OFL: Hyper-Genericity for Meta-Programming

– An application to Java –

Adeline Capouillez
Adeline.Capouillez@unice.fr

Pierre Crescenzo
Pierre.Crescenzo@unice.fr

Philippe Lahire
Philippe.Lahire@unice.fr

Laboratoire I3S (UNSA/CNRS)
Projet OCL
Les Algorithmes
bâtiment Euclide B
2000, route des lucioles
B.P. 121
F-06903 Sophia Antipolis CEDEX
France

ABSTRACT

OFL is the acronym for *Open Flexible Languages* and the name of a meta-model for object-oriented programming languages based on classes. *OFL* intends to describe them, especially by promoting capabilities such as introspection, modification and extension. *OFL* relies on three essential concepts of these languages: the descriptions which are a generalisation of the notion of class, the relationships such as inheritance or aggregation and the languages themselves. *OFL* provides a customisation of these three concepts in order to adapt their operational semantics to the programmer's needs. This paper summarises the main characteristics of the *OFL* model, shows how to create an application using this model and describes the *Java* language according to *OFL*.

1. INTRODUCTION

One of the project manager's main goals is to bring down the cost of software production. Their cost mainly depends on two steps: programming and maintenance. During these phases, the balance must be found between fastness and high quality. Several approaches are often used to solve this problem. Examples of some of these approaches can be given, keeping well in mind that none actually solves completely this problem at present.

- In a well determined context, such as the design of graphical interfaces or Web sites, the capacity to generate source code automatically brings valuable help.
- The efforts made to obtain more readable programming languages thanks to an ameliorated syntax contribute to improving the readability of the source code written in those languages.
- Reducing the gap between the design phase and the programming phase is also aimed at to reduce the time spent in programming.
- Libraries of reusable components allow not to start from scratch for each new piece of software.
- As for the design patterns, they offer architecture models used for specific programming problems.
- Aspect programming addresses separation in terms of orthogonal services of an application's features, such as persistence or distribution of objects.

We will deal with several of those solutions in a common approach starting with the idea that relationships between classes in object-oriented languages, and especially inheritance, are low-level mechanisms which it would be interesting to specify better. This approach is materialised in the definition of the *OFL* (Open Flexible Languages) model [1].

OFL was first designed as a meta-object protocol such as that of *CLOS* [2]. However, more open and complete that *CLOS*, it has quickly become very difficult and boring first to program and then to use it. So we switched to an hyper-generic approach to solve this problem [3]. Rather than allowing to redefine behaviours thanks to algorithms, we propose a set of parameters. These algorithms, already implemented, take into account, the values of these parameters to achieve the desired behaviour. These algorithms are called actions and they define the operational semantics. We promote the idea that it is much more convenient for the meta-programmer (faster, more efficient and reliable, etc.) to set parameter values which drive well-tested actions, than to change the source code of several methods which describe altogether the semantics of the language.

2. OFL APPROACH

At first reading the *OFL* approach can be summed up as the search for a set of parameters whose value determines the operational semantics of an object language based on classes.

2.1 Hyper-Genericity

Genericity is the ability to customise the behaviour of a class in an object language just as in the *Eiffel* [4] or *C++* (`template`) [5] generic classes. Hyper-genericity is the ability to customise the behaviour of the language itself. More precisely we have chosen to customise the behaviours of three important notions of object languages based on classes:

1. relationships such as generalisation and composition [6],
2. descriptions which describes the application's objects, such as the *Java* classes and interfaces [7, 8, 9], and
3. languages themselves.

2.1.1 Parameters

We have defined a set of parameters [1] which represents the main features of the behaviours of these three important notions which are called *concept-relationship*, *concept-description* et *concept-language*. For instance, concerning the concept-relationship, the value of the `Cardinality` parameter allows to specify if it is simple or multiple. As for the concept-description we have for instance the `Generator` parameter which determines whether the concept-description can or cannot create own instances.

2.1.2 Actions

The operational semantics of each concept must adapt to the value of its parameters. This is achieved thanks to a set of actions algorithms whose execution depends on these values. For example, the assignment of an object to an attribute, the dynamic binding of the features, the sending of messages and lots of other behaviours are expressed according to parameters of concept-relationship and concept-description. *OFL* links two facets to each action: the first illustrates the static part inside an interpretor or a compiler; the second represents the dynamic aspect integrated within the runtime. The distribution of the code into these two facets depends on implementation choices of the *OFL* model.

2.2 OFL Objectives

The first one is to improve the readability of the code written in an object language based on classes. Indeed *OFL* allows to specify the relationships between the descriptions whose semantics is more precise than inheritance or aggregation. Since inheritance and aggregation are often used for very different purposes (for example: generalisation, specialisation, code reuse, ...), we aim to offer the possibility to create a relationship which is specific to each of those uses. Let us precise that in order to remain pragmatic, we do not aim to force the programmers out of their habits and to interchange the relationships there are used to with the ones we propose. When a specific relationship is used, readability of the code is simplified. Furthermore, it will be easier to generate a relevant automatic documentation

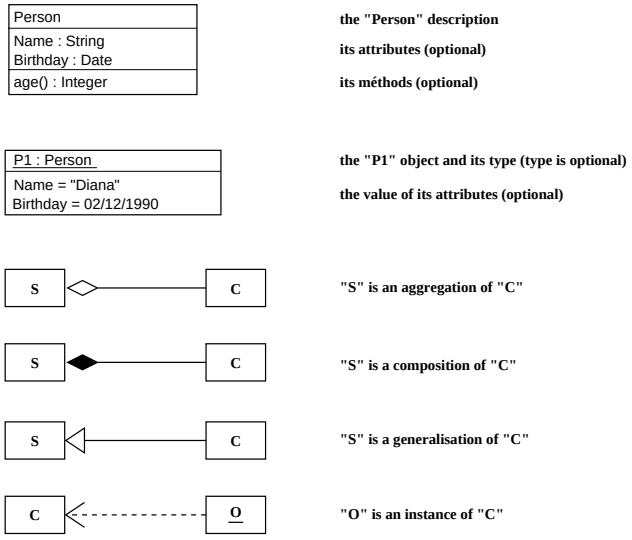

Figure 1: General graphic conventions

and the interpreter or compiler will be able to achieve more appropriate controls. As a consequence, it will be easier to maintain a program and to ensure further developments of the application.

