

MQL: a Powerful Extension to OCL for MOF Queries

David Hearnden
School of ITEE,
University of Queensland.
hearnden@dstc.edu.au

Kerry Raymond
CRC for Enterprise Distributed
Systems (DSTC)
kerry@dstc.edu.au

Jim Steel
CRC for Enterprise Distributed
Systems (DSTC)
steel@dstc.edu.au

Abstract

The Meta-Object Facility (MOF) provides a standardised framework for object-oriented models. An instance of a MOF model contains objects and links whose interfaces are entirely derived from that model. Information contained in these objects can be accessed directly, however, in order to realise the Model-Driven Architecture™ (MDA), we must have a mechanism for representing and evaluating structured queries on these instances.

The MOF Query Language (MQL) is a language that extends the UML's Object Constraint Language (OCL) to provide more expressive power, such as higher-order queries, parametric polymorphism and argument polymorphism. Not only do these features allow more powerful queries, but they also encourage a greater degree of modularisation and re-use, resulting in faster prototyping and facilitating automated integrity analysis.

This paper presents an overview of the motivations for developing MQL and also discusses its abstract syntax, presented as a MOF model, and its semantics.

1 Introduction

The Model-Driven-Architecture™ (MDA) [1] requires all its meta-models to be defined in the Meta-Object Facility (MOF) [2]. These meta-models include the Unified Modelling Language (UML), the Common Warehouse Metamodel (CWM) and the MOF itself. Since the MOF plays such a central role in the MDA, it is essential that information in MOF models can be accessed and manipulated by a standard mechanism. Currently, the *interfaces* for an instance of a model are standardised by mappings to OMG IDL. The OMG has standard mappings from IDL to several languages, such as Ada, C, C++, Java, Python and LISP. This means that any instance of a MOF model can be accessed programmatically in a standardised way, enabling interoperability.

However there is no standardised syntax (concrete or abstract) for representing *queries*. The closest thing is a somewhat backdoor mechanism of post-condition constraints defined on operations; however this is not a conceptually satisfactory solution.

Constraints are typically written in the Object Constraint Language (OCL), which currently has only a standardised concrete syntax and no formal semantics. Also, OCL is defined as part of the UML 1.4 standard [3], not the MOF standard. Moreover, OCL constraints on MOF models are not explicitly *modelled*, but rather represented using a string-valued attribute containing the OCL constraint in a textual form!

Currently, any structured queries on data or metadata must be done programmatically. Clearly, to achieve the consistency and interoperability that benefits the MDA, there must be a standard mechanism for representing queries.

1.1 Existing Work

Many technologies have successful and powerful query mechanisms that are widely known, understood and used. The Structured Query Language (SQL), adopted as an industry standard in 1986, is a very successful language for relational databases. More recently, SQL-99 [4] has introduced object-oriented concepts into the language. However, there are significant differences between the object models of the MOF and of object-relational databases (SQL-99 is also restricted to using only linear recursion).

Since MOF models and instances can be mapped to XML documents (XMI [5]), an XML query mechanism can be easily integrated with MOF technology. XQuery [6] and XPath [7], standardized by the W3C, are some of the many query languages for XML documents. The problem of querying in the MOF can be reduced to an already-solved problem of how to query XML documents. However, the lack of object-orientation (such as inheritance or polymorphism) in XML would constrain the expressive power of an XML-based query approach.

The Object Query Language (OQL) is a query language based on SQL defined by the Object Database Management Group (ODMG) as part of the Object Data Standard [8]. However, the standard does not define the language's abstract syntax nor formal semantics.

The Object Constraint Language (OCL) was originally defined in the UML 1.4 standard. Although only a concrete syntax, an abstract syntax and formal algebra for OCL 2.0 (in UML 2.0) is being finalised in the OMG.

This model of OCL can be very easily integrated with the MOF, and since OCL is already a familiar language for expressing constraints, could be an ideal basis for MOF-based queries. For the remainder of this document, the term OCL shall mean OCL 2.0, not the current OCL standard in the UML 1.4 specification.

