Automatic Profile Generation for OFL-Languages

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,
Bâtiment Euclide B BP121 F-06903,
Sophia-Antipolis CEDEX, FRANCE

November 10, 2003

Contents

Chapter 1

Introduction

The OFL-ML modeling language specification is designed to provide a standard way to express the semantics of an OFL-*language* application using UML-like notation and thus to support OFL applications modeling with standard UML tools.

The term OFL-*language* means a language reified or expressed in OFL[1]. For example it can be : OFL-Java, OFL-C++, OFL-myJavaExtension, etc.

OFL-ML could be considered as a "meta-profile" or a "meta-model" for UML Profiles dedicated to a better support of the semantics of object-oriented language. Each instance of OFL-ML in the context of a particular OFL-*language* is an UML Profile for that language. We name this profile "OFL-ML-Profile for OFL-*language*". It will exists for example an "OFL-ML Profile for OFL-*Java*", an "OFL-ML Profile for OFL-*myExtendedJava*" or an "OFL-ML Profile for OFL-*C++*".

An OFL-ML *Profile* is an UML Profile that is generated automatically and customized for every language expressed in OFL. Indeed, each existing language reified in OFL or a possible extended language expressed in OFL will have its own associated OFL-ML *Profile*.

OFL-ML could be considered as a template for profiles. To obtain a specific UML Profile for an OFL-language, OFL-ML has to be instantiated using OFL meta-information (*components*, *parameters*, *characteristics* and *modifiers*).

All properties of UML meta-model elements contained in OFL-ML may be used to express an object model that conforms to the resulting profile. Based on that, modeling tools that handle UML Profiles could generate an XML representation of an OFL-language application.

The main purpose of OFL-ML is to provide to programmer an UML Profile designed to support the development of OFL applications. Using this profile with a modeling tool, the programmer could generate a representation for the application that could be processed later by an OFL-compiler, an OFL-interpretor or other tools.

UML Profiles provide a generic extension mechanism for building UML models dedicated to a particular domain. They are based on additional Stereotypes and Tagged values that are applied to Elements, Attributes, Methods, Links, Link Ends, etc. A profile is a collection of such extensions that together describe some particular modeling problem and facilitate modeling constructs in that domain. In [Des99] it is discussed how specific domains that require a specialization of the general UML meta-model can define an UML profile which customize UML in order to better address specificities of a given domain. Even if concrete UML profiles have started to emerge, the use of the profiling mechanism is still discussed [DSB99, AK00]. To define OFL-ML profile generation we rely on the recommendation found in the "UML Profile White Paper" [Des99]. Because it is not a final accepted opinion about Profiles, this paper is not yet an official OMG white paper.

An OFL-ML profile is planed to be used with standard UML modeling tools or with new modeling tools especially designed for it. It could be used to test and validate the model, to apply design patterns in an automatic way, to collect metrics or to generate XML representation of OFL-code. The OFL information contained in OFL-ML entities represent a real help

[1]For more information on the OFL Model, to read the thesis of Pierre Crescenzo [Cre01b].

to achieve all these goals. It is obvious that in the last case, all this information will fill the XML representation of application elements.

Chapter 2

Supported Elements and Definitions

2.1 OFL Model

Specification of OFL-ML is based on OFL model definition found in [Cre01b] extended with OFL Modifiers [PL03, PCL03]. The OFL elements modeled by OFL-ML are:

OFL-atoms OFL-*atoms* represent the reification of the non-customized entities of the model. Example of atoms are AtomAttribute, AtomMethod, AtomParameter etc.

OFL-components OFL-*components* inherit from OFL-*atoms* and represent reification of customized language entities (*relationships* and *descriptions*).

OFL-component parameters and characteristics OFL-*parameters* and OFL-*characteristics* contains values that determine the operational semantics of an object oriented language. OFL-ML use only parameters and characteristics that have impact at the level of application model.

OFL-component properties Each OFL-*component* keeps a set of *reified properties* that represents meta-information for program entities such as lists of attributes and methods for a description component or lists of redefined features for relationship components. As specified, OFL-ML use only the reified properties that have an impact at the level of application model.

2.2 OFL-Modifiers

OFL-Modifiers [PL03] represent an extension of the OFL Model as presented in [Cre01b]. They are used to express additional semantics that is not customizable by OFL. OFL-ML will express this semantics using mainly tagged values. These tagged values will be added to the generated UML-Profile. Moreover, modifiers assertions, which contain most of the semantics, have to be translated into Profile constraints. In this paper we try to identify assertion transformation rules that are necessary if we address an automatic generation of profile.

2.3 UML Profile

UML profiles appeared with the UML 1.3 standard as a way to structure UML extensions (tagged values, stereotypes and constraints). UML is a modeling language which intend to be used in a large number of application domains and for all types of software applications.

However, each domain has specific notions and particular needs, which are handled by UML through extensions which are grouped into *UML Profiles*.

OFL-ML is based on the UML Profile specification found in [Des99, OMG02, Sof99]. An UML Profile:

- Identifies a subset of the UML meta-model (which may be the entire UML meta-model).
- Specifies *well-formedness rules* beyond those specified by the identified subset of the UML meta-model. *Well-formedness rule* is a term used in the normative UML meta-model specification [OMG03] to describe a set of constraints written in natural language and UMLŠs Object Constraint Language (OCL) that contributes to the definition of a meta-model element.
- Specifies *standard elements* beyond those specified by the identified subset of the UML meta-model. *Standard element* is a term used in the UML meta-model specification to describe a standard instance of an UML *stereotype*, *tagged value*, or *constraint*.
- Specifies semantics, expressed in formal or natural language, beyond those specified by the identified subset of the UML meta-model.

2.4 OCL

The OCL convenience operations for UML Meta-model elements presented in this section can be applied generally to UML version 1.5 (01.03.2003) and are not specific to the UML Profile defined by OFL-ML. They are defined in order to produce more compact and readable OCL. Indeed, they are used in UML profiles already approved by OMG [OMG02, OMG01] in the same way we intend to do here.

2.4.1 For ModelElement.

[1] The operation *allStereotypes* results in a Set containing the Stereotype of ModelElement and all Stereotypes inherited by this Stereotype (not to be confused with all Stereotypes inherited by the ModelElement).

```
allStereotypes : Set(Stereotype);
allStereotypes = self.stereotype->union
    (self.stereotype.generalization.parent.allStereotypes)
```

[2] The operation *isStereotyped* determines whether the ModelElement has a Stereotype whose name is equal to the input name.

```
isStereotyped : (stereotypeName : String) : Boolean;
self.stereotype.name = stereotypeName
```

[3] The operation *isStereokinded* determines whether the *ModelElement* has a *Stereotype* whose name is equal to the input name or if one of the ancestor of one of its stereotypes has a name which is equal to the input name.

```
isStereokinded : (stereotypeName : String) : Boolean;
self.allStereotypes->exists (
        stereotype | stereotype.name = stereotypeName)
```

There are some OCL convenience operations defined in this specification that apply more narrowly to certain extensions of UML that the profile defines. These operations appear in line with the Constraints for those specific extensions.

2.4.2 For Classifier

[1] The operation *navigableOppositeEnds* results in a Set containing all navigable *AssociationEnds* that are opposite to the Classifier.

```
navigableOppositeEnds : Set(AssociationEnd);
navigableOppositeEnds
        = self.oppositeAssociationEnds ->
            select(end | end.isNavigable)
```

[2] The operation *allEnds* results in a Set containing all AssociationEnds for which the Classifier is the type.

```
allEnds : Set(AssociationEnd);
allEnds = self.associations ->
            collect(assoc | assoc.connection)
```

[3] The operation *nonNavigableNearEnds* results in a Set containing all *AssociationEnds* that are adjacent to the Classifier and that are non-navigable.

```
nonNavigableNearEnds : Set(AssociationEnd);
nonNavigableNearEnds =
    self.allEnds->select
        (end | end.type = self and not end.isNavigable)
```

[4] The operation *navigableEnds* results in a Set containing all navigable *AssociationEnds* for which the *Classifier*; that is to say, *self* is the type.

```
navigableEnds : Set(AssociationEnd);
navigableEnds = allEnds ->
        select (end | end.isNavigable)
```

Chapter 3

OFL-ML Definition

3.1 Identified Subset of UML

OFL-ML diagrams are based on UML Static Structures Diagrams (Class Diagrams). An UML class diagram is a graph of Classifier elements connected by their various static relationships. These elements belong to standard UML packages.

The OFL-ML extends the following standard UML packages: Core and Model Management. Figure 3.1 shows the model elements that form the structural backbone of the meta-model and figure 3.2 shows the model elements that define relationships. The abstract syntax for the Model Management package is expressed in graphic notation in Figure 3.3.

UML use standard visibility markers to express access control at the level of a classifier and feature. These markers has no meaning for an OFL-ML profile. They are covered by tagged values that represents corresponding access control modifiers. The reason resides in difficulty of an automatic translation between access control modifiers and these markers. But, if a meta-programmer manual intervention is accepted, mapping between these elements should be considered.

The following concrete meta-classes, and implicitly all super-meta-classes of these meta-classes, are used:

3.1.1 From Core - Backbone

The backbone of the core package is shown in fig. 3.1.

Attribute An attribute is a named slot within a classifier that describes a range of values that may be hold by instances of the classifier.

Class A class is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics.

Classifier A classifier is an element that describes behavioral and structural features; it comes in several specific forms, including class, data type, interface, component, artifact, and others that are defined in other meta-model packages.

Comment A comment is an annotation attached to a model element or a set of model elements. It is not relevant for the definition of the semantics but it may contain information useful for people in charge with the project.

Constraint A constraint has a semantic which deals with either a condition or a restriction and it is expressed in the model specification.

DataType A data type is a type whose values have no identity (i.e., they are pure values). Data types include primitive built-in types (such as integer and string) as well as customizable enumeration types (such as the predefined enumeration type boolean whose literals are false and true).

ElementOwnership Element ownership defines the visibility of a ModelElement contained in a Namespace.

Feature A feature is a property, like operation or attribute, which is encapsulated within a Classifier.

Namespace A namespace is a part of a model that contains a set of ModelElements; each of them has a name which designates an unique element within the namespace.

Operation An operation is a service that can be requested from an object to implement its behavior. An operation has a signature, which describes the actual parameters that are possible (including possible return values).

Parameter A parameter is an unbound variable that can be changed, passed, or returned. A parameter may include a name, a type, and a direction of communication.

ProgrammingLanguageDataType A data type is a type whose values have no identity (i.e., they are pure values). A programming language data type is a data type specified according to the semantics of a particular programming language, using constructs available in that language.

3.1.2 From Core - Relationships

The UML relationships described in the core package are presented in fig. 3.2.

Abstraction An abstraction is a Dependency relationship that relates two elements or sets of elements that represent the same concept at different levels of abstraction or from different viewpoints.

Association An association declares a connection (link) between instances of the associated classifiers (e.g., classes). It consists of at least two association ends, each specifying a connected classifier and a set of properties that must be fulfilled for the relationship in order to be valid.

AssociationEnd An association end is an endpoint of an association, which connects the association to a classifier. Each association end is part of one association.

Dependency A term of convenience for a Relationship other than Association, Generalization, Flow, or meta-relationship (such as the relationship between a Classifier and one of its Instances).

Generalization A generalization is a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element (it has all of its properties, members, and relationships) and may contain additional information.

Usage An usage is a relationship in which one element requires another element (or set of elements) for its full implementation or operation.

3.1.3 From Model Management

The main elements of the model management package are shown in fig. 3.3.

ElementImport An element import defines the visibility and alias of a model element included in the namespace within a package, as a result of the package importing another package.

Package A package is a way to group several of model elements which deal with the concern.