Also, we wish to contribute to reducing the gap between the expressiveness of design methods and programming languages. Indeed, one can be particularly pleased with a very suitable *UML* representation but this is often difficult to implement straightforwardly using one's favourite programming language. *OFL* allows to define relationships with semantics closer to those of design methods and thus to program faster.

In order to obtain a realistic use of *OFL*, the programmer has to have access to libraries of concepts-relationships and concepts-descriptions in which he can select whatever he wishes to use. This method is similar to that which provides reusable software components [10].

3. OFL MODEL

First in figure 1, we determine the graphic conventions that are applied to the whole document. They are almost identical to those of *UML*.

3.1 General Architecture

Figure 2 illustrates how to use the *OFL* model to describe an application. In this figure the three necessary levels of modelisation are shown:

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

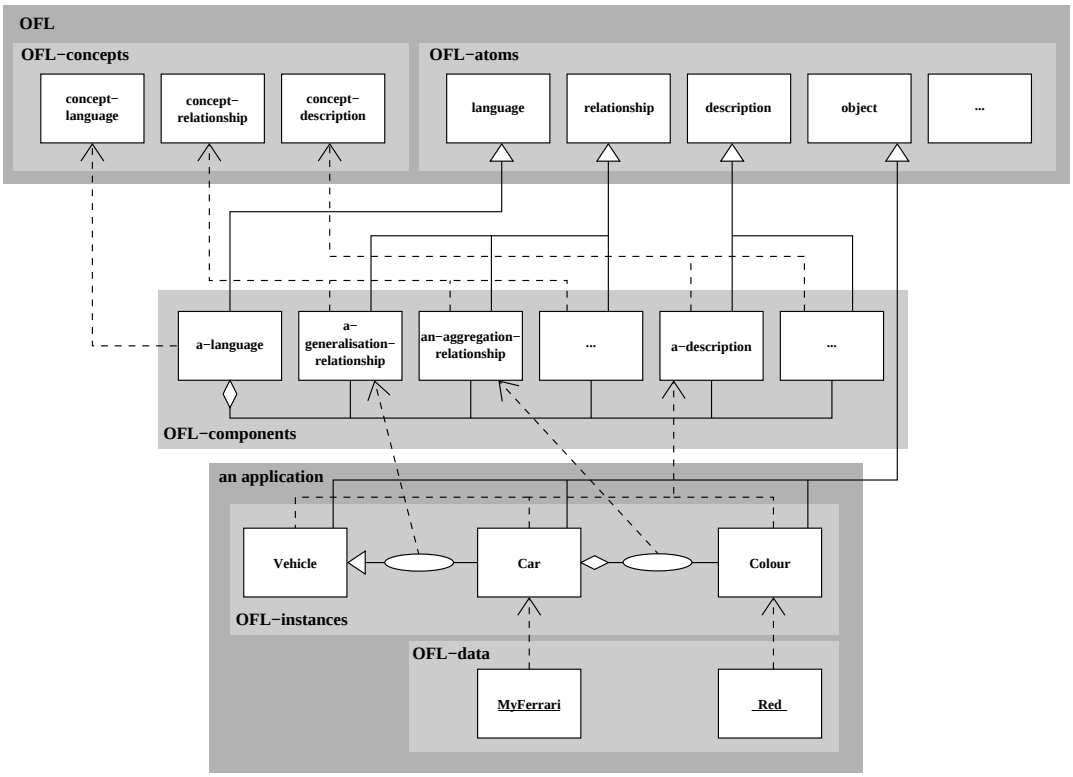

Figure 2: The *OFL* architecture

3.2 Application Level

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

3.2.1 OFL-Instances

Each description or relationship described by the programmer is modelised by an *OFL*-instance. Figure 2 gives an example of an application which includes five *OFL*-instances:

- three descriptions: `Vehicle`, `Car`, and `Colour`,
- one generalisation relationship: `Car` inherits from `Vehicle`, and
- one aggregation relationship: `Car` has an attribute of the `Colour` type.

3.2.2 OFL-Data

In the application, each description instance is modelised at runtime by an *OFL*-datum. Figure 2 shows two of them:

- `MyFerrari`, an instance of the `Car` description, and
- `Red`, an instance of the `Colour` description.

Let us point out that the *OFL*-instances which are descriptions specialise the *OFL*-atom `object`. Indeed, `object` is the reification of data of the application (*OFL*-data). So it represents the root of the specialisation tree of the *OFL*-instances which are descriptions.

3.3 Language Level

The language level describes different types of relationships and descriptions which can be used in the modelised language. The relationships are instances of `concept-relationship`, the description are instances of `concept-description`. The language itself is an instance of `concept--language`. Its main function is to put together the relationships and descriptions which are supplied to the programmer.

3.3.1 OFL-Components

The language level is solely composed of *OFL*-components. Figure 2 lists:

- some concepts-descriptions among with `a-description`,
- some concepts-relationships among with `a-generalisation-relationship` and with `an-aggregation-relationship`, and
- a concept-language `a-language`.

