures->forall(f:Feature |
    f.typeOfFeature = method
        implies
    NOT f.modifiers->includes('final'))
```

The modifier *Final* is compatible with following modifiers : *public*, *protected*, *package* and *private*. This means that *Final* can be set simultaneously with any of these modifiers. Its invariant will be added to the invariant of the corresponding component (for example *Feature* or more precisely *method*).

5.2.2 Modifier Actions.

Complex access control modifiers for methods require sometimes the rewriting of the OFL-Action *execute*.

5.3 Complex Access Control Modifiers for Attributes

Rights concerning attribute usage address the ability to read or write the content of fields (defined by an attribute). Protection on writing is achieved by a pre-condition in OFL-Action *assign*. We can consider here a proposal of Cook and Rumpe [CKMR99] for defining a read-only modifier for attributes. They

conclude that it 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.

5.3.1 Modifier Assertions

Assertions for attribute complex modifiers deal only with pre and post conditions added to the OFL-action *assign*.

5.3.2 Modifier Actions

It is necessary to redefine or rewrite OFL-action only if the semantics associated to the modifier is enough complex. As an example we consider a modifier that implements a *strong* protection against attribute modification. By strong protection we mean to protect not only the reference of the object against modification but also the internal state of the referred object.

A solution that lacks in efficiency is to use a clone of the object which contains the field (corresponding to the attribute) and to look if any changes appear; so that the access is allowed or not. To ensure this control, the call to the OFL-actions which deal with the access to attribute should be embedded in the following code:

```
// cloning the original object
    aux = deep_clone(f)
// original action
// ( any kind of action that may imply modification
//   of the internal state of attribute)
    *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)
```

OFL-*Actions* that permit the change of the internal state of attributes are the following : *evaluate-parameters*, *attach-parameters*, *detach- parameters*, *assign*, *execute*, etc.

5.4 Complex Access Control Modifiers for Descriptions

The specification of an application may lead to the extension of descriptions, to their use through the declaration of features or to the creation of instance of them. These situations involve different kinds of relationships.

5.4.1 Modifier Assertions

The extension is controlled through invariant on OFL-*ImportRelationship* components and the control of client-supplier relationship is made through invariants on OFL-*UseRelationship* components.

Let us consider the following example related to the Java modifier *final* applied to descriptions. The invariant for the Java extends-relationship check absence of this modifier in the target description of the relationship.

```
context ComponentJavaClassExtends
inv: NOT self.target.modifiers->includes('final')
```

5.4.2 Modifier Actions

For description modifiers, OFL-actions participate very much to the control of instantiation. Some rewriting may be needed but most of the times a precondition in the OFL-*action create-instance* is enough to ensure all semantics of the control.

Chapter 6

Optimization Modifiers

Optimization modifiers are used to transmit hints to the compiler in order to generate a smaller or faster code. Because these modifiers have no impact on the semantics of the application model they have only to be passed to final compiler.

6.1 Examples of Native Optimization Modifiers

6.1.1 Java Language.

Java has one optimization modifier for attributes - *volatile* - two optimization modifiers for methods - *native* and *strictfp* - and one optimization modifier for descriptions - *strictfp*.

An attribute which is declared as *volatile* refers to objects and primitive values that can be modified asynchronously by separate threads at runtime. They are treated in a special way by the compiler to control how they can be updated.

A *native* method is a method written in a language other than Java. In a way it is declared like an abstract method.

The effect of the *strictfp* modifier is to make all float or double expressions within the method body be explicitly FP-strict. Within a FP-strict expression, all intermediate values must be elements of the float value set or the double value set, implying that the results of all FP-strict expressions must be those predicted by IEEE 754 arithmetic on operands represented with single and double formats.

The effect of the *strictfp* modifier in context of a class or an interface is to make all float or double expressions within the class or interface declaration be explicitly FP-strict. This implies that all methods declared in the class, and all nested types declared in the class, are implicitly *strictfp*. Also all float or double expressions within all variable initializers, instance initializers, static initializers and constructors of the class will also be FP-strict.