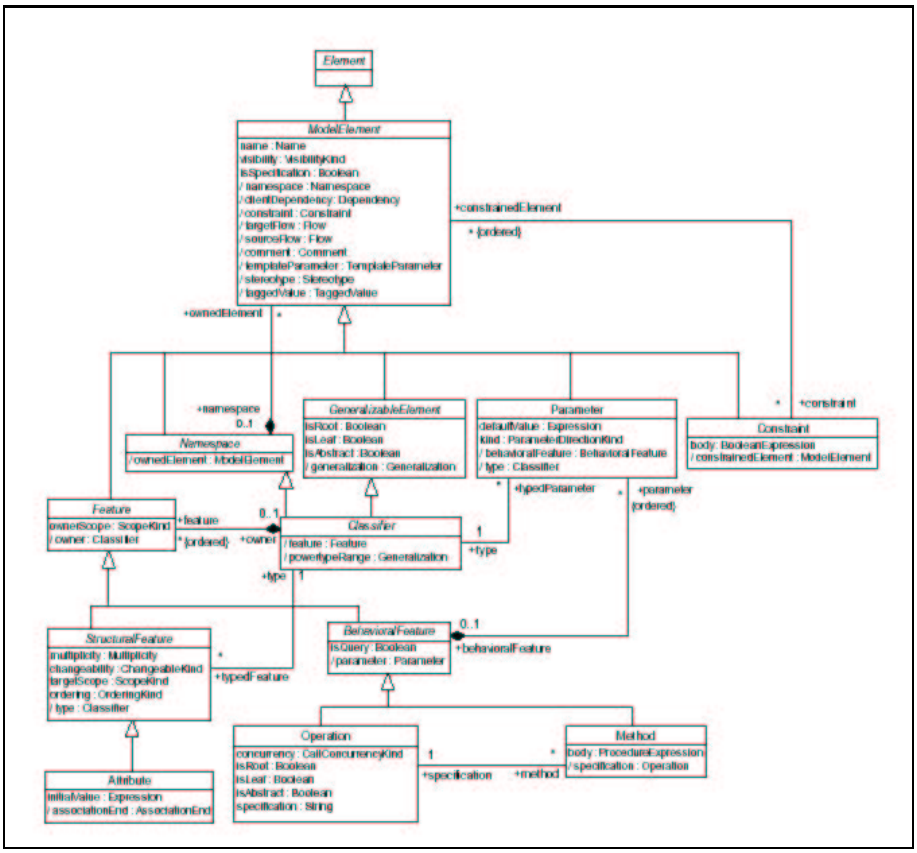

Figure 3.1: The UML Core Package - Backbone

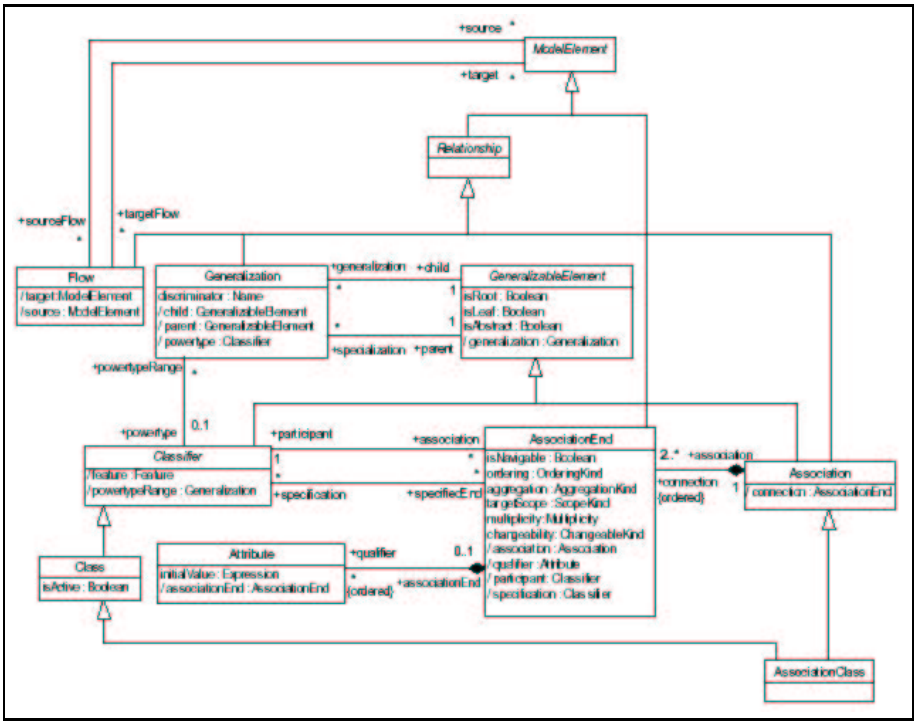

Figure 3.2: The UML Core Package - Relationships

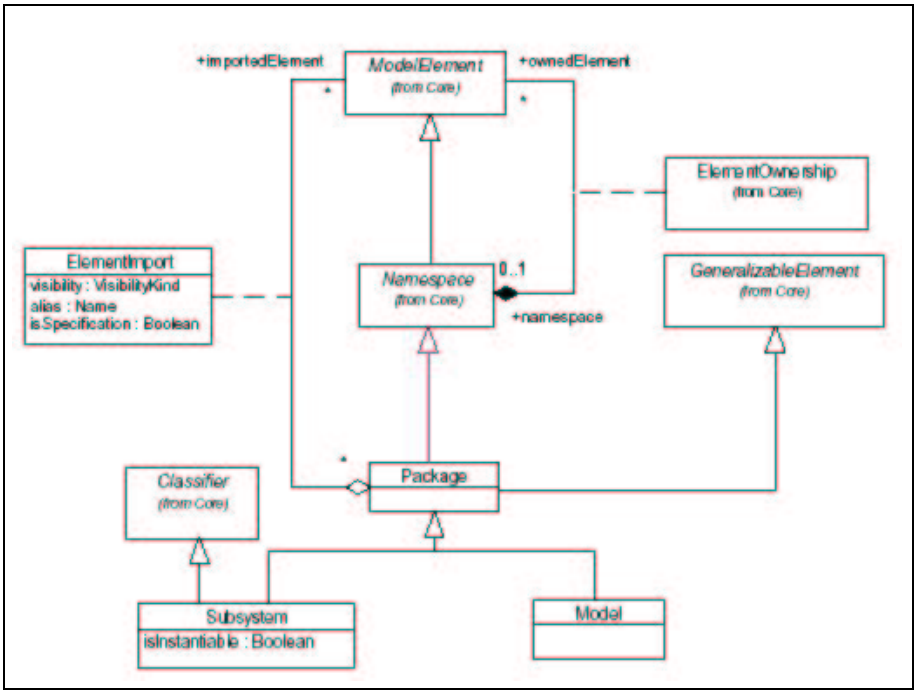

Figure 3.3: The UML Model Management Package

3.2 The Virtual Meta-model

3.2.1 Definition.

A virtual meta-model is a formal model of a set of UML extensions, expressed in UML. The virtual meta-model for the UML Profile for OFL-ML is presented in this chapter as a set of class diagrams. More information about virtual meta-models can be found in [OMG02, OMG01]. The semantics of stereotypes described in this virtual meta-model is given in the next sections.

Representation of Stereotypes. The virtual meta-model represents a *Stereotype* as a *Class* stereotyped $\ll$*stereotype*$\gg$. The *Class* that represents the Stereotype is the client of a Dependency stereotyped $\ll$*baseElement*$\gg$, whose supplier is the UML meta-model element being extended.

Representation of Tagged Values. The virtual meta-model represents a *TaggedValue* associated with a Stereotype as an *Attribute* of the *Class* that represents the Stereotype. The Attribute is stereotyped $\ll$*TaggedValue*$\gg$. An expression of the form $< x, y, ..., z >$ indicates that the *TaggedValue* value is a comma-delimited tuple. An expression of the form $(x, y, ..., z)$ indicates that the value is an enumeration.

A big challenge for OFL-ML is to generate a clean and understandable profile in an automatically way. To following rules are specified in order to address this objective:

- every OFL-*component* will be represented through an individual stereotype
- every combination of the characteristics values of an element of OFL which is non-customizable (reified by OFL-*atoms*) generates a different stereotype. This rule is based on the UML stereotype definition:"... *a stereotype may be used to indicate a difference in meaning or usage between two model elements with identical structure*".

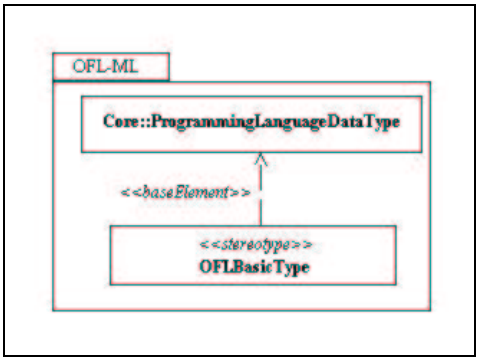

Figure 3.4: Virtual Model for OFL Basic Types

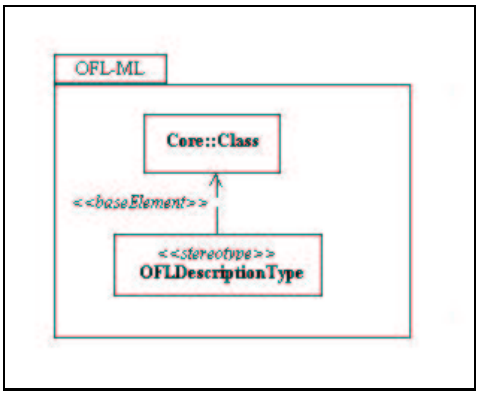

Figure 3.5: Virtual Model for OFL-description Components

- additional OFL-elements (like OFL-*modifiers* or OFL-*assertions*) are associated to tagged values or constraints which are created in the generated profile.

3.2.2 Virtual Meta-model of OFL-ML.

Figure 3.4 presents a stereotype used to model the basic types defined by a language. These types are managed as a characteristic of OFL-*language* component, which is actually a list. This Stereotype is derived from UML *programming language data type*.

Figure 3.5 shows a stereotype used to model OFL-*description* components. This stereotype is derived from UML *class*. An UML *class* is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics.

Figure 3.6 presents a stereotype used to model an *External Description*. This element does not exists in the OFL-model. It is defined only at the level of OFL-ML and it specifies a *Description* that has no OFL reification. It is necessary to include such entity in order to allow the integration of class libraries that have no OFL representation. The stereotype is derived from UML *classifier*.

Figure 3.7 shows how to represent an OFL-*package*. Generated profile will contain entities that inherit from this stereotype and denote specific language class organization mechanisms. The stereotype is derived from UML *package*.

In figure 3.8 we show the stereotypes used to represent OFL-*features*. Stereotypes are derived from UML *attribute* and *method*. Also, stereotypes are specialized according to the characteristics of OFL-*AtomAttribute* (*isDescriptionAttribute* and *isConstant*), and of OFL-*AtomMethod* (*isConstructor* and *isDestructor*).

Stereotypes for association end that belongs to OFL-*UseRelationship* are presented in figure 3.9. These stereotypes follows same rules as the stereotypes dealing with features. Figure

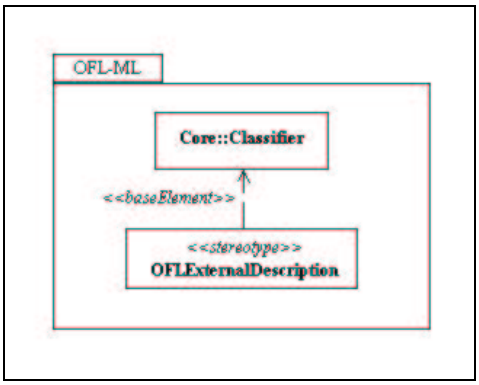

Figure 3.6: Virtual Model for External Description

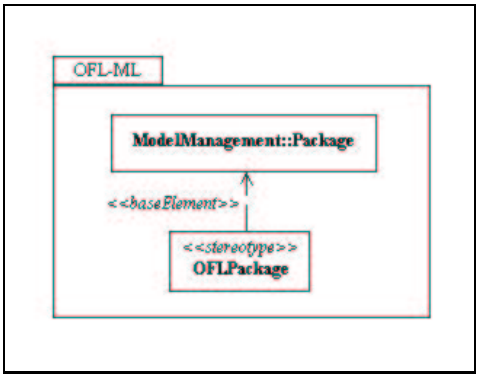

Figure 3.7: Virtual Model for OFL Package

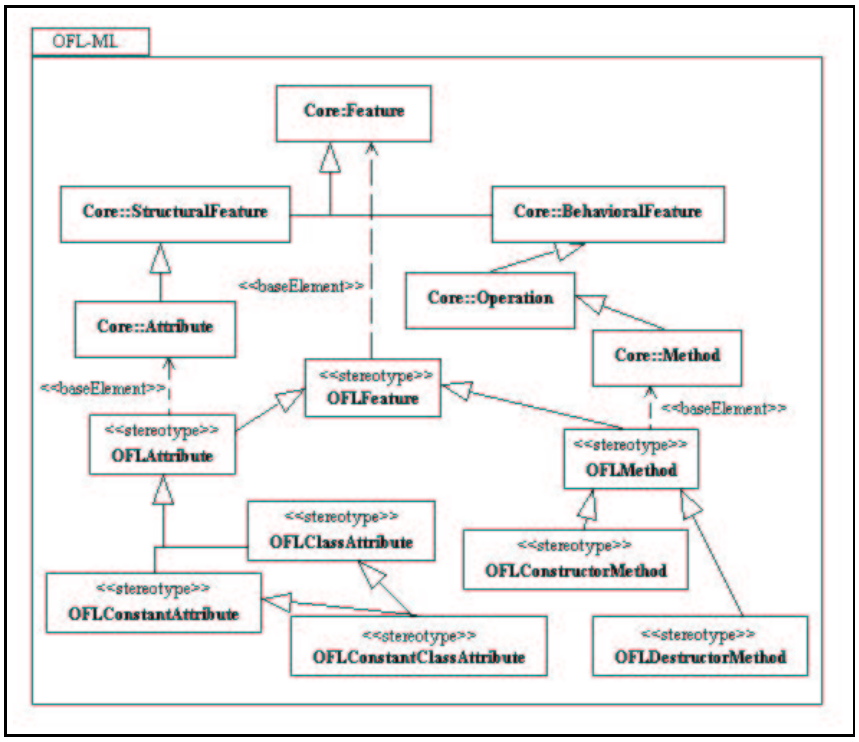

Figure 3.8: Virtual Model for OFL Features - attributes and methods

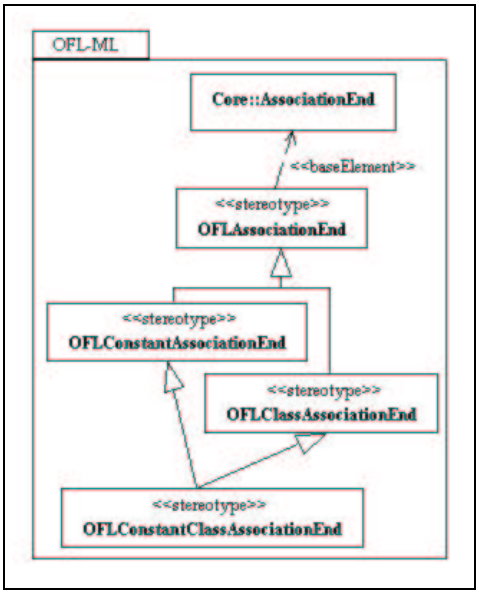

Figure 3.9: Virtual Model for Association End

3.10 presents stereotypes used to represent OFL-*relationships*. These stereotypes are derived from UML *generalization* and *association*.

