OFL: An open object model based on class and link semantics customization

Robert Chignoli, Pierre Crescenzo & Philippe Lahire

I3S Laboratory, University of Nice Sophia Antipolis, Les Algorithmes,
bâtiment Euclide, 2000 Route des Lucioles, Sophia Antipolis 06140 Biot - France
{lahire, chignoli, crescenz}@i3s.unice.fr

Abstract. The need to "open" object langages in order to make them more able to fit new situations appears as a major issue.More and more, objects will be persistent and mobile and will have to get adapted to the evolution of their original pattern. In the context of software engineering class languages, these ideas mean that the basic mechanisms (classes, inheritance and client-supplier links) need more flexibility. According to that, we propose a general model which allows some customization of class language semantics. This model provides a reflective and structured framework for the encapsulation of first level object semantics into three main types of meta-programmable components ("classes", "links" and "languages") which role and potential interest are presented in this paper.

1 Introduction

The starting point of this project is the integration of persistency in class-languages [1] and by the way, the necessity to consider a set of situations (version management, interoperability, etc.) which can be required for writing and maintaining applications.

The variety of the requirements in these fields has led to the definition of three main targets : *a)* to allow the customization of the concept of type in order to - for instance - arm or disarm the insertion of an object into the persistent collection of its type; *b)* to allow the customization of links between classes in order to - for instance - restrict a general mechanism of inheritance to a specific use; *c)* to allow the building of new links between classes as - for instance - a link of «is a version of».

Through these targets, we are in some way addressing the well known idea of «open language» that we propose to apply to the field of industrial object-oriented languages in order to make them more flexible [1].

In order to reach the targets mentioned above, we have defined the OFL model which formalizes, in Eiffel3, our approach. This model is currently undergoing implementation as the OFL/VM platform (Virtual Machine) [2].

[1] Hence the name of the project : Object Flexible Language. This may be also read as "Open Eiffel" (Eiffel is pronounced FL in French) because most statements proposed by OFL are derived from Eiffel.

OFL/VM is designed as an open application which provides two programming levels: *a)* a meta-level which provides a framework for the concept definition (classes and links between classes), and the handling of relationships between those definitions; *b)* a base-level which allows the programmer to develop his application using the concepts of classes and links between classes defined at the meta-level. From another point of view, OFL/VM may be shown as a library of components which may describe the semantics of classes and links implemented in languages such as Eiffel, Java, Smalltalk or C++.

The first version of OFL/VM relies on a reflective architecture. This technology (which current bounds regarding efficiency are well known) provides, according to our purpose, the flexible framework which is perfect for first experiments.

This paper lays out successively: *2)* the outlines of OFL model, through an example which describes the computation of a remote-call statement; *3)* the set of customization facilities defined within our approach. Finally, *4)* we position our approach according to related work and *5)* we conclude with current state of the project and perspectives.

2 The OFL model through an example

2.1 Description of the example

Figure 1 shows a toy application built with three components: *EXAMPLE*, *SQUARE* and *RECTANGLE* (where class *EXAMPLE* acts as the main program). Components which are shown in this figure are written using the Eiffel3 language syntax and semantics [3]. Semantic links[2] are *client-supplier* relationship and *(Eiffel3) inheritance* whereas the concept of *class* is the *Eiffel3* one's. In our exemple there are several *client-supplier* links (*is-client-of*):

- one from class *EXAMPLE* to class *RECTANGLE* (attribute *rectangle*),
- one from class *EXAMPLE* to class *SQUARE* (attribute *square*).

There are also inheritance links (*inherit-from*):

- one from class *SQUARE* to class *RECTANGLE*,
- three others, implicit within *Eiffel3* syntax, which have class *ANY* as **target** and respectively, *EXAMPLE*, *RECTANGLE* and *SQUARE* as **source** classes.

[2] In the following, the class which declares a link is the **source class**, whereas the one which ends the link is the **target class**.

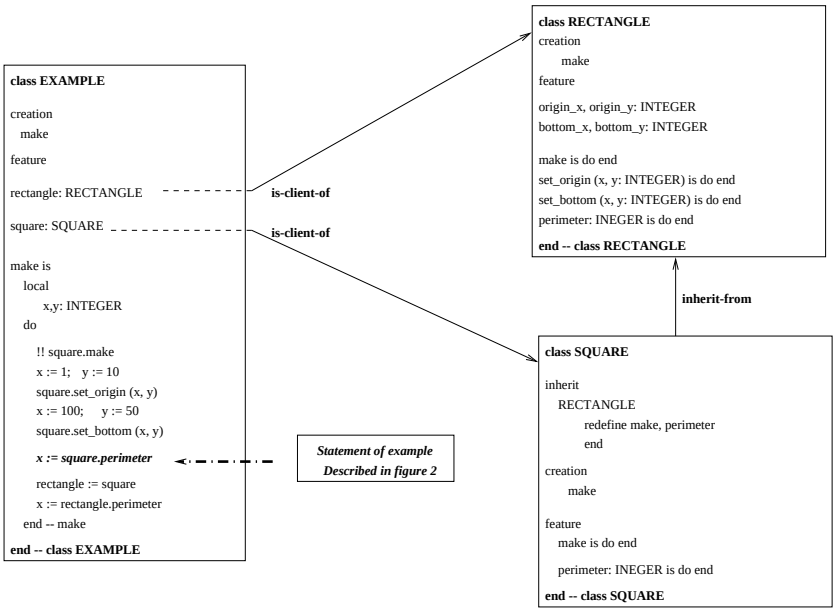

Fig. 1. A basic application (*Eiffel3* syntax)

2.2 Overview of OFL model entities

Each class of the application (*EXAMPLE, SQUARE, RECTANGLE*) is represented by an instance of the meta-class *M_CLASS*. It corresponds on the first hand to the reification of a class (attribute, procedure, function, inheritance link, etc), and on the other hand, to the semantics of the *class* concept within the language (in this example, the semantics of an *Eiffel3* class). In *Eiffel3,* every class inherits from class *ANY*. Within our model we propose the *ROOT_M_CLASS*, a predefined class which implements the basic services that any object should be equipped with; it will provide all necessary features for an easy implementation of the class *ANY* as it is implemented in Eiffel3 (for making the figure 2 more readable we do not draw it).