It is possible to represent a concept-description as a meta-class and a concept-relationship as a meta-relationship and similarly a concept-language as a meta-language.

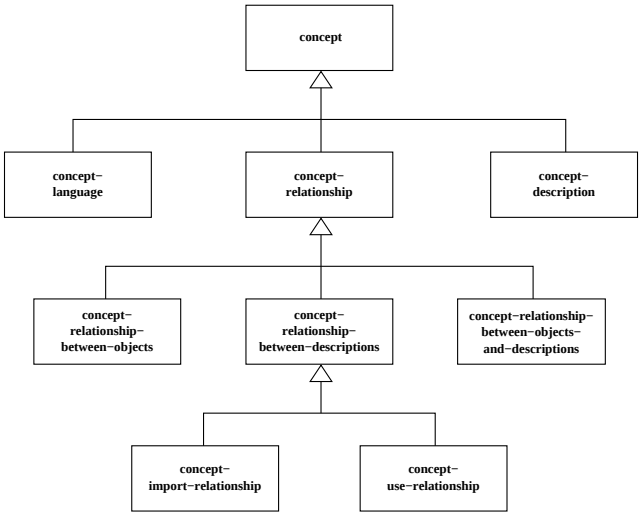

Figure 3: The *OFL*-concepts

3.4 OFL Level

The *OFL* model is a meta-model for the programming language (language level) and a meta-meta-model for the programs (application level). As was said in part 2.1, we have chosen to customise three important notions: relationships, descriptions and languages. However, a lot of other components need to be reified such as objects, methods, assertions, etc. in order to modelise a language completely. The *OFL* level includes two types of entities:

1. the *OFL*-concepts which describes the customisable part of the relationships, descriptions and languages, and
2. the *OFL*-atoms which describes the non-customisable part of these three concepts as well as all the other components.

Also assertions are described in each *OFL*-concept and *OFL*-atom in order to keep the model consistent.

3.4.1 OFL-Concepts

Figure 3 shows the whole of the classification of the *OFL*-concepts. Let us recall that only the *OFL*-concepts are customised in our model.

The meta-programmer's task is to create an *OFL*-component, i.e. an instance of an *OFL*-concept, by giving a value to each of its parameters. Thus he decides on the behaviour of each future instance of the *OFL*-component. If the operational semantics which the meta-programmer wants to bind to an *OFL*-component does not match the actions planned, then he has to modify the code of those actions. The *OFL* model is left open by this possibility which should not be used but in very specific context. Indeed, in that case, the meta-programmer's job is much heavier than just giving values to parameters.

3.4.1.1 Concepts-Relationships

We call a concept-relationship the entity representing a kind of relationship. A concept-relationship is consequently a meta-relationship. Among the relationships which are to be found in lots of object-oriented languages based on classes and object design methods, we may mention for example inheritance, aggregation, composition, generalisation, etc. However a given method or language seldom owns all of these relationships and usually uses some of them in order to simulate others. For example the generalisation in *UML* describes a generalisation as well as an inheritance, a strict sub-typing[1], ...

Around thirty parameters define the semantics of all the *OFL* model's concept-relationship. Part 4.1.1 lists the concepts-relationships representing the relationships of the *Java* language.

Figure 3 illustrates our classification of the concepts-relationships. Concerning the inter-description relationships, we distinguish between the import relationships (generalisation of the inheritance mechanism) and the use relationships (generalisation of the aggregation mechanism). As for figure 2 it illustrates one instance of an import concept-relationship (`a-generalisation-relationship`) and one example of an use concept-relationship (`an-aggregation-relationship`).

OFL also takes into account the relationship between objects and classes which are used for example to modelise the instantiation relationship existing between an object and its class. It is also possible to modelise the relationship between objects. Yet, we mainly care about inter-description relationships.

3.4.1.2 Concepts-Descriptions

A concept-description allows to define the notion of class and all that looks like a class such as the interfaces in *Java*. Therefore a concept-description is a kind of meta-class.

For instance we can notice that even if they look the same the *Eiffel*, *C++* or *Java*, classes show some notable differences. Figure 2 gives one instance of concept-description called `a-description` as an example.

Around twenty parameters are necessary to describe the behaviour of a description in the *OFL* model. Each concept-description is compatible with a set of concepts-relationships. For instance, in *Java*, the concept-description `interface` is compatible with the concept-relationship `implementation` but it is incompatible with `between-classes-inheritance`.

3.4.1.3 Concepts-Languages

The concept-language is an important and yet simple notion. It modelises a language. In particular, each language includes a set of concepts-descriptions and a set of concepts-relationships which are compatible with at least one of the concepts-descriptions. In figure 2 there is only one concept-language's instance (`a-language`) which represents the modelised language.

[1] These three relationships have different semantics even if they are similar enough to be often mistaken.

The concepts-languages are hardly customised. Their main function is to federate the concepts-relationships and concepts-descriptions which are compatible with them.

3.4.2 OFL-Atoms

They represent the reification of the non-customised entities of the model. Figure 4 illustrates a part of those *OFL*-atoms. The relationships, descriptions and languages have their own *OFL*-atoms to describe the part of their structure and their behaviour which are not customised. For instance, in figure 2, we may say that the *OFL*-component called `an-aggregation-relationship` is a specialisation of the *OFL*-atom `relationship`.

For instance in an application all the features of a description are instances of an heir of `feature`, all the expressions are instances of `expression` or of one of its heirs and all the objects are instances of `object`. Thus *OFL* gives a full reification of the entities found at the application runtime.