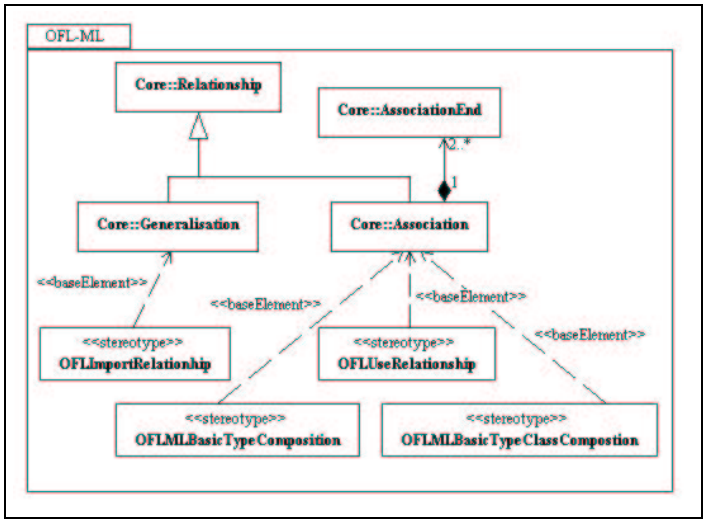

Figure 3.10: Virtual Model for OFL Relationships

Chapter 4

The OFL Type Representations

This section describes all the *Stereotypes* introduced in the Virtual Meta-model for OFL-*BasicType*, OFL-ML-*ExternalDescription* and OFL-*Description*. It adds the necessary *Tagged-Values*, *Constraints*, and *Common Model Elements* to complete the *Profile* specification.

These stereotypes could be used by modeling tools to generate corresponding instances of OFL elements and to fill them with appropriate information. Thereby, the following elements are considered to be generated: instances of *OFL-PrimitiveType* components and *OFL-Description* components. The result will be an OFL representation for application in XML.

4.1 The OFL BasicType Element

An OFL *BasicType* is a model of a primitive type found in the language binding such as *int*, *boolean*, *char* (from Java) etc.

4.1.1 Stereotypes and Tagged Values.

The OFL-ML basic types are represented by UML *ProgrammingLanguageDataType* from *Core* package with the ≪*OFLBasicType*≫ stereotype.

4.1.2 Constraints.

All ≪*OFLBasicType*≫ stereotyped elements is strongly related to the characteristic identified by *OFL-language.basicTypes*.

4.1.3 Elements Generation.

A profile element stereotyped ≪*OFLBasicType*≫ will be generated for each element of the list *OFL-language.basicTypes*. All strings contained by this list will became a name of one profile element.

4.1.4 Example.

If we consider Java language, eight elements will be considered. Those elements will have following names:

- boolean
- char
- byte
- short

- int
- long
- float
- double

4.2 The OFL Description Element

OFL Description Components represent the reification of Class types in different programming languages. They are created by the meta-programmer during the language modeling phase. If we consider the support for automatic code generation, OFL-ML has to include the representation of elements for all these components.

4.2.1 Stereotypes and Tagged Values.

The abstract stereotype ≪*OFLDescriptionType*≫ is the base for all the concrete stereotypes representing OFL Description of the considered language. The name of a generated stereotype is the name of the corresponding OFL component but without "Component" prefix. For a component *ComponentJavaClass*, a stereotype named ≪*JavaClass*≫ will be created.

Tagged values are created to express all OFL-*modifiers* associated with that component. These tags have boolean values and take the name from the attribute *keyword* of the modifier.

4.2.2 Constraints.

Constraints related with components stereotypes must address parameter values, characteristics and associated OFL Modifiers constraints for that component. Not all OFL parameters are considered but only those which have an impact on the static model of the application[1].

This paragraph presents constraints that have to be generated for all stereotypes derived from abstract stereotype ≪*OFLDescriptionType*≫. Each of them will consider parameter values, characteristics and modifiers associated with corresponding OFL component. Thus all constraints related with the stereotype ≪*JavaClass*≫ take into account parameter values, characteristics and modifiers associated with the component *ComponentJavaClass* defined for OFL-Java.

Parameter ConceptDescription::attribute. This parameter specify if the description could declare or not attributes. Legal values are *allowed* and *forbidden*. The constraint related with the value *forbidden* of this parameter will ensure an empty attribute compartment:

```
context: OFLDescriptionType (Core::Class)
self.allAttributes->size = 0
```

The operation *allAttributes* results in a Set containing all Attributes of the Class itself and all its inherited Attributes. It is defined in [OMG03] as a standard operation on classifiers.

```
allAttributes : set(Attribute);
allAttributes =
       self.allFeatures->select(f | f.oclIsKindOf(Attribute))
```

[1] See the full list of information associated to each OFL-component in the annexes, chapter 10.

Parameter ConceptDescription::methods. This parameter specifies if the description could declare or not methods. Legal values are *allowed* and *forbidden*. The constraint related with the value *forbidden* of this parameter will ensure an empty method compartment:

```
context: OFLDescriptionType(Core::Class)
self.allMethods->size = 0
```

The operation *allMethods* results in a Set containing all Methods of the Class itself and all its inherited Methods.

```
allMethods : set(Methods);
allMethods =
       self.allFeatures->select(f | f.oclIsKindOf(Method))
```

OFL Modifiers Constraints. All modifier constraints defined for the considered description component will be added in the generated profile. These constraints have to be transformed to deal with profile tagged values and stereotypes instead of OFL entities. Transformations that should be made to deal with profile tagged values are very basic. The purpose is to translate attributes of *OFL-Atoms* and *OFL-Components* into the corresponding tagged values.

According to modifier assertions which deal with *OFL-Description components*, only parameter *modifier* inherited from *OFL-AtomDescription* is involved. It has to be translated into a *taggedValue* which has the same name as the modifier. Indeed, transformations rely on the following two rules:

- *Syntax:*

  ```
  self.modifiers->includes('modifier_name')
  ```

  is translated in:

  ```
  self.stereotype.taggedValue
          ->select(name = 'modifier_name')->size = 1
  ```

- *and syntax:*

  ```
  NOT self.modifiers->includes('modifier_name')
  ```

  is translated in:

  ```
  self.stereotype.taggedValue
          ->select(name = 'modifier_name')->size = 0
  ```

These constraints check whether a tagged value exists or not. Each tagged value corresponds to a given modifier for the entity which is considered.

4.2.3 Elements Generation.

A profile stereotype derived from ≪*OFLDescriptionType*≫ will be generated for each OFL component. For a language which contains description types that are reified in OFL by components, such as ComponentLanguageDescriptionType1, ComponentLanguageDescriptionType2 etc, the resulting hierarchy is presented in figure 4.1.

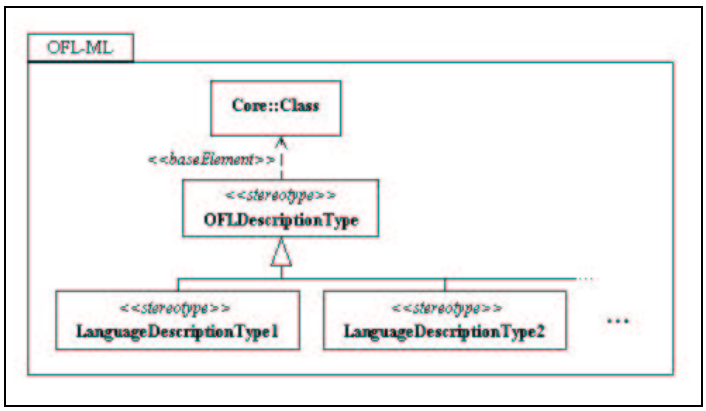

Figure 4.1: Generated stereotypes for Descriptions Components

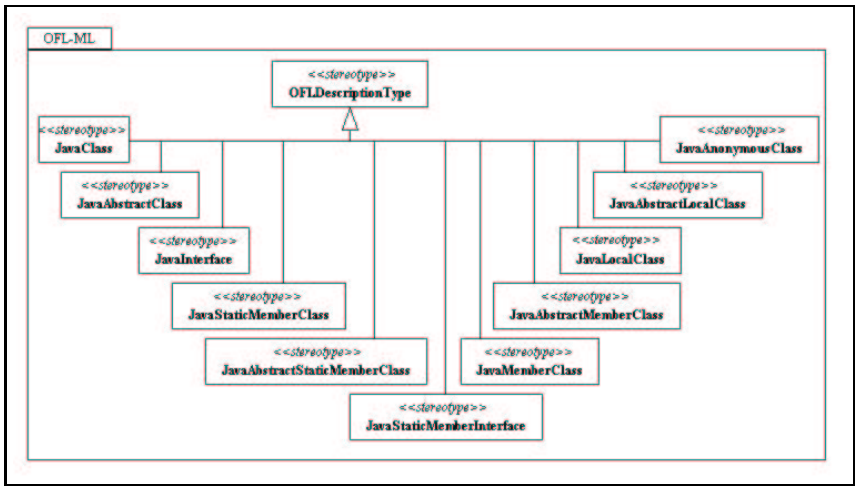

Figure 4.2: Generated stereotypes for OFL-Java Descriptions Components

\ Modifier Description	Basic Access Control	Complex Access Control	Optimization	Service	Additional
Class	public, package	final	strictfp	-	-
AbstractClass	public, package	-	strictfp	-	-
Interface	public, package	final	strictfp	-	-
StaticMemberClass	public, protected private, package	final	strictfp	- -	- -
AbstractStaticMemberClass	public, protected private, package	-	strictfp	- -	- -
StaticMemberInterface	public, protected private, package	final	strictfp	- -	- -
MemberClass	public, protected private, package	final	strictfp	- -	- -
AbstractMemberClass	public, protected private, package	-	strictfp	- -	- -
LocalClass	-	final	strictfp	-	-
AbstractLocalClass	-	final	strictfp	-	-
AnonymousClass	-	final	strictfp	-	-

Table 4.1: Modifiers for Java Description Components

4.2.4 Example.

Considering Java language, following description types are identified [CCL02, Cre01a]: *class*, *abstract class*, *interface*, *static member class*, *abstract static member class*, *static member interface*, *member class*, *abstract member class*, *local class*, *abstract local class* and *anonymous class*. Indeed, the OFL model for Java will contains eleven components derived from OFLComponentDescription.

Stereotypes generated for Java language are shown in figure 4.2.

Modifiers supported by these description components are summarized in table 4.1.

Table 4.2 presents the generated tagged values corresponding to these modifiers.

Stereotype	**Tagged Values**
JavaClass	{public}, {package}, {final}, {strictfp}
JavaAbstractClass	{public}, {package} {strictfp}
JavaInterface	{public}, {package}, {final}, {strictfp}
StaticMemberClass	{public}, {protected}, {final}, {strictfp} {private}, {package}
AbstractStaticMemberClass	{public}, {protected}, {strictfp} {private}, {package}
StaticMemberInterface	{public}, {protected}, {final}, {strictfp} {private}, {package}
MemberClass	{public}, {protected}, {final}, {strictfp} {private}, {package}
AbstractMemberClass	{public}, {protected}, {strictfp} {private}, {package}
LocalClass	{final}, {strictfp}
AbstractLocalClass	{final}, {strictfp}
AnonymousClass	{final}, {strictfp}

Table 4.2: Tagged Values for Java Description Components Stereotypes

No OFL-Java component has OFL parameters *ConceptDescription::attribute* and *ConceptDescription::methods* set to *forbidden*. Indeed, even Java interface could have attributes (*final static*). As a result, no constraints will be added to the generated profile for these parameters.

For Java components, only constraints dealing with incompatible modifiers are defined regarding basic access control modifiers and optimization modifiers.

If we consider JavaClass component, action control modifier assertion for that component is:

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

The constraint after the transformation which is included in the generated profile is very close from the original one:

```
context JavaClass::OFLDescriptionType (Core::Class)
inv: self.stereotype.taggedValue
             ->select(name='public')->size=1
    implies
     self.stereotype.taggedValue
             ->select(name='package')->size=0
```

Complex modifier *final* will be considered when the constraints associated to relationships will be described.

4.2.5 Additional constraints.

OFL parameters, characteristics and modifiers do not cover all language semantics. At the stage of our work there is no way to extract automatically constraints from the body of OFL actions. To take this situation into account the meta-programmer must add additional constraints. These constraints follow the same rules as OFL Assertions added with the same goal. As an example, if we consider Java Interfaces, the following rule has to be specified:

An interface should not contain attributes that are not final (constant) and static (class attribute).

This rule will have an associated OFL-assertion at the level of ComponentJavaInterface.

```
context: ComponentJavaInterface inv: self->features->forAll(
          a:OFLAttribute |
                a.isConstant and a.isDescriptionFeature )
```

The OCL constraint added into the profile in order to implement this rule is (for transformation see Section 5.1):

```
context: JavaInterface:OFLDescriptionType(Core::Class)
self->allAttributes
      ->forAll ( a | a.oclIsKindOf(Attribute) implies
            a.isStereokinded("OFLConstantClassAttribute") )
```

4.3 The External Description Element

The *External Description* element does not exists in the OFL-model. It is defined at the level of OFL-ML and specify a Description which is not described with OFL but with an external language, so that it is not associated to any OFL information and it is outside the scope of OFL. This element is useful especially when an application is linked to descriptions coming from existing class libraries which are not imported into the OFL universe.

OFL-ML could not handle the *External Descriptions* in the same manner as normal OFL-*Descriptions* are treated. The main impediment is their *opacity*. Their internal structures are hidden and could not be seen through usual OFL-*relationships*. As consequence, only very few profile constraints could be defined for them.

OFL-ML defines special relationships to deal with external descriptions. Those relationships are called "external relationships". For more information see the section "External Relationships". An external description could be involved only in external relationships and can act only as a target.

The usage of external descriptions is adequate only if the goal of OFL-application modeling is to obtain executable code. Control of semantics involved by these entities is done in that case by the native compiler or linker of the language of the generated code.