The *is-client-of* **use link** and the (*Eiffel3*) inheritance **importation link**, are represented by instances of the meta-classes *USE_LINK_M_CLASS* and *IMPORTATION_LINK_M_CLASS*. Those meta-classes are specialization of class *M_CLASS* so that, their instances are described by predefined classes: *USE_LINK* and *IMPORTATION_LINK* (see section 3.1).

Instances of class *EXAMPLE* and *SQUARE* are described by class *M_OBJECT*. Each object contains the fields that match the attributes described within its class (e.g.: *att_square* for class *EXAMPLE*). Each object[3] may become persis-

[3] Within our model, routines and classes are also objects.

tent and can access to its type through an instantiation link (*is-an-instance-of*). Class instances do not contain any customizable semantic operators.

An instantiation link is a link of type **object to class**, and is represented by an instance of meta-class *OBJECT_TO_CLASS_LINK_M_CLASS*, (see section 3.1). These links are customizable, they provide a support for the modeling of complex treatment allowing the **delegation** of an **object request** to a class. For any language, it is possible to define an instantiation link with its own semantics, so that it allows the integration of a customized handling of persistency, and additional controls or actions for the object contents conversion or adaptation.

A **job request** corresponds to a statement which does not have a universal semantics but a semantics which depends on the language which defines it. For instance, a conditional schema (**if** *condition* **then** *code-then* **else** *code-else*) relies on a well known semantics, whereas, for instance, a dot expression (*attribute.procedure(...)* or *attribute.function(...)*), depends on the semantics of the language. Section 2.3 gives more details about a **job request** with the *Eiffel3* semantics (*att_square.perimeter*).

2.3 Modeling of a statement execution

Each meta-class (M_CLASS, USE_LINK_M_CLASS, IMPORTATION_LINK-_M_CLASS), provides a set of customizable semantical operators (object attachment, routine invocation, etc), which are activated according to needs, all along the application execution (tables shown in section 3.3 provide a full list of these operators). Our computational model is based on two principles: **the delegation** of semantical actions and controls (**SAC**) between meta-class instances, and **the submission** of **job request** between class instances. The concept of **delegation** implies that the whole semantics is not centralized but distributed within the different meta-classes and that coordinating actions are mandatory. The concept of **submission** means that each object which receives a job request is up to choose whether it handles it *now*, *never*, *for some objects only*, *later*, *etc.*

The statement which is shown in this example is *att_square.perimeter*. Its definition is located in routine "*make*" of class *EXAMPLE* which is the root class of the toys application (see figure 1). In *Eiffel3*, when the program is launched, one instance of class *EXAMPLE* is created; this is on this instance that the statement should be applied[4]; it becomes the first current object[5]. Figure 2 describes main steps (numbered from 0 to 11), of **job request** processing[6].

The instance of class *EXAMPLE* receives the **submission** of this **job request** (*steps **0 and 1***) which handling is **delegated** to the class *EXAMPLE*

[4] The reader may note that all prior operations are not defined within the figure.
[5] The object which is currently running the routine which holds the statement.
[6] Operator names mentioned in this section are described in section 3.3.

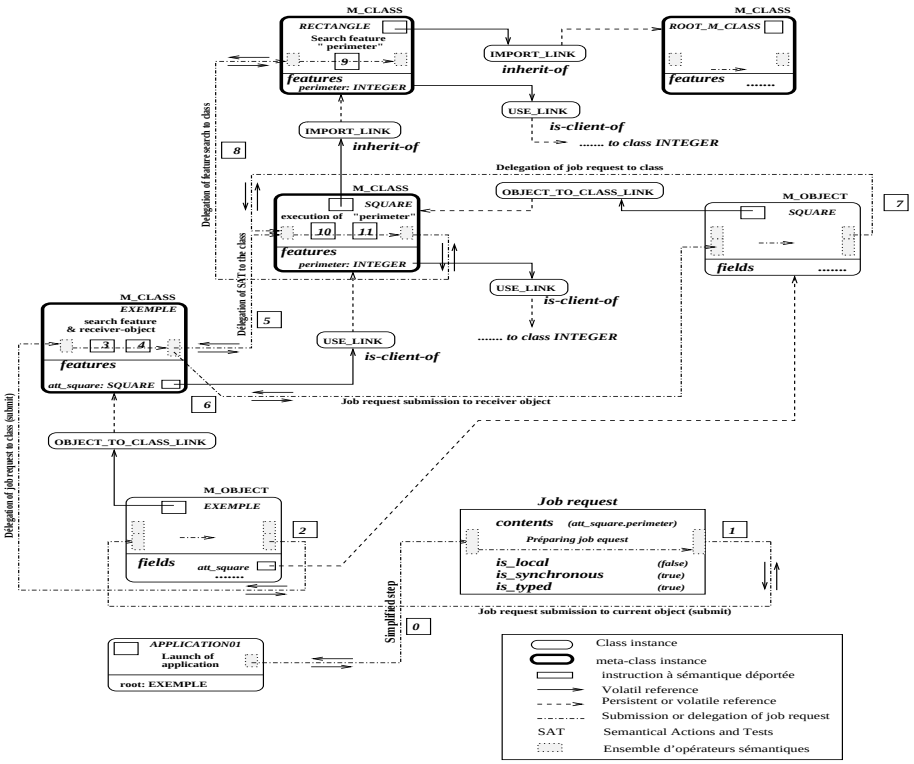

Fig. 2. An example of function-call computation for an *Eiffel3* application

(*step 2*) through the *is-an-instantiation-of* link. Since this link is customizable, it is possible to use the operator *submit* of *OBJECT_ TO_ CLASS_ LINK* object in order to activate SAC before (or after) the **delegation** becomes effective.