4. IMPLEMENTATION OF OFL

After describing the different elements composing the *OFL* model, we shall now use *OFL* to modelise the descriptions and relationships of the *Java* language. To illustrate this part we shall give a value of all hyper-generic parameters for one component-relationship and one component-description. We shall then deal with other meta-models available within the framework of object-oriented technologies. Finally we shall describe a set of tools allowing the use of our model.

4.1 Intuitive definition of Java

We are listing now the different semantics of the descriptions and of the relationships between the descriptions of the *Java* language. Each of these semantics is represented by an *OFL*-component. The reader can go to figure 5 to get the full list of the *Java* *OFL*-components.

For *Java* we have found:

- obviously, one concept-language,
- eight concepts-relationships i.e. four import ones and four use ones, and
- ten concepts-descriptions.

The reader familiar to *Java* may find the number of *OFL*-components is high. This is the result of the accuracy of our parameter system which provides a rather fine granularity. The differences in the semantics between the relationships or the descriptions are often hidden from the programmer by the use of the same keyword in different contexts.

What we say about the *Java* *OFL*-components is not the value of each of the parameters but rather a presentation of their main features. The keywords linked to each of the *OFL*-components are put in brackets.

4.1.1 Concepts-Relationships of Java

The first four concepts-relationships are import ones, the following four as use ones.

4.1.1.1 Between Classes Inheritance (`extends`)

This relationship is used *to refine the implementation of the specification of a data type.* A specification is implemented in a class; so that a class is specialised by an between classes inheritance. This is simple inheritance in which cycles are forbidden. The features of an inherited class are imported into the heir class. It is possible to replace the attributes and to redefine the methods. Polymorphism is applied in a bottom-up way, i.e. any instance of the heir may be seen as an instance of the inherited one.

4.1.1.2 Between Interfaces Inheritance (`extends`)

This relationship allows *to refine the specification of a data type.* It is applied between interfaces, the heir specialising the inherited ones. Contrary to the between classes inheritance, this relationship is multiple indeed. Polymorphism works in the same way as in the between classes inheritance.

4.1.1.3 Concretisation (`extends`)

This relationship allows *to materialise the implementation of the specification of a data type.* It is applied between an abstract (inherited) class and a non-abstract (inheriting) class. This relationship is identical to a between classes inheritance but it is compulsory to provide a body to the ancestor's abstract methods in the heir.

4.1.1.4 Implementation (`implements`)

This relationship modelise *the implementation of the specification of a data type.* A class can thus implements one or several interfaces. Therefore implementation is a multiple relationship. If the class is concrete, it must provide a body to all the methods which are specified in the interfaces. If the class is abstract, it can provide a body to some methods and keep the others abstract. Its polymorphism is identical to that of the inheritance relationships.

4.1.1.5 Aggregation

This relationship modelises *how to use the services of the descriptions.* In order to implement such a use, one just has to declare an attribute with the type of the used description[2]. Contrary of the four previous import relationships, cycles are allowed for aggregations. The attributes of the description used is directly accessible i.e. the use of accessors is not compulsory[3]. The life of the used object is independent from that of the using object and the same object can be used by several using objects.

4.1.1.6 Class Aggregation (`static`)

This relationship modelises the notion of *class attribute* well known in object-oriented languages. It is identical to aggregation but the used object is associated to the using class and not to its instances.

4.1.1.7 Composition

This relationship modelises *the strong use of the description services.* It is similar to an aggregation but the used entity

[2]The use of a method parameter, of a function result or of a local variable with the type of the used description resembles aggregation.

[3]Except if the attribute visibility is restricted, for instance declaring it as `private`.

