Using both Specialisation and Generalisation in a Programming Language: Why and How?

Pierre Crescenzo[1] and Philippe Lahire[1]

Laboratoire I3S (UNSA/CNRS), Projet OCL
2000, route des lucioles, Les Algorithmes, Btiment Euclide B
BP 121 F-06903 Sophia-Antipolis CEDEX, France
{Pierre.Crescenzo, Philippe.Lahire}@unice.fr
http://www.i3s.unice.fr/{ crescenz/, lahire/}

Abstract. The reuse of libraries of classes by client applications is an interesting issue quite difficult to achieve, especially when modification of the class tree is needed but not possible because of the context. We propose a solution which is based on the presence of both specialisation and generalisation relationships in an object-oriented programming language. The specification of both relationships is based on a meta-model called *OFL* which provides a support for describing the operational semantics of a language through the definition of parameters and semantical actions. We propose an overview of the expressiveness of OFL and of its implementation and we give also some other interesting applications.

1 Introduction

In this paper we address the problem of the reuse of libraries of classes by client applications when modifications of the class tree is needed. We propose a solution which is based on the introduction of both specialisation and generalisation relationships in future object-oriented programming langages. This idea to combine both relationships altogether is also pointed out in [7] which focuses more on the integration feasibility within existing OOPL. According to the handling of libraries of classes there are other problems to solve like the maintenance of classes (removal of deprecated features, redefinitions, etc.) that may be solved using interclassing [6]. Even if those approaches deal with the use of libraries of classes by client applications, the philosophy is quite different: our approach deals with existing libraries that may not be modified by client applications whereas the other approach is related to the modification of libraries of classes themselves and their consequences in client applications.

To develop this idea, we present two main parts. Firstly, in section 2, we describe a very pragmatic situation where specialisation and generalisation are useful in the graph of types. You will see that the use of only one of them would lead to only a poor approximation.

Secondly, in section 3, we present a practical solution to define a new programming language with both specialisation and generalisation, or to improve an existing language with a reverse inheritance. This solution is based on the

Model *OFL* ("Open Flexible Languages") [2]. The section 4 presents some implementation issues which handle principles of the Model *OFL*. Then, we conclude the paper in the last section, 5.

2 Why both Specialisation and Generalisation?

Our approach is defined in the context where a programmer uses a software library of components (these components could be classes). He may have written this library or not, but he can't modify it. This situation happens very often, for instance when the code is not provided, when it is *copyright*ed, when it has to be left unchanged for existing applications, and so on. The figure 1 give an example of such a very simple library with two very typical classes.

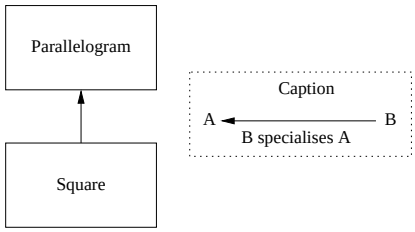

Fig. 1. An existing and unmodifiable hierarchy of classes

Now, for a specific program need or to make the library evolve, we want to add a component in the library (*i. e.* a class in the graph). This fact is illustrated in figure 2.

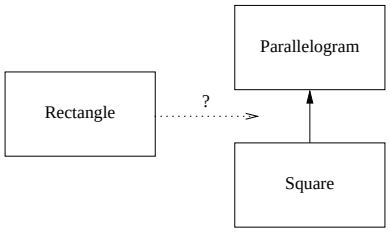

Fig. 2. A new class in the unmodifiable hierarchy

We can imagine three solutions to integrate Rectangle in the hierarchy:

1. The first is the most simple: "If we want to add a class, we must reorganise all the hierarchy!" This solution, illustrated in figure 3, is obviously the best one. But the best one if we can modify the hierarchy and an impossible

one otherwise. And even if we could modify the existing classes, it could be a bad idea: we could add some bugs in some other applications which use these existing library and, in the example, the stability of Square is called into doubt by introducing Rectangle.

