Bridging the Gap between Object Oriented Modeling and Implementation Languages Using a Meta-Language Approach

Dan Alexandru Pescaru

PhD Thesis

Faculty of Automation and Computer Science
"POLITEHNICA" UNIVERSITY OF TIMISOARA

Timisoara
November 2003

Advisor:
Prof.Dr.Ing. Ionel Jian

to Alex and Maria

Motto

All of physics is either impossible or trivial. It is impossible until you understand it, and then it becomes trivial.

- Ernest Rutherford

ACKNOWLEDGMENTS

I am honored to have Prof. Dr. **Ionel Jian** as advisor of my thesis. I would like to thank him for his continuous support on both a personal and professional level over the past years.

I am specially grateful to Dr. **Philippe Lahire** from Sophia Antipolis University of Nice, France. He has been a constant source of advice, guidance and support during all this time. Without his full support, I would probably have never reached the stage where I am today. I could say that most of my work is due to him.

My special thanks go to **Emanuel Tundrea** and **Constantin Papandonatos** for their invaluable help on prototypes of the OFL-compiler and OFL-ML tool, which were presented in theirs diploma thesis. The discussions I had from time to time with Dr. **Pierre Crescenzo** were also highly stimulating.

I want to give tanks to **Diana Andone** and Dr. **Radu Vasiu** for their friendship and all the logistic support, which made possible my research visits at Sophia Antipolis University of Nice.

Numerous colleagues at Computer Science Department, UPT, have indirectly contributed to this thesis, among them **Ciprian Chirila**, **Sorin Serau**, **Doru Todinca** and **Dan Cosma**, not at least by taking over some of my daily obligations during the time of the writing and help me with correcting, printing etc. Many thanks to all of them.

I would like to thank to all my family, my wife Maria, my child Alex, my grandmother, my parents and my parents-in-law for all their love, all their support, including financial one, and for believing in me even when I lost my confidence.

(*Timisoara, November 2003*)

Contents

Chapter 1

Introduction

The starting point of this thesis is the general opinion about the evident gap between object-oriented modelling languages and programming languages. Many companies do not use yet UML, which is the standard of object-oriented modeling languages since many years. Indeed, even they use UML in the analyzing phase they prefer to jump over implementation model for application. Instead they are using to have only an ad-hoc model that resides directly in implementation. First explanation consists in contradiction between generality of UML and specificity of application model after implementation in a programming language.

1.1 The Problem

UML is used in a wide area of contexts, as explained by OMG, by people coming from different cultures, many of them considering (more or less justified) their case special. Therefore they are asking for a particularization of the standard in the form of a tuned version of UML. However, as they said, a hard-coded UML precise semantics would preclude the existence of these tunings and thus would be practically unacceptable.

Developers are in a situation where on one hand need precise semantics to really make the UML a communication mean, while on the other hand they do not need too much specificity due the domains on which UML is to be applied are so different that they can not be unified under a uniform semantics. The response of OMG is the introduction of profiles as standard means to adapt the UML to some domain-specific needs.

1.2 The Goal

The goal of this research, as the title say, is to bridge the gap between object-oriented modeling and programming languages.

The problem appear when the UML is used to create an implementation model. After the implementation of this model, the application will contain itself an intrinsic model. Because a programming languages has a more precise semantic than UML, this two models will be different. If the specification change the problems will appear at reengineering phase.

If we think at UML Profile solution, the problem is how to specify, or to generate in our case, this profile in order to fill the gap. This problem is harder if we think in terms of diversity of existing programming languages, each of them with different versions and flavors.

The approach presented in this thesis try to use meta-information about a programming language described in a meta-language and to generate automatically a well tailored profile adapted to it.

This solution goes beyond the presented goal. This suggest a wider framework which allow also extensions of the programming languages not only generation of tailored profile. This way the programming language and the corresponding profile can become closer to real applications and domains.

1.3 Organization

Chapter 2 presents an overview on state of the art in the fields of modeling languages and meta-programming. It contains discussions and critics about existing approaches and try to define exactly the problem.

The *OFL-Model* is presented in **Chapter 3**. This model represents a parametric meta-model that allows description of object oriented programming languages. This chapter presents main concepts behind this model and represents an analyze of good and poor aspects of this model.

The model extension proposed in **Chapter 4** is the result of conclusions extracted from previous section. It enable customization of language modifiers and enrich the adaptable semantics of the *OFL-Model*.

Chapter 5 represents the main section of this thesis. During this chapter is developed the approach regarding generation of OFL-ML profiles. It describe the OFL-Meta-Profile and the Virtual Meta-Model that resides behind it. It contains also all the rules that are used to generate a specific OFL-ML Profile.

Chapter 6 presents working framework and the tools support necessary to validate the approaches presented in the previous two chapters.

Chapter 7 summarizes the thesis and point out the research perspectives of presented approach.

Chapter 2

State of the Art

There are several options in the field of object oriented modelling languages and meta-languages extensions. There are also several approaches to reduce the gap between models and implementations. This section try to offer an overview of some of those in a critical manner.

2.1 Design Methods

2.1.1 Unified Modelling Language - UML

The Unified Modelling Language (UML) [OMG03], as defined by OMG, is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.

The Logical Model in UML can be used to model the static structural elements of a software system. It captures and defines the building blocks (artefacts) as objects and entities of a software system. Classes are the generalized templates used to create run-time objects. Components are built from classes. Classes and Interfaces are the design elements that correspond to software artefacts created on the implementation step of an application.

The Unified Modelling Language is, as its name says, a modelling language, and cannot be used as a programming language or a process. It consists in a specific notation and a related grammatical rules for developing software applications models. It does not specify the way to use that notation in a software development process or, as in our case, as part of an object-oriented design methodology. However, there is a strong link, but also a gap between the UML model and the intrinsic implementation model. UML contains a complete set of graphical notation used for describing classes, components, activities, work flow, objects, and relationships between these elements. Stereotyped elements can be used for custom extensions. The language provides important benefits to programers and software companies by helping them to build rigorous, and maintainable models at all levels of software development process.

The interest of this theses concentrates mainly in Class model. As defined by OMG a Class is a standard UML construct used to detail the pattern from which objects will be produced at run-time. AS in many object oriented languages an object is an instance of a class. To extend the expresiveness, classes may be inherited from other classes. This way they inherit all the properties and behavior, but also could add new functionality. They could have attributes of the type of other classes, they can delegate responsibilities to other classes. They also can implement abstract interfaces. To gain a more precise semantic real models use a variety of *stereotypes* and related constraints applied to UML class element. They are used to model different kinds of classes belonging to object oriented languages (like Java Abstract Class, Java Interface, C++ abstract class etc.) .

I focus here on the Class Model because it is the core of object-oriented development and design. As mentioned in [Fla99a], it covers the persistent state of the system, and the behavior of the system. A class encapsulates state (attributes) and offers services to manipulate that state (behavior). Good object-oriented design limits direct access to class attributes and offers services which manipulate attributes on behalf of the caller. And very important, this data hiding and services exposing supports data updates that are done in a single place and following specific rules. A class model contains in some cases the implementation encapsulated at the level of method body.

To design a model logical UML elements may be related in a variety of ways. The following relationships are the most used in practice: association, aggregation and inheritance relationships.

Association relationships [OMG03] express the static relationships between entities. These generally relate to one object having an instance of another as an attribute or being related in the sense of owning (but not being composed of, as defined by OOP paradigm). Programers will use this element but in conjunction with stereotypes that define relationship semantics.

Aggregation relationships [OMG03] define a kind of whole/part relationships. A stronger form of aggregation is named composition. It specify that a class not only collects another class, but also is actually composed of that collection. There exists also some other flavors for aggregation.

Inheritance [Ewi, Car88, Por92] describes the hierarchical relationship between classes. It describes a *family tree*. Classes may inherit attributes and behavior from a parent class (which may in turn be the child of another class). This tree of inherited characteristics and behavior allows the designer the ability to collect common functionality in root classes (ancestors) and refine and specialize that behavior over one or more generations (children).

In the proposed methodology I pay a special attention to scope modifiers, as public, protected and private. They determine in a UML Class model which elements may be inherited and which are not visible. UML use generalization relationship element to denote inheritance. However, an enhanced model for Inheritance is obtained through stereotypes and tagged values added to UML generalization. This way it could make distinction between different types of import relationships like class inheritance and sub-typing. UML visibility mod-

ifiers represents a real problem for programmers [FS01]. They are used to define scope of inherited elements through relationship but also to define some level of protection inside the code. Instead of that, I will try to define a adaptive specification for this modifiers [PL03] that model the existing access control and protection specifiers from the target language. Moreover, following this way supportcan be provided for other king of features modifiers that address various semantic (like optimization or services).

UML Dynamic Relationships [Aal02] represents the messages exchanged between classes as objects at application running time. A Sequence Diagram is used to analyze the message passing and the sequence in which it hapapend. These model elements have an association to each other reflected at run time by the passing of messages to each other. I do not consider Dynamic Relationships in this thesis. However, a future research can analyze the opportunity to provide Instantiation Relationships (that are objects to classes relationships). This relationships model the link between application run-time instances (objects) and class meta-instances (meta-objects).

In UML, a logical model element and the attributes, associations and operations that it has, may all be further specified with constraints. Such constraints are important to catch a specific semantic in the code.

Consgtraints are expressed by UML contractual rules that apply to an element and/or its features. Typically they fall into one of three types:

- Pre-conditions. They must be true prior to the operation execution or prior to creation/existence of an element;
- Post-conditions. They must be validated after the destruction/deletion of an element or object;
- Invariants. They must always be true along the entire life spanning of an entity or object.

Any modeling language need support for these rules. This ussualy imply support for assertions. In UML they are modelled most of the time using the Object Constraint Language OCL [R. 02, BH00].

As the class model develops, classes (and interfaces) may be organized into logical units or packages in UML. These collect related elements (and in some implementations will govern the visibility of operations and attributes) by elements outside the package.

In most programming languages packages are used also as structures design to support a better organization of user application elements. This corresponds to UML package concept and to class libraries organizational model in object oriented programming languages. Generally, a package can be ssen as a grouping of model elements. It may contain a set of different kinds of model elements. Each model element can be directly owned by a single package. However, packages can reference other packages, so the whole usage network can be seen as a graph structure. As defined by UML the dependencies among the elements belonging to different packages create dependency relationships between packages. This dependencies could be used to write specific constraints.

2.1.2 UML Profile

UML is defined by its meta-model [Obj01, Lem98]. In [Des99] it is discussed how specific domains that require a specialization of the general UML meta-model can define a UML profile to focus UML to more precisely describe the domain. Even as concrete UML profiles have started to emerge, use of the profiling mechanism is still discussed [DSB99, AK00].

An UML Profile is created as a collection of UML extensions. Such extensions can be elements as stereotypes, tagged values or constraints specific for a domain. A profile is completed with specifications of the mappings of some domain concepts to such extensions, and specifies additional well-formedness rules. These are commonly written in OCL or using a subset of natural language.

Virtual Meta-Model of Stereotypes The UML specification makes the following comment in its discussion of Stereotypes [OMG03]:

The stereotype concept provides a way of classifying (marking) elements so that they behave in some respects as if they were instances of new "virtual" metamodel constructs.

In the UML meta-model, a Stereotype is a GeneralizableElement [OMG03]. Therefore it is a common practice to define Generalization Relationships among Stereotypes. Furthermore, a GeneralizableElement is a ModelElement, therefore Dependency Relationships can be defined among ModelElements. As consequences, it is accepted for Stereotypes to participate in Dependency Relationships.

In the UML meta-model, a Stereotype can be used to extend elements of the meta-model.

The notations used in this thesis for Stereotypes folows the notation defined by OMG in UML specification. Some abstract Stereotypes are defined and, in line with the UML notation, the abstract nature of them is emphases using italic fonts for the Stereotype name. Also, in line with programming languages, an abstract GeneralizableElement cannot be instantiated in UML. The abstract Stereotypes are useful for avoiding repetition in multiple Stereotypes that logically have common properties.

Using UML Notation for Virtual Meta-modeling. In light of these facts, the specification takes the following approach to using UML notation to express the virtual meta-model:

- The model is expressed via class diagrams.
- Each Stereotype plays the client role in a Dependency Relationship with the UML metaclass that it extends. These Dependencies are stereotyped ≪baseElement≫. We use this as non-standard notation because relationships afford greater clarity than TaggedValues.
- Each Stereotype is expressed via a Classifier box, even though a Stereotype is not a Classifier. The keyword ≪stereotype≫ does NOT represent

a stereotype itself - it is simply a notational marker for the underlying Stereotype meta-class.

- Generalization Relationships among Stereotypes are expressed in the standard UML fashion.

2.1.3 Considerations About UML Semantics Included in Profiles

Almost everyone admit needing a precise UML semantics [BCR00, Obe00, Hey01]. Next issues reveal the problems that appear when try to achieve this goal. I try to respond at least partially to this problems when propose OFL-ML meta-profile.

The granularity problem. Part of the precise UML semantics should be contained in domain-specific parts, i.e. profiles. The question that naturally arises from this is what do we put in the basic UML and what do we put in the domain specific parts?

Semantics defined in profiles. A lighter approach is to leave the UML definition as it was proposed. Each profile should contain all the semantics that describes it. The advantage of this lighter approach is that it allows almost any "UML semantics" to exist. However, this will not determine an increase in the UML modeling precision. Indeed, it only add some semantics to UML variants. I am not interested by this approach in my research considering the following reasons. An important goal of developing a precise UML semantics is to ensure that UML offers a communication means between modelers. However, it is compromised as the same UML model may be understood differently by different peoples. Actually, this approach would lead to the transformation of UML from a modelling language to a modelling paradigm. If no concept had any semantics, then UML would only be a vocabulary of terms with different meanings in different contexts. Concretely this approach would consist of leaving the UML definition as it is today (possible removing inconsistencies and omissions, if they are found), and adding precise semantics into UML profiles.

Semantics shared between profile and meta-model. Another approach propose to add semantics into profiles, and to also add more information into the UML definition. A possible solution is to define the concepts, relationships between them, constraints, some more precise semantics of them in the common UML. Moreover, for each concept it is stated explicitly in the common UML whether it can or not be refined, or redefined in a profile. The advantage is that the impact of different variants is reduced and localized and anytime someone will look at a UML model it will be clear which elements are susceptible of having a semantics different from the common one. In the same spirit we could imagine not only having concepts whose semantics can be refined, but also to

having without semantics, so that any UML profile should clearly state what is the meaning of that concept in its context.

Flavors of the flavors Assuming we know how to partition the semantic between the common UML and the specific profile, the next questions that comes is how much information the couple common UML plus specific profile should contain? I can take as a concrete example the under-worked UML profile for real-time [RFPRT], as it is one of the first profiles approved by the OMG. The purpose of the real-time profile is to offer specific means appropriate for the real-time domaina applications. Although it solves many of the UML problems, it still may need further refinement to be valuable in practice. The real-time application domain is itself vast, therefore a unique profile could not address every specific demand. As a result, if one would compare the UML profile for real-time with a solution dedicated to a specific real-time field, such as the SDL [SDL] primary designed for telecommunication, the conclusion would be that the real-time profile offers less than the existing solutions and may still need further refinements.

Although the problem of not having a precise semantics has been often signaled, having a precise semantics of UML is still an aspiration.

2.1.4 UML Action Semantics Model

Actions Semantic Model [OMG03, SPH$^+$01] is a promising technology included in last version of UML specification. It aims to provides both a meta-model integrated into the UML meta-model, and a model of execution for the statements for both application code or constraints. As a OMG standard, the Action Semantics provides a simplier way the achieve tools interoperability. Moreover, it also supports executable modeling and simulation.

The fundamental elements defined by this model are the next:

- Action - fundamental unit of computational behavior
- Action semantics are based on proven concepts from computer science
- Action semantics remove assumptions about specific computing environments in user models:
  - execution engines, PLs, implementation details
  - do not require specification of software components, tasking structures or forms of transfer of control
  - yet allows modelers to produce executable specifications

In a general case, an action takes a set of inputs and converts them into a set of outputs as a imperative programming function does.

- Input pins - hold values to be consumed by the action
- Output pins - hold values generated by the action

- Pins are type conform - The type of the output pin is the same as or is a descendant of the type of the input pin
- Fan out of output pins is allowed
- No fan in of input pins is possible

A data flow manage execution of two actions by transferring data between them. It ensures action sequencing. Such data flow connects source and destination pins. The output data of one action represents the input of the next one. A control flow defines a sequencing dependency between two or more actions. It ensure a kind of explicit sequencing of them. The second action of a control flow may not execute until the first action has completed execution. The specification tres to maximize the action concurrency. All actions are considered as executing concurrently unless they are explicitly sequenced by a flow of data or a flow of control.

Primitive actions do not contain any sub-actions (e.g. some nested actions). Procedure is defined as an action container. They rerpesents a set of actions within a model such as a body of a method in a programming language. Procedure provides a context for action execution. Inside the model, a procedure takes a single object as input and produces a single object as returning result. Multiple arguments or results can be implemented if they are represented as object attributes. A procedure may be attached to a method.

UML: Kinds of actions. New Data Types may be defined using the UML meta-model. If we consider a UnlimitedInteger type - it can represent a data type whose range is the nonnegative integers augmented by the special value unlimited. It can be used to represent the upper bound of multiplicities. It can support read and write actions applied to variables, attributes, links etc. It can also support composite actions or group actions, conditional actions and loop actions. Computation actions can be defined as needed as ApplyFunctionAction, CodeAction, MarshalAction etc. However, more discussion on UML meta-model for Data Types is beyond the scope of the this thesis.

A special kind of actions are represented by collection actions. They contain a subaction, which is an embedded action that is executed once for each element in the input collection. We can aply iterate procedure that execute a subaction on each element that belongs to a collection, repeatedly within a loop. It exists also a special filter procedure that selects a subset of elements in a collection and creats a new collection, or a map action that applies a subaction to each of the elements in a collection, in parallel fashion. Actions for synchronous, asynchronous invocation are denoted as messaging. A special kind of action is represented by jumps as break, continue, or exceptions. Derived languages may define their own actions.

2.1.5 J-UML

In most parts J-UML [Kai99], as an extension of UML, can be understood as a subset or customisation of UML. J-UML model is totally Java oriented. It supports a full graphical OOD/OOA of the actual Java source code intrinsic model. Howevern, the J-UML is not by any means trying to reduce the generality of language independent modelling. On the contrary, it tries to build a bridge between these two distant worlds as a kind of adaptation. It defines how to map the UML elements into actual Java implementations.

The J-UML motto is:

"You can't design anything that can't be straightforwardly transferred to Java."

I can say that our goal is quite the same with a small change: Java will be replaced by "target programming language (Java, C++,)". Because of this change, the manner of UML customization differs hardly from the original representation of J-UML.