When the **job request delegation** reachs it, class *EXAMPLE* handles the **submission** of this **job request**. The description of the semantics associated to the **submission** (operator ***submit*** of *M_ CLASS*) depends on the language attached to the job request. Mostly, the handling of the submission of a job request is made up following actions:

- To find the kind of job request (*also **step 2***) : a **job request** may be internal to the object (object attachment, creation or deletion of object, routine invocation or access to fields corresponding to attributes defined within the class[7]), or may need to be encapsulated within a message in order to be **submitted** to another object; this is the case for all access to attributes or routine call when they are performed thru a *use link*; this kind of job request is commonly called *dot expression*.
- Case of a dot expression (it is the case in 1), it is necessary to:
  - search, using name and context of the statement, the attribute or the function which allows to access after evaluation to the object-receiver

*(step **3**)*. These actions are performed thru the operator ***item*** of *M_CLASS* which returns the appropriate feature. This customization allows to implement all types of feature overloading, all possible feature search mechanisms, according to the importation links starting from the class; these links are represented by objects of type *IMPORTATION_LINK*; this kind of link allows to describe adaptation clauses (as those that may be found in most object-oriented languages[8]), and to delegate feature searches to the classes reached by the link.

- find object-receiver(*step **4***), thru attribute access or thru function computation (the attribute *att_square* points out receiver-object), and to perform controls corresponding to the language semantics (e.g. feature exportation in Eiffel3). All these actions are encapsulated within the operator ***execute*** of *M_CLASS* which can possibly be implemented, as it is proposed in section 3.4. This operator (and those it is using) allows to plan and perform all predefined control and actions (loading of persistent object, updating of class extension, etc), according to the *use link* associated to the typed feature and to the semantics of *class* concept. The reader may note that it is a link between classes and that it is represented by an object of type *USE_LINK*.
- encapsulate the **submission** of **job request** within a message and forward it to receiver-object *(step **6**),* using actions defined within the operator *send* (in this example the object pointed out by *att_square*); the receiver object becomes the new *current object*. A message should always be sent according to the constraints associated to the *use link*, that is to say using the semantics attached to this link (for an Eiffel3 application it is a *client-supplier* link). Typical actions that should be performed before forwarding the message deal with the loading of persistent objects, information recording, controls; for an Eiffel3 application it will be necessary to check[9]that the feature *att_square* is visible *(exported)* to class *EXAMPLE (step **5**)*.

– Case of an internal **job request**. The job request which is received by the object attached to *att_square* is *perimeter*; this object (the new *current object*) **delegates** the job request to its class, in the same way as the object of type *EXAMPLE* (see above, ***step 7***). The class finds out that it is an internal job request. It is necessary to perform following actions:

- To search the feature that should be accessed (same as feature-search described in the case of a dot expression). In this application, the search is extended to the class which is imported, that is to say, target of an importation link (*steps **8, 9***);
- To compute the routine (procedure or function depending on the application) or to access the attribute (*step **11***). Some **SAC** may be performed before (see explanations given in the case of a dot expression for the operator ***execute***, ***step 10***)[10].

We mentioned above several operators of *M_ CLASS*, especially those that deal with the customization of class semantics ***submit***, ***send***, ***execute***, *etc.* Of course those operators have twins in order to be able to return the result of a function or the value of an attribute when it is required (***execute_ with_ result***, ***submit_ with_ result***), or a response when the message needs one (***send_ with_ response***).

3 A model for customizing language semantics

In the previous section, we presented the main aspects of the OFL object protocol at run-time, and some of the possible customization that may be done with our model. This section intends to provide a more comprehensive list of operators and to propose a classification of customization facilities.

3.1 Definition of the concepts of *language*, *class* and *link*

Figure 3 gives a synthetic overview of the OFL design according to the semantic customization. This design relies on the concepts of *language*, *class* and *link* :

- **The reification of class semantics**. It allows to customize, among other things, routine invocation, message sending, object attachment and object creation. The choice and the contents of operators implementing the customization of class semantics rely in particular upon following remarks:
  - The need to coordinate the use of the semantics located within links (whatever is the class concept in which the description of the link semantics is associated, this description should not be dependent on the other types of link; this is the role of the class to handle these dependency problems).
  - The semantics of an action initiated by a class may be influenced by the semantics of the links which plays a part in the action.
- **The reification of link semantics** depends on the kind of link :
  - The importation links (object structuring)
  - The use links (structuring of exchanges between instances of classes)
  - The "object to class" links (structuring of the exchanges between an object and a class)

  We do not provide any support for the modeling of "**object to object**" links. According to our approach, all exchanges between instances of classes are defined by a link between object classes (*importation links* and *use links*).

[7] It includes attributes and routines defined within a class reachable thru an importation link.

[8] We allow the definition of complex adaptation clauses : redefine, rename, undefine and forget a feature (see section 3.1).

[9] These controls may be static or dynamic (see section 5).

[10] For readability purpose, links with class INTEGER are not fully described but it looks like the link between *EXEMPLE* and *SQUARE*.

– **The reification of language semantics** groups together: *a)* the semantics of classes (a given language may possibly define several concepts of *class*, for instance a "usual" concept of *class* and then a concept of *version* or *view* [4]), *b)* the semantics of the links defined within the language, and *c)* several other informations, controls or rules regarding the combination of *link types* and concepts of *class*.