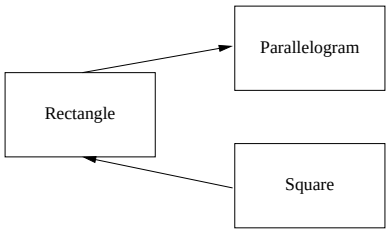

Fig. 3. The first solution: a total reorganisation of the hierarchy

2. A second solution respects the constraint of the unmodifiable existing hierarchy. The idea is to insert Rectangle as a specialisation of Parallelogram, as you can see on figure 4. Here there is no problem with existing classes and the relationship between Parallelogram and Rectangle is correct. But the instances of Square logically have to be instances of Rectangle and this is not the case here.

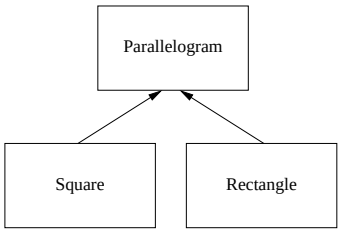

Fig. 4. The second solution: Rectangle specialises Parallelogram

3. The third solution is to take advantage of the fact that, in the end, Rectangle is closer from Square than from Parallelogram. So, the idea is to specialise Square rather than Parallelogram as you can see on figure 5. This solution is valid as long as polymorphism capabilities between Square and Rectangle are not used. The instances of Square logically have to be instances of Rectangle and this is the contrary here.

As we just see, if we have only specialisation (the problem is the same if we have only generalisation), we can make evolution of a graph of classes without risk (e.g. without modifying existing classes) but we can't have, simultaneously,

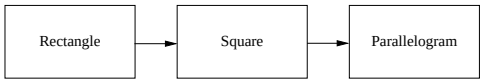

Fig. 5. The third solution: Rectangle specialises Square

a valid behaviour of the types (e.g. correct polymorphism capabilities) in the result graph.

Our proposition is very simple: to add a generalisation relationship in the language. Generalisation is only the reverse link of specialisation so, theoretically, only one of them is sufficient. But practically, we could perfectly resolve our evolution problem with both.

Let's show you a new figure, 6. It demonstrates a good way to handle evolution in our graph of classes. Rectangle is integrated as a specialisation of Parallelogram and a generalisation of Square. But what are the advantages in comparison with the three previous solutions?

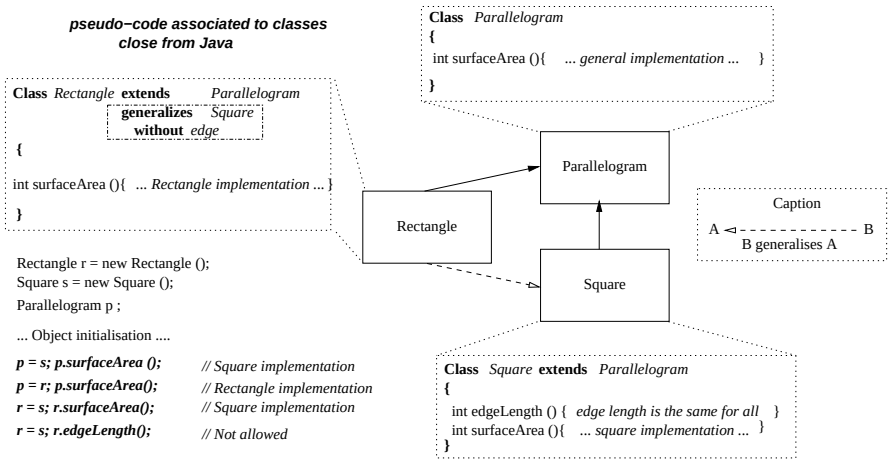

Fig. 6. A satisfactory integration of Rectangle with possible pseudo-code

1. The advantage of our solution in relation to the first one is that no class is modified in the initial graph. If we haven't the code or the right to modify it, we can nevertheless apply a relevant adaptation of the graph. And even if we can modify the initial graph, our solution protects the quality of Square since the new Rectangle must be compatible with the well-tried Square and not the contrary.
2. In comparison with the second proposition, to use both links allows to make capital out of a correct behaviour of polymorphism between Square and Rectangle.