J-UML try to solve such contradiction specific to any language independent modeling environment by providing several ways to implement any language independent model (like UML) in any specific language environment (like Java).

The basic understoonding is that J-UML ensure notation for a Java Class as an extension of UML Class representation. To achieve that it provides new compartments to the rectangle representing the class. Such example are the Events compartment and Exception compartment able to handle specific Java class syntax.

J-UML use UML visibility specifiers as (+, #, -) to define the visibility of class for proprieties interchangeable with keywords public, protected and private. The main difference is on interpreting the absence of specifiers, seen as default syntax. The UML considers by default the propriety as public but J-UML considers it to have package visibility to comply with the Java semantic. In addition of that, all Java modifiers, like static or synchronised, could be used. Howeve, using of UML specifiers is not a good point.

Moreover, J-UML has a special notation to refer the Java API classes. Using this notation, classes that do not belong to application model can be referred in an opaque manner, even if their implementation is not visible from the model. UML does not provide anything like this but other systems, like Express-G with "defined data types", does. The intent here is to cover two aspects of programming. The first one is to allow use of basic types defined by the language binding. The second is to support code reusing from libraries or other projects. The main impediment is the "opacity" related with the second aspect. The internal structures of basic types are hidden, and therefore it cannot be examined. The escape from this deadend imply to use a kind of "no control" policy. The meaning of such policy is to pass all responsibilities to the language-binding compiler. Indeed, no verification can be carried by the modeling tool related with the usage of such descriptions.

2.2 Programming Languages Extensions and Meta-languages

2.2.1 Java Extensions

OpenJava [TCKI00] is an extensible language based on Java. The OpenJava MOP (Metaobject Protocol) rerpesents the extension interface of the language. Through the MOP, the programmers can customize the language to implement a new language mechanism. As stated by its definition the OpenJava helps programmers who want to develop better Java libraries in the sense of easy-to-use and efficient ones. It also helps programmers who intend to define their own extended Java languages. On the other hand OpenJava can be regarded as a toolkit useful to construct a Java preprocessor. The special feature of the OpenJava MOP is its class meta-object API. Using this meta level, a programmer can manage source code in a object oriented language way by accessing classes, methods, fields, etc. Having in mind that its translation is performed at compile-time a parralel can be made with Java Reflection API at runtime. Therefore it is easy to use for high-level translations [Gui98]. Indeed it is useful when the programmer needs to extract information about methods, or she/he needs to add methods, to modify methods and so on in an easier manner.

Other extensions of the Java reflection are Refelxive Java [Wu98], Dalang [WS98] and metaXa [GK98]. They all share the same orientation to Internet, and same concerns as transactions, security, concurrency, distribution, mobility and persistency. All of them make use of separation between meta-code and the application code. Same time they provide the customization of methods invocation. For instance metaXa offers "before" and "after" routines for method invocation. This makes possible to customize routine computation on one object through several meta-objects that are associated with it by links.

All the models briefly described above aim to extend and open Java language in a kind of structured way. Each of them provides characteristics close to OFL approach : *before* and *after* methods routines, attaching of several meta-objects to the same base level object of the application etc. The main differences is the goal of OFL to obtain the language independence.

2.2.2 C++ Extensions

As is mentioned within its name, OpenC++ [Chi99] has been designed in order to provide new capabilities to C++ language. It main goal is to simplify the tasks for programmer, such as the modelling of a new type system by using meta facility constructs. An important target of the OpenC++ is the development of syntactical/semantical extensions of C++. This approach, as declared in the documentation, focuses on efficiency and handles meta-information at compile time. OpenC++ supports facilities as object assignment, handling of different kind of expressions, function invocation, creation and deletion of instances, access and updates of variables. In order to handle its customization, the meta-programmer has to build a meta-class, which inherits from the meta-class Class.

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

Another example is Iguana [GC96]. It enable a meta-programmer to select the concepts that should be reified independently from each other. Iguana allows the alteration of default semantics of entities by inheriting from the class that describes the realization and specializes the methods of it. The set of meta declarations is used in conjunction with the concept of protocol and it is allowed to build a new protocol. Therefore, such protocol is created staring from existing one. The protocols used in a class are selected at declaration time in this case. The main customizable concepts, as defined by Iguana, are method invocation, creation and deletion of objects. In addition it supports customization of message passing, feature search, and activation/deactivation of semantical controls.

Both OpenC++ and Iguana are based upon a same existing language for which an open programming environment is proposed. OFL approach is somehow different considering the fact that it propose a model that is not based on any particular programming language. Another important distinction is marked by the central position of links in OFL. As explained by its developers, this corresponds to a strong determination to isolate the meta-code that handles the relationships between entities from the meta code that handle the class semantics.

2.3 Design Patterns

Anoter notion reffered in this thesis is represented by the design patterns. A design pattern [GHJV94] provides a scheme for refining the subsystems or components of a software system, or the relationships between them. A design pattern has capabilities to describes commonly recurring structure of linked components that solves a general design problem considering a particular context. A design pattern is a template described by means of software design constructs. Such constructs can be objects, classes, inheritance, aggregation or use-relationship. A design pattern identifies the involved set of classes and corresponding objects, their roles and collaboration relationships, and their responsibilities in the cosidered context. A design pattern express a particular object-oriented design problem or issue with a recurrent ocurrance in practice. To be useful it has attaced information regarding where to apply it, whether or not can be used in presence of other design constraints. In addition it has to explain the consequences and trade-offs of its use in a particular situation. Design patterns are focussed as the name said on design problems and not on a particular programming language. Despite of this, programming languages give their own flavors to usage of a design pattern as explained in [Coo98].

2.4 Discussion

The easiest way to fill the gap between design and implementation model is to restrict UML to an existing language's capabilities. In that case we can speak about Java-UML, C++-UML etc. The main problem resides in loosing the "universal" characteristic of UML and in problems related in addition of some "non-standard" elements. A relevant example (J-UML) was analyzed in this chapter.

Using OFL I try to make a kind of "open restriction" of UML. It is like creating an open set of UML restrictions related with languages like Java, C++ etc. or with extension of that of languages. In that case we want to move the "universality" at the level of OFL meta- programming instead of level of modelling. As the result, my approach will represent in a way a collection of UML restriction. Since the collection is open we can say that OFL and OFL-ML does not really restrict in fact UML, it just uses it's model in other way.

The approach presneted here do not change the meaning of UML:

- I proposes the user to define its semantics for a class or a relationship using the semantics of OFL and to associate to these semantics a set of tags.
- I want to nudge the way to use the UML elements. On the one hand is possible to define whatever programming element a programmer needs at the level of OFL meta-programming and to use it in OFL-ML. On the other hand is not allow to use a element in OFL-ML, which have no corespondent at the level of OFL.

The main benefit of the approach presnted in this thesis approach is represented by the possibility to have a direct and an exactly matching implementation for the model but not loosing the fact that we are at the design level, because of the extended number of class and relationship semantics.

Chapter 3

The OFL model

3.1 Intuitive approach

OFL is the acronym for Open Flexible Languages [Cre01b, A. 00, P. 02, CCL99]and the name of a meta-model for object oriented programming languages based on classes. It was developed in France at University "Sophia Antipolis" of Nice. It relies on three essential concepts covering important aspects of object oriented languages: the descriptions used as a generalization of the notion of class, the relationships concepts like the inheritance or aggregation and finaly the languages themselves. OFL works based on customization aplied on these three concepts. The main goal is to adjust their operational semantics to the programmer's needs. This allows to specify new types of relationships and classes that can be then adapted to an existing programming language in order to improve its expressiveness. This way the described mechanism can also increase the code readability and the language capability to evolve.

The OFL-ML (OFL Modelling Language) is designed as an meta-profile that allows automatic generation of UML profiles tailored for OFL-languages. It is based on OFL and on UML profiles. OFL-ML will be design as a key feature in implementation of the OFL Framework [Pes01, PL00]. It is intended to allows using of OFL extension for existing object programming languages close to application models. The meaning of extension is that: is not possible to remove any kinds of classes and relationships that already exist within the language but only to add new kind of classes and relationships [CL02b]. For example it is not possible to remove the kind of class called "interface" in Java but it is allowed to add another kind of classes if needed.

The existing programming language is selected by defining a binding between an UML Profile and this language. All method bodies will be implemented according to the syntax of this language. Indeed, OFL does not provide customization at the code level of methods body.

One of the main goal of this approach is to allow programmer to reduce the gap between UML modelling structures and the *target language* used for

implementation. By *target language* we mean the object oriented programming language reified or extended using OFL.

The intent is to avoid the necessity to develop separate UML extension for every target object oriented language. As intended, this modelling language will be closer with implementation language than UML is. This goal could be achieved based on OFL feature to extend modelling capabilities of target language. This way our approach will avoid usage of general modeling features. Instead, the OFL specific features will be used.

The advantage of this solution comparing with *custom UML* languages like J-UML [Kai99] resides in its independence to the implementation language. The advantage related with reflective languages, like Iguana [GC96], Open C++ [Chi99] or Open Java [TCKI00], consists in a considerably less meta-programming work, tanks to the OFL. Also, unlike OFL Framework, the reflective language does not provide support to graphical modeling. The modeling tools in this framework will be a combination between an existing modelling tool, like Rational Rose, and an IDE (Integrated Development Environment), like IBM Visual Age. Although, commercial modeling tools can be adapted to use an OFL-ML Profile, using standard UML Profile mechanisms [Des99].

3.2 Overview of OFL Model

OFL was first designed as a meta-object protocol such as that of CLOS (Common Lisp Object System) [KDRB91], but it is more open and complete that CLOS. The concept evolves then to a hyper-generic approach to solve the problem of distance between modeling and coding. The central approach here is based on generecity. Genericity can be seen as the ability to customize the behavior of a class in an object oriented language just as for Eiffel [Mey02, Mey97] or C++ (template) [Str97] generic classes. Going further, hyper-genericity rerpesents the ability to customize the behavior of the language itself. The main mechanism is based on using a set of OFL parameters. The language adaptetion is based on selecting appropriate values for such parameters used by a set of algorithms instead of redefining language behaviors. Indeed, these predefined algorithms, already implemented in OFL framework, take into account the values of these parameters to achieve the desired behavior. The algorithms are called "OFL actions" and they define in fact the operational semantics of the language.

On a basic undertanding, the OFL approach can be reduced to the search for a set of parameters whose value determines the operational semantics of an object oriented language based on classes. Indeed, it defines a set of parameter [CCCL01], which represents the main features of the behaviors of these three central notions known as concept-relationship, concept-description, and concept-language. For instance, concerning the concept-relationship, the value of the Cardinality parameter will allow meta-programer to decide betwen simple or multiple cardinality. Another example is the Generator parameter associated with concept-description. Its value determines whether the concept-description

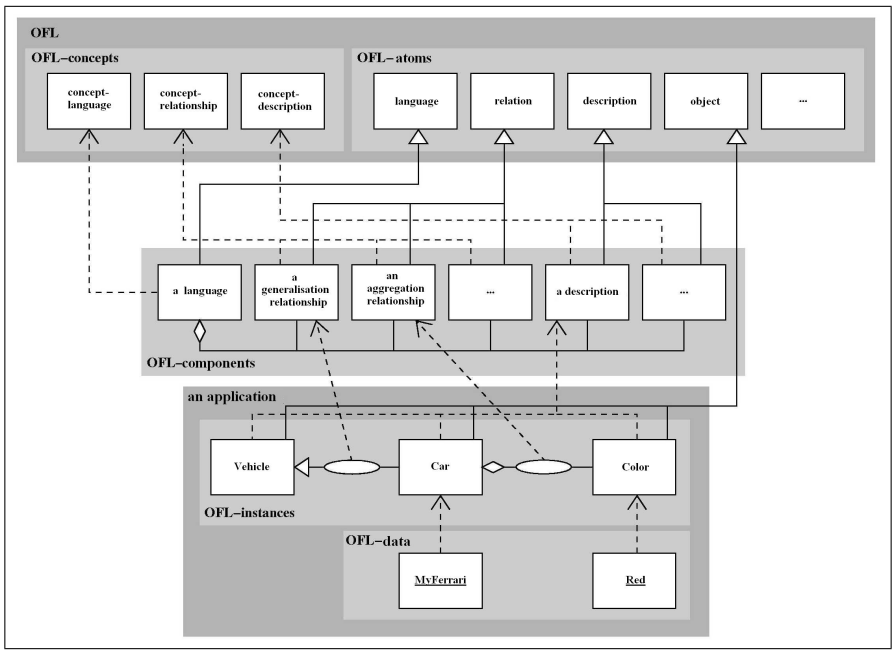

Figure 3.1: The OFL Architecture

can or cannot create own instances.

The operational semantics of each concept must adapt to the value of its parameters. This is achieved thanks to a set of action's algorithms whose execution depends on the parameters values. E.g., the assignment of an object to an attribute can be expressed using parameters of the concept-description; the dynamic binding of the features can be also enabled using parameters of concept-description; the sending of messages behavior is decided according to parameters of concept-relationship etc.

OFL links two facets to each action. The first aspect express the static part inside an interpreter or a compiler. The second facet covers the dynamic aspect generated as specific code executed at runtime by the compiler. The distribution of the code between these two facets depends on implementation choices of the OFL model. Figure 3.1 illustrates how to use the OFL Model to describe an application. The notation follows the UML convention. Three levels of modelling 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* like *ComponentJavaClass* or *ComponentJavaExtends*), and

3. the OFL level represents the reification of those components (OFL-*concepts* and OFL-*atoms*).

The OFL *atoms* represent the reification of the non-customized entities of the model. The *relationships*, *descriptions* and *languages* have their own OFL atoms to describe the part of their structure and their behavior, which are not customized.

The OFL *components* inherit from *atoms* and represents reification of language entities (*relationships* and *descriptions*). All components have a set of characteristics keeping meta-information for programming language entities (OFL *instances*) such as lists of attributes and methods for a description component. Another example is a lists of features for a relationship component. Ultimatelly, the language itself is a component that ensemble together the relationships and descriptions which are part of the application code.

In order to model an application, the programmer uses the services supplied by the language level. For that he masters OFL-*instances* in form of descriptions and the relationships as part of the model. This can be done by instantiation of various OFL-*components*. On the runtime level there will be some OFL-*data* corresponding to the application objects. Indeed, their denotes OFL-*instances* representing the descriptions.

3.2.1 OFL Level: OFL-Concepts and OFL-Atoms

The OFL model is a meta-model for object oriented programming languages, considering the language level. At the same time it represents a meta-meta-model for the programs (applications) itself at an application level. To acieve that the OFL customize three important notions as the relationships, the descriptions and the languages itself. Moreover, some others components need to be described such as the objects, the methods, the assertions, etc. These extra modeling is necessary in order to describe a language in a more completely manner. Indeed, the OFL level includes two types of entities:

- the OFL-Concepts. They reifies the adaptable aspects of components such as relationships, descriptions and languages, and
- the OFL-Atoms. They describes the fixed part of these three concepts as well as all the other components included by OFL.

Moreover, supplimentary assertions are necessary to be included in each OFL-concept and OFL-atom to keep the model consistent.

OFL-Concepts Figure 3.2 shows a complete set of the classification of the OFL-concepts. As previously described, only the OFL-Concepts are customizable in the OFL model. Therefore, the meta-programmer work consists in creating a OFL-Components as instances of OFL-Concepts. This can be achieved by setting values to each of components' parameters. This represents a way to describe the behavior of each instance of designed OFL-Component. However,

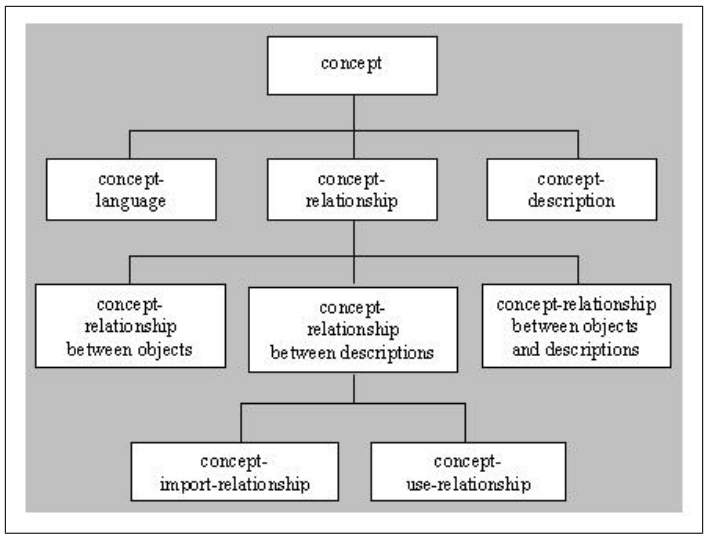

Figure 3.2: The OFL Concepts

if the operational semantics intended for an OFL-Component does not match the intended action, the code of these actions can be modified accordingly.

However, this is not a very common practice. The OFL model is kept open, but this feature should be used only in very specific context. In this case, the complexity of the meta-programmer job will increase a lot as it implies a lot more than just giving appropriate values to OFL componets' parameters.

The Concepts-Relationships. OFL Concept-relationships repesents entities modeling various kind of languge relationships. Therefore, a concept-relationship represents in fact a meta-relationship. Among other relationships, that may appear in object-oriented languages based on classes and objects, common examples are inheritance, aggregation, composition, generalization, etc. Moreover, a designer can use some of them in order to simulate some others. For example the UML Generalization relationship describes both a generalisation and an inheritance, a strict sub-typing. Some other example can be also easily found.

In order to describe the concepts at a very fine grade, a complete set of parameters are defined in OFL. For example more than thirty parameters are used to describe the semantics of all concept-relationships in the OFL model. Figure 1 depicts a expressive classification of the concepts-relationships. Concerning the inter-description relationships, a distinction should be made between the import relationships, which represents generalisation of the inheritance mechanism, and the use relationships, which corresponds to a generalisation of the aggregation mechanism.

The modeling power of OFL span also over the relationship between objects and classes. Among other possibilities, they can be used to model the instantiation relationship that links an object and its class in an object ori-

ented progamming language. Moreover, it is possible to model the relationship between objects itself, at the runtime level. However, the OFL model mainly concerns inter-description relationships, as they represnets main concept in modeling an application.

The OFL *Concept-Description* is used to describe the notion of a class and others class like entities such as the interfaces in Java. Therefore a concept-description is a kind of meta-class for describing a class like entity.

A skilled programer knows that even if class entities look the same in Eiffel, C++ or Java, they demonstrate important differences. Teherfore around twenty parameters are necessary to implement the behavior of a description (rerpesenting a class) in the OFL model. As for programming language classes each such OFL concept-description is compatible with a set of OFL concepts-relationships. Taking Java as an example, the OFL concept-description for a Java interface is compatible with the OFL concept-relationship for Java implementation, but it is not compatible with the inheritance between Java classes.