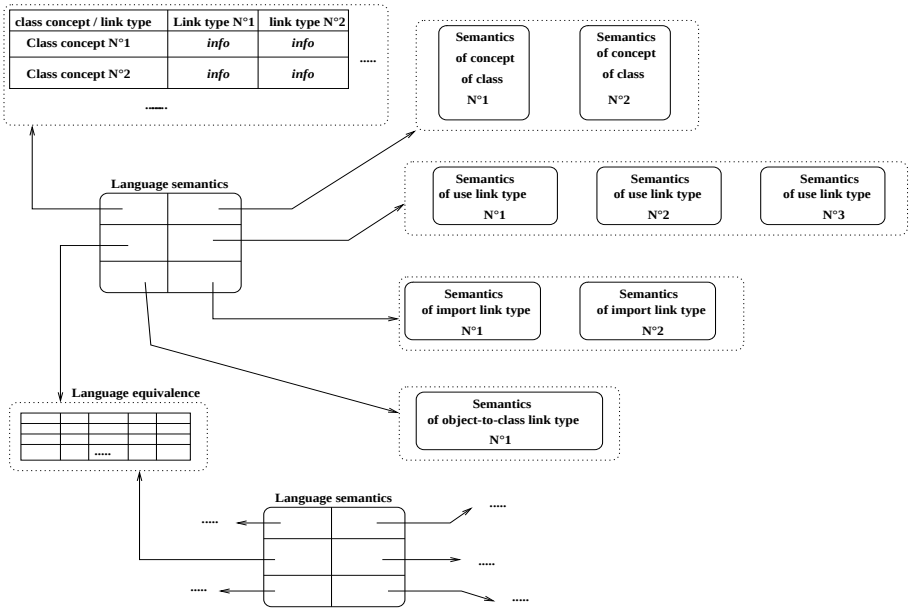

Fig. 3. Reification of language semantics

3.2 Operators for customizing the concept of *language*

This section goes back over the operators for the language semantics customization; these operators allow to access to information which are useful for the handling of interoperability between objects built with different languages, and for the implementation of normalized programming rules (for instance within a company). We present below the set of the main operators, with possible interesting examples of use.

– Recording of the meta-programming level which is used (see section 3.4 for the definition of meta-programming level);
– Recording of the number of *importation links* (of same type) that it is possible to declare in the same **source** class, both for the same or different **target** classes. This is especially interesting for the control of the *class importation* semantics:

  - importation of only one class (to be related with *single inheritance*);
  - importation of n classes (to be related with *multiple inheritance*). Being able to set the number n may help for the implementation of a programming methodology within a group of programmers (for instance, no way to inherit from more than 2 classes);
  - importation several times of the same type (to be related to *repeated inheritance*), still with the ability to set the upper bound.
- Recording of several rules dealing with the compatibility between the concepts of *class* and *link*; these rules take into account the role of the concept of *class* (**source** or **target** of the *link*). For instance, if there are two concepts of *class* (*abstract class* and *implementation class*), meta-programmers may allow the use of an *is-a-subtype-of* **importation link**, only within an *abstract class* and the use of an *is-an-implementation-of* **importation link** only within an *implementation class* (see [3] and [5] for the description of inheritance link usages).
- Access to a shared ressource which allows to describe, when it is possible, the equivalence between *link types* or concepts of *class* defined in two different languages.

Information associated to the operators described above are particularly useful when people deal with persistent objects created by an application written with another language; they provide a way to control the distance between language semantics. The meta-programmer may use those information when he describes the operators for which we propose a classification in the next section.

3.3 Classification of operators for customizing classes and links

Operators dealing with the customization of the semantics of classes and links are split in several categories :

- Feature looking,
- Semantical controls,
- Computation of job request (*routine invocation, attribute access, message sending, etc*),
- Handling of class instances and class extensions (*creation, deletion and object attachment, etc*), and
- Generic operators (*copy, clone, equality, etc*).

The reader will note that the set of features which handle the modification of class contents (feature adding, etc) is not customizable; there are only involved in the reification of *class* concept. For instance, consequences of attribute adding or removal should be handled by the definition of new ad hoc link types such as an *is-a-version-of* link (*importation link*), and an *"object to class"* link; the *structural reification* may not be customizable whereas the *computational reification* does.

The classification we propose is divided in two parts of unequal size:

- The customization of classes and links between classes which is described in the next five sections. In each section, we give a list of operators which customize class semantics; for each class operator we mention the operators which customize link semantics[11].
- The customization of objects thru links of type "*object to class*"; this provides a way to customize the concept of object (instance of class) which is not customizable by itself. The description of those operators may be found in section 3.3.

Each operator takes a **job request** as input; its contents (at least the statement which corresponds to the job) is updated as long as the different operators are activated. For instance, contents updating deal with the feature which is returned by feature-searching operators, the effective parameters after evaluation and, possible intermediate result(s) in the case of dot expression processing.

Feature looking. Before any feature computation, it is necessary to find the feature associated to a given name according to the statement context (type, number of effective parameters, etc); the operators which customize those actions are given in table 2.

Class operators	Description of class operator	Related link operators
match	returns true if the feature attached to parameter satisfies overloading rules. (definition of overloading rule).	is_match
local_lookup	returns feature of class (if it exists) which satisfies overloading rules. (definition of search sequence)	
lookup	returns feature of class (if it exists) which satisfies overloading rule; class may be found in linked classes by importation link (definition of search sequence is extended to imported classes	lookup lookdown
item	returns feature of class (if it exists), which is accessible directly or thru imported classes. (coordination of operators *match, local_lookup and lookup)*	
all_features	Returns the set of all class features	

Table 2. Customization of feature search

Let us take an example with the operator **item** of class *M_CLASS*. It allows the coordination of the feature search, thru the processing of the **local_lookup** et **lookup** operators following the chronological order defined by the meta-programmer. These operators handle respectively the feature looking within the class only, and the feature looking in the **target** classes of *importation links*.