3. In relation with the third approximative solution, the idea to use generalisation is better because the graph of types is relevant : a square is a rectangle and not the contrary!

Obviously the pseudo-code inserted in figure 6 is not self-sufficient to explain the semantics and it should be deeply discussed. Particularly, he has to be linked to the possible parameter values presented in 3.2 and should be further specified by assertions built on the model reification. However, in order to give a flavour of the capabilities which may be provided to programmers, we could say that in a *Java*-like language, the two keywords *extends* and *generalizes* are strongly related to lookup operation (see 3.3) in order to implement polymorphism and to allow to access to class instances as if the class hierarchy was the one described in figure 3.

3 How? The Model *Open Flexible Languages*

3.1 *OFL* in a nutshell

This section presents the *OFL* model (Open Flexible Languages) and its capability to define easily both specialisation and generalisation.

The *OFL* Model aims to describe the main object-oriented programming languages (such as *Java* [3], *C++* [9], *Eiffel* [4], ...) to allow their evolution and their adaptation to specific programmer's needs. To reach this goal, *OFL* reifies all elements of an object-oriented programming language in a set of components of a language. Thus classes, methods, expressions, messages, and so on are the *OFL*-components and are integrated in a specific MOP (Meta-Object Protocol) which allows to extend the set of entities needed for the reification of both languages and user applications.

The meta-programmer creates a language by selecting adequate *OFL*-components in predefined libraries. He can also specialise a given *OFL*-component in order to generate one dedicated to some specific uses. To distinguish the default *OFL*-components from the *OFL*-components created for a specific language, we call *OFL*-Atom the default one. In other words, *OFL*-Atoms are supplied by the model, other *OFL*-components, created for a specific language, are not.

Classes are reified by *OFL*-components. Take the example of *Java*. We have ComponentJavaClass, ComponentJavaInterface, ComponentJavaArray, ... An originality of *OFL* is that relationships are also reified. So, we have for *Java*: ComponentJavaExtendsBetweenClasses, ComponentJavaExtendsBetweenInterfaces, ComponentJavaImplements, ... A more complete list of *OFL*-components for *Java* is given in [1].

To facilitate the creation of an *OFL*-component, *OFL* provides some meta-components, called *OFL*-concepts. So, we have a ConceptRelationship and a ConceptDescription (the word *description* has been chosen to represent classes and all entities which look like classes, such as interfaces). Thus, ConceptDescription is equivalent to a meta-meta-class. In each concept, a set of parameters gives the meta-programmer powerful possibilities to create or adapt an *OFL*-component.

These parameters are detailed in section 3.2 wheras section 3.3 gives an overview of the semantical actions which describe the behavior attached to combinations of those parameters.

3.2 Hyper-Generic Parameters

But how can the meta-programmer easily define the *OFL*-components for the language he wants to create or adapt? In fact, this work may be very difficult and tedious because he would have to rewrite a lot of algorithms such as type controls, dynamic links, use-of-polymorphism verifications, inheritance rules, and so on.

In *OFL*, we provide a way to simplify this task: hyper-generic parameters. All the algorithms are predefined and are customized by hyper-generic parameters which have a value in each *OFL*-components.

In the sequel, we illustrate a subset of the hyper-generic parameters which can be applied to a relationship *OFL*-component to customize it. We explain each parameter and its capabilities of customization, and as an example, we give its value to define ComponentSpecialisation and ComponentGeneralisation. A summary of the differences according to the parameter values of these two kinds of relationships is proposed at the end of the section.

Name This is the most simple hyper-generic parameter. It is the name of the *OFL*-component and it must be unique in a language. For ComponentSpecialisation and ComponentGeneralisation, the name is respectively "ComponentSpecialisation" and "ComponentGeneralisation".