On the top level, the *Concept-Language* represents an important but simple notion used to models a language. Indeed, an object oriented programming language includes a set of concepts-descriptions and a set of concepts-relationships linked with at least one concepts-description. The concept-languages are hardly customized, and their main goal is to ensemble together concepts-relationships and concepts-descriptions described in context of corresponding language.

OFL-Atoms OFL Athoms represent the realization of the fixed entities of the model. Indeed, OFL Athoms represents non-customizable entities. Figure 3.3 illustrates a part OFL-Atoms hierarchy. The OFL relationships, OFL descriptions, and OFL languages have their own OFL-Atoms to describe the part of their structure and their behavior. But all thses are not customizable entities. Considering an object oriented application, all the features of a description are instances of an heir of feature. Same way expressions are instances of expression or of one of its heirs. Therefore the OFL Model gives a full reification of the entities found at the application runtime.

3.2.2 Language level: OFL-Components

The language level describes different types of relationships and descriptions, which can be used in the targeted language. As mentioned, relationships are instances of concept-relationship when descriptions are instances of concept-description. The language itself is an instance of OFL concept-language. Its main function is to group relationships and descriptions which are supplied to the programmer.

3.2.3 Application Level: OFL-Instances and OFL-Data

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

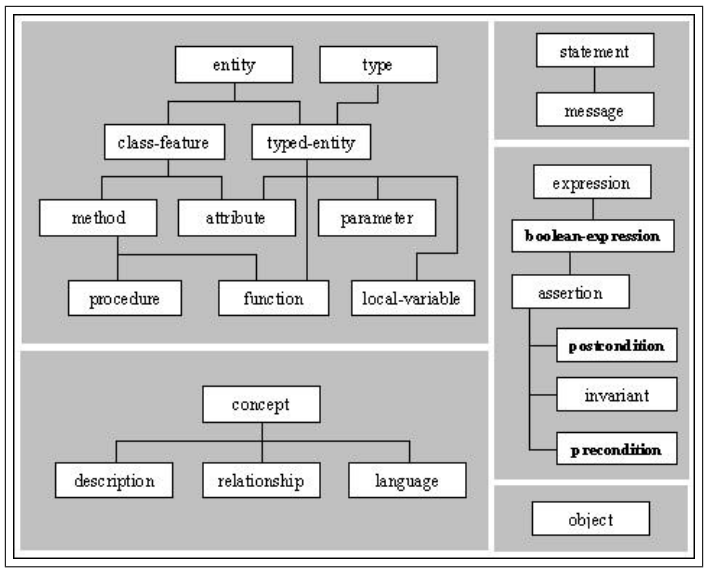

Figure 3.3: The OFL Atoms

runtime, the application objects modeled by OFL-Data are instances of the OFL-Instances. Indeed they represents descriptions.

OFL-Instances Each description or relationship described by the programmer is modelled by an OFL-Instance. The OFL Instances are analogues to a class written by a programmer in an object-oriented language. The OFL Instances representing relationships describe information necessary for relationship customization.

OFL-Data In the application, each description instance is modelled at runtime by an OFL-Data entity. However, OFL-Data entities are not customizable. Indeed, they are not instances of OFL-Components because the behaviors of such objects are not customizable according to the OFL.

3.3 Programmer and Meta-programmer: separation of tasks

In order to avoid confusions the OFL make a clear distiction between the programmer and the meta-programmer tasks, which terefore are clear separated. The programmer has to describe the application model and she/he has to write the implementation code. Indeed, hers/his work will be on the application level. For model specification she/he has to use a modelling language designed

to provide OFL features. For code implementation she/he can use different object oriented languages supported by the OFL implementation. To avoid confusion, the syntax is keept as in the original language. However, the semantics of model reified in application types will follow the meta-programmer OFL definitions. Usually this means much more constrains in accessing classes features. Teherfore, the meta-programming task is localized at the OFL Components level. Her/his work consists in three different tasks. The first task consists in creating components that wrap over the language entities. These are descriptions and relationships. The second task consists in implementing new components by changing parameters values. The last one implies much more work in both defining parameters and changing the action code for new components. The programmer will use those components to create the application model.

3.4 The Integration in the Existing Meta-Models

As presented before, the OFL is a meta-model that describes object-oriented languages based on classes. Its main goal is to customize the operational semantics targeted language descriptions and relationships. As mentioned in the second section of the thesis, the state of the art in the field of meta-model reveals diversity. Examples are Reflective Java, Dalang, metaXa, OpenC++, Iguana etc. These meta-models are usually able to describe one another. From a general point of view, OFL is close to OpenC++ by its customization model expressiveness. It is also comparable with Iguana by several details as the customization at the meta level and the encapsulation of semantics. Here we can compare the OFL concept of language with the concept of protocol in Iguana. Generally, OFL is original since its main strength is the language independence.

For OFL, the most significant player is represented by the MOF (Meta Object Facility) [Obj01]. However, the OFL do not has the first aim to compete against MOF because the lower abstractization. It target in fact less general model closer to the programmer. Indeed, MOF is based on 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. An iportant aspect that has to be well undertud is that OFL and MOF have both the same approach concerning the method bodies. Indeed, in both cases the body has to be written using the targeted language. Moreover, both OFL and MOF starts their artefacts from the OMG UML and IDL 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 using attributes and operations of the MOF classes. The MOF packages is porposed to group MOF classes and MOF associations. On the other hand, OFL may be described according to MOF and supply the latter with an additional layer on top of it. This can be done in order to customize the operational semantics of the MOF classes and associations. The OFL can

also be described thanks to XML [W3C00] and XML-Schemas.

Finally, we can consider relationships between OFL and Design Patterns. Design Patterns technology takes in account a lot of aspects controlled by OFL model. Using OFL parameters, meta-programmer could control the granularity of classes, could make distinction between different relationships like inheritance and sub-typing and could implement several types of use relationships like delegation or aggregation. Inportant here is the expresiveness reveald be thw fact that OFL make a clear distinction between association and aggregation. Because OFL model has OFL-Data wntities involved in run-time relationships, it can manage compile-time and run-time structures in an explicit manner. The OFL model can be used to simplify usage of design patterns or even to apply some of patterns automatically. Other way is to integrate directly some patterns into the OFL application model in order to support programmer to choose the best model for its application or to do automatic transformations to application model.

Chapter 4

Extending the OFL Model Through OFL-Modifiers

OFL model provides a customization of main aspects of the semantics of a language through actions and parameters, but the customization provided can deal only with features than are enough general for being applicable to most existing object oriented programming languages. Practical experience points out the necessity to capture more of the semantics of these languages. To achieve that it is necessary to add new elements to the original OFL Model [Cre01b, CL02a].

In order to preserve simplicity, a large part of the language reification is not customizable in the OFL Model philosophy. However, in order to achieve acceptance in programmers' community, some other customizations are needed. Generally, this additional semantics is handled by keywords (modifiers) in existing languages.

One main goal of introducing modifiers is to limit the number of components within an OFL-*language*. Using modifiers we avoid necessity to define one different component for any different combination of parameters. For instance, is better not to have both *public java-class* and *package java-class* components differentiated by a parameter *visibility*. Instead, we can imagine just one java-class component and something else (like modifiers) allowing ensuring that access is *public*.

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

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

Comparing with other approaches found in [ACL03, Sch02, BR01], we focus on a generic technique independent from languages. Also, instead to define a formalism which depicts access control mechanisms, we propose an approach that describes how to implement these mechanisms at a meta-programming level.

Following those goals we pay a special attention to not change the general aspect of the OFL model.

Considering these issues we propose to add at the level of language components the ability to define different kinds of *modifiers* and to add reification elements according to that.

OFL *modifiers* are used together with other language entities in order to change protection or other semantic aspects of them. Some of them have an equivalent in keywords that may be found in some programming languages, others could be added in order to simplify programming task.

4.1 The OCL Language

Starting from the point that most of the OFL *modifiers* relay on constraints [Pes03] to be applied to the program entities, we choose OCL as the language for specifying these constraints. OCL [CW02, WK98] is a formal language which allows to express side effect-free constraints. The Object Management Group (OMG) defines OCL (Object Constraint Language) [OMG00] as a part of UML 1.3 standard specification. Main motivation regarding that choice is programming language independence of OCL and general acceptance of this language.

OCL is designed to express side effect-free constraints. It was used by OMG in the *UML Semantics* document [Sof97] to specify the rules of the UML metamodel. Each rule in the static semantics sections in the UML Semantics document contains an OCL expression, which is an invariant for the involved class.

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

OCL has been developed to fill this gap. It is a formal language that remains easy to read and write. It has been developed as a business modelling language within the IBM Insurance division, and has its roots in the Syntropy method [CD94].

OCL is a pure expression language [Gri99]. Therefore, an OCL expression is guaranteed to be without side effect; it cannot change anything in the model. This means that the state of the system will never change because of an OCL ex-

pression, even though an OCL expression can be used to specify a state change, e.g. in a post-condition. All values for all objects, including all links, will not change. Whenever an OCL expression is evaluated, it simply delivers a value.

OCL is not a programming language, so it is not possible to write program logic or flow control in OCL.

OCL is a typed language, so each OCL expression has a type. In a correct OCL expression all types used must be type conformant.

OCL can be used for a number of different purposes:

- to specify invariant on classes and types in a class model
- to specify type invariant for UML Stereotypes
- to describe pre- and post conditions on Operations and Methods
- to describe Guards
- as a navigation language
- to specify constraints on operations

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

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

In OCL, a number of basic types are predefined and available to the modeler at all time: Boolean, Integer, Real, String and Enumeration. It is also defined a number of operations on these predefined types.

In addition, all descriptions coming from the OFL Model are types in OCL that is attached to the model.

The type Collection, which is predefined in OCL, plays an important role in writing constraints. It includes a large number of predefined operations to enable the OCL expression author (the modeler) to manipulate collections. Consistent with the definition of OCL as an expression language, collection operations never change collections. They may result in a collection, but rather than changing the original collection they project the result into a new one.

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

OCL defines a number of operators for collection manipulation:

- SELECT and REJECT - allows to specify a selection from a specific collection

- COLLECT - allows to specify a collection which is derived from some other collection, but which contains different objects from the original collection (i.e. it is not a sub-collection)
- FORALL - allows to specify a Boolean expression, which must hold for all objects in a collection
- EXISTS - allows to specify a Boolean expression which must hold for at least one object in a collection
- ITERATE - allows building one accumulation value by iterating over a collection. It is a very generic. The operations Reject, Select, forAll, Exists and Collect can all be described in terms of Iterate

4.2 The OFL Modifiers

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

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

Not all language modifiers are intended to be reified by OFL *modifiers*. Semantics changes induced by some of them are very deep and relay in different OFL components. We name them *component modifiers*. Following list presents situation for three well known object-oriented languages: Java [GJSB00, Fla99b, LYJW96], C++ [Str97, Lip99, Str94] and Eiffel [Mey02, Mey91].

4.2.1 Component Modifiers in Commercial Languages

Java language.

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

final {attribute declaration} A *final* attribute may only be assigned to once. Once a *final* attribute has been assigned, it always contains the same value. To model this kind of attribute in OFL we use an *OFL-AtomAttribute* that has property *isConstant* set to *true*.

static {feature declaration} If a feature (attribute or method) is declared *static*, there exists exactly one incarnation of the feature, no matter how

many instances (possibly zero) of the class may eventually be created. A static attribute, sometimes called a class variable, is incarnated when the class is initialized. A static method, called as class method, is always invoked without reference to a particular object. The *OFL-AtomAttribute* and *OFL-AtomMethod* that reifies these entities has the *isDescriptionFeature* property set to *false*.

C++ language.

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

Eiffel language.

expanded {class declaration} Declaring a class as *expanded* indicates that entities declared of the corresponding type will have objects as their run-time values. (By default, values are references to objects.). These classes will be reified by description components corresponding to *expanding class* and *generic expanding class*. Those components could not be target for client aggregation relationship or generically derivation. Instead, they could be target only for inheritance, expanded client relationship and expanded generically derivation.

Figure 4.1 illustrates the OFL model extended with *OFL-Modifiers*. We define three kind of modifiers for entities which support their semantics. These types are: *description-modifier*, *method-modifier* and *attribute-modifier*. The OFL *modifiers components* inherit from OFL-*modifiers* and represent reification of language modifiers.

4.2.2 Definition of an OFL-Modifier

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

Modifier Name. The name is used to identify the modifier. It should be a legal identifier related with OFL and the language binding.

Modifier Context. Type of entity that accepts the modifier is denoted by its context. Context could be description, relationship, attribute or method.

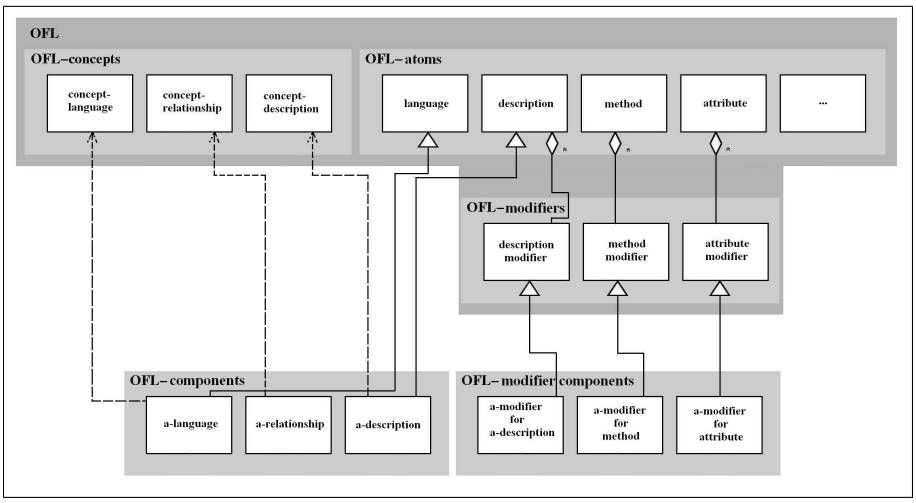

Figure 4.1: The extension of the OFL model through OFL-Modifiers

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

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

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

Another solution could be to define the assertion within the *OFL-Modifier* itself but the drawback is that one modifier has to know about other modifiers and this decrease its reuse capabilities.

Considering that, the role of an *OFL-Modifier* is to help meta-programmer to manage and organize assertions.

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

The OCL modifier assertions are written in context of the OFL model definition; as a result of that, all types defined by the OFL model could be used in assertions.

Some component features correspond to OCL collection type and support OCL collection operators. For instance,

component.modifiers $\rightarrow$ *includes('modifier name')*

that tests if the component has modifier 'modifier name' attached to it or not.

Modifier's Actions. Modifier's actions are OFL-*Actions* rewritten to consider new semantics. The modifier keeps references to all rewritten action, helping meta-programmer to manage them. Actions play different roles depending of the complexity of the considered modifier. Most modifiers do not need action rewriting. They have just a set of assertions attached to them.

In order to build a complex semantics from simpler ones and to extend modifiers, we define a modifier composition operator. This operator specifies how to combine assertions and actions that belongs to composed modifiers. In the context of composition operation we state the definition of "compatible modifiers" and "incompatible modifiers". Two modifiers defined in the same context are compatible if they can be the parts of a composition. They are incompatible if their actions and assertions are not disjunctive. Actions and assertions are not disjunctive if their semantics interfere. According to that we extent the definition of OFL-*Modifier* by adding a characteristic named *incompatible modifier set*. One modifier keeps in this set information about all modifiers that are incompatible with it.

In the composition process, two aspects of modifiers are addressed: the assertions and the actions associated with it. For compatible modifiers all interactions will be just cumulative. For the assertions, which are OCL expressions, other constraints can be composed using the AND logical operator. Because OCL avoids side effects, composition of assertions is commutative. Actions may be called in a random order. Indeed, if there are some interactions at the level of action semantics, the modifiers are incompatible and the composition operator cannot be applied.

To deal with incompatible modifiers we define an invariant at the level of OFL entity representing the modifier context.

Following example consider the Java *public* modifier for attributes. For better understanding we consider a 'package' modifier replacing all default visibility for attributes. The OFL reification for an attribute is the *OFL-AtomAttribute*. When define access control modifiers for Java attributes, we attach an invariant to this entity.
incompatible modifiers set for *public* is {*protected*, *private*, *package*}

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

In order to cover all situations an invariant should be added for each modifier considered.

In the context of a language extension made by a meta-programmer we can distinguish two kind of modifiers. An OFL-*modifier* could represent the reification of a modifier that belongs to the language binding - we name it *native modifier* - or could be a *custom modifier* added by the meta-programmer in order to enrich language semantic.

The native modifiers will have the same meaning, related with the language binding components, like in the original language. The meta-programming task will consist in describing the meaning and the behavior of modifiers according with their definition. When a meta-programmer adds new extension for the language (new components) he has the responsibility to extend the definition of the modifiers according to the new entities.

In the following sections we try to provide an orthogonal approach in order to define both native and custom modifiers.

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

4.2.3 Modifiers Classification Regarding OFL Implementation Issues

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

"When you are using visibility, use the rules of the language in which you are working. When you are looking at UML model from elsewhere, be wary of the meaning of visibility markers, and be aware how those meanings can change from language to language."

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

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

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

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

Optimization Modifiers These modifiers have no impact at the level of application model semantics. They are used only to establish optimization strategies for compilers or, more generally, translators (ex. inline, volatile, register etc.)

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

Additional Modifiers In addition to previous considered modifiers languages has also other keywords used to change semantics in a not customizable manner in OFL. The meaning of these additional modifier is to force compiler to treat in a special way the entity that declare the modifier. This category does not include modifiers that change the reification component for considered entity (this subject was discussed in sec. 4.2). The modified semantics is handled by the native compiler (ex. explicit, agent etc.).

4.3 Basic Access Control Modifiers

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

4.3.1 Examples of Native Basic Access Control Modifiers

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

Java class members (attributes and methods) that are declared *public* can be accessed anywhere that the class in which they are declared can be accessed.

Members that are declared as *protected* can be accessed within the package in which they are declared and in subclasses of the class in which they are declared.

Members that are declared as *private* are only accessible in the class in which they are defined and not in any of its subclasses.

Class members that have no access control modifier associated is considered to have default visibility. These members can be accessed only from within the package in which they are declared.

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

To achieve symmetry on defining modifiers we augmented the *default* Java visibility for both class and members with an implicit *package* modifier.

C++ Language. For C++ language [Str97] the *public*, *protected* and *private* modifiers has slightly different meaning as in Java [Ard02]. It has no "package" resolution but has instead a special class of visibility denoted by *friend*.

Using the *friend* keyword, a class can grant access to non-member functions or to another class. These friend functions and friend classes are permitted to access private and protected class members. The *public* and *protected* keywords do not apply to friend functions, as the class has no control over the scope of friends.