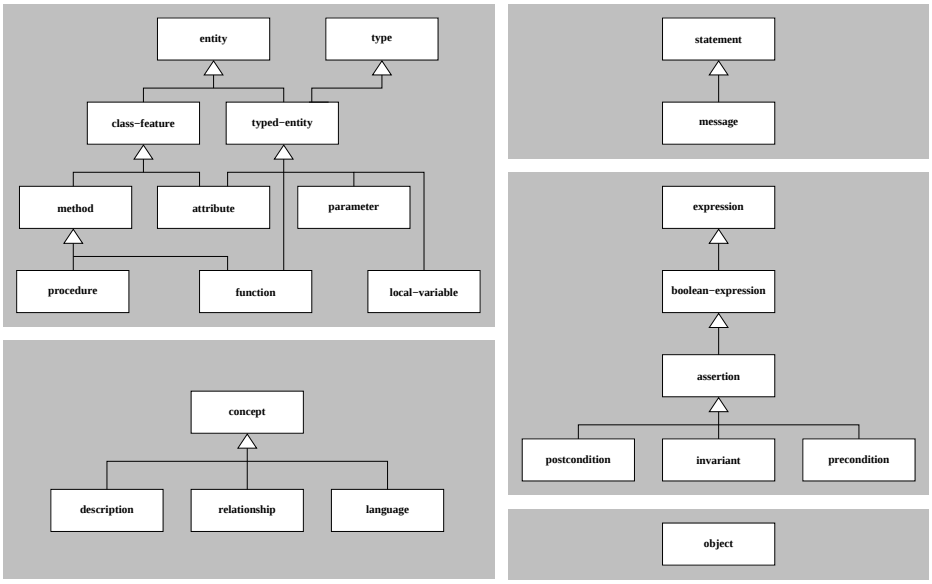

Figure 4: The *OFL*-atoms

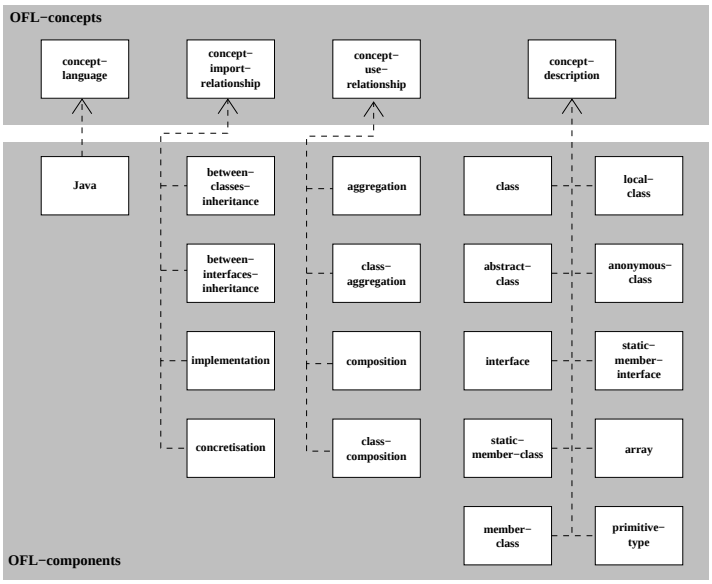

Figure 5: The *OFL*-components of *Java*

is not an object because it is of a primitive type (for example: `int`). Thus we consider that the life of the used entity depends on that of the using object.

4.1.1.8 Class Composition (`static`)

The class composition is similar to composition but it defines a *class attribute* as class aggregation does.

4.1.2 Concepts-Descriptions of *Java*

In general, we call *inner-classes* the static member classes, the member classes, the local classes, the anonymous classes, and the member static interfaces too. We have listed ten concepts-descriptions[4].

4.1.2.1 Class (`class`)

A class is the *concrete implementation of a data type*. It is a non-generic description which may contain methods[5] and attributes. It is visible within the limits of the package but this visibility can be extended or restricted by a qualifier (`public` or `private` for instance). It is able to create instances but cannot destroy them, this task is assigned to a garbage-collector. Finally it allows the overloading without taking into account the type of function result.

4.1.2.2 Abstract Class (`abstract class`)

An abstract class is *an abstract implementation of a data type*. This description as the same features as a class but it can describe abstract methods (without a body) and cannot have own instances.

4.1.2.3 Interface (`interface`)

This is the *specification of a data type*. Contrary to a class, an interface cannot define attributes. Moreover, all its methods are abstract and therefore it cannot create instances.

4.1.2.4 Static Member Class (`static class`)

A static member class is *an implementation of a data type, local to a class*. When compared to a class, its characteristic is to be defined inside a class and not at the highest level. Besides it is not accessible but through the class that defines it.

4.1.2.5 Member Class (`class`)

A member class is *an implementation of a data type, local to a class* too. But unlike the static member class, an instance of the member class is automatically associated to each instance of the class that defines it.

4.1.2.6 Local Class (`class`)

A local class is *an implementation of a data type, local to a method*. It is only visible within the method that defines it. Besides that, it is equivalent to the other class concepts-descriptions.

4.1.2.7 Anonymous Class (`class`)

An anonymous class is *an implementation of a data type, local to an expression*. It is similar to a local class but only visible within the expression that defines it. Moreover, as it has not got a name, it cannot be referred to and so it cannot be inherited. Finally because of its syntactic structure, if it implements an interface, it can only implement one.

4.1.2.8 Static Member Interface (`static interface`)

A static member interface is *a specification of a data type local to a class*. It is equivalent to a static member class with the shape of an interface.

4.1.2.9 Array

It represents the structure of data that has the same name, which is well-known by computer scientists. It is thus *an indexed and permanent-sized collection of entities of a given type*. In *Java*, array is a particular case. Each array is an instance of a virtual class[6] representing its type (for example: an array of integers is of type `int[]`).

4.1.2.10 Primitive Type

A primitive type is *the representation of a language basic type*. It allows essential elements of the application to be described: booleans, characters, bytes, shorts, integers, longs, floats, and doubles. Let us point out that each primitive type describes a value and not an object and that a class exists to represent each of them. For example, the `Integer` class allows to consider an `int` as an object.

4.1.3 Constraints between Java *OFL*-Components

The semantics of the *OFL*-components allows to answer the question "Which concepts-descriptions are valid as source or as target of each concepts-relationships?". To understand the problem better, the following definition can be useful:

- The source of a relationship is the description which declares the relationship. This description is the one that needs the service. There can be several sources, as in the *UML* association, even if it is quite a rare case.
- The target of a relationship is the description quoted by the source when declaring the relationship. The target provides the service. It is possible to find several targets, it is more frequent to find several targets than several sources (for example: multiple inheritance).

As far as import concepts-relationships are concerned, we can mention following constraints, for example.

- It is impossible for between classes inheritance to have as target and source an interface or a static member interface. Furthermore, anonymous classes cannot be targeted by such an inheritance.
- The between interfaces inheritance is applied between interfaces (whether they are static member or not).
- The concretisation must take place between an abstract class and a non-abstract class.
- Finally, the implementation is a relationship which always targets one or several interfaces (whichever they are) and a class (whichever it is).

[4] We doesn't consider abstract inner-classes in this paper.

[5] As well as constructors and initialisers.

[6] This class does not exist but everything goes as if it did.

The same type of constraints is found for use concepts-relationships. Here are a few examples of it.

- The aggregation may have as a source all kinds of classes and only them. As for the targets, there are also two kinds of interface.
- The class aggregation has the same constraints as the aggregation.
- The composition resembles the aggregation but the only possible target is a primitive type.
- We should apply the same constraints for class composition and composition.

4.2 OFL definition of Java

The semantics of the *Java*'s components had been described above. This section aims to show how they are defined in *OFL*.