Kind It allows to determine the sort of the *OFL*-component. In *OFL*, we have four kinds of relationships:

- import for inheritance and all other importation links between descriptions,
- use for aggregation, composition, and all other use links between descriptions,
- type-object for all links between types and objects such as instantiation, and
- objects for all links between objects.

The value of Kind for ComponentSpecialisation and ComponentGeneralisation is import.

Cardinality The parameter Cardinality defines the maximal cardinality of a relationship. For example, the value of Cardinality is $1-1$ for a single inheritance and $1-\infty$ for a multiple one. So, with Cardinality, we can customize the relationship to be single or multiple with a single value! All the difficulty of the lookup algorithm, which searches the relevant method in the graph of descriptions, is encapsulated in a predefined action which takes care of the Cardinality value for all relationships used in the application. Cardinality is also useful to limit the multiplicity. We want to specify single links, so Cardinality for ComponentSpecialisation and ComponentGeneralisation has the value $1-1$.

Repetition This parameter is useful if and only if Cardinality is not $1-1$. Repetition indicates if repetition of source-descriptions and target-descriptions are valid for this *OFL*-component (to make repeated inheritance, for example). Repetition is defined as a pair of boolean. For ComponentSpecialisation and ComponentGeneralisation, the value of Cardinality is $1-1$, so the value of Repetition is ignored.

Circularity This is a boolean and it expresses if the *OFL*-component admits a circular graph (value: true) or not (value: false). Often, *use* relationships allow circularity and *import* ones don't. Circularity is forbidden in specialisation or generalisation (we don't consider the case of a class which specialises or generalises itself), so the value of Circularity is false for ComponentSpecialisation and ComponentGeneralisation.

Symmetry This parameter points out if the *OFL*-component provides relationships that are symmetrical. Most of traditional links are not, but we can imagine a ComponentIsAKindOf where the semantics is bidirectional: a boat is-a-kind-of submarine and a submarine is-a-kind-of boat (they resemble each other but none are a specialisation of the other). Neither ComponentSpecialisation nor ComponentGeneralisation is symmetrical so the value for is false.

Opposite We may have, in a language, two *OFL*-components with reversed semantics. This is an essential information for all actions which need to travel through the graph of descriptions. ComponentSpecialisation is the opposite of ComponentGeneralisation and vice versa! We can set a value to Opposite even if ComponentSpecialisation and ComponentGeneralisation are described in the Context of a library because they are in the same library. But if we use one of them separately, the parameter Opposite has to be ignored.

Direct_access In traditional inheritance, features of the ancestor are directly visible in the heir, as if they are declared in the heir. The parameter Direct_access gives the capability to choose the policy of this visibility. If the value is mandatory then all features are inevitably visible. If it is forbidden, none are directly visible (but they can be indirectly visible as we will see in the next parameter). And if the value is allowed then some are visible, some not and the differentiation may be done, for example, by a keyword (such as public, private, ...). For ComponentSpecialisation and ComponentGeneralisation, a relevant value is allowed.

Indirect_access This is the same idea as for the previous parameter but for indirect accesses. Indirect accesses mean accesses naming the target-description. In *Java*, we can use super in constructors, finalisers or redefined methods. By this way, we can access to some features of the ancestor, but we have to specify an indirect access. So, for ComponentSpecialisation and ComponentGeneralisation, the value can be forbidden. This means we are in the context where to put in source-description a method which have the same signature than a method also present in the target-description implies that any method body of source-description cannot access to the "old version" in target-description.