If a member of a C++ class is *private*, its name can be used only by member functions and friends of the class in which it is declared.

A *protected* member can be used only by member functions and friends of the class in which it is declared and by member functions and friends of classes derived from this class.

A *public* member can be used by any function.

The default access for C++ class members is *private*.

These modifiers could be used to change access control through inheritance between classes.

When preceding the name of a base class, the *public* keyword specifies that the public and protected members of the base class are public and protected members, respectively, of the derived class.

The *protected* keyword use for inheritance specifies that the public and protected members of the base class are protected members of its derived classes.

Finally, when preceding the name of a base class, the *private* keyword specifies that the *public* and *protected* members of the base class are *private* members of the derived class.

Eiffel Language. In Eiffel [Mey02] there are two constructions that can deal with access modifiers; these are *feature* and *export*. In this language some of the protection semantics are hidden in the language philosophy. For instance, the writing protection has no direct meaning for an attribute because access to an attribute from outside class is considered as a query (and it is not possible to write *into* a result of a query).

4.3.2 Basic Access Control Modifiers for Features

Modifier Assertions. The assertions of basic access control modifiers for features (attributes and methods) are defined at the level of *OFL-Relationship*

components that manage export of those features. They should be tested each time a relationship involving that feature is created. An invariant at the level of description that own the feature is not necessary. Basic modifiers do not protect features against the description itself. Independently of the language syntax we can consider three possibilities: the feature belongs to current class or it is inherited through an inheritance relationship from a direct or indirect ancestor or it is accessed through an use relationship (current class is a client of description that owns the feature). In the last situation we consider that the current description could access supplier description. Indeed, this problem is covered by description's access control. By current class we mean the class that accesses the feature.

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

The following example presents invariants for *extends* Java inter-class relationship and Java *aggregation* relationship.
Java features basic modifiers: {*public, protected, private, package*}

```
context ComponentJavaClassExtends
inv: self.showedFeatures->forall(f:Feature |
    f.modifiers->includes('public')
        OR
    f.modifiers->include('protected'))
inv: self.redefinedFeatures->forall(f:Feature |
    f.modifiers->includes('public')
        OR
    f.modifiers->include('protected'))
inv: self.hiddenFeatures->forall(f:Feature |
    f.modifiers->includes('private'))
```

The invariant says that all *showed* and *redefined* features through an *extend relationship* should have modifiers *public* or *protected* attached. All *hidden* features have *private* modifier.

```
context ComponentJavaAggregation
inv: self.showedFeatures->forall(f:Feature |
    f.modifiers->includes('public')
    OR
    (( f.modifiers->include('package') OR
               f.modifiers->include('protected'))
       AND
       self.source.package = self.target.package)))
inv: self.hiddenFeatures->forall(f:Feature |
    f.modifiers->includes('private')
```

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

In addition to previous assertion, this one tests also information about description's packages. In this assertion the descriptions are accessed through *source*[1] and *target*[2] members of the relationship component instance (*self*).

All these modifiers are incompatible. For methods, the incompatible modifiers set contains also the *abstract* modifier.

Modifier Actions Interference with model actions is minimal. Assertions are added to control features access through relationships and no action rewriting is necessary. Indeed, modifiers for basic access control generally do not redefine any actions.

As an exception we can consider *protected* modifier for Java features. Action is needed in this case to express a particular semantic presented in Figure 4.2. Method m of class C have access to protected member f of B. This happens because class A, which declare the member f, and class C belongs to the same package. To express this semantics we need to rewrite the lookup action for features. This action has to ensure access to protected members for any feature that is declared by an ancestor belonging to same package with the class that access the feature.

4.3.3 Basic Access Control Modifiers for Descriptions

Modifier Assertions. The assertions of basic access control modifiers for descriptions are defined at the level of relationship components and at the level of description component itself. They should be tested each time a relationship involving that description is created and each time an instance of description is created. The last situation deals with relationships that enable polymorphism. According to these assumptions, the assertion associated to such modifier should become a post-condition for the *look-up* OFL action.

The following example refers the Java language semantics for class access control. Please note that this example does not consider interfaces, abstract classes and inner classes.
Java class modifiers: { *public*, *package*}

```
context ComponentJavaClassExtends
inv: self.source.package = self.target.package
     OR
     ( self.source.package <> self.target.package
```

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

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

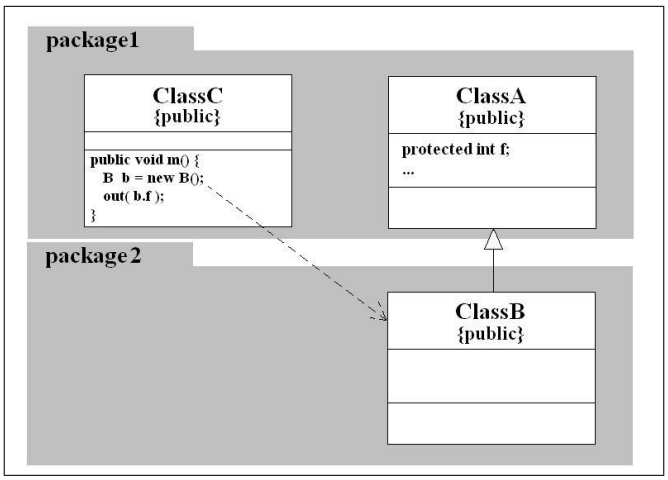

Figure 4.2: Java *protected* modifier semantics

```
      implies
    self.source.modifiers->includes('public'))
```

A class can extend another class from the same package and a class can extend a *public* class from other package.

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

The following assertion address dependencies between classes, which are not covered by OFL customization.

```
context Description::
    lookup(accessed: Description):Description
post: self.package = result.package
    OR
      self.package <> result.package
    implies
        result.modifiers->includes('public')
```

Next we consider the Java language semantics for interfaces access control. The example does not consider inner interfaces.
Java interface modifiers: { *public*, *package*}

```
context ComponentJavaInterfaceExtends
```

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

An interface can extend another interface from the same package and an interface can extend a *public* interface from other package.

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

A class can implements an interface from the same package and a class can implements a *public* interface from other package.

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

A class can declare an attribute of a type of an interface from the same package and of a type of a *public* interface from other package.

To handle dependencies between classes and interfaces we use the same post-condition for *lookup* action previous defined for class modifiers.

Modifier Actions For those modifiers, assertions are also added to control features access through relationships. Post-conditions are used to filter the *lookup* action result. Modifiers do not redefine any actions.

4.4 Complex Access Control Modifiers

Complex access control modifiers define protection at the level of special rights like writing / reading an attribute, calling / redefining a method or extending / instantiating a description.

4.4.1 Examples of Native Complex Access Control Modifiers

Java Language. Java language does not include complex access control modifiers for attributes. It includes *final* modifier for methods and classes and interfaces.

Modifier *final* associated to a method disallow redefinition.

A modifier with same name in context of classes and interfaces is used to avoid extension.

Other language mechanisms (like making all constructors *private*) could be used to control instantiation of classes.

C++ Language. C++ does not provide any specific modifiers to control rights for using an entity.

Changing access rights to constructor does also control at the level of class instantiation like in Java.

Eiffel Language. *Frozen* and *deferred* modifiers from Eiffel could be considered in this category.

Frozen, appearing before a feature name express that the declaration is not subject to redefinition in descendants.

Deferred modifier permits declaration of a feature without an implementation. This transfers to proper descendants the responsibility for providing an implementation through a new declaration, called an "effecting" of the feature.

4.4.2 Complex Access Control Modifiers for Methods

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

Modifier Assertions. Implementation of control implies assertions at the level of OFL entities reifying corresponding mechanisms. Redefinition mechanism is reified in OFL by *redefinedFeatures* characteristic of relationship components. Access control is done by invariant for these components. Calling mechanism is reified in *execute* action. Assertion concerning calling rights is implemented in a post-condition for this action.

The following example is an implementation of *final* modifier for Java methods.

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

Final modifier is compatible with *public*, *protected*, *package* and *private* modifiers and can be present in a composition to them. Its invariant will be added to the component invariant.

Modifier Actions. Complex access control modifiers for methods require some times rewriting of the *execute* OFL action.

4.4.3 Complex Access Control Modifiers for Attributes

Rights concerning attribute usage address control against reading or writing. Protection on writing is achieved by a pre-condition at the level of assign action. We can consider here a proposal of Cook and Rumpe [CKMR99] for defining a read-only modifier for attributes. They conclude that is useful to constraint the visibility of an attribute to be readable, but not changeable. The concept of a read-only-modifier is introduced in combination with private and protected modifiers.

Modifier Assertions Assertions for attribute complex modifiers resides in pre and post conditions at the level of assign OFL action.

Modifier Actions Necessity for action writing resides in complexity of considered semantic.

As an example we consider a modifier that implements a *heavy* writing protection for an attribute. By heavy protection we mean to protect not only the reference of the object against writing but also the internal state of the referred object.

A solution that lacks in efficiency is to give access to a clone of the object that contains attribute and to look after that if any changes appear. To ensure this control, attribute access action should be embedded in the following code:

```
// cloning the original object
    aux = deep_clone(f)
// original action
// ( any kind of action that may imply changing
// of attribute's internal state )
    *action(aux)
// test if the object preserve same state
    if (not deep_compare(f, aux) )
        generate_error("Could not write attribute")
    end_if
    destroy_object(aux)
```

Actions that permit changing of attribute's internal state are considered the following OFL-*actions*: *evaluate-parameters*, *attach-parameters*, *detach- parameters*, *assign*, *execute* etc.

4.4.4 Complex Access Control Modifiers for Descriptions

Description may be extended, used or instantiated.

Modifier Assertions Extension is controlled through invariant on inheritance relationship components. To control client-supplier relationship, invariant is attached to use relationship components.

As am example we consider the Java *final* modifier in context of a description. The invariant for Java extends relationship will check absence of this modifier at the level of target description of relationship.

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

Modifier Actions For description modifiers, actions are necessary to control instantiation. Instead, most of the times a precondition at the level of *create-instance* action is enough to ensure all semantics.

4.5 Optimization Modifiers

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

4.5.1 Examples of Native Optimization Modifiers

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

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

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

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

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

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

The *inline* modifier for a member function is a hint for the compiler that is should attempt to generate code far a call od function inline rather through the usual function call mechanisms.

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

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

Eiffel Language. Analyzing Eiffel we find also optimization modifiers. *Indexing* and *obsolete* modifiers for a class could be considered in this category .

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

The *obsolete* clause in a class indicates that the class does not meet current standards. The advice for developers is against continuing to use it as supplier or parent but without to harm existing systems which rely on this class. Declaring a class as Obsolete does not affect its semantics. Instead, some language processing tools may produce a warning when they process a class that relies, as client or descendant, on an obsolete class.

4.5.2 Optimization Modifiers for Attributes

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

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

If we consider Java modifiers, *volatile* is incompatible with *final*. Because *final* keyword has no reification in OFL (4.2) the assertion have to ensure that the propriety *isConstant* is set to *false*.

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

```
        self.isConstant = false
```

Modifier Actions In case of using an OFL translator to native code, actions for these modifiers have just to copy them to the final translated code.

In case of an OFL compiler, it could consider directly those modifiers to make optimizations. Another possibility is to ignore these modifiers if that optimizations are not compulsory.

4.5.3 Optimization Modifiers for Methods

Optimization modifiers for methods concerns in accelerating calling mechanism and in dealing with methods written and compiled in other languages.

Modifier Assertions Assertions for optimization modifiers concerns usage of incompatible modifiers. No other constraints are necessary.

In the case of *native* modifier in Java, it is incompatible with *synchronized* modifier. Also, a constructor method could not be declared as *native*. The lack of a possible native constructors is an arbitrary language design choice that makes it difficult for an implementation of the virtual machine to verify that superclass constructors are always properly invoked during object creation.

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

Modifier Actions These modifiers needs same kind of actions as optimization modifiers for attributes. In case of designing of an OFL compiler for the OFL language reification, attention must be payed to make a correct linking with outside code.

4.5.4 Optimization Modifiers for Description

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

Modifier Assertions No assertion are needed.

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

4.6 Service Modifiers

4.6.1 Examples of Native Service Modifiers

Java Language. Java has three modifiers that could be part of this classification. These are *synchronized* for methods and *transient* for attributes.

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

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

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

Eiffel Language. Eiffel also does not include any service modifier.

4.6.2 Service Modifiers for Attributes

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

Modifier Assertions Most of the assertions for these modifiers deal just with incompatible modifiers. A particular situation result because OFL does not provide customization at the level of attributes. To cover this situation, modifier assertion has to test if usage of the considered service is permitted or not in context of description that declare the attribute.

Modifier Actions Service modifier actions will implement the service or will make link with components that provide considered service.

4.6.3 Service Modifiers for Methods

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

Modifier Assertions Service modifier assertions has to ensure that a particular kind of method (ex: a constructor or a destructor) support or not targeted service. Similarly to attributes, OFL does not provide customization at the level of methods. Because all methods have same kind of reification, as *OFL-AtomAttribute* instance, information regarding them are characteristics at the level of those instances.

Additionally, incompatible modifiers have to be considered.

Modifier Actions Service modifier actions will implement the considered service. Most of those actions will be dynamic actions injected at compiling time.

4.6.4 Service Modifiers for Descriptions

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

Modifier Assertions Assertion will have to ensure that all relationships that involve the current description are compatible with the service provided. If we consider persistency, a composition relationship could imply that target of relationship should be also persistent if the source is persistent. In other words, assertions have to verify that all composition parts could be made persistent.

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

4.7 Additional Modifiers

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

4.7.1 Examples of Native Additional Modifiers

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

C++ Language. In this category, C++ has modifiers like *const* for methods and *explicit* for constructors (that are also a kind of method).

The *const* modifier used for a method indicated that the method do not modify the state of an object.

In C++, *explicit* constructors will be invoked only explicitly. That disallows implicit conversions.

Eiffel Language. Eiffel contains *agent* keyword that modify the semantics of a method parameter.

The keyword *agent* is used to transmit a routine as a parameter for other routine. It avoids confusion with an actual routine call when transmit parameter. Indeed, when transmit the parameter, the routine is not called yet. Instead, the routine is pass to calling routine as an *agent*.

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

Modifier Actions We can assume that all modifiers from this category involve hard action writing. Each of them address a very specific sematic. Metaprogrammer has to identify first what OFL actions are involved in expressing considered semantics.

As example, if we consider the *explicit* native C++ modifier, semantics are expressed at the level of *before-create-instance* and *create-instance* OFL actions.

4.8 Conclusion and discussions

In this paper we proposed to extend the OFL Model. The main goal of this extension was to add customization of the access control mechanism and of additional non-covered semantics. We introduced the notion of OFL modifier to provide a clean way for control implementation. For better understanding of the concept we present in sections 4 and 5 examples of several native modifiers reification.

As future work we proposed to add support for OFL modifiers and to integrate them in all OFL tools. We also plan to extend the modifiers with high level actions. The OFL modeling tool will execute these actions to ensure automatic model correction.

Chapter 5

The OFL-ML Meta-Profile

The specification for an OFL modeling language (OFL-ML) [PCL03b] set out the necessity to provide a standard way to express the semantics of an OFL-*language* application using UML-like notation and thus to support OFL applications modelling with standard UML tools. The term OFL-*language* means a language reified or expressed in OFL (ex: OFL-Java, OFL-C++, OFL-myJavaExtension etc.).

We define an OFL-ML *Profile* as an UML Profile that is generated automatically and customized for every language expressed in OFL. Indeed, each existing language reified in OFL or a possible extended language expressed in OFL need their own associated OFL-ML *Profile*.

Our goal is to design a meta-model which allow us to generate OFL-ML *Profiles*. We name this meta-model as *OFL-ML meta-profile*.

Considering that, the OFL-ML will be a meta-profile for each possible UML *Profile* designed for a programming language. Indeed, each **instance** of OFL-ML in context of a particular OFL-*language* is an UML *Profile* for that language. We name this profile "OFL-ML-Profile for OFL-language". This way will exists "OFL-ML Profile for OFL-Java", "OFL-ML Profile for OFL-myExtendedJava" or "OFL-ML Profile for OFL-C++" and so on.

In a simplified way, OFL-ML could be considered as a kind of *Profile-Template*. To obtain a specific UML Profile for an OFL-language, OFL-ML has to be instantiated using OFL meta-information as *components*, *parameters*, *characteristics* and *modifiers*.

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

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

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

An OFL-ML profile are planed to be used with standard UML modeling tools or with new modeling tools special designed for it. It could be used to test and validate the model, to apply design patterns in automatic way, to collect metrics or to generate XML representation of OFL-code. The OFL information contained by OFL-ML entities represent a real help to achieve all these goals. It is obvious that in the last case, all this information will fill the XML representation of application elements.

5.1 Supported Elements and Definitions

5.1.1 OFL Model

Specification of OFL-ML meta-profile is based on OFL model definition found in [Cre01b] extended with OFL Modifiers [PL03, PCL03a]. The OFL elements modelled by OFL-ML meta-profile are:

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

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

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

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

5.1.2 OFL-Modifiers

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

5.1.3 UML Profile

The notion of the UML profile appeared in the UML 1.3 standard as a means of structuring UML extensions (tagged values, stereotypes and constraints). UML is a modelling language used in a large number of application domains and all types of software applications. However, each domain has specific notions and particular needs, which are handled by UML through extensions which are grouped into *UML Profiles*.

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

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

5.1.4 OCL

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

For ModelElement.

[**1**] The operation *allStereotypes* results in a Set containing the ModelElements Stereotype and all Stereotypes inherited by that Stereotype (as opposed to all Stereotypes inherited by the ModelElement).

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

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

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

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

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

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

For Classifier

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

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

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

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

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

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

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

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

5.2 OFL-ML Definition

5.2.1 Identified Subset of UML

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

The OFL-ML extends the following standard UML packages: Core and Model Management. Figure 5.1 shows the model elements that form the structural backbone of the meta-model and figure 5.2 shows the model elements that define relationships. The abstract syntax for the Model Management package is expressed in graphic notation in Figure 5.3.

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

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

5.2.2 From Core - Backbone

The backbone of the core package is shown in fig. 5.1.

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

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

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

Comment A comment is an annotation attached to a model element or a set of model elements. It has no semantic force but may contain information useful to the modeler.

Constraint A constraint is a semantic condition or restriction expressed in text.

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

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

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

Namespace A namespace is a part of a model that contains a set of ModelElements each of whose names designates an unique element within the namespace.

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

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

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

From Core - Relationships The UML relationships described in the core package are presented in fig. 5.2.