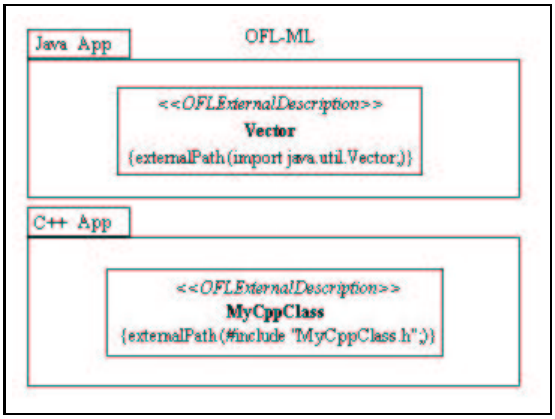

Figure 4.3: Example of using External Description Stereotype

4.3.1 Stereotypes and Tagged Values

There is only one stereotype involved in the representation of an external description. It is presented in figure 3.6.

Moreover, one tagged value is also specified. This is the *taggedValue* { externalPath = *importPathSpecification* }. It allows to specify the location of the resource. The value of this tag is a string that depends much on the syntax used by the language in order to integrate existing resources. For example, following values are legal : "*import java.util.Vector*" for Java or "*#include 'MyApp.h'*" for C++.

4.3.2 Constraints

The use of external descriptions is strongly linked with specific language semantics. Only very light control may be achieved. Constraints related with external descriptions are added at the level of relationships that could involve these elements.

4.3.3 Elements Generation

Only one profile element stereotyped as ≪OFLExternalDescription≫ will be generated. As it has been already mentioned, this stereotype will be tagged with an *externalPath* tagged value. The value of this tag will be included in the generated source file. For models that are intended to be used for other purposes than the generation of executable code, this tag may be ignored.

For languages with a complex syntax for addressing resources, the meta-programmer may define additional tags for this stereotype.

4.3.4 Example

Figure 4.3 shows some examples of the description of external descriptions for Java and C++.

Chapter 5

The OFL Feature Representations

OFL-Features deals with features declared by an OFL-Description. They describe the state (attributes) and the behavior (methods) of the description which is considered. Every feature is associated to a name and a list of modifiers.

These stereotypes could be used in modeling tools to generate corresponding instances of OFL elements and to fill them with appropriate information. Thereby, the following elements are considered to be generated: instances of *OFL-Attribute* atom and *OFL-Method* atom.

5.1 The OFL Attributes

An Attribute inherit from the properties of a feature and keep the values that describe the state of the description. An attribute has a name, a type, an initial value and a set of modifiers. Whether the attribute type is basic or not, we have to consider one of the two following cases :

- An attribute definition whose type is a language basic type (modeled as an OFLBasicType) is represented as an UML Attribute. This is an attribute of a Class stereotyped with a stereotype derived from ≪OFLBasicType≫.
- An OFL-attribute whose type is an OFL-description is represented as an UML Association. This association is between the Class stereotyped with a stereotype derived from the ≪OFLDescriptionType≫ which declares the attribute and the UML stereotype that represents the OFL-description type of the attribute. The name of the attribute is used as the role name for the attribute type AssociationEnd of this Association.

5.1.1 Stereotypes and Tagged Values.

Instance Attributes. Whenever a new instance of a description is created, a new attribute associated with that instance is created for all *instance attributes*. OFL handles them by setting the value of *isDescriptionAttribute* characteristic of the *AtomAttribute* instance to *false*. OFL-ML represents those attributes using ≪OFLAttribute≫ stereotype for basic type attributes or ≪OFL-AssociationEnd≫ for attributes that represent aggregation with other descriptions.

Class Attributes. For a given class it exists exactly one incarnation of each *class attribute*, no matter how many instances (possibly zero) of the class may eventually be created. For such attributes the characteristic *isDescriptionAttribute* of *AtomAttribute* instance, is set with *true*. OFL-ML represents these attributes using ≪OFLClassAttribute≫ stereotype for basic type attributes and ≪OFLClassAssociationEnd≫ for attributes that represent aggregation with other OFL-descriptions.

Constant Attributes. Constant attributes are attributes that could not change their value after initialization. OFL uses the characteristic *isConstant* of *AtomAttribute* in order to take them into account. If this characteristic is set to *true*, the attribute is constant and OFL-ML will represent it through ≪OFLConstantAttribute≫, ≪OFLConstantClassAttribute≫, ≪OFLClassAssociationEnd≫, respectively ≪OFLConstantClassAssociationEnd≫ stereotype.

Tagged values are created to express all OFL-*modifiers* associated with an OFL-*attribute*. These tags have boolean values and take the name from the attribute *keyword* of a modifier.

5.1.2 Constraints.

All modifiers constraints defined for *AtomAttribute* will be added in the generated profile. For incompatible modifiers, constraint transformation is the same as presented in Section 4.2. Transformation of constraints regarding stereotypes for attributes are the following:

- *Syntax*:

```
a.isConstant
```

  is translated into:

```
a.isStereokinded("OFLConstantAttribute")
```

This transformation refers to constant attributes. OFL uses *AtomAttribute.isConstant* to keep this information. OFL-ML will represent this as an UML Attribute stereokinded as ≪OFLConstantAttribute≫.

- *Syntax*:

```
a.isDescriptionAttribute
```

  is translated into:

```
a.isStereokinded("OFLClassAttribute")
```

This transformation refers to class attributes. OFL uses *AtomAttribute.isDescriptionAttribute* to keep this information. OFL-ML will represent this as an UML Attribute stereokinded as ≪OFLClassAttribute≫.

- *Syntax*:

```
a.isConstant
    AND
a.isDescriptionAttribute
```

  is translated into:

```
a.isStereokinded("OFLConstantClassAttribute")
```

This transformation refers to class attributes that are constant. OFL uses *AtomAttribute.isConstant* and *AtomAttribute.isDescriptionAttribute* to keep this information. OFL-ML will represent this as an UML Attribute stereokinded as ≪OFLConstantClassAttribute≫.

Stereotype	Applies To	Definition
≪OFLAttribute≫	Attribute	An attribute of a basic type
≪OFLConstantAttribute≫	Attribute	A constant attribute of a basic type
≪OFLClassAttribute≫	Attribute	A class attribute of a basic type
≪OFLConstantClassAttribute≫	Attribute	A constant class attribute of a basic type
≪OFLAssociationEnd≫	Attribute	An attribute that represents an OFL use relationship
≪OFLConstantAssociationEnd≫	Attribute	A constant attribute that represents an OFL use relationship
≪OFLClassAssociationEnd≫	Attribute	A class attribute that represents an OFL use relationship
≪OFLConstantClassAssociationEnd≫	Attribute	A constant class attribute that represents an OFL use relationship

Table 5.1: OFL-ML Attribute Stereotypes

5.1.3 Elements Generation.

Four profile stereotypes will be generated automatically for the basic-type attributes and four for the association-ends that correspond with relationships known as OFL-UseRelationships. These stereotypes are presented in table 5.1.

To increase the expressiveness of the generated profile, meta-programmer can derive new stereotypes from ≪OFLAttribute≫ and can choose for them a suggestive name such as ≪OFLJavaStaticAttribute≫, ≪OFLJavaFinalAttribute≫ or ≪OFLJavaFinalStaticAttribute≫. Same work could be done also for *AssociationEnd* stereotypes. In order to make this task easier, some "wizard" may be added to the profile generator tool. The additional stereotypes will inherit from the standard ones, all generated constraints.

5.1.4 Example.

Table 5.2 shows some profile elements which are mapped to Java attributes.

Stereotype	Java Mapping	Example
OFLJavaAttribute (:OFLAttribute)	instance non-final Java basic types attributes (for Java basic types see Section 4.1)	char a
OFLJavaFinalAttribute (:OFLConstantAttribute)	instance final Java basic types attributes	final char a
OFLJavaStaticAttribute (:OFLClassAttribute)	static (class) non-final Java basic types attributes	static char a
OFLJavaFinalStaticAttribute (:OFLConstantClassAttribute)	static (class) final Java basic types attributes	final static char a
OFLJavaAssociationEnd (:OFLAssociationEnd)	instance non-final Java aggregation attributes	AClass a
OFLJavaFinalAssociationEnd (:OFLConstantAssociationEnd)	instance final Java aggregation attributes	final AClass a
OFLJavaStaticAssociationEnd (OFLClassAssociationEnd:)	static (class) non-final Java aggregation attributes	static AClass a
OFLJavaFinalStaticAssociationEnd (:OFLConstantClassAssociationEnd)	static (class) final Java aggregation attributes	final static AClass a

Table 5.2: OFL-ML Stereotypes of Java Attribute

Table 5.3 presents tagged values generated for modifiers associated with Java attributes.

These tagged-values correspond to *public*, *protected*, *package* and *private* access control modifiers, respectively *volatile* optimization modifier and *transient* service modifier.

Stereotype	**Tagged Values**
OFLAttributes	{public}, {protected}, {private}, {package} {volatile}, {transient}
OFLConstantAttributes	{public}, {protected}, {private}, {package} {transient}
OFLClassAttributes	{public}, {protected}, {private}, {package} {volatile}, {transient}
OFLConstantClassAttributes	{public}, {protected}, {private}, {package} {transient}
OFLAssociationEnd	{public}, {protected}, {private}, {package} {volatile}, {transient}
OFLConstantAssociationEnd	{public}, {protected}, {private}, {package} {transient}
OFLClassAssociationEnd	{public}, {protected}, {private}, {package} {volatile}, {transient}
OFLConstantClassAssociationEnd	{public}, {protected}, {private}, {package} {transient}

Table 5.3: Tagged Values for Java Attribute Stereotypes

5.2 The OFL Methods

Methods inherit from features and specify the behavior of the description. Method elements may represent both procedures and functions. Functions differs from procedures because they return a result. The method declaration specifies a list of parameters. This list could be empty or not. If not, it contains a list of OFL-*parameter* elements. Abstract methods are methods that are not implemented. An abstract methods has an empty body. Additionally, OFL make the distinction between normal methods, constructors and destructors.

5.2.1 Stereotypes and Tagged Values

Three stereotypes defined in the OFL-ML virtual meta-model are used also in the generated profile: ≪OFLMethod≫, ≪OFLConstructorMethod≫ and ≪OFLDestructorMethod≫. In addition, an ≪OFLParameter≫ is derived from UML-*parameter* element to express method parameters. The returned value is indicated thanks to the following UML convention : the method has a parameter which implements an attribute '*kind* = *return*'.

The standard attribute *body* of *UML-Method* element is used to keep the list of statements that represents the method body. UML represents it as a list of *ProcedureExpression*, which are actually strings. When the code is generated from the model, these strings have to be translated into OFL-*Statement* elements. Another possibility is to represent the body using the UML-ActionsSemantic Model. This option will be discussed at the end of this chapter.

For abstract methods, OFL-ML uses an attribute *isAbstract* inherited from UML-*Operation* element. If it is set to *true*, then the operation does not have an implementation and the method body will be empty. If *false*, the operation must have an implementation specified within the description or within one of its ancestors.

To end the method overriding, UML uses the boolean attribute *isLeaf* from *Operation*. If it set to *true*, then the implementation of the operation may not be overridden by a descendant class. If *false*, then the implementation of the operation may be overridden by a descendant class but it is not compulsory to be overridden. If we consider automatic generation of profile, OFL-ML cannot use directly this attribute. In OFL, rights about method overriding or redefining are specified through modifiers rather than characteristics.

Method parameters are represented as a list of UML-*parameter* elements. An UML-*parameter* is an unbound variable that can be changed, passed, or returned. A parameter may include a name, a type, and a direction of communication. If we consider the reification of parameter semantics (as the Eiffel *agent* parameter modifier) constraints have to be added at the level of these elements.

Other constraints could be added related to parameter semantics. The standard attribute *kind* of the UML-*parameter* element can take following values:

in An input Parameter (may not be modified).

out An output Parameter (may be modified to communicate information to the caller).

inout An input Parameter that may be modified.

return A return value of a call.

Tagged values are created to express all OFL-*modifiers* associated with an OFL-*method*. These tags have boolean values and take the name from the attribute *keyword* of a modifier.

5.2.2 Constraints

Some constraints are imported from UML semantics. In fact, any usage of standard UML attributes implies also constraints.

In this context, according to UML-BehavioralFeature which is inherited by an UML-Method, we have:

- All Parameters should have a unique name.

```
self.parameter->
        forAll(p1, p2 | p1.name = p2.name implies p1 = p2)
```

- The type of the Parameters should be included in the Namespace of the Classifier.

```
self.parameter->forAll( p |
    self.owner.namespace.allContents->includes (p.type))
```

Moreover, as for attributes, all modifiers constraints defined for AtomMethod must be added.

5.2.3 Elements Generation.

Four stereotypes will be generated for methods. These are ≪*OFLMethod*≫, ≪*OFLConstructorMethod*≫, ≪*OFLDestructorMethod*≫ and ≪*OFLParameter*≫. The first tree stereotypes apply to the UML Method element. The last one apply to the UML Parameter one. ≪*OFLConstructorMethod*≫ corresponds to an OFL-method which has the attribute *isConstructor* set to *true*. ≪*OFLDestructorMethod*≫ corresponds to an OFL-method which has an attribute *isDestructor* set to *true*. As mentioned in section 5.1, meta-programmer may decide to derive specific stereotypes from the generated ones in order to increase the expressiveness of the profile elements.

Generated tags will correspond to the OFL-method modifiers defined for the language which is considered.

No tags will be generated for abstract methods and or eventually for non-overriding methods. Instead, the profile will use standard UML attributes as mentioned in the previous section.

Following list presents the transformation rules for the constraints dealing with methods.

- *Syntax*:

```
m.isConstructor
```

is translated into:

```
m.isStereokinded("OFLConstructorMethod")
```

This transformation refers to constructor methods. OFL uses the characteristic *isConstructor* from *AtomMethod* to keep this information. OFL-ML will represent it as an UML Method stereokinded as ≪OFLConstructorMethod≫.

- *Syntax*:

```
m.isDestructor
```

is translated into:

```
m.isStereokinded("OFLDestructorMethod")
```

This transformation refers to destructor methods. OFL uses the characteristic *isDestructor* from *AtomMethod* to keep this information. OFL-ML will represent it as an UML Method stereokinded as ≪OFLDestructorMethod≫.