Class operators	Description of class operator	Related link operators
is_import_links_valid	definition of validity rules for the use of importation links	
is_use_links_valid	definition of validity rules for the use of use links	
is_arguments_compatible	definition of compatibility between effective and formal arguments	is_arguments_control_valid before_arguments_control after_arguments_control
is_type_conformance	definition of type conformance rules	
is_same_type	definition of rules for type equality	
is_generics_valid	definition of compatibility between effective and formal generic parameters	
is_valid_request	definition of job request validity (visibility, ...)	is_feature_valid, is_send_valid

Table 4. Customization of semantical controls

Semantical controls. This section deals with operators that implement possible dynamic control. If those controls are handled statically, the operators are empty (see table 4).

For instance, let us consider the operator **is_import_links_valid**. This is typically the operator which should contain the controls dealing with importation links, using the information located at the language level (see section 3.2). Such controls are particularly useful in a universe of persistant objects attached to different language semantics.

Computation of routine or access to attribute. Operators of this section deal with the customization of routine computation and attribute access for *current object* or for distant object thru a *use link* (see table 6).

Class operators	Description of class operators	Related link operators
arguments_evaluation	definition of argument evaluation rules	before_argument_evaluation, after_argument_evaluation is_argument_evaluation_valid
effective_to_formal	attachment of effective arguments to formal ones	before_effective_to_formal, after_effective_to_formal is_effective_to_formal_valid
detach_effective	detachment of effective arguments	is_detach_effective_valid before_detach_effective, after_detach_effective
before_request	Source code in front of the first statement of one routine body	before_feature
after_request	Source code after the last statement of one routine body	after_feature
execute	coordination of routine computation or of attribute access	
execute_with_result	idem as *execute* but with a result returned after computation	
send	coordination of message sending for dot expression	before_send, after_send
send_with_response	idem as *send* but with a response to message (asynchronous or not)	before_send, after_send
submit	coordination of job request submission	
submit_with_response	idem as *submit* but with a response or result after request processing	

Table 6. Customization of routine computation and attribute access

We consider the operator **effective_to_formal**[12]. This operator customizes the attachment of effective parameters to formal ones when the routine is being processed. The result of this operator computation depends a lot on the

[11] We present in this paper a first version of the set of link operators, it will be refined as long as we will extend the link-type library.

[12] In section 3.4, we describe the contents of operator ***execute*** of M_CLASS.

semantics of links attached to routine parameters. According to the type of links, it is necessary to perform controls and actions dealing with the attachement of effective parameters to formal ones, and describing the point of view of links (for instance a copy of objects attached to effective parameters in the case of a composition link); these actions are described in link operators such as **before_effective_to_formal**, **after_effective_to_formal** or else **is_effective_to_formal_valid**. The operator **effective_to_formal** coordinates and controls the computation of those operators.

Handling of class instances and class extensions Operators located in this section allow the customization of instance creation or deletion, object attachment, and class extension update (see table 8).

Class operators	Description of class operators	Related link operators
create	instance creation, coordination between generic and non generic instances	is_creation_valid, before_creation after_creation
instance_creation	allocation of non generic instance	is_creation_valid, before_creation after_creation
generic_instance_creation	allocation of generic instance	is_creation_valid, before_creation after_creation
instance_deletion	deletion of instances, activate desallocation of generic and non generic instances	is_deletion_valid before_deletion after_deletion
instance_desallocation	desallocation of generic and non generic instances	is_deletion_valid before_deletion after_deletion
assign	Rules and action related to object attachment	is_assign_valid before_asign after_assign
make_instension	creation of class extension	
update_extension	Update of class extension according to source and target classes	update_extension update_extension_of_to_c update_extension_of_from_c

Table 8. Customization of the management of class instances and class extensions

In table 8, we mention the creation of generic class instances; the reification of concept of *class* described by the meta-class *M_CLASS* provides the opportunity to create generic classes. So that, we should take into account the specificity of generic instance creation (control of parameter instantiation, etc). Furthermore, the model provides also the ability to handle class extension, that is to say the set of its instances (see operators **make_extension**, **update_extension**); this is very useful, especially in the framework of the integration of universal operators ($\forall$, $\exists$) within a query language or assertion mechanism [1] [6].

Object generic operators. This section deals with basic operators that every objects should be equipped with. The customization of those operators according to classes and links should make easier the modeling of links such as an *is-a-version-of* link (see table 9).

Class operators	Description of class operators	Related link operators
equal_reference	definition of equality of object reference	before_equal_reference after_equal_reference is_equal_reference_valid
equal_contents	definition of equality of object contents, not propagated to referenced objects	before_equal_contents after_equal_contents is_equal_contents_valid
deep_equal_contents	definition of deep equality of object contents (equality is tested even for referenced objects)	before_deep_equal_contents after_deep_equal_contents is_deep_equal_contents_valid
copy_instance	definition of object copy, not propagated to referenced objects	before_copy, after_copy is_copy_valid
deep_copy_instance	definition of deep object copy (copy is propagated to referenced objects)	before_deep_copy after_deep_copy is_deep_copy_valid
clone_instance	definition of object cloning, not propagated to referenced objects	before_clone, after_clone is_clone_valid
deep_clone_instance	definition of deep object cloning (cloning is propagated to referenced objects)	before_deep_clone after_deep_clone is_deep_clone_valid
adapt_instance	definition of instance adaptation according to its (new or modified) type	before_adapt_instance after_adapt_instance is_adapt_instance_valid
conforms_to_instance	definition of compatibility rules between instances	

Table 9. Customization of generic operators

Customizing operators such as **clone_instance** allows, for instance, to implement a use link which takes into account the update of all object clones when the contents of one of them is modified, in order to ensure that objects will remain identical during their whole life cycle.