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

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

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

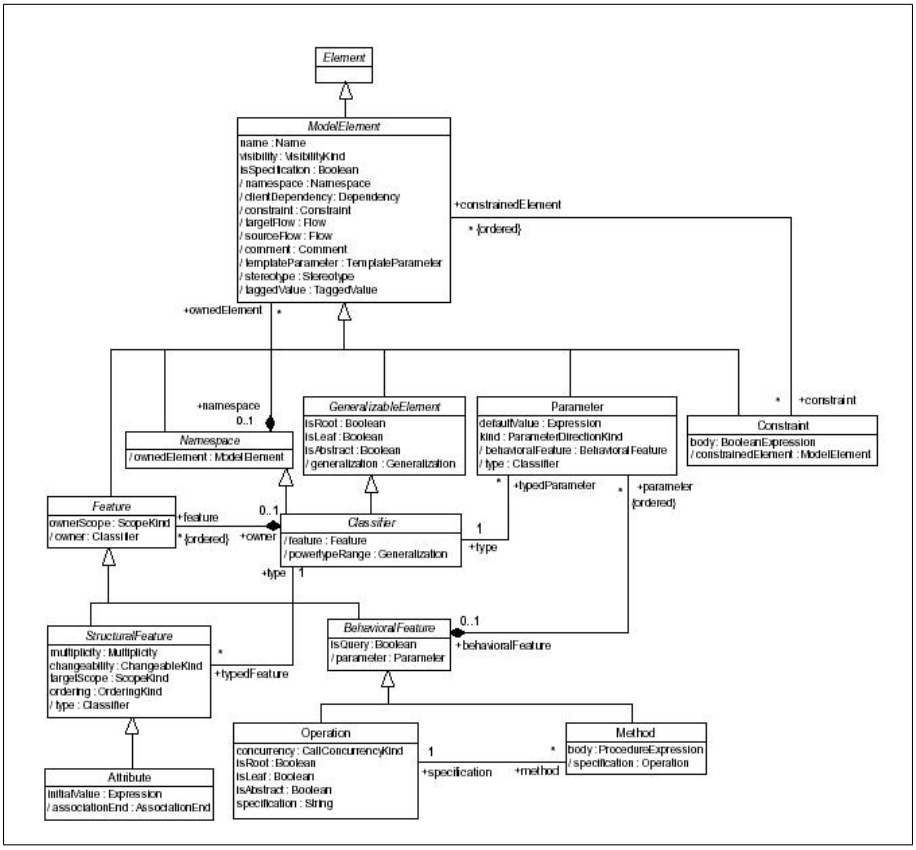

Figure 5.1: The UML Core Package - Backbone

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

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

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

From Model Management The main elements of the model management package are shown in fig. 5.3.

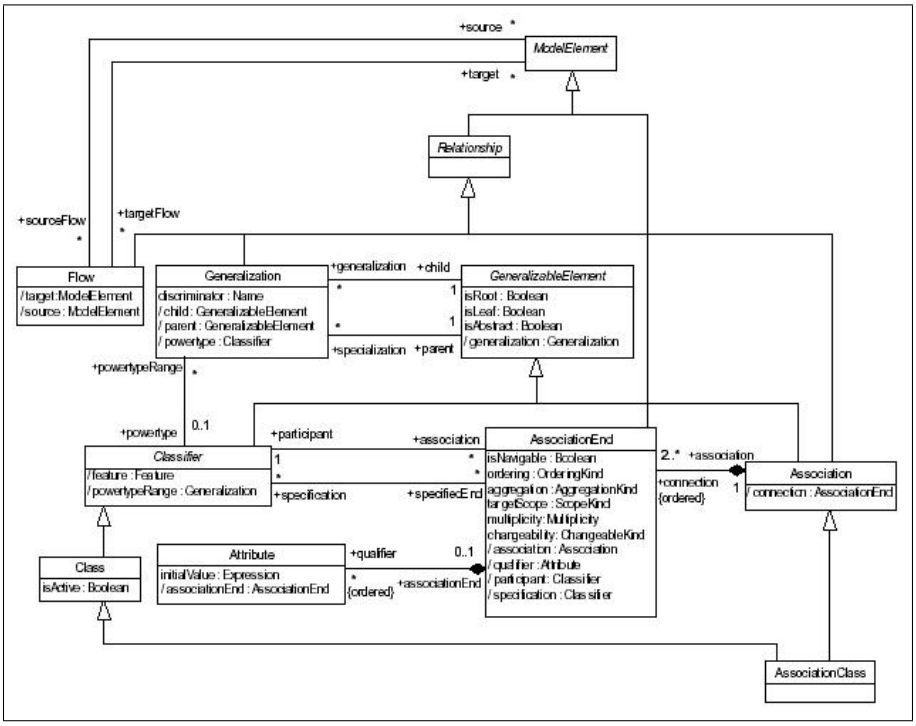

Figure 5.2: The UML Core Package - Relationships

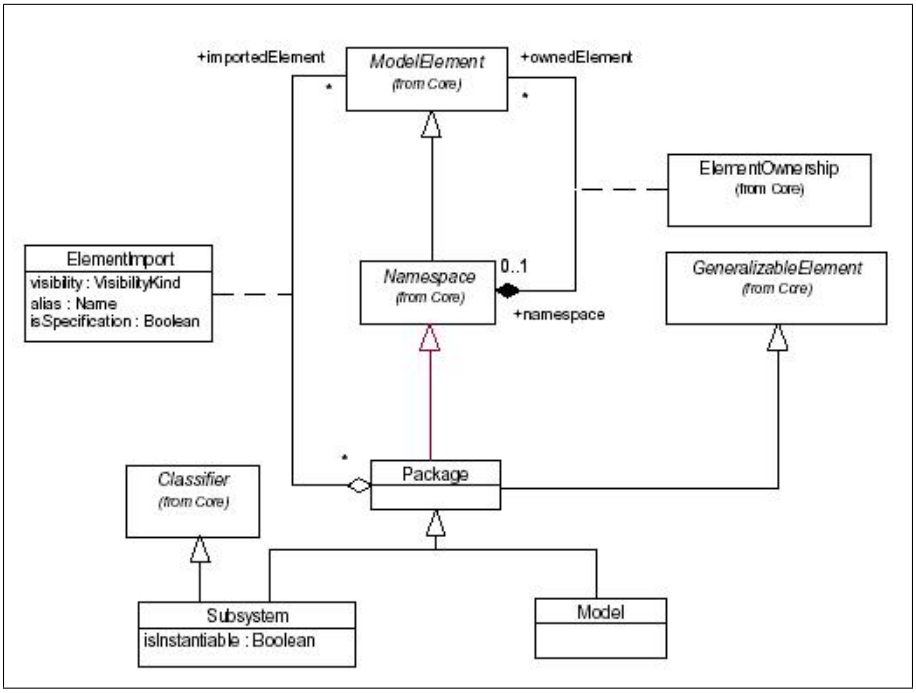

Figure 5.3: The UML Model Management Package

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

Package A package is a grouping of model elements.

5.2.3 The Virtual Meta-model

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

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

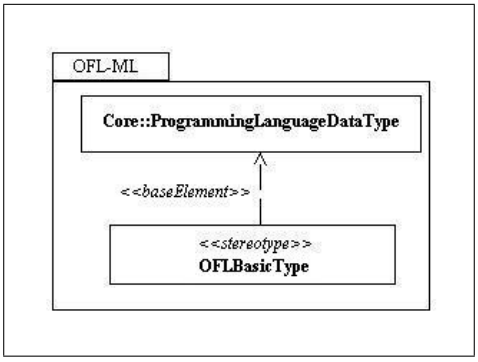

Figure 5.4: Virtual Model for OFL Basic Types

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

A big challenge for OFL-ML is to generate a clean and understandable profile in an automatically way. To following rules are designed to help this aspect:

- every OFL-*component* will be represented through an individual stereotype
- every combination of characteristics values of OFL non-customizable elements(reified by OFL-*atoms*) will generate a different stereotype. This rule is based on the UML stereotype definition:"*... a stereotype may be used to indicate a difference in meaning or usage between two model elements with identical structure*".
- additional OFL-elements (like OFL-*modifiers* or OFL-*assertions*) will be considered in generated tagged values or constraints of constructed profile

5.2.4 Virtual Metamodel of OFL-ML.

Figure 5.4 presents stereotype used to model the basic types defined by a language. These types are managed as a characteristic of OFL-*language* component, which is actually a list. Stereotype is derived from UML *programming language data type*.

Figure 5.5 shows stereotype used to model OFL-*description* components. This stereotype is derived from UML *class*. An UML *class* is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics.

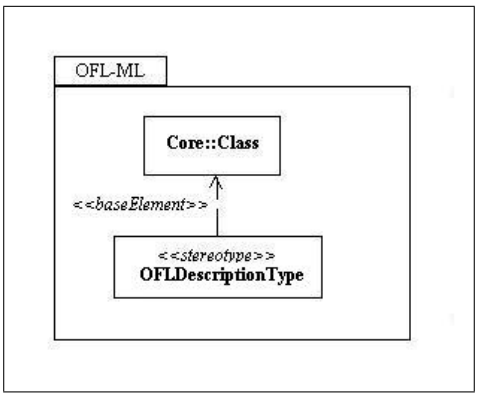

Figure 5.5: Virtual Model for OFL-description Components

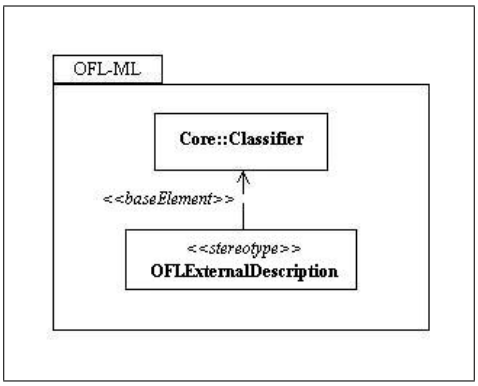

Figure 5.6: Virtual Model for External Description

Figure 5.6 present a stereotype used to model an *External Description*. This element does not exists in the OFL-model. It is defined at the level of OFL-ML and specify a *Description* that has no OFL reification. It is necessary for helping usage of class libraries that have no OFL representation. The stereotype is derived from UML *classifier*.

Figure 5.7 shows how to represent an OFL-*package*. Generated profile will contain entities that inherit from this stereotype and denote specific language class organization mechanisms. The stereotype is derived from UML *package*.

In figure 5.8 we show the stereotypes used to represent OFL-*features*. Stereotypes are derived from UML *attribute* and *method*. Also, stereotypes are specialized based on OFL-*AtomAttribute* characteristics: *isDescriptionAttribute* and *isConstant*, and OFL-*AtomMethod*: *isConstructor* and *isDestructor*.

Stereotype for association end that belongs to OFL-*UseRelationship* are presented in figure 5.9. These stereotypes follows same rules as features stereotypes.

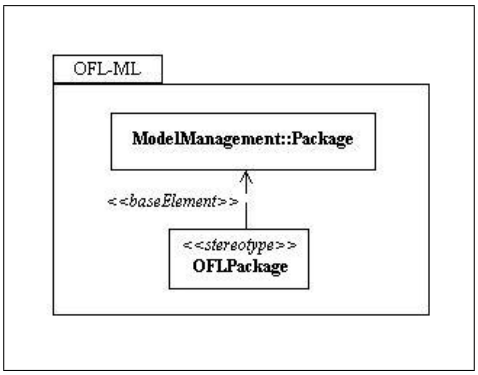

Figure 5.7: Virtual Model for OFL Package

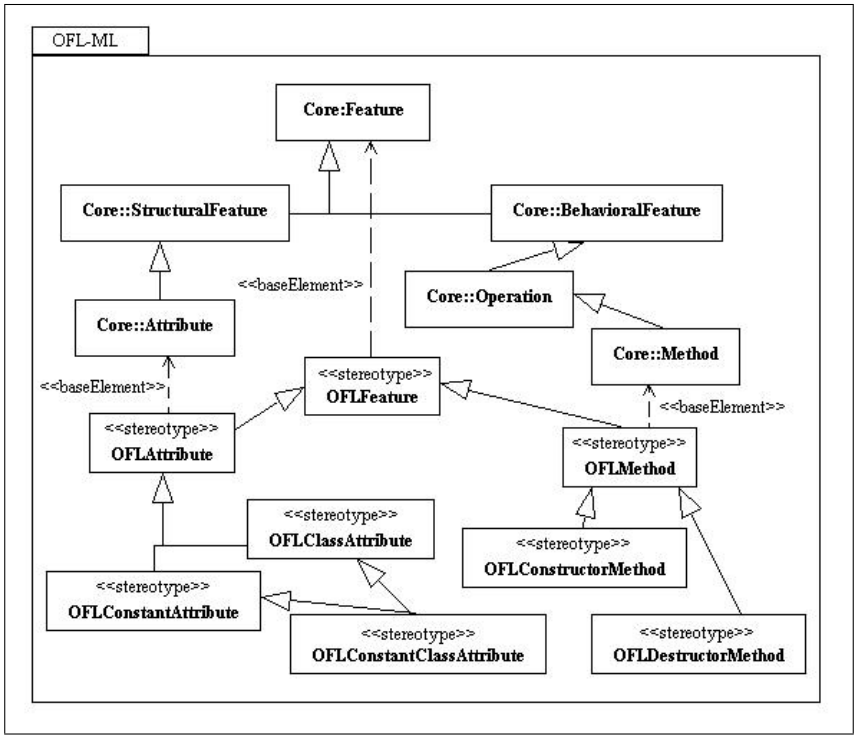

Figure 5.8: Virtual Model for OFL Features - attributes and methods

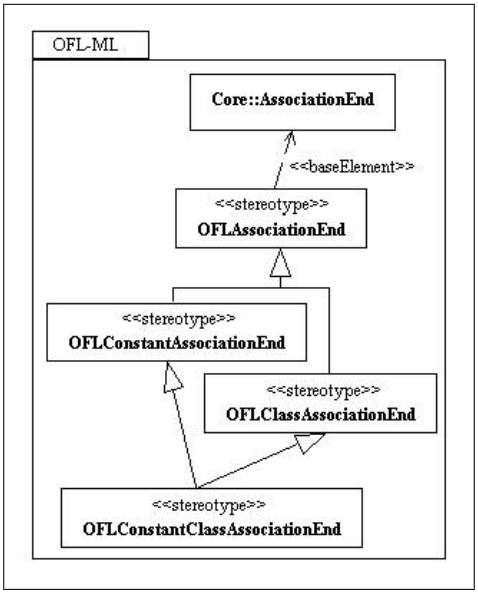

Figure 5.9: Virtual Model for Association End

Figure 5.10 presents stereotypes used to represent OFL-*relationships*. Stereotypes are derived from UML *generalization* and *association*.

5.3 The OFL Type Representations

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

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

5.3.1 The OFL BasicType Element

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

Stereotypes and Tagged Values. The OFL-ML basic types are represented by UML *ProgrammingLanguageDataType* from *Core* package with the $\ll OFLBasicType \gg$ stereotype.

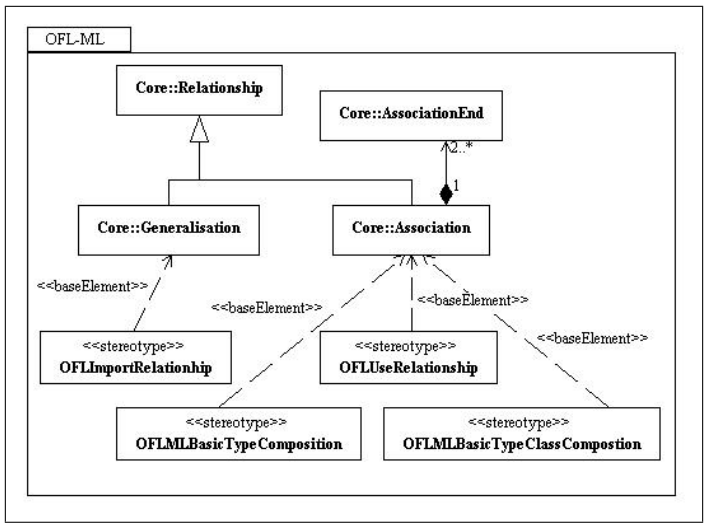

Figure 5.10: Virtual Model for OFL Relationships

Constraints. All $\ll OFLBasicType \gg$ stereotyped elements has direct correspondence in characteristic *OFL-languge.basicTypes*.

Elements Generation. A profile element stereotyped $\ll OFLBasicType \gg$ will be generated for each element of the list *OFL-language.basicTypes*. All strings contained by this list will became a name for a profile element.

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

- boolean
- char
- byte
- short
- int
- long
- float
- double

5.3.2 The OFL Description Element

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

Stereotypes and Tagged Values. The abstract stereotype $\ll$*OFLDescriptionType*$\gg$ is the base for all the concrete stereotypes representing OFL Description of the considered language. The name of the generated stereotypes are the name of the OFL components with "Component" prefix removed (ex. for a component *ComponentJavaClass*, a stereotype named $\ll$*JavaClass*$\gg$ will be created).

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

Constraints. Constraints related with components stereotypes have to consider parameter values, characteristics and associated OFL Modifiers constraints for that component. Not all OFL parameters are considered but only that one which have impact on static model of the application.

This paragraph presents constraints that have to be generated for all stereotypes derived from abstract stereotype $\ll$*OFLDescriptionType*$\gg$. Each of them will consider parameter values, characteristics and modifiers associated with corresponding OFL component. Thus all constraints related with stereotype $\ll$*JavaClass*$\gg$ consider parameter values, characteristics and modifiers associated with component *ComponentJavaClass* defined by OFL-Java.

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

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

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

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

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

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

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

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

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

Regarding modifiers assertions that deal with *OFL-Description components*, only parameter *modifier* inherited from *OFL-AtomDescription* is involved. It has to be translated into *taggedValue* with same name like the modifier. Indeed, transformations are based on the following two rules:

- *Syntax:*

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

  is translated in:

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

- *and syntax:*

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

  is translated in:

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

These constraints test presence or absence of tagged value that corresponds to a given modifiers in context of considered entity.

Elements Generation. A profile stereotype derived from $\ll OFLDescriptionType \gg$ will be generated for each OFL component. For a language with description types reified in OFL by components: ComponentLanguageDescriptionType1, ComponentLanguageDescriptionType2 etc, resulting hierarchy is presented in figure 5.11.