Polymorphism_implication This parameter is very important. Polymorphism_implication can take four values:

- up means that all instances of the source-description (heir in an inheritance link) must be also instances of the target-description (ancestor in an inheritance link). This is the traditional direction for polymorphism.
- down points out the contrary: all instances of the target-description must be also instances of the source-description. This value is very useful to create *OFL*-components like ComponentGeneralisation.
- both is an interesting value. It means that source-description and target-description have the same instances. This can be relevant to describe other derivations of inheritance, such as ComponentVersion. We can imagine two versions of class linked by this *OFL*-component. The two versions represent the same type, so they must have the same list of instances, and dynamic link has to find the good version of features to execute.
- none is the last possible value and allows to define other kinds of inheritance, such as ComponentCodeReuse where features are imported from the target-description to the source-description, but where we need to ensure that polymorphism capabilities are avoided.

The value of Polymorphism_implication for ComponentSpecialisation is up. And for ComponentGeneralisation, this value is down!

Polymorphism_policy This parameter is ignored if Polymorphism_implication has the value none. Polymorphism_policy indicates if a new declaration of a feature in the source-description hides the feature in target-description (value: hiding) or redefines it (value: overriding). This value is double, one for attributes, one for methods. For ComponentSpecialisation and ComponentGeneralisation, we can use a traditional value: hiding for attributes and overriding for methods. Note that on *OFL*, capabilities of overloading is not handled by relationships but by descriptions.

Feature_variance This parameter indicates the kind of variance rule for redefinitions of features, if these redefinitions are allowed (we will see the parameter Redefining later). Four values are possible:

- covariant The type indicated in the source-description must be the same or a subtype (let A be the source and B the target. A is a subtype of B if the value of Polymorphism_implication is up, and B is a subtype of A if Polymorphism_implication is down. If the value is both, A and B represent the same type and if it is none, there is no subtype link between A and B) of the type given in the target-description. This is the relevant value for the parameters of methods of *Eiffel*.
- contravariant This is the reverse of the previous value. The type indicated in the target-description must be the same or a subtype of the type given in the source-description. This choice has been made, for example, by *Sather* [8].
- nonvariant The type indicated in the source-description must be the same than the type given in the target-description. This is the case in *Java*. (if type of parameters of methods are not exactly the same, in *Java* this is overloading and not overriding)
- non_applicable is the last possible value. Meta-programmer uses it if he wants no feature-variance control.

The value of Feature_variance for ComponentSpecialisation can be covariant for method parameters, nonvariant for function results, and non_applicable for attributes (these values could be chosen arbitrarily by the meta-programmer, we select here some typical values). In this case, for ComponentGeneralisation, we would have contravariant, nonvariant, and non_applicable.

Assertion_variance *OFL* is able to describe languages with assertions (precondition, postcondition, and invariant) like *Eiffel*. So, we have a parameter to indicate the kind of variance for assertions:

- weakened The assertion in the source-description must be implicated by the assertion in the target-description.
- strengthened This is the reverse value of the previous one. The assertion in the source-description must implicate the one of target-description.
- unchanged The assertions in source-description and target-description must be equivalent.
- non_applicable means that controls of assertion variance must be avoided.

If we imagine that ComponentSpecialisation and ComponentGeneralisation could be used in a language which handles assertions (if, after all, the language is without assertion, this hyper-generic parameter is ignored), we could give the traditional behaviour:

- **for ComponentSpecialisation:** weakened for preconditions and strengthened for postconditions and invariants,
- **for ComponentGeneralisation:** the contrary, strengthened for preconditions and weakened for postconditions and invariants.

Renaming This parameter points out if the programmer can rename a feature using a relationship defined by the *OFL*-component. For example, renaming is possible in *Eiffel* but not in *Java* or *C++*. The accepted values are forbidden to prevent renaming, allowed to authorise renaming, or mandatory to oblige it. To rename a feature can be very useful in specialisation as in generalisation, so the value for our ComponentSpecialisation and ComponentGeneralisation is allowed.

On the same idea than for Renaming, we have parameters to customize the capability to add (Adding), to remove (Removing), to redefine (Redefining for assertions, method's signatures, method's bodies, and method's qualifiers), to mask (Masking), to show (Showing), to abstract (Abstracting), or to make effective (Effecting) the imported features.