6.1.2 C++ Language.

C++ language contains also optimization modifiers. The C++ specification defines *inline* for functions and *mutable* and *volatile* for member attributes.

The *inline* modifier for a member function is a hint for the compiler which ensure that when it encounter a function call it should rather generate the code corresponding to the function body than the usual function call mechanisms.

The *mutable* modifier specifies that a member attribute should be stored in a way that allows its update - even when it is a member of a const object. In other words *mutable* means "can never be const". Declaration of *mutable* member is appropriate when only a part of the object is allowed to change.

A *volatile* specifier is a hint to a compiler which means that an attribute may change its value in a way not specified by the language, so that aggressive compiler optimization must be avoided.

6.1.3 Eiffel Language.

Analyzing Eiffel we find also optimization modifiers. *Indexing* and *obsolete* modifiers for a class could be considered in this category . The optional *Indexing* parts have no direct effect on the semantics of the class. They serve to associate information to the class which will be used by tools for archiving and retrieving classes according to their properties. This is particularly important in the approach to software construction promoted by Eiffel, based on libraries of reusable classes: the designer of a class should help future users to find out about the availability of classes fulfilling particular needs. We choose to implement that part like a modifier because OFL does not contain any customization according to that. Because indexing part could appear in two different places - one at the beginning and one at the end - we define two different modifiers *StartIndexing* and *EndIndexing*.

The *obsolete* clause in a class indicates that the class does not meet current standards. The advice for developers is against continuing to use it as supplier or parent but it does not prevent existing systems which rely on this class, to compile and run. Declaring a class as Obsolete does not affect its semantics. Instead, some language processing tools may produce a warning when they process a class that relies, as client or descendant, on an obsolete class. The same mechanism exists for features.

6.2 Optimization Modifiers for Attributes

Optimization modifiers for attributes deal mainly with memory allocation and persistency.

6.2.1 Modifier Assertions

Assertions for optimization modifiers have to be written just to avoid usage of incompatible modifiers. No other constraints are necessary.

If we consider Java modifiers, *volatile* is incompatible with *final*. Because *final* keyword has no reification in OFL (3) the assertion have to ensure that the propriety *isConstant* is set to *false*.

```
context AtomAttribute
inv: self.modifiers->includes('volatile')
    implies
        self.isConstant = false
```

6.2.2 Modifier Actions

If an OFL translator is used in order to generate native code, it is only necessary that OFL-*actions* ensure that these modifiers are copied to the final generated code. If we deal with an OFL-interpretor, it could consider directly those modifiers to make optimizations. Another possibility is to ignore these modifiers if that optimizations are not compulsory.

6.3 Optimization Modifiers for Methods

Optimization modifiers for methods are related *i)* to the need to get more efficient mechanism for calling method and *ii)* to deal with methods written and compiled in other languages.

6.3.1 Modifier Assertions

Assertions for optimization modifiers deal with incompatible modifiers. No other constraint is necessary.

In the case of the modifier *native* in Java, it is incompatible with modifier *synchronized*. Moreover, a constructor method may not be declared as *native*. Not to be able to declare native constructors is an arbitrary design choice of the language. It makes difficult an implementation of the virtual machine which verify that superclass constructors are always properly invoked during object creation.

```
context AtomMethod
inv: self.modifiers->includes('native')
    implies        self.isConstructor = false
        and
        self.body->isEmpty()
        and
        NOT self.modifiers->includes('synchronized')
```

6.3.2 Modifier Actions

These modifiers needs the same kind of actions as the optimization modifiers for attributes. If it is decided to build an OFL compiler for the OFL language

reification, attention must be payed to make a correct linking with the external code.

6.4 Optimization Modifiers for Description

Optimization modifiers for descriptions are used for version and documentation management. They could be used also to organize library of classes.

6.4.1 Modifier Assertions

No assertion are needed.

6.4.2 Modifier Actions

Actions could be designed to generate errors or warnings in case of version conflicts or to generate class documentation. These actions could be executed by modeling tools or by translators or compilers. Special tools could also run them in order to find desired classes in libraries or to check compatibilities.