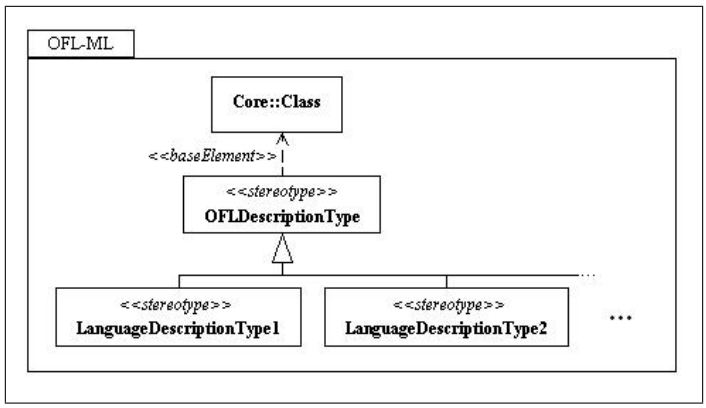

Figure 5.11: Generated stereotypes for Descriptions Components

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

Stereotypes generated for Java language are shown in figure 5.12.

Modifiers supported by these description components are summarized in table 5.1.

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

Table 5.1: Modifiers for Java Description Components

Table 5.2 presents the generated tagged values corresponding to these modifiers.

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

For Java components, only constraints dealing with incompatible modifiers

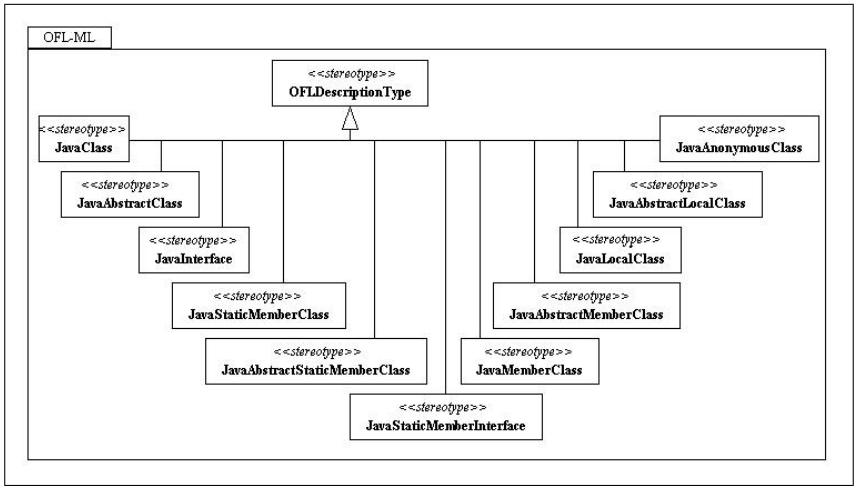

Figure 5.12: Generated stereotypes for OFL-Java Descriptions Components

are defined regarding basic access control modifiers and optimization modifiers .

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

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

The transformed constraint for generated profile is very close to the original one:

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

Complex modifier *final* will be considered further, when relationships constraints will be presented.

5.3.3 Additional constraints.

OFL parameters, characteristics and modifiers does not cover all language semantic. There is no option for automatic extraction of constrains from OFL actions. To solve these situations, additional constraints should be added by

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

Table 5.2: Tagged Values for Java Description Components Stereotypes

meta-programmer. These constraints follow the same rules like OFL Assertions added with the same goal. As an example, if we considering Java Interfaces, the following rule has to be expressed:

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

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

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

The OCL constrain added into profile to cover this rule is (for transformation see Section 5.4.1):

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

5.3.4 The External Description Element

The *External Description* element does not exists in the OFL-model. It is defined at the level of OFL-ML and specify a Description that belong to "outside world" (outside current project). This description is written usually in original

language and have no OFL information associated. It is useful especially when application access descriptions coming from class libraries.

OFL-ML could not treat the *External Descriptions* in the same manner as normal OFL-*Descriptions* are treated. The main impediment is their *opacity*. The internal structures of them are hidden and could not be seen through usual OFL-*relationships*. As a result of that, just few profile constraints could be defined for them.

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

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

Stereotypes and Tagged Values There is only one stereotype involved in external description representation. It is presented in figure 5.6.

Also, one tagged value are specified here. This is the *taggedValue* { externalPath = *importPathSpecification* }. It allows specification of the place where the resource is originated. The value of this tag is a string that depends much on language syntax related with using external resources (ex. of legal values are "import java.util.Vector" for Java or "#include 'MyApp.h'" for C++).

Constraints Using of external descriptions are heavy liked with specific language semantics. Just light control could be made. Constraints related with external descriptions are added at the level of relationships that could involve these elements.

Elements Generation Only one profile element stereotyped as ≪OFLExternal Description≫ will be generated. As already presented, this stereotype will be tagged with an *externalPath* tagged value. The value of this tag will be included in the generated source file. For models that are intended to be used in other purpose than execution this tag may be ignored.

In case of languages with complex importing syntax, meta-programmer could define additional tags for this stereotype.

Example Figure 5.13 presents examples of external descriptions representation for Java and C++.

5.4 The OFL Feature Representations

Features represents primitives declared by an OFL-Description. They describe the state (attributes) and the behavior (methods) of the considered description. Every feature has associated a name and a list of modifiers.

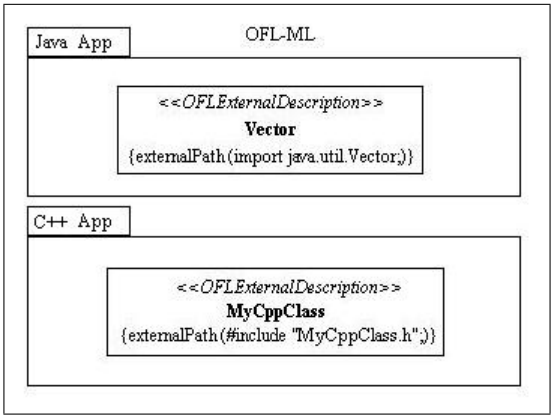

Figure 5.13: Example of using External Description Stereotype

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

5.4.1 The OFL Attributes

Attributes inherit form feature and keep values that describe the state of the description. An attribute has a name, a type, an initial value and a set of modifiers.

An OFL-attribute definition whose type is a language basic type (modelled as OFLBasicType) is represented as:

- An UML Attribute of a Class stereotyped with a stereotype derived from ≪OFLDescriptionType≫ corresponding to the OFL-description that the attribute is defined in.

An OFL-attribute whose type is an OFL-description is represented as:

- An UML Association between the Class stereotyped with a stereotype derived from ≪OFLDescriptionType≫ that declare the attribute and the UML stereotype that represents the OFL-description type of the attribute. The name of the attribute is used as the role name for the attribute type AssociationEnd of this Association.

Stereotypes and Tagged Values.

Instance Attributes. Whenever a new instance of a description is created, a new attribute associated with that instance is created for all of this

kind of attributes. OFL treats them by setting the value of *isDescriptionAttribute* characteristic of the OFL-*attribute* instance to *false*. OFL-ML represents those attributes using ≪OFLAttribute≫ stereotype for basic type attributes or ≪OFL-AssociationEnd≫ for attributes that represent aggregation with other descriptions.

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

Constant Attributes. Constant attributes are attributes that could not change their value after initialization. OFL use the OFL-*attribute*'s *isConstant* characteristic to model them. If this characteristic has value *true*, the attribute is constant and OFL-ML will represent it through ≪OFLConstantAttribute≫, ≪OFLConstantClassAttribute≫, ≪OFLClassAssociationEnd≫, respectively ≪OFLConstantClassAssociationEnd≫ stereotype.

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

Constraints. All modifiers constraints defined for AtomAttribute will be added in the generated profile. For incompatible modifiers, constraint transformation is the same as presented in Section 5.3.2. Transformation of constraints regarding stereotypes for attributes are the following:

- *Syntax*:

```
a.isConstant
```

  is translated into:

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

This transformation refer constant attributes. OFL use AtomAttribute.isCostant to keep this information. OFL-ML will represent this as an UML Attribute stereokinded as ≪OFLConstantAttribute≫.

- *Syntax*:

```
a.isDescriptionAttribute
```

is translated into:

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

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

- *Syntax*:

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

is translated into:

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

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

Elements Generation. Four profile stereotypes will be generated automatically for basic types attributes and four for association end that corresponds with relationships of the kind of OFL-UseRelationships. These stereotypes are presented in table 5.3.

To increase expressiveness of the profile, meta-programmer could derive new stereotypes from ≪OFLAttribute≫ and give them suggestive named as ≪OFLJavaStaticAttribute≫, ≪OFLJavaFinalAttribute≫, respectively ≪OFL JavaFinalStaticAttribute≫. Same work could be done also for *AssociationEnd* stereotypes. To help this task, a kind of "wizard" could be add to the profile generator tool. The additional stereotypes will inherit all generated constraints from the standard ones.

Example. Profile elements mapping to Java attributes are presented in table 5.4.

Table 5.5 presents tagged values generated for modifiers associated with Java attributes. This corresponds to *public*, *protected*, *package* and *private* access control modifiers, respectively *volatile* optimization modifier and *transient* service modifier.

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

Table 5.3: OFL-ML Attribute Stereotypes

5.4.2 The OFL Methods

Methods inherit from features and specify the behavior of the description.

Method elements could represent both procedures and functions. Functions differs from procedures because they return a result.

Method declaration specify a list of parameters. This list could be empty or not. If not, it contains a list of OFL-*parameter* elements.

Abstract methods are methods that are not implemented. An abstract methods has an empty body.

Additionally, OFL make distinction between normal methods, constructors and destructors.

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

The standard attribute *body* of *UML-Method* element is used to keep the list

Stereotype	Java Mapping	Example
OFLJavaAttribute (:OFLAttribute)	instance non-final Java basic types attributes (for Java basic types see Section 5.3.1)	char a
OFLJavaFinalAttribute (:OFLConstantAttribute)	instance final Java basic types attributes	final char a
OFLJavaStaticAttribute (:OFLClassAttribute)	static (class) non-final Java basic types attributes	static char a
OFLJavaFinalStaticAttribute (:OFLConstantClassAttribute)	static (class) final Java basic types attributes	final static char a
OFLJavaAssociationEnd (:OFLAssociationEnd)	instance non-final Java aggregation attributes	AClass a
OFLJavaFinalAssociationEnd (:OFLConstantAssociationEnd)	instance final Java aggregation attributes	final AClass a
OFLJavaStaticAssociationEnd (OFLClassAssociationEnd:)	static (class) non-final Java aggregation attributes	static AClass a
OFLJavaFinalStaticAssociationEnd (:OFLConstantClassAssociationEnd)	static (class) final Java aggregation attributes	final static AClass a

Table 5.4: OFL-ML Stereotypes of Java Attribute

of statements that represents the method body. The UML represents that list like *ProcedureExpression*, that is actually a list of strings. When code are generated from the model, these strings have to be translated into OFL-*Statement* elements. Other possibility is to represent the body using UML-Actions Semantic Model. This option will be discussed at the end of this chapter.

For abstract methods, OFL-ML use attribute *isAbstract* inherited from UML-*Operation* element. If true, then the operation does not have an implementation and the method body will be empty. If false, the operation must have an implementation in the description or inherited from an ancestor.

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

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

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

in An input Parameter (may not be modified).

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

inout An input Parameter that may be modified.

return A return value of a call.

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

Table 5.5: Tagged Values for Java Attribute Stereotypes

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

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

In this context, from UML-BehavioralFeature which UML-Method inherit from, we have:

- All Parameters should have a unique name.

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

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

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

Also, as for attributes, all modifiers constraints defined for AtomMethod will be added.

Elements Generation. Four stereotypes will be generated for methods. These are ≪*OFLMethod*≫, ≪*OFLConstructorMethod*≫, ≪*OFLDestructorMethod*≫ and ≪*OFLParame-ter*≫. First tree stereotypes apply to UML Method element. The last one apply to UML Parameter. ≪*OFLConstructorMethod*≫ corresponds to a OFL-method that have attribute *isConstructor* set to true. ≪*OFLDestructorMethod*≫ corresponds to a OFL-method that have attribute *isDestructor* set to true. As mentioned in the section 5.4.1, to increase expressiveness of the profile elements, meta-programmer could decide to derive specific stereotypes from the generated ones.

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

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

Following list presents transformation rules for constraints related with methods.

- *Syntax*:

  ```
  m.isConstructor
  ```

  is translated into:

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

This transformation refer constructor methods. OFL use AtomMethod. isConstructor to keep this information. OFL-ML will represent this as an UML Method stereokinded as ≪OFLConstructorMethod≫.

- *Syntax*:

  ```
  m.isDestructor
  ```

  is translated into:

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

This transformation refer destructor methods. OFL use AtomMethod.isDestructor to keep this information. OFL-ML will represent this as an UML Method stereokinded as ≪OFLDestructorMethod≫.

Both characteristics *body* and *parameters* are collection also in OFL and in UML, so collection operation could be applied on both in same way.

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

Table 5.6: OFL-ML Stereotypes of Java Method

Example. Profile elements mapping to Java methods are presented in table 5.6.

Tagged values generated for modifiers associated with Java methods are presented in table 5.7. This corresponds to *public*, *protected*, *package*, *private* and *final* access control modifiers, respectively *native* and *strictfp* optimization modifiers and *synchronized* service modifier.

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

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

Table 5.7: Tagged Values for Java Method Stereotypes

Constraints that correspond to access control modifiers are generated using same translation as presented in the previous section.

For native modifier the assertion has also to be transformed.

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

Transformation are made using already presented transformation rules.

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

Additional constraints. As we mentioned in 5.3.2, the generated constraints will not cover all language model semantics.

For Java, all method parameters have to have attribute *kind* set to value *in*, except one that is set to *return*.

An Java method could not be abstract unless it is contained be a Java Interface or a Java abstract class.

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

Following Java constraint is related with a *finalize* method. A *finalize* method has to be declared as *protected*, return no value (has *void* as return type) and throws *Throwable* exception.

```
context OFLFinalizeMethod (Core::Method)
inv: self.stereotype.taggedValue
              ->select(name = 'protected')->size = 1
        and
     self.parameter->select(p |
            p.kind = return
            implies
                ( p.type.isStereotyped('OFLBasicType')
                    and
                p.type.name = 'void')
```

```
        )
    and
self.stereotype.taggedValue
          ->select(tag | tag.name = 'javaThrows'
                        implies tag.value = 'Throwable')
```

5.5 The OFL Relationship Representations

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

These stereotypes could be used in modelling tools to generate corresponding instances of OFL elements and to fill them with appropriate information. Thereby, the following elements are considered to be generated: instances of *OFL-Import Relationship* components and *OFL-Use Relationship* components.

This version of OFL-ML does not consider dynamic relationships reified by OFL-ObjectToClassRelationhip and OFL-ObjectToObjectRelationship. That is because OFL-ML profiles could represent only static models corresponding to UML Static Class Diagrams.

5.5.1 The OFL Import Relationship

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

5.5.2 Stereotypes and Tagged Values.

The abstract stereotype ≪OFLImportRelationship≫ is the base for all the concrete stereotypes representing OFL *ImportRelationhip* components of the considered language. The name of the generated stereotypes are the same as the name of the OFL components with "Component" prefix removed (ex. for a component "ComponentJavaExtends", a stereotype named ≪JavaExtends≫ will be created).

All relationships stereotyped as specialization of ≪OFLImportRelationship≫ will have associated a set of tagged values. Values of these elements correspond to some OFL-*AtomRelationship* characteristics. These tagged values are presented in table 5.8.

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

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

Table 5.8: OFL-ML Tagged Values for OFLImportRelationhip

Constraints. All modifiers constraints defined at the level of relationship components will be added. Transformation rules will translate all characteristics of relationships components into corresponding tagged values. Following rules will apply:

- *Syntax:*

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

is translated in:

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

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

- *Syntax:*

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

is translated in:

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

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

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

Constraint related with this parameter will check conformance with cardinality specification. If *cardinality* is ∞ no constraint is necessary.

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

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

Parameter ConceptRelationship::repetition. Repetition denote if a direct repetition of target (parent) is permitted or not. The possible values of this parameter are *allowed* and *forbidden*. Value *allowed* make sense just in a relationship with cardinality $n < 1$ (1-1).

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

OFL-ML: if cardinality ≠ 1 and repetition = forbidden

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

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

OFL-ML: if circularity = forbidden

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

First OCL *let* expression (*dp*) calculates all direct parents of a Classifier in the meaning of considered relationship. Expression *allParents* calculates all parents of a *Classifier*. Parameter *p* contain all already visited parents and is used to stop recursions. Constraint check if the source of relationship is included or not in its list of parents.

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

covariant elements that change on redefinition need to have same type or a sub-type like original one (defined by the source).

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

nonvariant elements could not change the type on redefinition.

non_applicable parameter is not applicable

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

All constraint use the following definitions for direct parent and all parents of a Classifier:

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

Constraints regarding method parameters variance are presented next.

OFL-ML: if feature_variance for method parameter = covariant

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

OFL-ML: if feature_variance for method parameter = contravariant

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

```
                    rp.name = p.name
                      implies
                    rp.allParents(Set{})
                          ->including(rp.type)->include(p.type)))
          ))
```

OFL-ML: if feature_variance for method parameter = nonvariant

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

For method result variance constraints are the same but the term '*p.kind <> return*' are replaced by '*p.kind = return*'.

Next list show constraints for attribute variance.

OFL-ML: if feature_variance for attributes = covariant

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

OFL-ML: if feature_variance for attributes = contravariant

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

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

OFL-ML: if feature_variance for method parameter = nonvariant

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

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

OFL-ML: if abstracting = mandatory

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

OFL-ML: if abstracting = forbidden

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

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

OFL-ML: if effecting = mandatory

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

OFL-ML: if effecting = forbidden

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

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

OFL-ML: if masking = mandatory

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

OFL-ML: if masking = forbidden

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

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

OFL-ML: if redefining = mandatory

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

OFL-ML: if redefining = forbidden

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

Parameter ConceptRelationship::renaming. This parameter indicate if renaming of features that pass considered relationship is *mandatory*, *allowed* or *forbidden*.

OFL-ML: if renaming = mandatory

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

OFL-ML: if renaming = forbidden

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

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

OFL-ML: if removing = mandatory

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

OFL-ML: if removing = forbidden

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

Parameter ConceptRelationship::showing. This parameter is opposite for masking. It indicate if the primitive is make again visible after it was masked. Possible values are *mandatory*, *allowed* and *forbidden*.

OFL-ML: if showing = mandatory

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

OFL-ML: if showing = forbidden

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

Characteristic AtomLanguage::validRelationships. This characteristic indicate descriptions types that could act as sources and targets for considered relationship. Values are triplets of <componentRelationship, componentDescriptionSource, componentDescriptionTarget>. The OFL-ML will add a constraint that check for legal source type according to that.

Next we present generated constraints consider the value {< componentRelationship, LanguageDescriptionTypeSource1, LanguageDescriptionTypeTarget1>, <componentRelationship, LanguageDescriptionTypeSorce2, LanguageDescriptionTypeTarget2>, ... } for this characteristic.