Now, let's study the value of these parameters, firstly for ComponentSpecialisation, secondly for ComponentGeneralisation:

1. **for ComponentSpecialisation:** It is correct to add a feature when we make a specialisation (we can add features with inheritance!), so the value for Adding is allowed but we can't remove a feature, so the value for Removing is forbidden. To redefine a feature is also correct (and what's more it is an essential characteristic of object-oriented languages), so the value of Redefining could be ¡allowed,allowed,allowed,allowed¿ and assertions and signature of each method will have to respect the policies described by Assertion_variance and Feature_variance. We want to make simple links so we prefer to avoid masking and showing, thus the value

of the corresponding parameters is forbidden. Lastly, we could say that specialisation is compatible with making an abstract feature concrete, so the value of Effecting is allowed but not with the contrary, thus the value for Abstracting is forbidden.

2. **for ComponentGeneralisation:** Following a similar thought process (with the opposite semantics), we can set Adding and Effecting with forbidden, and Removing and Abstracting with allowed. We keep forbidden for Masking and Showing. Finally, Redefining could be ¡allowed,allowed,allowed,allowed¿ like for ComponentSpecialisation but with Assertion_variance and Feature_variance of ComponentGeneralisation!

What are the main differences between ComponentSpecialisation and ComponentGeneralisation? The previous list of parameters is given also to present the *OFL* hyper-generic parameters system and could hide the important difference between our two relationships. Here is a reminder of the parameters which have different values:

- Name is obviously different.
- Polymorphism_implication is ascendant for a specialisation and descendant for a generalisation.
- Feature_variance of parameters of methods is covariant for specialisation, contravariant for generalisation.
- Assertion_variance is weakened for preconditions and strengthened for postconditions and invariants in specialisation and exactly the contrary in generalisation.
- Adding and Effecting are allowed in specialisation and forbidden in generalisation.
- Removing and abstracting are forbidden in specialisation and allowed in generalisation.

3.3 Actions

To put a relevant value to a set of parameters gives a good way to describe the wanted behaviour of a sort of relationship. But it's not sufficient to allow pertinent control and execution of these links. So, in *OFL*, a list of actions is present.

Each action defines the operational semantics of a part of work traditionally handled during the compilation or execution time. And each action takes into account the value of the hyper-generic parameters. So the behaviour of the defined language is adapted to the value of each parameter of each component. We have classified our actions in seven categories:

1. **Actions to search a feature** For example: lookup which makes the research of the relevant feature in the graph of descriptions in relation to a message; or match (called by lookup) which verifies if a given message is compatible with a given feature.

2. **Actions to execute a feature** For example: execute_with_result which allows to execute a message and to return an objet as the result (useful for functions or other expressions with a result); or attach_parameters which makes the control and the attachment of the effective parameters of a method.
3. **Actions to make a control** For example: verify_circularity which controls that none relationship with hyper-generic parameter Circularity=false have a circular graph; or are_valid_parameters (called by attach_parameters) which controls that effective parameters are compatible with the definition of the formal ones.
4. **Actions to handle instances of descriptions** For example: create_instance or destroy_instance.
5. **Actions to handle extension of descriptions**.
6. **Base operation** For example: assign which gives the algorithms of the assignment, or copy.

Each action uses the value of the hyper-generic parameters to choose the pertinent way to make its job. For example, lookup has to take into account the very important value of the parameter Polymorphism_implication to go through the graph of descriptions.

How to write an action? An action could be simple. For example, we have an action called verify_circularity which controls that all relationships with false in the parameter Circularity don't make circular graph. The algorithm of this action is simple: to go all over the graph for this relationship and to verify that none description are direct or indirect target of itself. The moment to execute verify_circularity is also easy to imagine: it could be launch once in a static tool like a compiler or a code checker.