Chapter 7

Service Modifiers

7.1 Examples of Native Service Modifiers

7.1.1 Java Language.

Java has two modifiers that could be considered as service modifiers. These are *synchronized* for methods and *transient* for attributes.

Java virtual machine can support many threads simultaneously at runtime. Threads may be supported by having many hardware processors, by time-slicing a single hardware processor, or by time-slicing many hardware processors. To help programmer to use threads, Java provide mechanisms for synchronizing the concurrent activity of threads through the keyword *synchronized*. A Java *synchronized* method is a method that must acquire a lock on an object or on a class before it can be executed. For a (*static*) method, this is the lock associated with the object *Class* corresponding to the class (which declare the method), which is used. For an instance method, this is the lock associated with *this* (the object for which the method was invoked), which is used.

An attribute which is declared as *transient* is not saved as part of an object when the object is serialized. The transient keyword identifies an attribute that does not maintain a persistent state.

7.1.2 C++ Language.

We do not identify any native service modifier in C++ language.

7.1.3 Eiffel Language.

Eiffel also does not include any service modifier.

7.2 Service Modifiers for Attributes

Service modifiers for attributes address services that deal with objects state (like persistency).

7.2.1 Modifier Assertions

Most of the assertions for these modifiers deal just with incompatible modifiers. A particular situation result from the fact that OFL does not provide any customization for attributes. All attributes are reified with the same set of properties (see *OFL-AtomAttribute*). To cover this situation, modifier assertion has to test if the usage of the considered service is permitted or not in the context of the description which declares the attribute.

7.2.2 Modifier Actions

Service modifier actions will implement the service or will make link with components that provide the considered service. this are OFL-Actions added to the list proposed in [Cre01] or modified by the meta-programmer in order to support a new service. It may be interesting to discus about the possibility to let meta-programmer to add new kind of actions.

7.3 Service Modifiers for Methods

Service modifiers for methods address services which deal with execution (ex. concurrency).

7.3.1 Modifier Assertions

Service modifier assertions has to ensure that a particular kind of method (ex: a constructor or a destructor) support or not a given service. In the same way as for attributes, OFL does not provide customization for methods. All methods are reified with the same set of properties (see *OFL-AtomMethod*). Additionally, incompatible modifiers have to be considered.

7.3.2 Modifier Actions

Service modifier actions will implement the considered service. Most of those actions will be dynamic actions integrated at compiling time. This actions should consider at runtime the usage of particular hardware (e.g. actions that support concurrency) or the usage of external resources (e.g. a database in case of persistent actions).

7.4 Service Modifiers for Descriptions

Service modifiers for descriptions have to deal with all kinds of services.

7.4.1 Modifier Assertions

Assertion will have to ensure that all relationships which involve the current description are compatible with the provided service. If we consider persistency, a relationship implementing object composition could imply that the target of relationship (in fact the referenced object) should be also persistent if the source (the object which contains the reference), is persistent. In other words, assertions have to verify that both objects are persistent or that both are not.

7.4.2 Modifier Actions

Service modifier actions will implement the service. Most of these actions will specialize actions of modifiers for attributes and methods.

Chapter 8

Additional Modifiers

We consider in this chapter all modifiers that could not be included in previous categories. These modifiers are used to change the semantics of the related entity when it is non-customizable in OFL. Semantics modification implied by native modifiers is handled by a native compiler of the corresponding language. When an OFL application model is translated in native language code these modifiers are just written into the generated source code. A custom OFL compiler for the considered language binding must take care to generate the correct semantic for native modifiers.

8.1 Examples of Native Additional Modifiers

8.1.1 Java Language.

For Java language we do not identify any modifiers that could be considered in this category.

8.1.2 C++ Language.

In this category, C++ has modifiers like *const* for methods and *explicit* for constructors (that are also a kind of method). The *const* modifier used for a method indicates that the method do not modify the state of an object. In C++, *explicit* constructors will be invoked only explicitly. This disallows implicit conversions.