In order to create new *OFL*-components, the meta-programmer needs to valuate the parameters of the *OFL*-concepts associated to them. We provide in the following the parameter values attached to two of the Java *OFL*-components : one component-relationship (the implementation) and one component-description (the abstract class). While reading the values of *OFL*-component parameters the reader may refer to the corresponding paragraph in section 4.1.

4.2.1 *The OFL-component* `Implementation`

People may distinguish `components-relationships` of a language using the parameter `Name`. For this component its value is `Implementation`. The kind of relationships is recorded within the parameter `Kind` which is set to `import` for this component. *OFL* allows to mention (parameter `Context`) if a component is specific to a language or if it belongs to a library of components[7]. We are describing a relation for *Java*: `Context` is set to `language`. `Implementation` is a multiple relationship and *Java* does not limit the number of target-descriptions so the parameter `Cardinality` is set to $1-\infty$[8].

The parameter `Repetition` indicates if a direct repetition in the sources and/or targets of the relationship is allowed. According to the parameter `Cardinality`, we only consider one source for the `Implementation` relationship of *Java*, so the `Repetition` for source is not applicable. Moreover it does not make sense to implement several identical interfaces and it is forbidden in Java. Thus, this parameter is set to `forbidden` for targets.

The parameter `Circularity` records if it is possible to get cycles thanks to this relationship (that means to be able to import directly or not the source of the relationship itself). In *Java*, cycles are not allowed for `Implementation` and `Circularity` is set to `forbidden`. Other possible value is `allowed`.

To set the parameter `Symmetry` to `true` would mean that if a description of type `Class` or `Abstract Class` named `A` implements another description of type `Interface` named `B` then the `B` implements `A`. It is meaningless so that `Symmetry` is set to `false`.

The parameter `Opposite` provides (if any) the name of the opposite relationship; otherwise it is set to `none`. For a symmetry relationship, the opposite is the relationship itself. The implementation has no opposite in *Java* and this parameter is set to `none`.

The parameter `Direct_access` specifies if the relationship provides a direct access to the features of the target-description from the source-description. Because the targets of a `Implementation` relationship are only `Interfaces`, all the features of the target-description are `abstract`, and to access to the target-description features has no meaning so that `Direct_access` should be set to `forbidden`[9].

The parameter `Indirect_access` means that any access to a feature of the target-description from the source-description should be performed trough the target-description itself (i.e. the target-class must be specified). For the same reasons as for `Direct_access`, this parameter is set to `forbidden`. Possible values are `mandatory`, `allowed` and `forbidden`

The parameter `Polymorphism_direction` and the parameter `Polymorphism` are specific to import relationships. The first indicates if the relationship handles polymorphism between source and targets and when it is useful indicates also the direction (possible values are `up`, `down`, `both` and `none`). The second records how the relationship handles name conflict when the first one allows some polymorphism. Two possibilities are proposed: a feature (we distinguish attribute and methods) of the source-description which has the same name in the target-description may hide or override the latter. According to the relationship being defined, all instances of the source-description are also instances of the target-description and `Polymorphism_direction` is set to `up`. `Polymorphism` is set to `overriding` for both attributes (only constant attributes are allowed) and methods.

The parameter `Feature_variance` is specific to import relationships. It gives the kind of variance associated to the parameters of methods, to the result of functions and to the attributes, when a feature is redefined. Several values are available: `nonvariant` (new type must be the same), `covariant` (new type must be a descendant), `contravariant` (new type must be an ancestor) and `non_applicable`. For this relationship `Feature_variance` is set to `nonvariant` for method parameters, result of functions and attributes.[10]

Another specific parameter of import relationships is `Assertion_variance`. It gives the type of the variance for the assertions when there is a redefinition. There are three kinds of assertion : invariants, preconditions and postcon-

[7] According to the value of this parameter it is possible to include or not additional information or assertions which improve *OFL*-component's consistency.

[8] For classical object-oriented languages one relationship involves only one source and one or several target(s): one if the relationship is simple and several if it is a multiple one.

[9] The reader should be aware of the case of constant attributes that may be defined within an interface. *OFL* may handle this case not trough parameter values (it is something to specific) but with a generic mechanism based on `before/after` routines included in the *OFL*'s actions.

[10] Remember that *Java* disposes of overloading capacity.

ditions. Several values are available: `strengthened` (new assertion must be stronger), `weakened` (new assertion must be lighter), `unchanged` (assertion must not be modified) and `non_applicable`. For this relationship `Assertion_variance` is set to `non_applicable` for all types of assertions because such feature is not present in Java.

The parameter `Adding` indicates if the relationship allows to add features in the source-description. In Java it is allowed for `Implementation`, so that `Adding` is set to `allowed`. Other possible values are `forbidden` and `mandatory`.

The parameter `Removing` indicates if it is allowed to remove features from the target-description trough this relationship. In Java it is not allowed, so that `Removing` is set to `forbidden`. Possible values are the same as those of `Adding`.

The parameter `Renaming` records if it is possible to rename features in the source-description. In Java it is not allowed wath'ever is the relationship. `Renaming` is set to `forbidden`. Possible values are the same as those of `Adding`.

The parameter `Redefining` indicates whether it is possible or not to redefine assertions, signature, body and modifier(s) (visibility, protection, constancy, ...) of features. For each category, possible values are the same as for `Adding`. According to `Implementation`, the redefinition is `non_applicable` for assertions, it is `forbidden` for signature and modifier(s) of features. It is `allowed` (but not `mandatory`) for a method body (imported features are all abstract and may be associated to a body in the source-description).

The parameter `Masking` is dealing with the capability for a relationship to mask features coming from the target-description. Possible values are the same as those of `Adding`. In *Java* the `Implementation` relationship does not allow to mask any feature of target-description. `Masking` is set to `forbidden`. The parameter `Showing` deals exactly with the contrary: the capability to show in the source-description features which had been previously masked. In the `Implementation` relationship of *Java*, `Showing` is also set to `forbidden`.