But other actions are a lot intricate! For example, let's examine the action lookup. To find the relevant feature in accordance with a message, the task may be difficult and the algorithm complicated. We have to take the value of many parameters into account. The value of Polymorphism_implication is used to build a graph of types. Polymorphism_implication will help to determine which policy (hiding or overriding) have to be considered. With Cardinality and Circularity, we can choose an efficient way to go all over the graph. Symmetry could help us to adopt a two-direction route. Direct_access and Indirect_access give information about the visibility of the target-description. Finally, Feature_variance, Assertion_variance, Adding, Removing, and so on, allow to know how features are imported. Furthermore, the moment when it is correct to execute the lookup is also not obvious. We can easily imagine that a first part of this task is static (determination of all unambiguous calls for example) and another one is dynamic (dynamic linkage at runtime for example). With all these data, it is theoretically possible to write the code of lookup.

Theoretically... But if we want to provide some useful model to the programmer, it is obviously necessary to help him to write actions. In this way, we supply three things. The first one is that we have split complex actions in

more elementary ones[1]. For example, we have a local_lookup which make the local (independently of all import relationships) research of a relevant feature in a description and a match which takes a feature and a message and determines if the second one is compatible with the first one... Thus, we split the difficulty of the complex lookup which has to call local_lookup, match and other actions to make its job.

Secondly, to solve the problem of the static and dynamic facets of our actions, we provide a way to define them in several parts. So, in fact, each action is split in a set of facets and each facets is declared static (used in a preliminary step like a compiler, a code checker, or a first access) or dynamic (used in an interpreter, an execution engine, or a virtual machine).

Thirdly, we intend to provide some default behaviours for all actions. Indeed, *OFL* could be used for a large variety of tools about source code. In this paper, we present a way to assist an extension of a programming language, but it is also possible to use our actions to make others tools like a code checker, some trace service, or a wizard for programming. So, our idea is to write typical algorithms for actions and to supply them in libraries.

Who write the actions? There are three possible answers to this question:

1. As we just explained, *OFL*-designers (we) have to provide libraries of actions for the more frequent usages. These libraries must be for very general purposes.
2. If a relevant solution is not given in these libraries, the meta-programmer (the person who makes a new language or a new tool for code) has to redefine some of the actions or, in a bad case, to rewrite all of them. It is here useful to create a kind of plug-in library which adds some interest to the *OFL* set of tools.
3. Finally, when the meta-programmer wants to add a very particular behaviour, he can redefine or write some actions in order to handle this behaviour. As this case is for a specific use (useful for an unique application, for example), creation of a library is not useful and the redefinition could be temporary.

4 Implementation issues

Firstly, an implementation of *OFL* (cf. fig. 7) is based on the reification of both language semantics (*OFL*-components instances of an *OFL*-concepts) and application entities such as method, attribute, statement, etc. Because it is not reasonable to design a reification which deal with any entity of any language, it is necessary to design an extensible reification model. All this issues are achieved through a Meta-Object Protocol (MOP) written in *Java* and called *OFL/J*. In order to make easier the coupling with other tools, an XML-DTD of *OFL/J* is

[1] Chapter 6 of [2] presents more than fifty actions.

generated whereas meta-information and application reification are stored under an XML representation which conforms to this DTD.

Secondly, the reification of application should be parsed and semantics actions should be performed on each entities according to the language semantics. This will be done by *SmartTools* [5] which allows to define visitors (design-pattern) in order to make possible the description of semantical actions to be associated to application entities. *SmartTools* apply all these actions, automatically to any node of the abstract syntax tree associated to the application reification.