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

context  ComponentRelationhip (OFLImportRelationship)
inv: let st = self.parent in
    (
       st.isStereotyped('LanguageDescriptionTypeTarget1')
          or
       st.isStereotyped('LanguageDescriptionTypeTarget2')
          or
       ...
    )
```

Elements Generation. A profile stereotype derived from ≪OFLImportRelationship≫ will be generated for each OFL component. For a language with import relationships reified in OFL by components: ComponentLanguageImportRelationship1, ComponentLanguageImportRelationship2 etc, resulting hierarchy is presented in figure 5.14.

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

Constraints are generated regarding OFL-ML generation-conditions. These condition was presented as statements like:

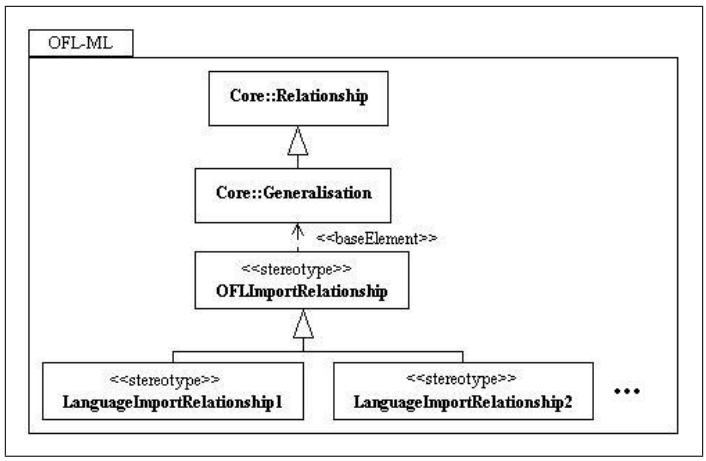

Figure 5.14: Generated stereotypes for Import Relationships Components

OFL-ML: if condition

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

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

TaggedValues that corresponds to these stereotypes are shown in table 5.9. Valid sources and targets for components are presented in table 5.10. Example

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

Table 5.9: Tagged Values for Java Import Relationship Components Stereotypes

of generated constraints for valid sources and targets for *JavaInterfaceExtends* relationship are given bellow.