The parameter `Abstracting` records whether a relationship may specify to remove the body of a feature of the target-description in the source-description. Such capability is `forbidden` for this relationship (all methods of an interface are already abstract methods). Possible values are the same as those of `Adding`.

`Effecting` is exactly the opposite of `Abstracting`. It indicates that a source-description may provide a body to abstract methods. For this relationship it is `allowed` (but not `mandatory`): source-description may be of type `Class` or `Abstract Class`. Possible values are the same as those of `Adding`.

4.2.2 *The OFL-component* `Abstract Class`

In previous section we described a component-relationship; the next one is dedicated to the description of a component-description. Like components-relationships, components-descriptions are associated to a set of parameters which describes their operational semantics.

A few parameters such as `Name` or `Context` which participate to the description of a component-relationship are also present in a component-description, with the same meaning. This section deals with the definition of abstract classes so that `Name` is set to `abstract class`. Moreover, we define an `abstract class` in the framework of *Java*, so that the parameter `Context` is set to `language` (cf. section 4.2.1).

The parameter `Genericity` records if a description is generic or not. There is no generic description in *Java*, so that `Genericity` is set to `false`.

The parameter `Generator` indicates if a component-description is able to create instances. If yes, it should be possible to define constructors and initialisers. In *Java*, for abstract classes as well as "normal" classes, this parameter is set to `true`. On the contrary, the parameter `Destructor` records if it is possible to free an instance. In *Java*, this is rarely used but possible, so this parameter is set to `true`.

The extension of a description is the set of its own instances and of instances of its heirs (more precisely, this set of instances depends on the parameters dedicated to polymorphism of the relationships where are this description is source or target). The parameter `Extension_creation` records if the extension of this kind of description is handled automatically or manually. In *Java* we can set this parameter to `automatically` (by default, at the description creation, its extension is empty) even if the extension is maintained only virtually.

The parameter `Encapsulation` indicates if features are encapsulated into descriptions; it is possible to distinguish between attributes and methods. In *Java* everything is encapsulated, so that the parameter is set to `<true, true>`. Let us mention that the ability to modelise non encapsulated method is useful for the specification of language such as *CLOS*.

`Sharing_control` is the parameter which records if the instances of a description may be shared and eventually how. An instance may be shared by all instances of one description (this represent the idea of class variable and the value of parameter is `description`), it may be shared by only some of the instances of one description (this is an instance variable and the value of parameter is `instance`). Finally, an instance may be attached to only one instance of one description (value `unique_instance`). A *Java* abstract class does not put constraints about the sharing of its instances (in fact the instances of its descendants) so `Sharing_control` is set to {`description, instance, unique_instance`}.

The parameter `Visibility` indicates entities from which the description is visible. Several values are possible for this parameter from a value which restricts the visibility to a single expression to one which open the description to the entire network. In *Java*, by default, the visibility is limited to the package which contains the description so the chosen value is `package`.

`Attribute` is the parameter which specifies if the description allows the definition of attributes (`allowed`) or not (`forbidden`). For an `Abstract Class` this parameter is `al-`

`lowed` (for constant attribute only). In the same way, there is a very similar parameter for methods, it is called `Method`. In an `Abstract Class` it is also possible to define methods, so that this parameter is set to `allowed`.

The parameter `Overloading` indicates if the component-description allows or prohibits (`allowed`, `forbidden`) the overloading of features. In fact it is constituted by four values which specify this overloading for the attributes, for the results of functions, for the number and for the type of method parameters. For a *Java* abstract class the parameter is set to `<forbidden, forbidden, allowed, allowed>`.

4.2.3 Some more about OFL-components

Perhaps the reader noticed some aspects of the Java language that were not covered by the parameters presented above. Let us take some examples:

- In section 4.2.1 the parameter `repetition` handles direct repetition of target descriptions, that is to say it allows or not to get several time the same description in the list of target. But what about the handling of indirect repetition (repetition of the same description several times in the same hierarchy but at different level)?
- In section 4.2.2 we deal with visibility of a description (parameter `visibility`) but nothing about the handling of the keyword `public` that may be attached to a *Java* class. On the other hand the keyword `abstract` is handled by the parameter `Generator`, why such inequity?
- The handling of the features' visibility within the hierarchy of classes is not even mentioned. The situation is the same for the handling of constructors and initialisers according to values of the parameter `Generator`. One may note that it is also necessary to ensure that constructors has same name as the description, that initialisers are anonymous and that the destructor is called "`finalize`".

According to those questions or remarks we need to make the three following comments and to propose an answer: the assertions and the qualifiers.

Firstly, the choice to define or not a new parameter for a component-relationship or a component-description is mainly based on the following question: "Is it an enough general and relevant aspect that could be integrated in more than one language and that may be found in existing literature?". To answer this question looks like very difficult and in some way quite arbitrary but let us give some examples in order to show the underlying philosophy. The handling of some language keywords is very specific to each language. Each language has its own rules for handling the visibility of routines within the hierarchy of descriptions (private, public, protected, export, ...). On the other hand, to say that a description is abstract or not, justifies from our point of view, to associate it to one or several parameters. However, the specificity of each language according to this aspect may not be fully handled by parameters (for instance when a language enforces an abstract description to contain only abstract methods).

Secondly, some semantics into a language may not fit within only one component. The handling of indirect repetition or the handling of feature visibility relies on several kinds of relationship (on several *OFL*-components), so that it is necessary to get a mechanism which is orthogonal to the kinds of relationship.