Both characteristics *body* and *parameters* are a collection whatever we consider OFL or UML. Collection operations may be applied on both in the same way.

5.2.4 Example.

Profile elements mapping to Java methods are presented in table 5.4.

Stereotype	Java Mapping	Example
OFLMethod (:Method)	standard Java method	returnType aMethod(listOfParameters)
OFLConstructorMethod (:OFLMethod)	a Java constructor method must have same name as the class itself and no return type	className(listOfParameters)
OFLFinalizeMethod (:OFLDestructorMethod)	a Java finalizer (not exactly a destructor)	protected void finalize()

Table 5.4: OFL-ML Stereotypes of Java Method

Tagged values generated for modifiers associated with Java methods are presented in table 5.5. This corresponds to *public*, *protected*, *package*, *private* and *final* access control modifiers, respectively *native* and *strictfp* optimization modifiers and *synchronized* service modifier.

To handle java language, an additional tagged value is necessary to express the exception mechanism. Considering that OFL does not provide any customization for exceptions handling, this tagged value have to be added manually. We propose a tag {javaThrows = string}. The value of this tag will represent a comma-delimited list of names of Java Exception Classes thrown by the method which is considered.

Constraints that correspond to access control modifiers are generated using the same translation as the one mentioned in the previous section. For native modifier the assertion has also to be transformed.

```
context AtomMethod
inv: self.modifiers->includes('native')
    implies
        self.isConstructor = false
```

Stereotype	Tagged Values
OFLMethod	{public}, {protected}, {private}, {package}, {final} {native}, {strictfp}, {synchronized}, {javaThrows}
OFLConstructorMethod	{public}, {protected}, {private}, {package} {javaThrows}
OFLFinalizeMethod	{protected} {javaThrows}

Table 5.5: Tagged Values for Java Method Stereotypes

```
      and
   self.body->isEmpty()
      and
   NOT self.modifiers->includes('synchronized')
```

Transformation are made using transformation rules which had been already presented.

```
context OFLMethod (Core::Method)
inv: self.stereotype.taggedValue
            ->select(name = 'native')->size = 1
     implies
        NOT self.isStereotyped('OFLConstructorMethod')
           and
        self.body->isEmpty()
           and
        self.stereotype.taggedValue
               ->select(name = 'synchronized')->size = 0
```

5.2.5 Additional constraints.

As we mentioned in 4.2, the generated constraints do not cover all the semantics of the language. For example, in Java, all method parameters must have an attribute *kind* set to *in*, except one that is set to *return*. Moreover, a Java method cannot be abstract unless it is declared in a Java Interface or a Java abstract class.

```
context OFLMethod (Core::Method)
inv: let owner:Classifier = self.specification.owner
     in
     ( owner.isStereokinded('JavaAbstractClass')
        or
       owner.isStereokinded('JavaAbstractMemberClass')
        or
       owner.isStereokinded('JavaAbstractStaticMemberClass')
        or
       owner.isStereokinded('JavaAbstractLocalClass')
        or
       owner.isStereokinded('JavaInterface')
        or
       owner.isStereokinded('JavaStaticMemberInterface'))
```

Following Java constraint is related with a *finalize* method. A *finalize* method has to *i)* be declared as *protected*, *ii)* not return any result (it must have *void* as return type) and *iii)* throw *Throwable* exception.

```
context OFLFinalizeMethod (Core::Method)
inv: self.stereotype.taggedValue
```

```
        ->select(name = 'protected')->size = 1
   and
self.parameter->select(p |
       p.kind = return
       implies
           ( p.type.isStereotyped('OFLBasicType')
               and
           p.type.name = 'void')
       )
   and
self.stereotype.taggedValue
          ->select(tag | tag.name = 'javaThrows'
                       implies tag.value = 'Throwable')
```

Chapter 6

The OFL Relationship Representations

This chapter describes all the Stereotypes introduced in the Virtual Meta-model for OFL-ImportRelationship and OFL-UseRelationship. It also adds the necessary TaggedValues, Constraints, and Common Model Elements to fulfill the profile.

These stereotypes could be used by modeling tools in order to generate corresponding instances of OFL elements and to fill them with the appropriate information. Thereby, the following elements are considered to be generated: instances of OFL-*ImportRelationship* components and OFL-*UseRelationship* components.

Current work on OFL-ML does not consider dynamic relationships which are reified by OFL-*ObjectToClassRelationship* and OFL-*ObjectToObjectRelationship*. The reason is that OFL-ML profiles may represent only static models corresponding to UML Static Class Diagrams.

6.1 The OFL Import Relationship

The OFL-import relationship is a generalization of the inheritance mechanism found in object oriented languages. The meta-programmer has the responsibility to create an OFL relationship component for each import relationships existing in the language which is modeled. OFL-ML generates the necessary elements in order to represents all these components.

6.1.1 Stereotypes and Tagged Values.

The abstract stereotype ≪OFLImportRelationship≫ is the base for all the concrete stereotypes representing OFL *ImportRelationship* components of the language which is considered. The name of a generated stereotype is the same as the one of the related OFL component but without "Component" prefix. For example a component "ComponentJavaExtends", leads to the creation of a stereotype named ≪JavaExtends≫.

All relationships stereotyped as specialization of ≪OFLImportRelationship≫ will have an associated set of tagged values. Values of these elements correspond to some of the *Atom-Relationship* characteristics. These tagged values are presented in table 6.1.

In addition, one tagged value will be generated for each modifier associated with a relationship component.

6.1.2 Constraints.

All modifiers constraints defined in relationship components will be added to the profile. Transformation rules ensure that all characteristics of relationships components lead to the creation of the corresponding tagged values. Following rules will apply:

TaggedValue Name	**TaggedValue Value**	**Comment**
abstractedFeatures	string (list of feature names)	list of concrete methods that are abstracted
effectedFeatures	string (list of feature names)	list of abstract methods that are effected
hiddenFeatures	string (list of feature names)	list of features that are hidden
redefinedFeatures	string (list of feature names)	list of features that are redefined
renamedFeatures	string (list of feature names)	list of features that are renamed
removedFeatures	string (list of feature names)	list of features that are removed
shownFeatures	string (list of feature names)	list of features that pass the relationship unchanged

Table 6.1: OFL-ML Tagged Values for OFLImportRelationship

- *Syntax:*

```
self.relationshipCharacteristic->forall(f:Feature |
        f.modifiers->includes('modifier_name'))
```

is translated in:

```
self.stereotype.taggedValue
        ->forall(t:taggedValue |
            ( t.name = 'relationshipCharacteristic' and
              t.values->includes(feature_name) )
                          imply
               self.parent.features->forall(f:Feature |
                  f.name = feature_name imply
                      f.stereotype.taggedValue->
                         select(name = 'modifier_name')->
                             size = 1))
```

Following example applies to the Java *private* modifier in the context of the ≪JavaClassExtends≫ stereotype.

- *Syntax:*

```
self.hiddenFeature->forall(f:Feature |
        f.modifiers->includes('private'))
```

is translated in:

```
self.stereotype.taggedValue
        ->forall(t:taggedValue |
            ( t.name = 'hiddenFeatures' and
              t.values->includes(feature_name) )
                          imply
               self.parent.features->forall(f:Feature |
                  f.name = feature_name
                              imply
```

```
                    f.stereotype.taggedValue->
                       select(name = 'private')->
                           size = 1))
```

Additionally, the generated profile will contains constraints regarding each stereotype which corresponds to the relationship components of the language. The generic name *ComponentRelationship* designates these stereotypes. Indeed, each OFL-ML generic constraint presented next will have one instance for each component into the generated OFL-ML Profile.

Parameter ConceptRelationship::cardinality. This parameter specifies the cardinality of relationship as an integer value n in the meaning of cardinality *1-n*. This specify that relationship has one source (child) description and could have between *1* and n target (parent) descriptions. As an example, for simple inheritance n = *1* and the cardinality is *1-1*. For a general relationship n could be ∞.

The constraint related with this parameter checks the conformance with cardinality specification. If *cardinality* is ∞ no constraint is necessary.

OFL-ML: if cardinality $\neq \infty$

```
context ComponentRelationship(OFLImportRelationship)
inv: self.child.generalization->select( gen |
        gen.isStereotyped('ComponentRelationship')
          and
        gen.child = self.child)->size = n
```

Parameter ConceptRelationship::repetition. This parameter indicates if a direct repetition of target (parent) is permitted or not. The possible values of this parameter are *allowed* and *forbidden*. Value *allowed* make sense only for a relationship with the cardinality $n > 1$ (1-n).

If the *cardinality* value n is *1* or if the *repetition* value is *allowed*, no constraint is necessary.

OFL-ML: if cardinality $\neq$ *1 and repetition* = *forbidden*

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.parent.generalization->select( gen |
        gen.isStereotyped('ComponentRelationship')
          and
        gen.child = self.child)->size = 1
```

Parameter ConceptRelationship::circularity. This parameter specifies the possibility to create cycles using considered relationship component. Constraint make sense only if parameter contain value *forbidden*.

OFL-ML: if circularity = *forbidden*

```
context  ComponentRelationship (OFLImportRelationship)
inv: let dp(d:Classifier) =
          d.generalisation.select(g |
           g.isStereotyped('ComponentRelationship'))
                                      ->collect(g.parent) in
         allParents(p:Set(Classifier)) =
           self.dp(self.child)->union((self.dp(self.child)-p)
               ->collect(np |
                   np.allParents(p->including(self.child)))) in
         NOT self.child.allParents(Set{})->includes(self.child))
```

First, OCL *let* expression (*dp*) calculates all direct parents of a Classifier in the meaning of considered relationship. The expression *allParents* returns all parents of a *Classifier*. Parameter *p* contains all parents which are already visited and is used to stop recursions. The constraint checks if the source of the relationship is included or not in its list of parents.

Parameter ConceptRelationship::feature_variance. This parameter specifies the type of variance of relationship concerning method parameters, method result and attributes. The value is a triplet where each component can have one of the following values:

covariant elements that corresponds to a redefinition need to have the same type or a subtype as the original one (defined by the source).

contravariant elements that corresponds to a redefinition need to have the same type or a super-type as the original one (defined by the source).

nonvariant elements that corresponds to a redefinition must have the same type as the original one (defined by the source).

non_applicable parameter is not applicable

Constraint has to consider the first three values separately for each triplet component.

All constraint use the following definitions for *i)* direct parent and *ii)* all parents, of a Classifier:

```
context Classifier
def: directParent =
           self.generalisation->collect(g.parent)
def: allParents(p:Set(Classifier)) =
           self.directParent->union((self->directParent-p)
               ->collect(np | np.allParents(p->including(self))))
```

Constraints dealing with method parameters variance are now presented.

OFL-ML: if feature_variance for method parameter = covariant

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.redefinedFeatures->forAll(
       m | m.oclIsKindOf(Method) implies
         m.parameters->forAll(
           p | p.kind <> return
             implies
           self.source.features->forAll( rm |
              rm.oclIsKindOf(Method)
                 implies
              if (rm.name = m.name and
                   rm.parameters->count() = m.parameters->count())
                     rm.parameters->forAll( rp |
                       rp.name = p.name
                         implies
                       p.allParents(Set{})
                            ->including(p.type)->include(rp.type)))
           ))
```

OFL-ML: if feature_variance for method parameter = contravariant

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.redefinedFeatures->forAll(
       m | m.oclIsKindOf(Method) implies
         m.parameters->forAll(
```

```
            p | p.kind <> return
              implies
            self.source.features->forAll( rm |
               rm.oclIsKindOf(Method)
                 implies
               if (rm.name = m.name and
                    rm.parameters->count() = m.parameters->count())
                     rm.parameters->forAll( rp |
                       rp.name = p.name
                         implies
                       rp.allParents(Set{})
                             ->including(rp.type)->include(p.type)))
            ))
```

OFL-ML: if feature_variance for method parameter = nonvariant

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.redefinedFeatures->forAll(
       m | m.oclIsKindOf(Method) implies m.parameters->forAll(
            p | p.kind <> return
              implies
            self.source.features->forAll( rm |
               rm.oclIsKindOf(Method)
                 implies
               if (rm.name = m.name and
                    rm.parameters->count() = m.parameters->count())
                      rm.parameters->forAll( rp |
                        rp.name = p.name
                          implies
                        p.allParents
                            ->including(p.type)->include(rp.type)))
            ))
```

For method result variance constraints are the same but the term '*p.kind* <> *return*' is replaced by '*p.kind* = *return*'. Following specifications show the constraints for attribute variance.