8.1.3 Eiffel Language.

Eiffel contains *agent* keyword that modify the semantics of a method parameter. The keyword *agent* is used to pass a routine as a parameter of another routine. It avoids the confusion with an actual routine call when effective parameter is computed. Indeed, when a routine is passed as an *agent* to another routine it is not called but only transmit to it.

8.1.4 Modifier Assertions

Assertions have to deal with incompatible modifiers for all additional modifiers. Because this category is a very general one, no other assumptions could be made regarding other necessary assertions.

8.1.5 Modifier Actions

We can assume that all modifiers from this category involve hard action writing. Each of them address a very specific semantic. Meta-programmer has to identify first how OFL actions are involved in expressing the considered semantics.

As example, if we consider the native C++ modifier *explicit*, the semantics is described in following OFL-Actions : *before-create-instance* and *create-instance*.

Chapter 9

Conclusion and perspectives

In this paper we proposed to extend the OFL Model. The main goal of this extension was to improve the customization of the access control mechanism and of additional non-covered semantics. We introduced the notion of OFL modifier to provide a clean way to control implementation. For providing a better understanding of the concept, sections 4 and 5 present 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 modeling tool will execute these actions to ensure automatic model consistency.

Bibliography

[Aba98] M. Abadi. Protection in Programming Language Translation. In *Automata, Languages and Programming: 25th International Colloquium, ICALP'98, Springer-Verlag*, July 1998.

[ACL03] G. Ardourel, P. Crescenzo, and P. Lahire. Lamp : vers un Langage de definition de Mecanismes de Protection pour les langages de programmation a objets. In *LMO 2003, Vannes, France*, February 2003.

[Ard02] G. Ardourel. Modelisation des Mechanismes de Protection dans les Langages a Objets. Phd thesis, University of Montpellier, France, December 2002. http://www.lirmm.fr/ ardourel/cv/theseArdourel.pdf.

[CKMR99] S. Cook, A. Kleppe, R. Mitchell, and R. Rumpe. The Amsterdam Manifesto on OCL. Technical Report TUM-I9925, Technical University of Munchen, Germany, 1999.

[CL02] P. Crescenzo and P. Lahire. Customisation of Inheritance. In *Springer Verlag, LNCS series, ECOOP'2002 (The Inheritance Workshop) and Proceedings of the Inheritance Workshop at ECOOP 2002, University of Jyvskyl, Finlande*, page 7, June 2002.

[CNP89] L. Cardelli, E. J. Neuhold, and M. Paul. Typefull Programming. In *IFIP Advanced Seminar on Formal Methods in Programming Langage Semantics, Lecture Notes in Computer Science. Springer Verlag*, 1989.

[Cre01] P. Crescenzo. OFL: un Modele pour Parameter la Semantique Operationnele des Langages a Objets - Application aux Relations interclasses. Phd. thesis, University of Nice, Sophia Antipolis, France, December 2001. http://www.crescenzo.nom.fr/.

[FS01] K. Flower and K. Scott. *UML Distilled Second Edition.* Addison-Wesley, 2001.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, 2000.

[Mey02] B. Meyer. *Eiffel: The Language*. http://www.inf.ethz.ch/ meyer/, 2002.

[OMG00] Object Management Group OMG. *Object Constraint Language Specification. Version 1.3*, March 2000. http://www.omg.org.

[OMG03] Object Management Group OMG. *Unified Modelling Language Specification, version 1.5, 1st ed.*, March 2003. http://www.omg.org.

[PL03] D. Pescaru and P. Lahire. Modifiers in OFL: An Approach for Access Control Customization. In *The 9th International Conferences on Object-Orinted Information Systems - OOIS'03, WEAR workshop, Geneva, Swizerland*, September 2003.

[Sch02] N. Schirmer. Analasyng the Java Package/Access Concepts in Isabelle/HOL. In *ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP'2002), Malaga, Spain*, June 2002.

[Sny86] A. Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages, and Applications.*, November 1986.

[Str97] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.