Operators of "object to class" links. Operators which describe links of type "*object to class*" are the only one that may modify the semantics of class instances; this choice can be justified by the fact that the OFL model is dedicated to the description of software engineering languages. The customization of the "*object to class*" link types deals with following operators:

- object generic operators (***clone***, ***equal_contents***, *etc*), see section 3.3,

– operators which handle object attachment and **job request submission** (see operators ***assign***, ***submit*** *and* ***submit_with_response*** of sections 3.3 and 3.3).

3.4 Strategies of meta-programming through an example

It is not acceptable to ask the meta-programmer the same effort (or the same level of knowledge of the meta-programming tools) whatever are his needs; this is why we provide a default strategy for meta-programming the large number of operators which all correspond to elementary needs of customization. But, of course, the meta-programmer can choose to develop its own sequence of operator invocations and its own operator bodies, or to mix specific and default programming. This section shows an example with the *execute* class operator.

We present now in this section the predefined sequence of operator invocations for the operator ***execute***. If a meta-programmer sets in the language semantics the indicator called ***is_default_handling***, then when operator ***execute*** is invoked, the predefined sequence is applied. In the following example, all operators of the model are indicated by a '<— (*)'. One may note that even if the meta-programmer chooses the predefined sequence, he can customize a part of the semantics of the operator ***execute*** for the *class* concept defined within the language that he addresses (***execute_semantics***). This possible customization should be used for handling the coordination of operators which define actions depending on the type of link semantics.

```
execute (m: INTRA_MESSAGE) is
    -- Computation of internal job request 'm'
require
  message_not_void: m /= void
  is_not_void: m.action /= void and m.parameters /= void
  exists: attached.has (m)
local
  f_tmp: like item
  b_tmp: BOOLEAN
do
  if is_default_handling then
     -- search of feature
     f_tmp := item (m) <---- (*)
     if not is_error (m) then
        --  setting of selected feature in job request
        m.set_candidate_feature (f_tmp)
        -- evaluation of parameters
        parameter_evaluation (m) <---- (*)
        if not is_error (m) then
           -- control of parameters
           b_tmp := is_parameters_compatible (m) <---- (*)
           if not is_error (m) then
              if b_tmp then
                 -- attachment of effective parameters
                 effective_to_formal (m) <---- (*)
              else -- semantical error (type mistmach)
              end -- if
              -- code executed before the computation of any routine
              before_request (m) <---- (*)
              -- customized part of operator 'execute'
              execute_semantics (attached, m)
              -- code executed after the computation of any routine
              after_request (m) <---- (*)
```

```
                -- detachment of effective parameters
                detach_effective (m) <---- (*)
            else -- Error handling
            end -- if
          else -- Error handling
          end -- if
        end -- if
    else -- fully free semantics
        execute_semantics (attached, m)
    end -- if
ensure
end -- execute
```

Comments: If the meta-programmer chooses to use the predefined semantics, the operator *execute* is equivalent to the processing of the following semantic operations (all these operations may contain specific code defined by the meta-programmer and take into account the semantics of associated links):

1. Feature looking
2. Evaluation of Parameters
3. Check of effective parameter compatibility
4. Attachment of effective parameters to formal ones
5. Computation of something equivalent to a *before* routine
6. Computation of the specific code, defined by the meta-programmer
7. Computation of something equivalent to an *after* routine
8. Detachment of effective parameters

otherwise, the meta-programmer build his own sequence of semantical operator invocations.

Reading the code which corresponds to the operator **execute**, one should be strongly convinced that it is very important to get a tools for semantics aided design (SAD). Such a tool would allow to:

- remove any dependency between the meta-programming task and the language used for implementation (Eiffel 3),
- make code optimization according to meta-programming information,
- decrease the complexity of meta-programming tasks.

4 Comparison with other systems

In this section, we propose to briefly position our model according to related works. We study characteristics of reflective system model built on top of Lisp, Smalltalk, C++ and Java. Main discussed aspects deal with available concepts with their ability to be customized, and granularity of reflectivity.

Systems built on top of Lisp: CLOS (Common Lisp Object System) [7] proposes a smart, compact and powerful solution based on a Meta-Object Protocol. Each semantical concept (invocation of function, instance creation, ...) is described by a function which is itself a meta-object. Among the main meta-objects of the protocol one may find, for instance, *metaobject*, *method*, *generic_function*, *method_ combination*, *slot_ definition*, *specializer*, or *class*; they are themselves descendants and their contents may be redefined by meta-programming.

According to this, our approach is different from the one of CLOS thru two aspects. On the first hand, with OFL, the language semantics is located in a smaller number of concepts (about five for languages such as Eiffel3); On the other hand, we strongly distinguish (in the description of *meta*-code) the classes, that mainly play a role of coordinator and the different link types implemented in those classes.

Systems built on top of Smalltalk : In *NeoClasstalk*, F. Rivard [8] is interested in the evolution of object behaviour in the framework of reflective languages which are dynamically typed; its motivation is to provide features for decreasing the distance that exists between the application design and its implementation in a language of classes, when people face objects which structure and behaviour change during their life cycle. Among the services provided by *NeoClasstalk*, one may note in particular: the control and the lazy update of instance variables, the ability for an object, to fit the semantics of another class through the modification of its instantiation link, the extension of the built-in *lookup* of Smalltalk in order to take into account multiple inheritance, the reification of method invocation, and its control thru the receiver-object class. A first use of *NeoClasstalk* is the *OpenCorba* environment [9]. Its objective is to extend the original model of the OMG [10] with meta-protocol libraries which specialize the distributed programming mechanisms in order to introduce new semantics (concurrency, replication, security, ...).