OFL-ML: if feature_variance for attributes = covariant

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.redefinedFeatures->forAll(
       a | a.oclIsKindOf(Attribute) implies
            self.source.features->forAll( ra |
               ra.oclIsKindOf(Attribute)
                 implies
               ra.name = a.name
                   implies
               a.type.allParents
                        ->including(a.type)->include(ra.type)))
```

OFL-ML: if feature_variance for attributes = contravariant

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.redefinedFeatures->forAll(
       a | a.oclIsKindOf(Attribute) implies
            self.source.features->forAll( ra |
               ra.oclIsKindOf(Attribute)
                 implies
```

```
            ra.name = a.name
                implies
            ra.type.allParents->including(ra.type)
                                    ->include(a.type)))
```

OFL-ML: if feature_variance for method parameter = nonvariant

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.redefinedFeatures->forAll(
       a | a.oclIsKindOf(Attribute) implies
           self.source.features->forAll( ra |
              ra.oclIsKindOf(Attribute)
                implies
              ra.name = a.name
                 implies
              a.type=ra.type)))
```

Parameter ConceptRelationship::abstracting. This parameter specifies if the relationship permits or not to abstract methods (to transform methods that pass relationship from implemented to abstract status). Permitted values are *mandatory*, *allowed* and *forbidden*. The OFL-ML constraint for this parameter refers only to the first and last values.

OFL-ML: if abstracting = mandatory

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.parent.features->forAll(
         m | m.oclIsKindOf(Method) implies
            NOT m.isAbstract
               implies
            self.stereotype.taggedValue
                ->forAll(t | t.name='abstractedFeatures'
                    implies t.value->include(m)))
```

OFL-ML: if abstracting = forbidden

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.stereotype.taggedValue
             ->select(name='abstractedFeatures')->size=0
```

Parameter ConceptRelationship::effecting. This parameter specifies if the relationship permits or not to effect methods (to implements methods that pass relationship). Permitted values are *mandatory*, *allowed* and *forbidden*. The OFL-ML constraint for this parameter refers only to the first and last values.

OFL-ML: if effecting = mandatory

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.parent.features->forAll(
         m | m.oclIsKindOf(Method) implies
            m.isAbstract
               implies
            self.stereotype.taggedValue
                ->forAll(t | t.name='effectedFeatures'
                    implies t.value->include(m)))
```

OFL-ML: if effecting = forbidden

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.stereotype.taggedValue
             ->select(name='effectedFeatures')->size=0
```

Parameter ConceptRelationship::masking. The masking parameter establishes if the features could be hidden or not when they pass a relationship. Legal values are *mandatory*, *allowed* and *forbidden*.

OFL-ML: if masking = mandatory

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.parent.features->forAll(
         f:Feature |
            self.stereotype.taggedValue
                ->forAll(t | t.name='hiddenFeatures'
                    implies t.value->include(f)))
```

OFL-ML: if masking = forbidden

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.stereotype.taggedValue
              ->select(name='hiddenFeatures')->size=0
```

Parameter ConceptRelationship::redefining. This parameter indicates if the redefinition of features is *mandatory*, *allowed* or *forbidden*.

OFL-ML: if redefining = mandatory

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.parent.features->forAll(
         f:Feature |
            self.stereotype.taggedValue
                ->forAll(t | t.name='redefinedFeatures'
                    implies t.value->include(f)))
```

OFL-ML: if redefining = forbidden

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.stereotype.taggedValue
              ->select(name='redefinedFeatures')->size=0
```

Parameter ConceptRelationship::renaming. This parameter indicates if the renaming of features that pass a given relationship is *mandatory*, *allowed* or *forbidden*.

OFL-ML: if renaming = mandatory

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.parent.features->forAll(
         f:Feature |
            self.stereotype.taggedValue
                ->forAll(t | t.name='renamedFeatures'
                    implies t.value->include(f)))
```

OFL-ML: if renaming = forbidden

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.stereotype.taggedValue
              ->select(name='renamedFeatures')->size=0
```

Parameter ConceptRelationship::removing. This parameter establishes if the removal of features is *mandatory*, *allowed* or *forbidden*.

OFL-ML: if removing = mandatory

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.parent.features->forAll(
         f:Feature |
            self.stereotype.taggedValue
                ->forAll(t | t.name='removedFeatures'
                    implies t.value->include(f)))
```

OFL-ML: if removing = forbidden

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.stereotype.taggedValue
              ->select(name='removedFeatures')->size=0
```

Parameter ConceptRelationship::showing. This parameter is the opposite as masking. It indicates if the feature is made again visible after it was masked. Possible values are *mandatory*, *allowed* and *forbidden*.

OFL-ML: if showing = mandatory

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.parent.features->forAll(
         f:Feature |
            self.stereotype.taggedValue
                ->forAll(t | t.name='shownFeatures'
                    implies t.value->include(f)))
```

OFL-ML: if showing = forbidden

```
context  ComponentRelationship (OFLImportRelationship)
inv: self.stereotype.taggedValue
              ->select(name='showed-Features')->size=0
```

Characteristic AtomLanguage::validRelationships. This characteristic indicates the descriptions types that may act as sources and targets for a given kind of relationship. Values are triplets of <ComponentRelationship, ComponentDescriptionSource, ComponentDescriptionTarget>. OFL-ML will add a constraint that check for legal source type according to that.

In the next lines, we present the generated constraints according to the value {<ComponentRelationship, LanguageDescriptionTypeSource1, LanguageDescriptionTypeTarget1>, <ComponentRelationship, LanguageDescriptionTypeSource2, LanguageDescriptionTypeTarget2>, ... } for this characteristic.

```
context  ComponentRelationship (OFLImportRelationship)
inv: let st = self.child in
    (
       st.isStereotyped('LanguageDescriptionTypeSource1')
          or
       st.isStereotyped('LanguageDescriptionTypeSource2')
          or
       ...
    )

context  ComponentRelationship (OFLImportRelationship)
inv: let st = self.parent in
```

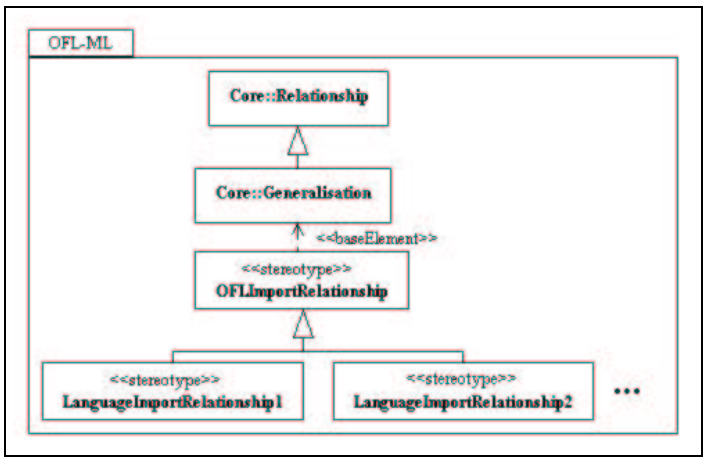

Figure 6.1: Generated stereotypes for Import Relationships Components

```
(
   st.isStereotyped('LanguageDescriptionTypeTarget1')
      or
   st.isStereotyped('LanguageDescriptionTypeTarget2')
      or
   ...
)
```

6.1.3 Elements Generation.

A profile stereotype derived from ≪OFLImportRelationship≫ will be generated for each OFL component. For a language which includes import relationships reified in OFL by the components ComponentLanguageImportRelationship1, ComponentLanguageImportRelationship2, etc, the resulting hierarchy is presented in figure 6.1.

Tagged values will be generated for each relationship component according to the values of OFL-*parameters*: *abstracting*, *effecting*, *masking*, *redefining*, *renaming*, *removing* and *showing*. Indeed, tags will be added taking into accounts the values *mandatory* and *allowed* for these parameters.

Constraints are generated regarding OFL-ML generation-conditions. These conditions will be described by statements like: *OFL-ML: if condition*

The test condition will be evaluated by the module that generates the OFL-ML profile.

6.1.4 Example.

Considering Java language, following import relationships are identified [CCL02, Cre01a]: between classes inheritance (JavaClassExtends), between interfaces inheritance (JavaInterfaceExtends), concretization (JavaConcretization) and implementation (JavaImplements).

TaggedValues that corresponds to these stereotypes are shown in table 6.2. Valid sources and targets for components are presented in table 6.3. Example of generated constraints for valid sources and targets for *JavaInterfaceExtends* relationship are given below.

```
context  JavaInterfaceExtends (OFLImportRelationship)
inv: let st = self.child in
    (
       st.isStereotyped('JavaInterface')
          or
       st.isStereotyped('JavaStaticMemberInteface')
```

Stereotype	Tagged Values
JavaClassExtends	{redefinedFeatures}, {hiddenFeatures} {effectedFeatures}
JavaInterfaceExtends	{redefinedFeatures}
JavaConcretization	{redefinedFeatures}, {hiddenFeatures} {effectedFeatures}(mandatory)
JavaImplements	{redefinedFeatures}, {effectedFeatures}

Table 6.2: Tagged Values for Java Import Relationship Components Stereotypes

```
    )

context  JavaInterfaceExtends (OFLImportRelationship)
inv: let st = self.parent in
    (
       st.isStereotyped('JavaInterface')
          or
       st.isStereotyped('JavaStaticMemberInteface')
    )
```

6.2 The OFL Use Relationships

The OFL-use relationship is a generalization of the aggregation mechanism found in object oriented languages. The meta-programmer has the responsibility to create an OFL relationship component for each kind of use relationships existing in the language which is modeled. OFL-ML generates the *stereotypes*, *tagged values* and *constraints* that are needed in order to represents all these components.

6.2.1 Stereotypes and Tagged Values.

The abstract stereotype ≪OFLUseRelationship≫ is the base for all the concrete stereotypes representing OFL *UseRelationship* components of the considered language. As for import relationships which had been presented in the section above, the name of a generated stereotype is the same as the name of the OFL component but without the "Component" prefix. for example a component "ComponentJavaAggregation", leads to the creation of a stereotype named ≪JavaAggregation≫.

In the same way as for import relationship, all use relationships stereotyped as specialization of ≪OFLUseRelationship≫ are associated to a set of tagged values which correspond to some of the characteristics of *AtomRelationship*. These tagged values are presented in table 6.4.

6.2.2 Constraints.

All associations which correspond to an OFL use relationship must have exactly two ends. They correspond to the source and target of the relationship.

```
context ComponentRelationship(OFLUseRelationship) inv:
self.allConnections->size = 2
```

Some constraints regarding parameters of OFL-*concept-relationship* which are generated for import relationships are also valid for use relationships. In this context, the OFLUseRelationship stereotype replaces OFLImportRelationship as ancestor of ComponentRelationship stereotype. Moreover, UML-*associations* attribute replaces the UML-*generalization*. This attribute is a set which contains all association relationships in which the classifier is involved.

Stereotype	Valid Sources	Valid Targets
JavaClassExtends	{JavaClass} {JavaAbstractClass} {JavaStaticMemberClass} {JavaAbstractStaticMemberClass} {JavaMemberClass} {JavaAbstractMemberClass} {JavaLocalClass} {JavaAbstractLocalClass} {JavaAnonymousClass}	{JavaClass} {JavaAbstractClass} {JavaStaticMemberClass} {JavaAbstractStaticMemberClass} {JavaMemberClass} {JavaAbstractMemberClass} {JavaLocalClass} {JavaAbstractLocalClass}
JavaInterfaceExtends	{JavaInterface} {JavaStaticMemberInteface}	{JavaInterface} {JavaStaticMemberInteface}
JavaConcretization	{JavaClass} {JavaStaticMemberClass} {JavaMemberClass} {JavaLocalClass} {JavaAnonymousClass}	{JavaAbstractClass} {JavaAbstractStaticMemberClass} {JavaAbstractMemberClass} {JavaAbstractLocalClass}
JavaImplements	{JavaClass} {JavaAbstractClass} {JavaStaticMemberClass} {JavaAbstractStaticMemberClass} {JavaMemberClass} {JavaAbstractMemberClass} {JavaLocalClass} {JavaAbstractLocalClass} {JavaAnonymousClass}	{JavaInterface} {JavaStaticInterface}

Table 6.3: Valid sources and targets for Java Import Relationship Components Stereotypes

According to the parameter ConceptRelationship::cardinality, the constraint will be the following after transformation :

OFL-ML: if cardinality $\neq \infty$

```
context ComponentRelationship(OFLUseRelationship)
inv: self.child.associations->select( assoc |
        assoc.isStereotyped('ComponentRelationship')
          and
        assoc.child = self.child)->size = n
```

The parameters which remain valid in the context of an use relationship are :

- cardinality
- repetition
- circularity
- masking
- renaming
- removing
- showing

TaggedValue Name	TaggedValue Value	Comment
hiddenFeatures	string (list of feature names)	list of features that are hidden
renamedFeatures	string (list of feature names)	list of features that are renamed
removedFeatures	string (list of feature names)	list of features that are removed
shownFeatures	string (list of feature names)	list of features that pass the relationship unchanged

Table 6.4: OFL-ML Tagged Values for OFLUseRelationship

Parameter ConceptRelation::dependence. This parameter specifies if the instances of the target description have a life time which is dependent or independent from the source description. Possible values are *dependent* and *independent*. This parameter has a meaning only for an use relationship.

OFL-ML links this parameter with the aggregation attribute of UML-association-End element. Possible values for this attribute are:

aggregate The target class is an aggregate; therefore, the source class is a part and must have the aggregation value set to *none*. The part may be contained in other aggregates. This value is mapped to *independent* values of the OFL *dependence* parameter.

composite The target class is a composite; therefore, the source class is a part and must have the aggregation value of none. The part is strongly owned by the composite and may not be part of any other composite. This value corresponds in OFL to the parameter *dependence* set to *dependent*.

OFL-ML: if dependence = independent

```
context ComponentRelationship(OFLUseRelationship)
inv: self.connection->select( assocEnd |
                        assocEnd.aggregation = aggregate )->size = 1
```

OFL-ML: if dependence = dependent

```
context ComponentRelationship(OFLUseRelationship)
inv: self.connection->select( assocEnd |
                        assocEnd.aggregation = composite )->size = 1
```

The constraints related with the characteristic AtomLanguage::validRelationships are the same as the one presented for import relationships (see section above).

6.2.3 Elements Generation.

OFL-ML generates one stereotype derived from ≪OFLUseRelationship≫ for each OFL use relationship component.

Tagged values are generated also for each use relationship according to the values of OFL-*parameters masking*, *renaming*, *removing* and *showing*. As it has been already mentioned, tags will be added taking into account the values *mandatory* and *allowed* of these parameters.

6.2.4 Example.

If we consider the Java language, we identify the following use relationship components [CCL02, Cre01a]: aggregation (JavaAggregation), class aggregation (JavaClassAggregation), composition (JavaComposition) and class composition (JavaClassComposition). Because the

last two components imply only Java primitive types, which correspond to OFL-ML basic types, they are represented by stereotypes derived from basic type composition (see section 6.3).

TaggedValues that correspond to these stereotypes are presented in the table 6.5. The deletedFeatures contain the features that are deleted when this relationship is crossed. This may correspond for example to features declared with *private* modifier.