```
context  JavaInterfaceExtends (OFLImportRelationship)
```

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

Table 5.10: Valid sources and targets for Java Import Relationship Components Stereotypes

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

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

5.5.3 The OFL Use Relationships

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

Stereotypes and Tagged Values. The abstract stereotype ≪OFLUseRelationship≫ is the base for all the concrete stereotypes representing OFL *UseRelationhip* components of the considered language. As for import relationships presented in the section above, the name of the generated stereotypes are the same as the name of the OFL components with "Component" prefix removed (ex. for a component "ComponentJavaAggregation", a stereotype named ≪JavaAggregation≫ will be created).

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

Table 5.11: OFL-ML Tagged Values for OFLUseRelationhip

Also, same way as for import relationship, all use relationships stereotyped as specialization of ≪OFLUseRelationship≫ will have associated a set of tagged values that corresponds to some OFL-*AtomRelationship* characteristics. These tagged values are presented in table 5.11.

Constraints. All associations that correspond to an OFL use relationship must have exactly two ends that correspond to source and target of relationship.

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

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

parameter ConceptRelationship::cardinality, transformed constraint will be the following:

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

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

The list of parameters that are valid in context of an use relationship is:

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

Parameter ConceptRelation::dependence. This parameter specify if instances of target description have a life time dependent or independent of source description. Possible values are *dependent* and *independent*.

This parameter has meaning just for an use relationship.

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

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

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

OFL-ML: if dependance = independent

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

OFL-ML: if dependance = dependent

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

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

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

Tagged values will be generated also for each use relationship according to values of OFL-*parameters* masking, renaming, removing and showing. As already presented, tags will be added considering values *mandatory* and *allowed* for these parameters.

Example. If we consider Java language, following use relationship components are identified [CCL02, Cre01a]: aggregation (JavaAggregation), class aggregation (JavaClassAgregation), composition (JavaComposition) and class composition (JavaClassComposition). Because the last two components imply only Java primitive types, which are OFL-ML basic types, they are represented by stereotypes derived from basic type composition (presented in section 5.5.4).

TaggedValues that correspond to these stereotypes are presented in table 5.12. The deletedFeatures specify the features that are deleted passing this relationship (ex. features declared with *private* modifier). Table 5.13 presents

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

Table 5.12: Tagged Values for Java Use Relationship Components Stereotypes

valid sources and targets for these relationships. Constraints and tags will be added regarding parameters values.

For JavaAggregation we will have:

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

For JavaClassAggregation we will have:

- cardinality = ∞ (no OFL-ML constraint)

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

Table 5.13: Valid sources and targets for Java Use Relationship Components Stereotypes

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

5.5.4 The Basic Type Composition

Basic type composition association stereotypes are used to represent composition with language primitive types. The relationship corresponds to primitive type attribute declaration by a description. This relationship is all time composition because basic types instances represents values but not objects.

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

Constraints. An OFLMLBasicTypeComposition represents a composition.

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

An OFLMLBasicTypeComposition could have as a target only an OFLBasicType.

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

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

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

An OFLMLBasicTypeComposition could have only OFLAssociationEnd as a target end.

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

An OFLMLBasicTypeClassComposition could have only OFLClassAssociationEnd as a target end.

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

Elements Generation. Usually maxim two stereotypes are generated: one derived from ≪OFLMLBasic-TypeComposition≫ and one from ≪OFLMLBasicTypeClassComposition≫. If considered language have more than two type of relationships involving basic types, additional constraints could be also necessary.

No tagged values are necessary.

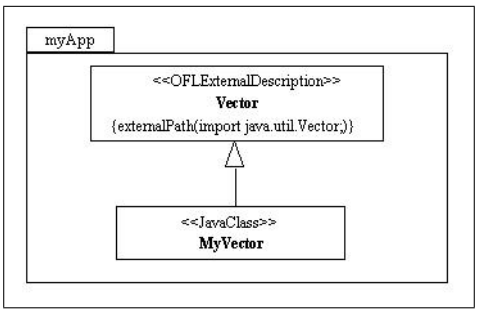

Figure 5.15: Example of using OFLML ExternalImportRelationship

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

5.5.5 The External Import Relationship

External import relationships involve external descriptions. External descriptions are presented in sec. 5.3.4 and represents descriptions imported from external class libraries. These descriptions are usually opaque and they could not be involved in OFL relationships.

OFL-ML use standard UML-*generalization* to represent these values.

Stereotypes and Tagged Values. No stereotypes and tagged values are necessary.

5.5.6 Constraints.

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

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

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

Example. An example of using an external import relationship in OFL-ML Java profile is presented in fig. 5.15.

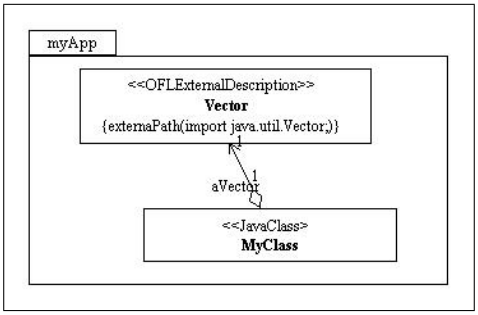

Figure 5.16: Example of using OFLML ExternalUseRelationship

5.5.7 The External Use Relationship

External use relationships involve external descriptions. Treatment of external use relationship is done in same way as for external import relationship.

OFL-ML use standard UML-*association* to represent these values.

Stereotypes and Tagged Values. No stereotypes and tagged values are necessary.

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

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

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

Example. An example of using an external use relationship in OFL-ML Java profile is presented in fig. 5.16.

5.6 The OFL Model Organization

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

5.6.1 The OFL Package

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

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

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

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

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

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

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

5.7 Modelling Example Using an OFL-Java Profile

As an example we consider the following Java code:

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

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

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

public class Color { }

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

public class Car {
   public Color color;

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

Figure 5.17 gives an example of a model for application which use an OLF-ML profile for OFL-Java:

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

The diagram corresponds to above Java code. The OFL-ML Java Profile elements used have bin defined according to previous sections. The diagram was generated with Objecteering UML Modeler version 5.2.2 [Sof03a].

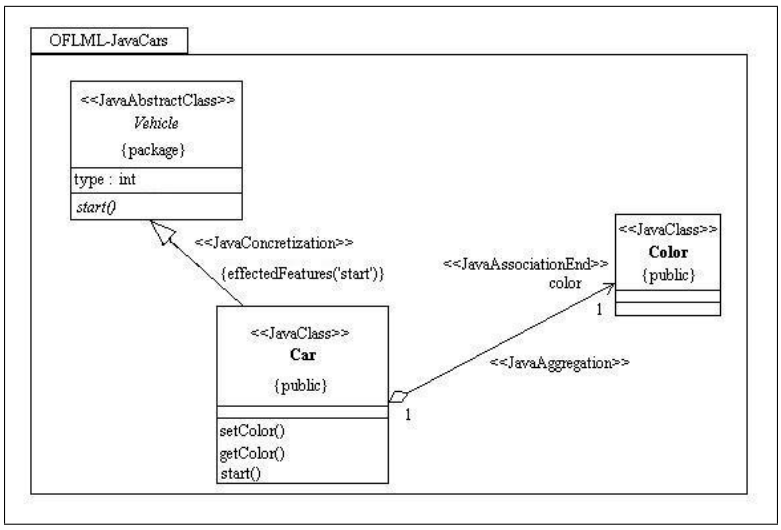

Figure 5.17: Example of using OFLML Java Profile

5.8 Conclusions and Future Work

5.8.1 Conclusions

This paper has presented an approach for generation of UML profiles for an object oriented languages described in OFL. This approach is based on a profiles meta-languages named OFL-ML. We present on detail generation mechanisms of OFL-ML and its drawbacks related with some language semantics. Then, based on this meta-language, we present an OFL-Java Profile that is generated based on OFL-ML rules.

To define a profile, OFL-ML use meta-information existing at the level of OFL. Profiles elements are generated based on following OFL entities:

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

To complete the Profile, for each elements, additional taggedValues and OCL constraints are also generated.

Because each OFL-ML Profile respect UML 1.5 standard specification, generated profiles are guaranteed to be used with commercial UML modelling tools that support profile mechanisms.

Presented approach has some limitations. It not consider following issues:

- other UML diagrams, additional to static class diagrams
- do not model OFLObjects
- do not address dynamic relationships like OFL-class-to-object-relationships and OFL-object-to-object-relationships
- do not treat type multiplicity (arrays or collection classes like java.util.Vector)

5.8.2 Future Work

We identify two main directions for future work.

First intend is to go deeper with language customization. Current version of OFL provides just a light reification and no customization of semantics at the level of routine body. Using UML definition of Action Model [OMG03, MTAL98], we intent to provide a way to represent also semantics at this level. Our proposal is to extend the generated OFL-ML profile with UML-Actions for routine body representation.

Briefly, UML actions represent:

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

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

ConceptDescription parameters .

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

ConceptRelationship parameters .

- direct_access - specify if the relationship allow direct access to a feature of target description. This parameter will be involved in constraints at the level of UML Read and Write Actions
- polymophism_implication - specify if considered relationship accept or not polymorphism for instances of classes involved in. This parameter will be involved in constraints at the level of UML Read and Write Actions and Messaging Actions

The second proposed task is to generate a representation in XML [CCCL00] or in a proprietary language representation of profile elements. We consider here specifications for profile representation provided by some major tools like Objecteering UML, Rational Rose etc.

Chapter 6

OFL-ML Tools Support and Validation

Tools are the way most people interact with a modeling language. Therefore one important concern is to help tools offer as much support as possible to the modeler. We also use tools support to demonstrate the validity of the presented approach.

6.1 The OFL Framework

The OFL framework presented here describes a set of tools that make possible the implementation and usage of the OFL model. This implementation could serve a language designer, to help him to try new modeling facilities (descriptions and relationships types). It can assist an analyst to validate a model or to extract metrics from application implementation model. Also it can help a programmer who needs an extension of an existing language to be closer to a specific domain. Basically there are four main tools included in the proposed framework: OFL-Meta - a tool for meta-programming work; OFL-ML tool for application design and implementation or as an alternative the OFL-ML profile generator; OFL-Parser for code generation and OFL Database that allows interactions between previously mentioned tools and keeps OFL languages and OFL applications meta-data. The framework architecture is presented in figure 6.1.

As an implementation language we considered Java, a modern object oriented language that permits a great portability and, furthermore, has powerful libraries, essential in implementation of complex applications. Parts of presented tools were developed or are under development in collaboration with researchers from "Sophia Antipolis" University of Nice. Some of them were created as diploma projects by graduating students from "Politehnica" University of Timisoara.

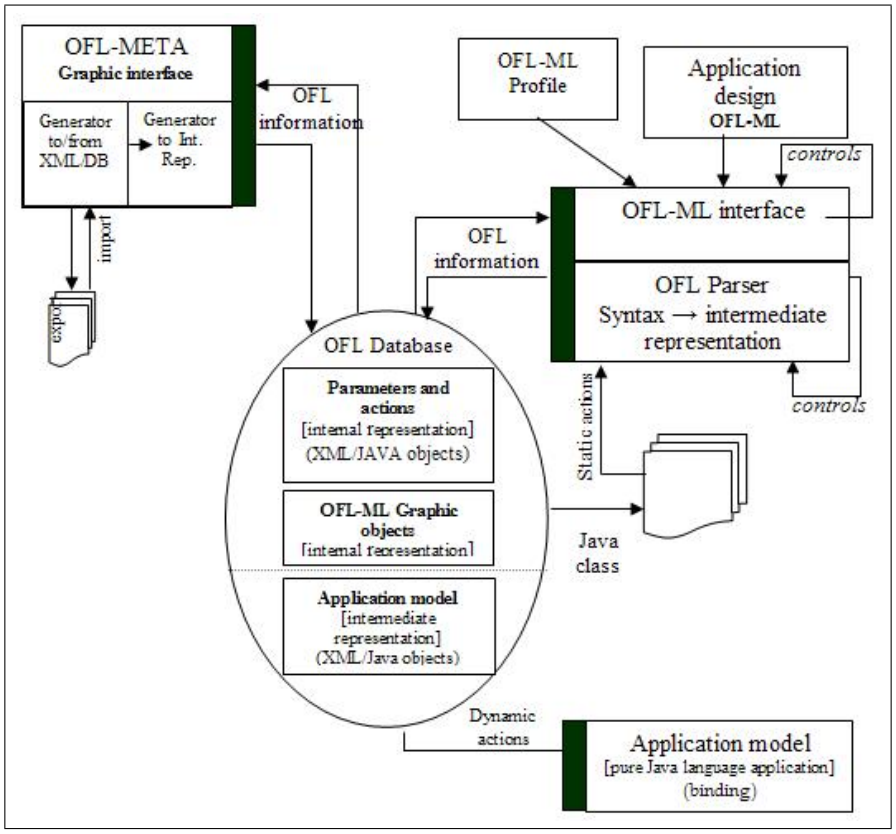

Figure 6.1: The OFL Framework for OFL Applications Development

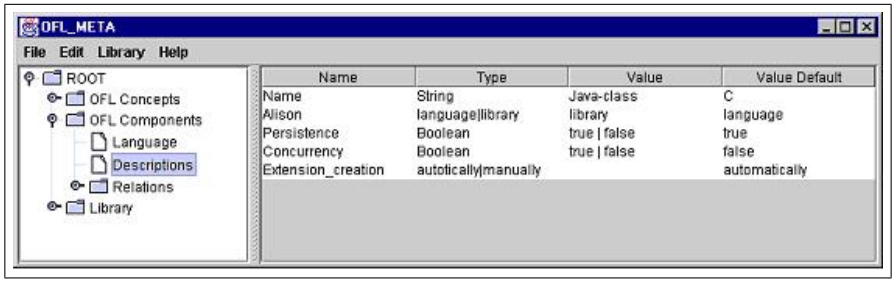

Figure 6.2: Using the OFL Meta Tool to describe an OFL-Language

6.1.1 The OFL-Meta Tool

The OFL-Meta tool is designed to help meta-programmers to describe an OFL-Language or to extend an existing one. It allows in fact to define a new OFL-Language and to add OFL-Components, OFL-Modifiers, OFL-Assertions and OFL-Actions to it.

It presents a synthetic view of the tree representing the OFL hierarchy. It allows inspection of already made components or creating of new components. These could be new components designed from the scratch or could be copies of existing components modified as needed as presented in figure 6.2.

6.1.2 The OFL-Database

All tools from OFL-Framework are designed around OFL-Database. It represents a repository for OFL language components and for OFL application entities. A meta-programmer will use OFL Database to store information about his OFL-Languages. A programmer will use OFL Database to retrieve components that he is planning to use and to store developed application. An early version of OFL Database was considered a **PJAMA** [ADJ+96, ADJS96] implementation. The current version is developed in **POET** [Sof02, Sof03b] which is a free object oriented database management system. References for OFL-Database implementation could be founded in [Pes98, Cap99].This system supports the ODMG specifications [CBB+97], allows storage of Java Objects and export to XML [Mic99].

6.1.3 The OFL-ML Modeling Tool

To implement OFL-ML we decide to implement both a dedicated modeling tool and a profile generator. The reason is the incipient support for UML Profiles included in standard UML modeling tools.

The OFL-ML modeling tool [PP01] is designed to help programmers create OFL models. The architecture used is showed in figure 6.5. In this version, we don't implement the package concept. Furthermore, we don't implement the

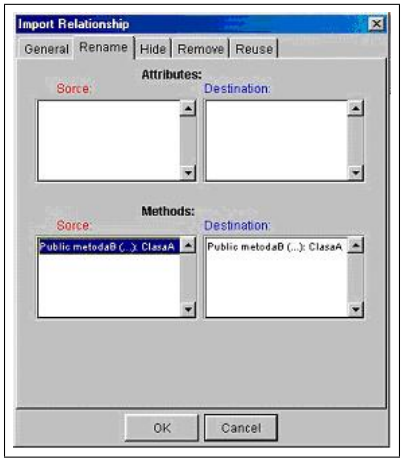

Figure 6.3: OFL-ML Tool: Import Relationship Dialog Window - the List of Characteristics

concept of local description, the reason for this decision being that the actual version of OFL reification did not support the local description. We study carefully the necessity to have two levels of visualization of the application model of OFL-ML and we adopt the following solutions as presented in figures 6.3 and 6.4:

- The visualization of the import relationship characteristics (level two of visualization) was not drawn on the model view, but we created a dialog window, which presents the list of its characteristics. The advantage of this solution is its capacity to allow visualization and modification of the relationship parameter at the same time and in the same mode (reduce the code). Instead of using two commands: one for the visualization (that draws in model view) and one for the modification of the import relationship parameters (normally a dialog box) the programmer has one command for both cases. Another problem of the specification was the overlapping of the draw parameters (of the import relationship), on top of other elements of the model and in this situation the visibility of the model is drastically reduced.

- To increase the contrast and visibility of the programmer's model we've introduced the full colored termination for relationship (we draw a solid triangle and a solid diamond instead of the empty geometrical shape).

- The use relationship is visible to the designer only on demand. This solution was adopted in order to reduce the complexity of the model view and to permit the programmer to concentrate on the modeling side of his project. In conclusion, the model view only includes representations of the

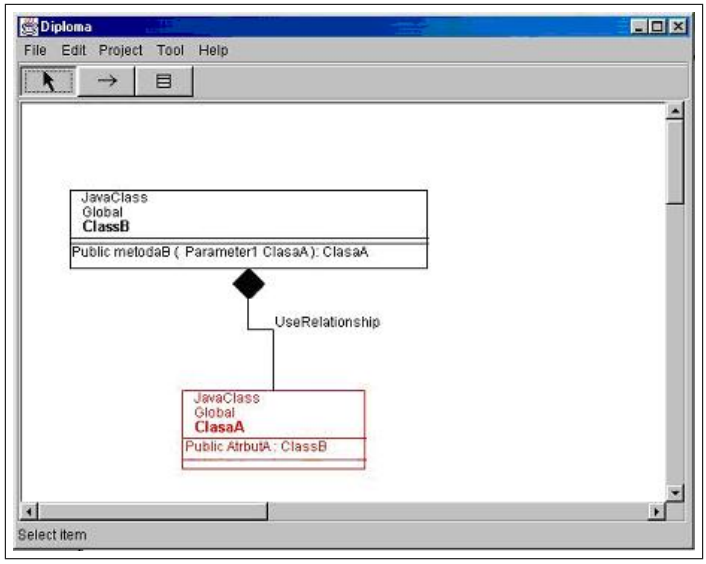

Figure 6.4: OFL ML Tool: Application Window

descriptions and import relationships, by default. The core of OFL doesn't provide any support for storage or usage of graphical information of the description representation. Consequently, we supplement the specification with additional the classes. Analyzing the necessities of description drawing, we find it essential to store the position of the description; other information (frame of the description, the position of the relation) will be generated at the run time. This approach reduces the space claimed by the saved file on the disk. Another solution, which we have considered, is to automatically generate the graph of the model, but in this way the organization of the model will be harder.

The application offers the programmer the possibility to create descriptions and relationships in his project. Both description and relationship have a second level of visualization, more detailed, which also permits the modification of the parameter's characteristics.

6.1.4 The OFL-ML Profiles Generator

The OFL-ML Profiles Generator in under development. It has the mission to generate both a Profile spcification in LaTex format and the XML Profile representation.

This generator will consider all the rules presented in Chapter 5. As an extension of it we think to also generate action routines in J-Language defined by Objecteering Software [Sof03a]. This actions could be used into the Objecteering

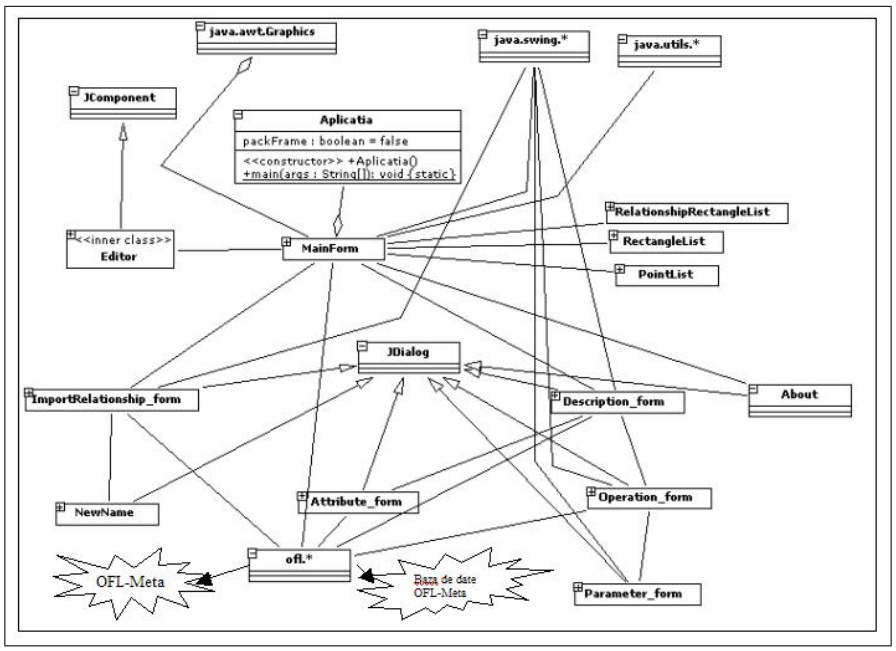

Figure 6.5: The OFL ML Tool Implementation Class Diagram

UML modeling tool in order to generate the XML representation of the OFL application.

6.1.5 The OFL Parser

The OFL Parser [PT01] could be described as a compiler for OFL applications. In the current version it is a translator which generates pure Java code. The generated code is augmented with OFL run-time information including both OFL assertions and OFL actions.

The OFL Parser has a modular construction and could be adapted to generate information like metrics or to do some formal verification of the application model.

6.2 Perspectives

We describe in this section five tools involved in OFL applications development. We plan to adapt this tools in the future to follow changing of the UML Profile standard. Also we start discussions with Objecteering for a future collaboration that have as goal a possible integration of our approach into tools developed by them.

Chapter 7

Conclusions and Perspectives

7.1 Conclusions

The main benefit of our approach is the possibility to have a direct and an exact matching between model and implementation of an application. This desiderate is achieved through two facilities supported by our approach. The first one is represented by the possibility of programming language tailoring through meta-language extension mechanisms. The second one resides in increasing sematic precision of modeling language based on generation of an UML Profile (OFL-ML Profile). The backbone of both facilities is the meta-information existing at the level of OFL.

The strong integration of our approach with standard programming and modeling tools and technology represents also a validation for it.

7.2 Author Contributions

The approach presented in this thesis brings a number of significant contributions to the field of object oriented programming and modeling languages. This contributions are presented split in three categories.

Contribution at the level of OFL model extension

- Analysis of the OFL non-customizable elements that are used frequently by programmers in practical works
- Definition of the Component Modifier and OFL Modifier
- Identification of Component Modifiers in Java, C++ and Eiffel

- Definition of new atoms and components in addition to original OFL Model
- Classification of modifiers based on origin and semantic
- Definition of implementation rules for each category
- Reification of several modifiers belonging to Java, C++ and Eiffel

Contribution at the level of OFL-ML meta-profile definition

- Analysis of main modeling and meta-modeling approaches
- Definition of a method which allowed to increase sematic precision for a modeling language (an UML Profile) based on OFL meta-information
- Definition of the notions of OFL-ML Profile and OFL-ML Meta-profile
- Identification of the UML subset covering all OFL-ML Profiles
- Definition of the Virtual Meta-Model for OFL-ML Profile
- Definition for all modelling elements belonging to a generated profile
- Definition of generation rules for all elements considering OFL components, parameters, characteristics and actions
- Definition of a mechanism which allow to add constraints for the generated profile
- Rules for automatic constraints generation
- Example of a elements generation considering OFL-Java language

Contribution at the level of tools implementation

- Definition of a framework which provides support for OFL application developments
- Development and integration of various tools into the OFL Framework

7.3 Perspectives

As perspective we plan to develop and refine OFL and OFL-ML approaches by adding direct support for metrics extraction, aspect oriented programming and service definitions. We plan also to keep the OFL-ML meta-profile up to date with new versions for UML Profile standard. We also intend to test our approach in at industry level by starting cooperation with modeling and programming tools vendors like *Objecteering Software*.

Bibliography

[A. 00] A. Capouillez, R. Chignoli, P. Crescenzo, and P. Lahire. Modeling Hypergeneric Relationships between Types in XML. In *In ETC2000, 4th Edition of Symposium of Electronics and Telecommunications*, November 2000.

[Aal02] W. M. Van Der Aalst. Inheritance of Dynamic Behaviour in UML. In *In Proceedings of the Second Workshop on Modelling of Objects, Components and Agents (MOCA 2002), Aarhus, Denmark, August 2002, University of Aarhus, D. Moldt, editor*, 2002.

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

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

[ADJ+96] M. P. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent java. *SIGMOD Record*, 25(4), 1996.

[ADJS96] M. P. Atkinson, L. Daynes, M. J. Jordan, and S. Spence. Design issues for persistent Java: A type-safe, object-oriented, orthogonally persistent system. In *Proceedings of the 7th Workshop on Persistent Object Systems (POS'96), Cape May (NJ), USA*, 1996.

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

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

[BCR00] E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML activity diagrams. In *Proceedings Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 2000*, LNCS. Springer, 2000.

[BH00] T. Baar and R. Hahnle. An integrated metamodel for OCL types. In *In Proc. OOPSLA 2000, Workshop Refactoring the UML: In Search of the Core, Minneapolis, Minnesota, USA*, 2000.

[BR01] C. Boyapati and M. Rinard. A Parameterized Type System for Race-Free Java Programs. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001), Tampa, Florida*, October 2001.

[Cap99] A. Capouillez. ROOPS: un Service parametrable de persistance pour OFL. Technical Report I3S/RR–1999-15–FR, Laboratoire d'Informatique, Signaux et Systmes de Sophia-Antipolis, France, September 1999. http://www.i3s.unice.fr/I3S/FR/.

[Car88] L. Cardelli. A semantics of multiple inheritance. In *Information and Computation, 76(2/3)*, February 1988.

[CBB+97] R. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, A. Strickland, and D. Wade. *Object Database Standard : ODMG 2.0.* Morgan Kaufmann Publishers, Inc, 1997.

[CCCL00] A. Capouillez, R. Chignoli, P. Crescenzo, and P. Lahire. Modeling Hypergeneric Relationships between Types in XML. In *ETC'2000, 4th Edition of Symposium of Electronics and Telecommunications*, November 2000.

[CCCL01] A. Capouillez, R. Chignoli, P. Crescenzo, and P. Lahire. Hypergénéricité pour les Langages a Objets : le Modèle OFL. In *LMO'2001 (Langages et Modèles a Objets)*, page 16. Hermes Science Publications, L'objet : logiciels, bases de donnes, réseaux, volume 7, nr. 1-2/2001, January 2001.

[CCL99] R. Chignoli, P. Crescenzo, and P. Lahire. OFL: An Open Object Model based on Class and Link Semantics Customization. Technical Report 99-08, Laboratoire d'Informatique, Signaux et Systmes de Sophia-Antipolis, France, March 1999. http://www.i3s.unice.fr/I3S/FR/.

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

[CD94] S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice Hall, 1st edition, November 1994.

[Chi99] S. Chiba. Open C++ 2.5 Reference Manual. University of Tsukuba, Japan, http://www.csg.is.titech.ac.jp/ chiba/, May 1999.

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

[CL02a] P. Crescenzo and P. Lahire. Customisation of Inheritance. In *Springer Verlag, LNCS series, ECOOP'2002 (The Inheritance Workshop) and Proceedings of the Inheritance Workshop at ECOOP 2002, University of Jyvskyl, Finlande*, page 7, June 2002.

[CL02b] P. Crescenzo and P. Lahire. Using both specialisation and generalisation in a programming language: Why and how? In *In OOIS 2002 (8th International Conference on Object-Oriented Information Systems) - MASPEGHI workshop, Montpellier, France*, September 2002.

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

[Coo98] J. W. Cooper. The Design Patterns Java Companion. In *Addison-Wesley*, 1998.

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

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

[CW02] T. Clark and J.B. Warmer. *Object Modeling With the Ocl: The Rationale Behind the Object Constraint Language*. Springer Verlag, Lecture Notes in Computer Science, 2263, April 2002.

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

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

[Ewi] G. Ewing. *Class inheritance: The mechanism and its uses.* http://citeseer.nj.nec.com/ewing94class.html.

[Fla99a] D. Flanagan. *Java in a Nutshell : A Desktop Quick Reference.* OReilly and Associates, 3rd edition, 1999.

[Fla99b] D. Flanagan. *Java in a Nutshell : A Desktop Quick Reference.* O'Reilly and Associates, 3rd edition, November 1999.

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

[GC96] B. Gowing and V. Cahill. Meta-Object Protocols for C++: The Iguana Approach. In *Proceedings of Reflexion'96, Ed. Kiczales, California*, April 1996.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object Oriented Software.* Addison-Wesley Publishing, 1994.

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

[GK98] M. Golm and J. Kleinoder. metaXa and the Future of Reflexion, PWRP'98. Technical Report UTCCP 98-4, University of Tsukuba, Japan, October 1998.

[Gri99] W Grieskamp. *A Set-Based Calculus and its Implementation.* Phd. thesis, Technischen Universität Berlin, Germany, November 1999. http://research.microsoft.com/users/wrwg/.

[Gui98] J. Guimares. Reflection for Statically Typed Languages. In *ECOOP 98, 12th European Conference on Object-Oriented Programming Brussels, Belgium*, July 1998.

[Hey01] T. Heyer. Semantic Inspection of Early UML Designs. In *Proceedings Workshop on Inspection in Software Engineering (WISE '01)*, July 2001.

[Kai99] K. Kaitanen. *J-UML Specification. Version 1.02*, February 1999. http://www.vtt.fi/tte/papers/j-uml.

[KDRB91] G. Kiczales, J. Des Rivieres, and D. Bobrow. *The Art of the MetaObject Protocol.* MIT-Press, 1991.

[Lem98] R. Lemesle. Meta-modeling and modularity : Comparison between MOF and CDIF formalisms. In *OOPSLA'98 Workshop Model Engineering, Methods and Tools*, October 1998.

[Lip99] S. B. Lippman. *Essential C++.* Addison-Wesley Pub. Co., 1st edition, October 1999.

[LYJW96] T. Lindholm, F. Yellin, B. Joy, and B. Walrath. *The Java Virtual Machine Specification*. Addison-Wesley Pub. Co., 3rd edition, September 1996.

[Mey91] B. Meyer. *Eiffel : The Language*. Prentice Hall, 1st edition, October 1991.

[Mey97] B. Meyer. *Object-Oriented Software Construction. Professional Technical Reference*. Prentice Hall, second edition, 1997.

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

[Mic99] A. Michard. *XML Language and Applications*. Eyrolles, 1999.

[MTAL98] S. J. Mellor, S. R. Tockey, R. Arthaud, and P. LeBlanc. An Action Language for UML: Proposal for a Precise Execution Semantics. In *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France*, LNCS, pages 307–318. Springer, June 1998.

[Obe00] I. Ober. Defining Precise Semantics for UML. In *Workshop at ECOOP'2000, Cannes, Fance*, June 2000.

[Obj01] Object Management Group - OMG. *Meta Object Facility Specification (MOF), Version 1.3.1*, November 2001. http://www.omg.org/technology/documents/formal/meta.htm.

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

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

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

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

[P. 02] P. Lahire, P. Crescenzo, and A. Capouillez. Le modele ofl au service du metaprogrammeur - application a java. In *In Proceedings of LMO 2002 (Langages et Modeles a Objets), Montpellier, France*, January 2002.

[PCL03a] D. Pescaru, P. Crescenzo, and P. Lahire. An Extension for OFL Model through modifiers. Technical report, Laboratoire d'Informatique, Signaux et Systmes de Sophia-Antipolis, France, Jully 2003.

[PCL03b] D. Pescaru, P. Crescenzo, and P. Lahire. Automatic Profile Generation for OFL-Languages. Technical report, submited to Laboratoire d'Informatique, Signaux et Systmes de Sophia-Antipolis, France, October 2003.

[Pes98] D. Pescaru. A Java Object Oriented Environment for Databases. In *Third International Conference on Technical Informatics, CONTI-98, Timisoara*, October 1998.

[Pes01] D. Pescaru. A Framework for an Hypergeneric System Implementation Based on OFL. PhD Report at "Politehnica" University of Timisoara, Romania, October 2001.

[Pes03] D. Pescaru. Implementation for OFL Modifiers Assertions. submited to Buletinul Stiintific al Univ. "Politehnica" din Timisoara, Vol.48(62)/03, ISSN 1224-600X, November 2003.

[PL00] D. Pescaru and P. Lahire. OpenIDL: an Open Modeling Language Based on IDL and OFL. In *Fourth International Conference on Technical Informatics, CONTI-2000, Timisoara, Romania*, October 2000.

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

[Por92] H. H. Porter. Separating the subtype hierarchy from the inheritance of implementation. In *Journal of Object-Oriented Programming*, February 1992.

[PP01] D. Pescaru and C. Papandonatos. An OFL-ML tool implementation. Buletinul Stiintific al Univ. "Politehnica" din Timisoara, nr. 46(60)/01, ISSN 1224-600X, October 2001.

[PT01] D. Pescaru and E. Tundrea. OFLParser - Code Generator for OFL-ML Models. Buletinul Stiintific al Univ. "Politehnica" din Timisoara, nr. 46(60)/01, ISSN 1224-600X, October 2001.

[R. 02] R. Hennicker, H. Hussmann, and M. Bidoit. Object Modeling with the OCL: The Rationale behind the Object Constraint Language. In *volume 2263 of LNCS. Springer*, 2002.

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

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

[Sof97] Rational Software. *UML semantics*, September 1997. http://www.rational.com/uml/html/semantics/.

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

[Sof02] Poet Software. Developing object oriented databases using POET. FastObjects web site - http://www.fastobjects.com, 2002.

[Sof03a] Objecteering Software. *Objecteering 5.2.2 Manual*, 2003. http://www.objecteering.com/.

[Sof03b] Poet Software. POET XML White Paper. Poet web site - http://www.poet.com/, 2003.

[SPH+01] G. Suny, F. Pennaneac, W. Ho, A. Guennec, and H. Jzquel. Using UML Action Semantics for Executable Modeling and Beyond. In *CAiSE 2001, LNCS 2068*, 2001.

[Str94] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Pub. Co., 1st edition, March 1994.

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

[TCKI00] M. Tatsubori, S. Chiba, M. Killijian, and K. Itano. A Class-based Macro System for Java. In *Reflection and Software Engineering, LNCS 1826, Springer Verlag*, 2000.

[W3C00] Org. W3C. Extensible Markup Language XML, Version 1.0 sec. ed., W3C Recommendation. http://www.w3c.org, October 2000.

[WK98] J. Warmer and A. Kleppe. *The Object Constraint Language : Precise Modeling with UML*. Addison-Wesley Pub. Co., 1st edition, 1998.

[WS98] I. Welch and R. Stround. Dalang - a Reflexive Java Extension, PWRP'98. Technical Report UTCCP Report 98-4, University of Tsukuba, Japan, October 1998.

[Wu98] Z. Wu. Reflexive Java and a Reflexive Component-based Transaction Architecture. Technical Report UTCCP Report 98-4, University of Tsukuba, Japan, October 1998.