As *NeoClasstalk*, our approach allows to modelize several link semantics. The links *object to class* allow to meta-program the mutation of type in the same way as *NeoClasstalk*. The *importation links* allow to modelize all kinds of inheritance, including one kind which may delete features. Finally, the *use links* may handle problems related to replication and security.

Systems on top of C++ : As it is mentionned within its name, *OpenC++* [11][12] has been designed in order to provide new capabilities to C++ but avoiding tedious tasks for the programmer, such as the programming of an analyser or the modeling of a type system. The main uses of OpenC++ are the development of syntactical and/or semantical extension of C++. This sytem focuses on efficiency and handles *meta*-information at compile time. Main services of *OpenC++* are object assignment, handling of different kinds of expressions, function invocation, creation and deletion of instances, access and update of variables.

In order to implement its customization, the meta-programmer have to build a meta-class which inherits from the meta-class *class* and redefines the routine

bodies that are selected according to the C++ extension that he intends to implement (each routine corresponds to a customizable concept); the new contents of these routines correspond to the new piece of generated code related to the semantical action which is considered.

DART project [13] is based on *openC++*. It proposes an extension of the *C++* syntax and relies on the *OpenC++*MOP; it aims to provide facilities for the development of distributed applications thru mechanisms for adapting their behaviour according to the system and network environment.

Iguana [14] allows the meta-programmer to select the concepts that should be reified independently from each other. The modification of semantics attached by default is implemented by inheriting from the class which describes the reification and specializes the methods that may be found within it. The set of meta-declarations is encapsulated within the concept of *protocol* and it is allowed to build a new protocol from existing ones. The protocols that are used in a class are selected at declaration time. The main reifed and customizable concepts are method invocation, creation and deletion of objects, message receiving, feature search, activation/desactivation of semantical controls.

Unlike OFL, OpenC++ and Iguana are based upon a same existing language for which an *open* programming environment is proposed. Our approach is a little bit different according to the fact that we propose a model which is not based on any particular language. An other important difference between OFL and these two models is the central position of links; this corresponds to the strong determination to isolate the meta-code which handles the relationships (the links) between entities, from the meta-code which handles class semantics. Furthermore, from a general point of view, OFL is closed from *OpenC++* by its customizing model expressiveness. Then OFL is close to Iguana by more technical aspects: the customizable meta-informations, and most of all, the encapsulation of semantics (concept of *language* in OFL vs concept of *protocol* in Iguana).

Systems on top of Java : Standard *Java* environments provide few capabilities for reflection. Thru the class loader, it is possible, for instance, to convert an array of bytes into class, but the reverse translation is not possible. It is also not possible to change the semantics of a class. The API (*java.lang.reflect*) provides services for reflection but reification is limited to structures. Several systems intend to improve the reflection capabilities of Java.

Systems such as *Reflexive Java* [15], *Dalang* [16], *metaXa* [17], *LEAD++* [18], extend the reflection in Java. They all share the same orientation to Internet, the same concerns (transaction, security, concurrency, distribution, mobility, persistence), and the idea dealing with the separation between the meta-code and the application base code.

Reflexive Java, *metaXa* and *dalang* provide the customization of method invocation; the last system offers *before* and *after* routines in front of and after routine invocation, and makes possible to customize routine computation on one object thru several meta-objects which are associated to it by linking.

MetaXa allows the customization of variable access, object creation, class loading; it is possible to attach several meta-objects to a same base-level objects

and to apply the semantics of one or several of them according to the event and to the use of an operator which provides a way to address the next meta-object. The information which are useful for the computation (parameter passing mode, protocols for object replication, persistence handling, message sending, etc.), are forwarded to meta-object at instantiation time or when operators are activated.

Another system, *guarana* [19][20], allows, when several meta-objects are attached to a same base-level object of application, to coordinate, put altogether, or filter, thru a special meta-object, the action of other meta-objects associated to the base-level object. This approach fits to the design of meta-object libraries such as *MOLDS* [21] which address distributed systems.

The *LEAD++* model relies on the concept of *adaptable procedures* (the idea deals with the adaptation of procedure computation according to run-time environment). These procedures are associated to methods; the method selection is made according to an *adaptation strategy*, to run-time environment, and to meta-information attached to it.

The models that are briefly described above aim to open one language and to structurize this openess. Each of them provides characteristics close to those of OFL : to meta-program the routines *before* et *after* (*dalang*), to attach several meta-objects to a same base-level object of application (*MetaXa*), to coordinate, to put altogether, to filter (*guarana*), to provide adaptation strategies (*LEAD++*) are services proposed by OFL and which may be found in previous sections.

5 Perspectives

Mobile objects We have integrated in OFL the model of persistence that has been proposed in [1] and that has been fully experimented in the FLOO project [22]. This model relies on a transparent handling of persistent objects thru a *podvm* (*Persistent Object Descriptor in Volatil Memory*); at this time we work on the extension of the concept of *podvm* to the one of *object finder* in order to be able to take into account the distribution and the mobility of objects upon the network.

Interoperability We have integrated facilities for interoperability thru the late binding of the language semantics to an object and thru the handling of meta-information which allows an object to check the validity of the language binding when a job request is submitted to it. It seems important to go deeper and to propose a set of criteria allowing to measure the distance of semantics between two *languages*.

Reflectivity and efficiency Efficiency is a key point of the implementation of reflective languages or virtual machine. It mainly depends on the expressiveness of the model for customization and on the system ability to support dynamic behaviour changes. The relative antagonism between these two legitim needs leads to mention two essential questions: should we prefer static semantical controls (performed at compiling time) or rather dynamic ones (at run time)? Is it

better to propose a full reification of concepts or on the contrary to reify only some of them ? Those questions do not address universal answers. Whatever is the technics, openess has a cost. In the domain of open reflective environments, there are, as everywhere else, two strategies for handling the problem: either to set the openess capabilities, or to allow meta-programmer to adjust the computation model openess (and the efficiency) according to its needs. In OFL we have chosen to give a greater place to openess; this will lead us to select the second strategy and to provide additionnal adjustment according to the ratio *openess of an application* (or *of a language*) / *efficiency*.