Stereotype	**Tagged Values**
JavaAggregation	{deletedFeatures}
JavaClassAggregation	{deletedFeatures}

Table 6.5: Tagged Values for Java Use Relationship Components Stereotypes

Table 6.6 presents valid sources and targets for these relationships.

Stereotype	**Valid Sources**	**Valid Targets**
JavaAggregation	{JavaClass} {JavaAbstractClass} {JavaStaticMemberClass} {JavaAbstractStaticMemberClass} {JavaMemberClass} {JavaAbstractMemberClass} {JavaLocalClass} {JavaAbstractLocalClass} {JavaAnonymousClass}	{JavaClass} {JavaAbstractClass} {JavaInterface} {JavaStaticMemberClass} {JavaAbstractStaticMemberClass} {JavaMemberClass} {JavaAbstractMemberClass} {JavaLocalClass} {JavaAbstractLocalClass} {JavaAnonymousClass} {JavaStaticMemberInteface}
JavaClassAggregation	{JavaClass} {JavaAbstractClass} {JavaInterface} {JavaStaticMemberClass} {JavaAbstractStaticMemberClass} {JavaMemberClass} {JavaAbstractMemberClass} {JavaLocalClass} {JavaAbstractLocalClass} {JavaAnonymousClass} {JavaStaticMemberInteface}	{JavaClass} {JavaAbstractClass} {JavaInterface} {JavaStaticMemberClass} {JavaAbstractStaticMemberClass} {JavaMemberClass} {JavaAbstractMemberClass} {JavaLocalClass} {JavaAbstractLocalClass} {JavaAnonymousClass} {JavaStaticMemberInteface}

Table 6.6: Valid sources and targets for Java Use Relationship Components Stereotypes

Both constraints and tags will be added according to the parameters values.
For JavaAggregation we will have:

- cardinality = ∞ (no OFL-ML constraint)
- circularity = allowed (no OFL-ML constraint)
- repetition = allowed (no OFL-ML constraint)
- removing = allowed (no OFL-ML constraint but 'removedFeatures' generated tag)

For JavaClassAggregation we will have:

- cardinality = ∞ (no OFL-ML constraint)
- circularity = allowed (no OFL-ML constraint)

- repetition = allowed (no OFL-ML constraint)
- removing = allowed (no OFL-ML constraint but 'removedFeatures' generated tag)

6.3 The Basic Type Composition

Basic-type composition association stereotypes are used to represent composition with language primitive types. The relationship corresponds to the declaration of primitive-type attribute within a description. This relationship is always a composition because basic types instances represents values but not objects.

6.3.1 Stereotypes and Tagged Values.

Stereotypes have to be derived from two stereotypes ≪OFLMLBasicTypeCompo-sition≫ and ≪OFLMLBasicTypeClassComposition≫. The first represents instance association and the second represents class association. No tagged values are necessary.

6.3.2 Constraints.

An OFLMLBasicTypeComposition represents a composition.

```
context OFLMLBasicTypeComposition (Core::Association)
inv: self.connection->select( assocEnd |
                  assocEnd.aggregation = composite )->size = 1
```

An OFLMLBasicTypeComposition can have only OFLBasicType as target.

```
context OFLMLBasicTypeComposition (Core::Association)
inv: self.connection->forAll( assocEnd |
                  assocEnd.aggregation = composition
                     implies
                  assocEnd.participant.isStereokinded(OFLBasicType))
```

A ≪OFLBasicType≫-stereotyped Classifier may not participate in any Associations with navigable opposite AssociationEnds.

```
context OFLBasicType (Core::ProgrammingLanguageDataType)
inv: self.navigableOppositeEnds->isEmpty
```

An OFLMLBasicTypeComposition could have only OFLAssociationEnd as a target end.

```
context OFLMLBasicTypeComposition (Core::Association)
inv: self.connection->forAll( assocEnd |
                  assocEnd.aggregation = composition
                     implies
                  assocEnd.isStereotyped(OFLAssociationEnd))
```

An OFLMLBasicTypeClassComposition could have only OFLClassAssociationEnd as a target end.

```
context OFLMLBasicTypeClassComposition (Core::Association)
inv: self.connection->forAll( assocEnd |
                  assocEnd.aggregation = composition
                     implies
                  assocEnd.isStereotyped(OFLClassAssociationEnd))
```

6.3.3 Elements Generation.

Usually a maximum of two stereotypes are generated: one derived from ≪OFLMLBasicTypeComposition≫ and one from ≪OFLMLBasicTypeClassComposition≫. If the language which is considered have more than two type of relationships involving basic types, then additional constraints could be also necessary. No tagged values are necessary.

6.3.4 Example.

For Java language we have two relationship components that involve Java primitive-types: composition (JavaComposition) and class composition (JavaClassComposition). The JavaComposition stereotype is derived from ≪OFLMLBasicTypeComposition≫ and the JavaClassComposition is derived from ≪OFLMLBasicTypeClassComposition≫.

6.4 The External Import Relationship

External import relationships involve external descriptions. External descriptions are presented in section 4.3. They represent descriptions imported from external class libraries. These descriptions are usually opaque and they could not be involved in OFL relationships. OFL-ML uses the standard UML-*generalization* to represent these values.

6.4.1 Stereotypes and Tagged Values.

No stereotypes and tagged values are necessary.

6.4.2 Constraints.

Any generalization relationship that is not stereotyped has to have an external description as target.

```
context generalization
inv: self.stereotype->isEmpty
                implies
            self.parent.isStereokinded(OFLExternalType)
```

6.4.3 Elements Generation.

No stereotypes or tagged values are generated. Only the constraint mentioned above is added to the profile.

6.4.4 Example.

An example of using an external import-relationship in OFL-ML Java profile is presented in fig. 6.2.

6.5 The External Use Relationship

An external use-relationships involves external descriptions. The handling of external use-relationship is done in same way as for external import-relationship. OFL-ML uses the standard UML-*association* to represent these values.

6.5.1 Stereotypes and Tagged Values.

No stereotypes and tagged values are necessary.

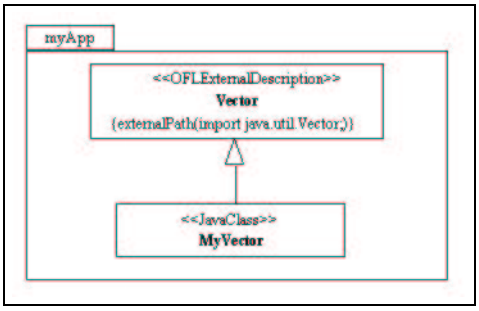

Figure 6.2: Example of using OFL-ML ExternalImportRelationship

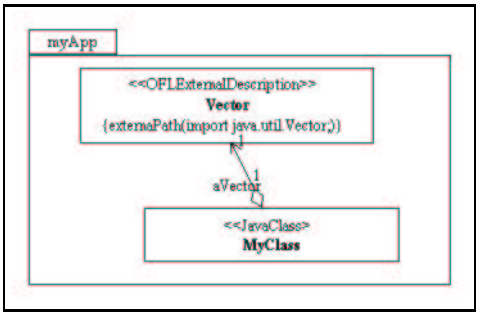

Figure 6.3: Example of using OFL-ML ExternalUseRelationship

6.5.2 Constraints.

Any association relationship that is not stereotyped has to have an external description at one end.

```
context association
inv: self.stereotype->isEmpty
        implies
     self.connection->select( assocEnd |
      assoEnd.participant.isStereotyped(OFLExternalDescription))
                                                        ->size = 1
```

6.5.3 Elements Generation.

No stereotypes or tagged values are generated. Only presented constraint is added to the profile.

6.5.4 Example.

An example of using an external use relationship in OFL-ML Java profile is presented in fig. 6.3.

Chapter 7

The OFL Model Organization

OFL organizes application elements into OFL-*packages*. An OFL-*package* contains a group of Descriptions, Relationships and other OFL-packages. An OFL-package is intended to maps to different module organization founded in existing object oriented languages.

7.1 The OFL Package

An UML-*package* is a set of model elements. In the meta-model, *Package* is a subclass of *Namespace* and *GeneralizableElement*. A *Package* contains ModelElements like *Packages*, *Classifiers*, and *Associations*. A *Package* may also contain *Constraints* and *Dependencies* between ModelElements of the *Package*.

7.1.1 Stereotypes and Tagged Values.

An OFL *package* is represented by an UML *package* (from *Model Management*) and is stereotyped as ≪OFLPackage≫. OFL package containment (nesting) is modeled by *Namespace* containment of one ≪OFLPackage≫-stereotyped UML *package* within another. For each OFL-language which is considered, stereotypes must be derived from ≪OFLPackage≫. Because current version of OFL does not provides customization for package organization, these stereotypes have to be created by the meta-programmer.

7.1.2 Constraints.

An OFLPackage may contain only OFLDescriptionTypes, OFLExternalDescriptions, OFLImportRelationships, OFLUseRelationships and other OFLPackages .

```
context OFLPackage (ModelManagament::Package)
inv: self.ownedElement->forAll(el |
                el.isStereokinded('OFLDescriptionType') or
                el.isStereokinded('OFLExternalDescription') or
                el.isStereokinded('OFLImportRelationships) or
                el.isStereokinded('OFLUseRelationships) or
                el.isStereokinded('OFLPackage'))
```

7.1.3 Elements Generation.

Profile package stereotypes must be generated manually by the meta-programmer. If necessary, it could add also tagged values to catch additional semantics of model organization.

7.1.4 Example.

A Java Package maps to an ≪OFLJavaPackage≫, which is derived from ≪OFL-Package≫. The name of the OFL Package is the name of the Java Package. A hierarchy of Java Packages maps to a hierarchy of OFL-packages.

PackageName is the fully-qualified name of the Java Package. The fully-qualified name of a top level Java Package is its name. The fully-qualified name of a Java Package contained by another Java Package is the fully-qualified name of the containing Java Package, followed by ".", followed by the name of the Java Package. The fully-qualified name of a Java Package maps to the fully-qualified name of the corresponding *OFLPackage* by replacing every occurrence of "." with "::".

Chapter 8

Modeling Example Using an OFL-Java Profile

As an example we consider the following Java code:

```
// file: Vehicle.java //
package OFLML_JavaCars;

abstract class Vehicle {
  public int type;
  public abstract void start();
}
/* Class Vehicle is the base for all vehicle hierarchy */

// file: Color.java //
package OFLML_JavaCars;

public class Color { }

// file: Car.java //
package OFLML_JavaCars;

public class Car {
   public Color color;

   public void setColor(Color c) {};
   public Color getColor() {
            return color;  };
   public void start() {};
}
```

Figure 8.1 gives an example of a model for an application which uses an OFL-ML profile for OFL-Java:

- three descriptions : *Vehicle*, *Car*, and *Color*,
- one Java concretization relationship : class *Car* is a concretization of the abstract class *Vehicle*,
- one Java aggregation relationship: class *Car* has an attribute of type *Color*.

The diagram corresponds to above Java code. The OFL-ML Java Profile elements used have been defined according to previous sections. The diagram was generated with Objecteering UML Modeler version 5.2.2 [Sof03].

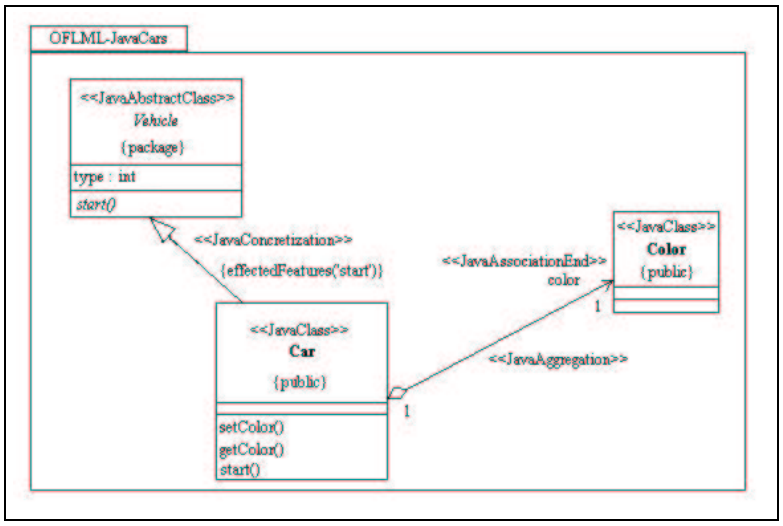

Figure 8.1: Example of using OFLML Java Profile

Chapter 9

Conclusions and Future Work

9.1 Conclusions

This report presented an approach for the generation of UML profiles for an object oriented languages described in OFL. This approach is based on a profile meta-language named OFL-ML. We present in detail the generation mechanisms of OFL-ML and its drawbacks when taking into account the semantics of some language. Then, based on this meta-language, we present an OFL-Java Profile that is generated according to OFL-ML rules.

To define a profile, OFL-ML use meta-information defined by the OFL model. Profiles elements are generated according to following OFL entities:

- OFL-DescriptionComponents
- OFL-AtomAttribute
- OFL-AtomMethod
- OFL-ImportRelationshipComponents
- OFL-UseRelationshipComponents
- OFL-Package

To fulfill the profile description, for each elements, additional taggedValues and OCL constraints are also generated.

Because each OFL-ML Profile conforms to UML 1.5 standard specification, generated profiles are guaranteed to be compatible with commercial UML modeling tools that support the profile mechanisms.

But the approach which has been proposed has some limitations. It does not consider following issues:

- other UML diagrams, in addition to the static class diagrams
- Modeling of OFL Objects
- dynamic relationships like OFL-class-to-object-relationships and OFL-object-to-object-relationships
- type multiplicity (arrays or collection classes like java.util.Vector)

9.2 Future Work

We identify two main directions for future work.