Thirdly, the *OFL* model is based essentially on two customisable concepts: relationships and descriptions. This intentional limitation means that the definition of the behaviour of entities such as methods or attributes is located within components-descriptions and components-relationships.

In order to take into account what we mentioned above, we integrated assertions within *OFL*-components: they may address any reified information including parameters value. They correspond to controls that the meta-programmer may need to set on the hierarchy of components. We add also to *OFL*-components the ability to define assertions associated to a keyword (we call them *qualifiers*). For example we may define an assertion which corresponds to the semantic of a private method in *Java* or another one which deals with the semantic of public class (also in *Java*). All controls specified by a language either explicitly through a keyword or implicitly which are not handled by existing parameters, are associated to one or several assertions; the tools mentioned in section 4.4, especially *OFL*-ML are in charge to implement those assertions.

4.3 Integration in the Existing Meta-Models

OFL is a meta-model which describes object-oriented languages based on classes and customises the operational semantics of their descriptions and relationships. The state of the art, in the field of meta-model shows quite a diversity. These meta-models are usually able to describe one another. For us, the most significant one is *MOF* (Meta Object Facility) [11]. We do not aim to compete against *MOF* but to offer a less general model closer to the programmer. *MOF* describes a *class* concept, an *association* concept and a *package* concept.

A *MOF* class allows to define attributes, the type of which can be simple or described by a class, and to specify operations. Let us point out that *OFL* and *MOF* have the same approach concerning the method bodies that have to be described according to an independent language (*binding*). *OFL* and *MOF* both draw from the *UML* and *IDL* [12] notation and syntax.

A *MOF* association allows to define any relationship that occurs between a number of *MOF* source classes and a number of *MOF* target classes. The semantics of the relationship described by such an association is implemented thanks to the attributes and the operations of the *MOF* classes.

The *MOF* packages allow to encompass the *MOF* classes and associations.

OFL may be described according to *MOF* and supply the latter with an additional layer on top of it allowing to cus-

tomise the operational semantics of the *MOF* classes and associations.

Besides, we have shown in figure 2 that *OFL* can be described with the *UML* formalism, in particular thanks to the possibility of tagging the new concepts (which was unnecessary in this figure).

Finally, *OFL* can also be described thanks to *XML* [13] and *XML-Schemas*.

4.4 OFL Tools

OFL can be used as a set of three tools which share the information contained in a database whose schema is described in *ODL* [14].

The first tool called *OFL-Meta* is for the meta-programmer. It allows to graphically create, modify or delete *OFL*-components i.e. the instances of the *OFL*-concepts. In other words, it allows to describe the operational semantics of a language which will be used when designing an application. The *OFL*-components which it handles can be stored by using various standard formalism such *XML* or *MOF*.

The second tool is called *OFL-ML* in reference to *UML*. It is intended for the application designer. It is also a graphical tool which allows to create, modify or delete *OFL*-instances, i.e. the instances of the *OFL*-components described in *OFL-Meta*. The programming task is included by a binding: the selected language is *Java*, that is body of all the methods is written in *Java*.

The third tool is called *OFL-Parser*. It is a translator, interpreter or compiler. Its task is to translate the *OFL*-instances and the body of methods (using the static facets of the actions) into a target language, *Java* in our case. The last step consist in executing the generated application and thus to use the methods and create the *OFL*-data, i.e. the instances of the *OFL*-instances (using the dynamic facets of the actions).

5. CONCLUSION AND FURTHER WORKS

The *OFL* meta-model, the features of which we have quickly summed up in this document, describes object-oriented languages based on classes. Its characteristic is to customise the operational semantics of the descriptions and the relationships of those languages. It is also possible to add other descriptions and relationships if their use is compatible with those that already exist. *OFL* also provides several assets which may allow a higher maintainability of the applications.

We have started implementing each of the three tools described in section 4.4; a prototype of the first two ones should be released in the next few months. The third tool will certainly take longer.

6. REFERENCES

[1] A. Capouillez, R. Chignoli, P. Crescenzo, and P. Lahire, "Hyper-généricité pour les langages à objets : le modèle OFL," in *Conférence LMO 2001 (Langages et Modèles à objets)*, Hermes Science Publications, janvier 2001.

[2] G. Kiczales, J. Des Rivières, and D. G. Bobrow, *The Art of the MetaObject Protocol*. MIT-Press, 1991.

[3] P. Desfray, *Object Engineering, the Fourth Dimension*. Addison-Wesley Publishing Co., 1994.

[4] B. Meyer, *Object-Oriented Software Construction*. Professional Technical Reference, Prentice Hall, 2nd ed., 1997.

[5] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley Publishing Co., 3rd ed., 1997.

[6] Object Management Group, *Unified Modeling Language Specification*, 1st ed., March 2000. Version 1.3.

[7] K. Arnold and J. Gosling, *The Java Programming Language*. The Java Series ... from the Source, Sun Microsystems, 3rd ed., 2000.

[8] D. Flanagan, *Java in a Nutshell: a Desktop Quick Reference*. O'Reilly, 3rd ed., December 1999.

[9] J. Gosling, B. Joy, G. Steele, and B. G., *The Java Language Specification*. The Sun Microsystems Press Java Series, Sun Microsystems, June 2000.

[10] M. Bouzeghoub, G. Gardarin, and P. Valduriez, *Les objets*. Eyrolles, 1997.

[11] Object Management Group, *Meta Object Facility (MOF) Specification*, March 2000. Version 1.3.

[12] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, February 2001. Version 2.4.2.

[13] *Extensible Markup Language (XML)*, 2nd ed., October 2000. Version 1.0, W3C Recommendation.

[14] R. Cattell, D. K. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade, *Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Inc., 1997.