One interesting thing is the flexibility of the system. Actions can be added or removed from *OFL/J* and they can implement the approach described above from different point of view:

- to control the appropriateness between the body of application methods and the relationships defined between the classes within the reification
- to generate pure *Java* code according to the information above
- to do both control and generation

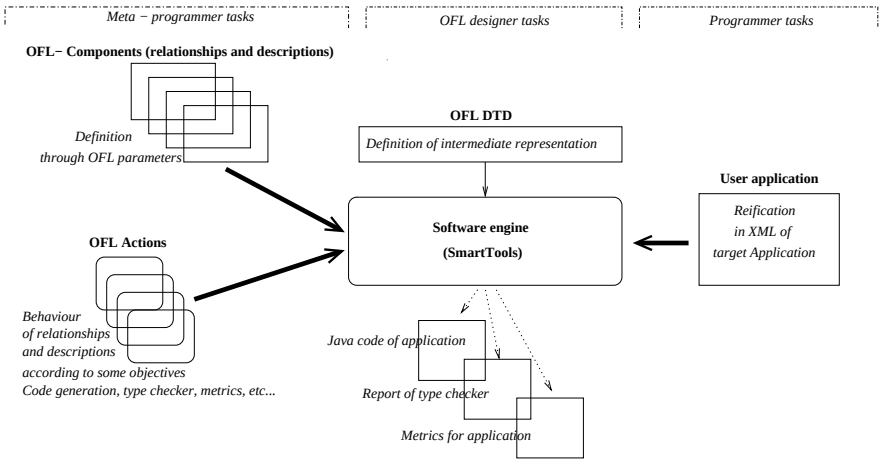

Fig. 7. Architecture of OFL implementation

Many other variants may be found according to the level of reification of statements and expressions. For example another variant could be to insert into action-semantics the code which is necessary to implement an open virtual machine, but at the moment we focus more on simplest solution for the validation of our approach.

5 Conclusion and future work

In this paper we demonstrated that it was interesting to make coexist both specialisation and generalisation relationships, in order to better handle libraries of classes in the design of application. Other relationships also could be useful such as a reuse-code relationship whose aim is to provide one class the capability to include some methods from existing classes without allowing any polymorphism for its instances with those classes. These are only examples of the kind of relationships that *OFL/J*, the implementation of *OFL* model could handle. the part of *OFL/J* which deals with meta and non meta information reification and with the *OFL* Mop for extending the capabilities of the reification are implemented. Now we are investigating how to implement a first version of the semantics actions into *SmartTools*.

References

1. A. Capouillez, P. Crescenzo, and P. Lahire. Le modle OFL au service du mta-programmeur - Application Java. In *LMO'2002 (Langages et Modles Objets)*. Hermes Science Publications, *L'objet : logiciels, bases de donnes, rseaux*, volume 8, numro 1-2/2002, January 2002. also Research Report I3S/RR–2001-04–FR (Laboratoire d'Informatique, Signaux et Systmes de Sophia-Antipolis), http://www.crescenzo.nom.fr/.
2. P. Crescenzo. *OFL : un modle pour paramtrer la smantique oprationnelle des langages objets - Application aux relations inter-classes.* PhD. Thesis, Universit de Nice-Sophia Antipolis, December 2001. http://www.crescenzo.nom.fr/.
3. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* The Sun Microsystems Press Java Series. Sun Microsystems, June 2000. http://java.sun.com/docs/books/jls/.
4. B. Meyer. *Eiffel: The Language.* Object-Oriented Series. Prentice Hall, 1992. http://www.eiffel.com/doc/.
5. D. Parigot. Web Site of *SmartTools*. World Wild Web, December 2001. http://www-sop.inria.fr/oasis/SmartTools/.
6. P. Rapicault and A. Napoli. Evolution d'une hirarchie de classes par interclassement. In *LMO'2001 (Langages et Modles Objets)*. Hermes Science Publications, *L'objet : logiciels, bases de donnes, rseaux*, volume 7, numro 1-2/2001, janvier 2001.
7. M. Sakkinen. Exheritance - Class Generalisation Revived. In *ECOOP'2002 (The Inheritance Workshop)*, june 2002.
8. D. Stoutamire and S. Omohundro. Sather Specification. Technical report, International Computer Science Institute, University of Berkeley, August 1996. Version 1.1, http://www.icsi.berkeley.edu/ sather/ Documentation/Specification/Sather-1.1/.
9. B. Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Co., 3rd edition, 1997. http://www.research.att.com/ bs/3rd.html.