A first short-term perspective is to refine and improve the language customization. Current version of OFL provides only a light reification and no customization of semantics at

the level of routine body. Using UML definition of Action Model [OMG03, MTAL99], we intent to provide a way to represent also semantics at this level. Our proposal is to extend the generated OFL-ML profile with UML-Actions for routine body representation.

Briefly, UML actions deal with :

- a fundamental unit of computational behavior
- action semantics which are based on proved concepts from computer science
- action semantics which remove assumptions about specific computing environments in user models:
  - execution engines, programming languages, implementation details
  - do not require specification of software components, tasking structures or forms of transfer of control
  - allows people in charge of the modeling to produce executable specifications

Considering the usage of Action, all OFL parameters should be considered into the Profile constraints. As some example we can consider:

ConceptDescription parameters .

- generator - specifies if a description may create or not instances. This parameter will be involved in constraints at the level of all UML Actions that implies creation of description

  instances.
- destructor - specifies if a description instance could be destroyed or not. This parameter will be involved in constraints at the level of all UML Actions that implies destroying of objects.

ConceptRelationship parameters .

- direct_access - specifies if a relationship allows a direct access to a feature of target description. This parameter will be involved in constraints at the level of UML Read and Write Actions.
- polymorphism_implication - specifies if the relationship which is considered accepts or not the handling of polymorphism for the instances of classes which are involved in. This parameter addresses in constraints at the level of UML Read and Write Actions and Messaging Actions

The second proposed task is to generate a representation in XML or in a proprietary language representation of profile elements. We consider here specifications for profile representation provided by some major tools like Objecteering UML, Rational Rose etc.

Chapter 10

Annexes

These tables, taken from the thesis of Pierre Crescenzo, make the synthesis of all the meta-informations related respectively to *OFL-language*, *OFL-description* and *OFL-relationship*.

[2]This parameter is specific to import relationships.
[3]This parameter is specific to use relationships.

General Information related to the Model *OFL*		
Assertion	**Type**	**Sample of value**
Structural invariants	set of condition	cf. examples in the thesis

Information which is specific to concept-language		
Element for the reification	**Type**	**Sample of value**
Extension	set of component-language	{MJava, MEiffel}

Information which is specific to each component-language		
Element for the reification	**Type**	**Sample of value**
Valid component-descriptions	set of component-description	{Class, Interface}
Valid component-relationships	set of component-relationship	{Inheritance, Implementation, Aggregation}
Valid relationships	set of <component-relationship, component-description, component-description>	{<Implementation, Class, Interface>}
Extension	set of language	{Java, Eiffel, CPP}
Parameter	**Type**	**Sample of value**
Name	string	"Java-like"
Services	set of <Service_Name, boolean>	{<"Persistence", false>, <"Concurrency", true>}
Redefinition	**Type**	**Sample of value**
Description limitations	set of <component-description, parameter-name, value>	{<cd, Generator, false>}
Relationship limitations	set of <component-relationship, parameter-name, value>	{<cr1, Circularity, forbidden>, <cr2, Polymorphism_implication, none>}
Valid-Qualifiers redefinitions	set of <component-description | component-relationship, atom_name, set of string>	{<cd1, method, {"public", "private"}>, <cd2, attribute, {"static", "final"}>}
Assertion	**Type**	**Sample of value**
Local invariants	set of condition	cf. examples in the thesis

Table 10.1: Synthesis about the structure of concept-language

General Information related to the Model *OFL*		
Assertion	**Type**	**Sample of value**
Structural invariants	set of condition	cf. examples in the thesis
Action	**Type**	**Sample of value**
Actions	set of routine	*cf. the thesis*

Information which is specific to concept-description		
Element for the reification	**Type**	**Sample of value**
Extension	set of component-description	{Class, Interface}

Information which is specific to each component-description		
Element for the reification	**Type**	**Sample of value**
Language	language	Java-like
Valid source component-relationships	set of component-relationship	{Inheritance, Aggregation}
Valid target component-relationships	set of component-relationship	{Inheritance, Aggregation, Implementation}
Valid Qualifiers	set of <atom_name, set of string>	{<description, {}>, <attribute, {"static"}>}
Extension	set of description	{Class1, Class2, Interface1}
Parameter	**Type**	**Sample of value**
Name	string	"Java-class"
Context	*language* | *library*	library
Services	set of <Service_Name, boolean>	{<"Persistence", false>, <"Concurrency", true>}
Genericity	boolean	false
Generator	boolean	true
Destructor	boolean	false
Extension_creation	*automatically* | *manually*	automatically
Encapsulation	<boolean, boolean>	<true, true>
Sharing_control	set of (*description* | *instance* | *unique_instance*)	{*description*, *instance*}
Visibility	*global* | *package* | *description* | *method* | *object* | *statement* | *expression* | *list*	global
Attribute	*allowed* | *forbidden*	allowed
Method	*allowed* | *forbidden*	allowed
Overloading	<*allowed* | *forbidden*, *allowed* | *forbidden*, *allowed* | *forbidden*, *allowed* | *forbidden*>	<forbidden, forbidden, allowed, allowed>
To be continued in table 10.3 on the following page		

Table 10.2: Synthesis about the structure of concept-description (1/2)

continuation and end of the table 10.2 on the page before		
Information which is specific to each component-description		
Redefinition	**Type**	**Sample of value**
Relationship limitations	set of <component-relationship, parameter-name, value>	{<cr1, Circularity, forbidden>, <cr2, Polymorphism_implication, none>}
Valid-Qualifiers redefinitions	set of <component-relationship, atom_name, set of string>	{<cr1, relationship, {}>}
To be continued in table 10.3		
Assertion	**Type**	**Sample of value**
Local invariants	set of condition	cf. examples in the thesis

Information which is specific to each description[1] (cf. the thesis)		
Element for the reification	**Type**	**Sample of value**
Source relationships	set of relationship	{Inheritance18, Inheritance19, Aggregation3}
Target relationships	set of relationship	{Aggregation4, Aggregation5, Aggregation6}
Features	set of feature	{Prim1, Prim2}
Formal generic types	set of type	{object}
Effective types	set of type	{ListOfObjects, ListOfCircles, ListOfMen}

Table 10.3: Synthesis about the structure of concept-description (2/2)

General Information related to the Model *OFL*		
Assertion	**Type**	**Sample of value**
Structural invariants	set of condition	cf. the thesis
Action	**Type**	**Sample of value**
Actions	set of routine	*cf. the thesis*

Information which is specific to concept-relationship		
Element for the reification	**Type**	**Sample of value**
Extension	set of component-relationship	{Inheritance, Clientele}

Information which is specific to each component-relation		
Element for the reification	**Type**	**Sample of value**
Language	language	Java-like
Valid source concept-descriptions	set of concept-description	{Class}
Valid target concept-descriptions	set of concept-description	{Class, Interface}
Valid Qualifiers	set of <atom_name, set of string>	{<relationship, {"in"}>}
Extension	set of relationship	{Inheritance1, Inheritance2}
Parameter	**Type**	**Sample of value**
Name	string	"Specialization"
Kind	*import* | *use* | *type-object* | *objects*	import
Context	*language* | *library*	library
Services	set of <Service_Name, boolean>	{<"Persistence", false>, <"Concurrency", true>}
Cardinality	<integer, integer>	<1, ∞>
Repetition	<*allowed* | *forbidden*, *allowed* | *forbidden*>	<forbidden, forbidden>
Circularity	*allowed* | *forbidden*	forbidden
Symmetry	boolean	false
Opposite	*none* | concept-relationship	none
Direct_access	*mandatory* | *allowed* | *forbidden*	mandatory
Indirect_access	*mandatory* | *allowed* | *forbidden*	forbidden
Polymorphism_implication[2]	*none* | *up* | *down* | *both*	up

To be continued in table 10.5 on the next page

Table 10.4: Synthesis about the structure of concept-relationship between descriptions (1/3)

continuation of the table 10.4 on the preceding page

Information which is specific to each component-relation		
Parameter	**Type**	**Sample of value**
Polymorphism_policy[2]	<*hiding* | *overriding*, *hiding* | *overriding*>	<overriding, hiding>
Feature_variance[2]	<*covariant* | *contravariant* | *nonvariant* | *non_applicable*, *covariant* | *contravariant* | *nonvariant* | *non_applicable*, *covariant* | *contravariant* | *nonvariant* | *non_applicable*>	<covariant, nonvariant, non_applicable>
Assertion_variance[2]	<*weakened* | *strengthened* | *unchanged* | *non_applicable*, *weakened* | *strengthened* | *unchanged* | *non_applicable*, *weakened* | *strengthened* | *unchanged* | *non_applicable*>	<strengthened, weakened, strengthened>
Dependence[3]	*dependent* | *independent*	independent
Sharing_level[3]	*global* | *package* | *description* | *instance* | *unique_instance*	instance
Read_accessor[3]	*optional* | *mandatory*	optional
Write_accessor[3]	*optional* | *mandatory*	optional
Adding	*mandatory* | *allowed* | *forbidden*	allowed
Removing	*mandatory* | *allowed* | *forbidden*	forbidden
Renaming	*mandatory* | *allowed* | *forbidden*	allowed
Masking	*mandatory* | *allowed* | *forbidden*	forbidden
Showing	*mandatory* | *allowed* | *forbidden*	forbidden

To be continued in table 10.6 on the next page

Table 10.5: Synthesis about the structure of concept-relationship between descriptions (2/3)

continuation and end of the table 10.5 on the preceding page		
Information which is specific to each component-relation		
Parameter	**Type**	**Sample of value**
Redefining	<*mandatory* | *allowed* | *forbidden*, *mandatory* | *allowed* | *forbidden*, *mandatory* | *allowed* | *forbidden*, *mandatory* | *allowed* | *forbidden*>	<allowed, forbidden, allowed, forbidden>
Abstracting	*mandatory* | *allowed* | *forbidden*	forbidden
Effecting	*mandatory* | *allowed* | *forbidden*	forbidden
Adaptation_advice	<*Adding* : *advisable* | *free* | *inadvisable*, *Removing* : *advisable* | *free* | *inadvisable*, *Renaming* : *advisable* | *free* | *inadvisable*, *Redefining* : *advisable* | *free* | *inadvisable*, *Masking* : *advisable* | *free* | *inadvisable*, *Showing* : *advisable* | *free* | *inadvisable*, *Abstracting* : *advisable* | *free* | *inadvisable*, *Effecting* : *advisable* | *free* | *inadvisable*>	<*Adding* : *advisable*, *Removing* : *inadvisable*, *Renaming* : *advisable*, *Redefining* : *free*, *Masking* : *inadvisable*, *Showing* : *inadvisable*, *Abstracting* : *inadvisable*, *Effecting* : *free*>
Assertion	**Type**	**Sample of value**
Invariants	set of condition	*cf. examples in the thesis*

Information specific to each relationship[4] (cf. the thesis)		
Element for the reification	**Type**	**Sample of value**
Source descriptions	set of type	{Class23, Class28}
Target descriptions	set of type	{Interface1, Class22}
Removed features	set of feature	{a_feature}
Renamed features	set of <feature, feature_name>	{<a_feature, its_new_name>}
Redefined features	set of <feature, feature>	{<a_feature, its_new_version>}
Hidden features	set of feature	{another_feature}
Shown features	set of feature	{a_feature}
Abstracted features	set of feature	{a_feature_again}
Effected features	set of feature	{another_feature}

Table 10.6: Synthesis about the structure of concept-relationship between descriptions (3/3)

Bibliography

[AK00] Colin Atkinson and Thomas Kühne. Strict profiles: Why and how. In *UML 2000 – The Unified Modeling Language, Third International Conference, University of York, UK*, LNCS 1939, page 13. Springer Verlag, October 2000.

[CCL02] A. Capouillez, P. Crescenzo, and P. Lahire. OFL: Hyper-Genericity for Meta-Programming: an Application to Java. Technical Report I3S/RR–2002-16–FR, Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis, France, April 2002. http://www.i3s.unice.fr/I3S/FR/.

[Cre01a] P. Crescenzo. OFL : les relations et descriptions d'Eiffel et de Java. Technical Report I3S/RR–2001-06–FR, Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis, France, April 2001. http://www.i3s.unice.fr/I3S/FR/.

[Cre01b] P. Crescenzo. OFL: un Modele pour Parameter la Semantique Operationnele des Langages a Objets - Application aux Relations inter-classes. Phd. thesis, University of Nice, Sophia Antipolis, France, December 2001. http://www.crescenzo.nom.fr/.

[Des99] P. Desfray. White Paper on the Profile Mechanism, OMG document ad/99-04-07. http://www.omg.org, 1999.

[DSB99] D. F. D'Souza, A. Sane, and A. Birchenough. First Class Extensibility for UML - Packaging of Profiles, Stereotypes, Patterns. In *2nd Int. Conf. on the Unified Modeling Language: UML'99, Fort Collins, CO, USA*, page 14. Springer-Verlag,LNCS series, UML'99, October 1999.

[MTAL99] S. J. Mellor, S. R. Tockey, R. Arthaud, and P. LeBlanc. An Action Language for UML: Proposal for a Precise Execution Semantics. In *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France*, LNCS, pages 307–318. Springer, June 1999.

[OMG01] Object Management Group OMG. UML Profile for EJB Specification, Version 1.0. http://www.omg.org, May 2001.

[OMG02] Object Management Group OMG. UML Profile for CORBA Specification, Version 1.0. http://www.omg.org, April 2002.

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

[PCL03] D. Pescaru, P. Crescenzo, and P. Lahire. An Extension for OFL Model through modifiers. Technical report, Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis, France, Jully 2003.

[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. also Research Report I3S/RR-2003-16-FR, Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis, UNSA, France, http://www.i3s.unice.fr/I3S/FR/.

[Sof99] SoftTeam. UML Profiles and the J Language: Totally control your application development using UML, 1999. http://www.softeam.fr/pdf/us/uml_profiles.pdf.

[Sof03] Objecteering Software. *Objecteering 5.2.2 Manual*, 2003. http://www.objecteering.com/.