6 Conclusion

Reflection is a promising support for the development of *open* object platforms. Actually, introduction of this openess in the world of object environment for software engineering should allow the orthogonal integration of both :

- first-level needs, regarding persistency for local or distributed objects,
- and also (an above all ?) meta-needs related to the evolution of the applications which generate those persistent objects (that may change of type, that is to say change of semantics).

In this paper we make a contribution to this problematics with the proposal of a reflective model in which :

- first-level needs are satisfied by the definition of a computational model based on the integration of a service of persistency, the **submission** of a **job request** between objects and the **delegation** between one object and its class.
- the fulfillment of meta-needs relies on the ability to meta-program three kinds of components : classes, links and languages . *The classes* contain the structural semantics and the behavioral semantics ("local lookup", etc) and delegate to *links* (relationships between classes), treatment that they do not know how to handle. The *languages* provide an additional methodological support useful to regulate combination between *link*s and *class* concepts .

  In fact, the OFL model may be seen as "Yet Another Son of MOP" particularly dedicated to the software engineering problematics. As all other models, its brothers, OFL proposes its own structuring of object-oriented concepts. From our point of view, the originality of OFL relies on the central position given to the concept of link; this concept which, under different names or semantics, is found at any stage of the application design: from specification to the implementation in a given language.

At that time, we are finalizing a first version of the virtual machine for OFL (OFL/VM) with a client-supplier link and a single-inheritance link. The implementation has been made with the Eiffel 3 language (version 3.3.7, Interactive Software Engineering). To give an idea of the project, 300 specific classes have been developped (around 40.000 of line code).

References

1. P. Lahire. Conception et réalisation d'un modèle de persistance pour le langage Eiffel. Thèse de doctorat de l'Université de Nice - Sophia Antipolis, Mai 1992.
2. R. Chignoli, P. Crescenzo & P. Lahire. OFL/VM: Une machine virtuelle à objets et ouverte, pour la gestion d'objets persistants et mobiles. Rapport de recherche, pp 65, Laboratoire I3S, Université de Nice - Sophia Antipolis, Déc. 1998.
3. B. Meyer. Object-Oriented software construction, 2^{nd}edition. Prentice Hall 1997.
4. S. Marcaillou. Intégration de la notion de points de vue dans la modélisation par objet. Thèse de doctorat de l'Université Paul Sabatier, Fév. 1995.
5. R. Chignoli, P. Crescenzo et Philippe Lahire. Liens entre classes dans les langages à objets. Rapport de recherche 97-22, pp. 26, Laboratoire I3S, Université de Nice - Sophia Antipolis, Juil. 1997.
6. P. Collet. Un modèle fondé sur les assertions pour le génie logiciel et les bases de données : application au langage OQUAL, une extension d'Eiffel. Thèse de doctorat de l'Université de Nice - Sophia Antipolis, Déc. 1997.
7. R.G. Gabriel, D.G. Bobrow, J.L. White. Clos in context - The shape of the design space. In Object Oriented Programming - The Clos perspective. MIT Press 1993.
8. F. Rivard. Evolution du comportement des objets dans les langages à classes réflexifs. Thèse de doctorat de l'Université de Nantes, Juin 1997.
9. T. Ledoux. OpenCorba: un bus réflexif adaptable, Actes de LMO'99, Villefranche sur mer, Janvier 1999.
10. Object Management Group. The Common Object Request Broker: architecture and specification, revision 2.0, Jul. 95.
11. S. Chiba. A Metaobject protocol for C++, Proceedings of the OOPSLA'95 ACM Conference, Vol. 30 of SIGPLAN Notice, ACM, 285-299.
12. S. Chiba. OpenC++ 2.5 Reference Manual. Institute of Information Science and Electronics, University of Tsukuba, 1998.
13. P-G. Raverdy, H. Le Van Gong, R. Lea. DART: A reflective middleware for adaptative applications
14. B. Gowing, V. Cahill. Meta-Object Protocols for C++: The Iguana Approach. Proceedings of reflexion'96, Ed. Kiczales, San Francisco, California, Apr. 1996
15. Z. Wu.Reflexive Java and a reflexive component-based transaction architecture.[23]
16. I. Welch and R. Stroud. Dalang - A Reflective Java Extension. in [23]
17. M. Golm, J. Kleinöder. metaXa and the future of reflection. in [23]
18. N. Amano & T. Watanabe. LEAD++: an object-oriented freflective language for dynamically adaptable software. in [23]
19. A. Oliva, L.E. Buzato. Composition of méta-objects in Guaranà. in [23]
20. A. Oliva, I.C. Garcia, L.E. Buzato. The reflective architecture of Guaranà. Technical report IC-98-14, instituo des computaçaos, Universidade Estadual de Campinas, Apr. 1998.
21. A. Oliva, L.E. Buzato. An overview of MOLDS: a meta-object library for distributed systems. Technical report IC-98-15, instituo des computaçaos, Universidade Estadual de Campinas, Apr. 1998.
22. R. Chignoli, J. Farré, P. Lahire, R. Rousseau. FLOO: un environnement pour la programmation persistante en Eiffel. Technique et sciences informatique, vol. 15, numéro 6 (Juin 1996), pages 735-763. Numéro spécial "systèmes objets: tendances actuelles et évolution" A. Napoli & J.F. Perrot (eds.)
23. J-C. Fabre & S. Chiba. Proceedings of Workshop on Reflective Programming in C++ and Java UTCCP Report 98-4, Center for Computational Physics, University of Tsukuba, Japan ISSN 1344-3135, Oct. 1998.