1.2 Aims

This paper outlines the results of the MQL project [10], whose aims were:

- to create a *platform-independent model* (PIM) defined in the MOF for querying the MOF,
- to create a concrete syntax for the language described by the PIM,
- to formally define the semantics of this language, and
- to create an interpreter to evaluate queries written in this language.

The MOF Query Language (MQL) was created to meet these goals, and a prototype interpreter has also been written that enables the definition and evaluation of MQL queries.

1.3 Structure of this paper

Section 2 outlines the requirements for a MOF query language that motivated the development of MQL. Section 3 presents the MQL abstract syntax as a MOF model, describing the extensions made to the OCL abstract syntax. Section 4 gives an overview of the semantic extensions needed to achieve the requirements for MQL. Some more MOF-specific extensions are covered in Section 5. Section 6 discusses issues raised through the development of MQL, Section 8 outlines some future work to be done, and conclusions are made in Section 9.

2 Architecture

Before queries can be explored in more detail, the purpose and the role of queries must first be established. Specifically, the architecture of queries must be decided based on MDA principles.

According to the MDA, relations between models should be defined on the same level of abstraction as the definition of the meta-models. Queries can be thought of as a relation from a model to itself, and therefore queries should be defined on the same level as the meta-model definition. In the context of the MOF, this means that queries on a given MOF model must be defined on the same level as that MOF model. This implies that the MOF itself must have facilities for describing queries.

For example, Table 1 describes a simple model and a query in MOF. In this context, MOF is being used as a meta-model. It can be clearly seen that queries must be defined on the same meta-level as the model, and queries can only be evaluated one meta-level lower.

Table 1. Queries and meta-levels.

Meta-model	<i>Hard-wired meta-model (MOF)</i>	
Model	Class ("Student", [Attr("name", string) Attr("age", integer)])	Query ("Under18", ...)
Instance	Student("Jane", 24) Student("Bob", 17) Student("Tom", 30)	Under18 (Student("Bob", 17))

Making conceptual distinctions between the three layers (meta-model, model and instance) allows some commonly used terms to be clarified:

- *query meta-model* refers to the model that exists at the same meta-level as the MOF model. The constructs defined in this model are used to define queries.
- *query* and *query model* refer to the specification of a query. A query model uses concepts defined in the query meta-model.
- The information at the instance meta-level is referred to as the *result* or *value* of a query.

The architecture of queries has now been established.

- There needs to be a query meta-model that is defined on the same meta-level as the MOF itself.
- Queries are defined on some source model at the same meta-level, using meta-model constructs.
- Queries are evaluated (and possibly instantiated) on an instance of that source model, at the same level as the instance.

2.1 Why MOF is different – object orientation

The MOF is an object-oriented framework. Therefore the modelling concepts used in the MOF exhibit the three principle features of object-orientation – encapsulation, inheritance and polymorphism. These features are essential for the design of MOF models, and querying in the MOF must incorporate these features.

Encapsulation allows information to be structured, and thus it should be the basis of navigation. Starting with an entity (e.g. an object), navigation to its features should be very straightforward, and references should be used to navigate across associations. This is the approach adopted by OCL.

Inheritance is arguably the most useful modelling technique that object-orientation provides. Inheritance

introduces the principle of *substitutability*, which is the ability of a subtype to be used in all places that any of its supertypes may be used. Therefore, queries should exhibit substitutability; queries that operate on a class must also operate on all subtypes of that class.

Polymorphism (in an object-oriented context) refers to the ability of a subtype to alter the behaviour of an inherited feature. This ability is most naturally realised by dynamic binding, where the decision of which feature to use is delayed until run-time. Polymorphism is not directly mentioned in the MOF specification, since it is an implementation feature, not part of a specification. However, in the MOF it is assumed that a subtype can override the implementation of inherited features.

Dynamic binding is a characteristic of evaluation (or execution), and the result (execution) of a query must be described by its language semantics. Therefore, the semantics of any MOF query language must define polymorphic behaviour.

Not only must a MOF query language have a strong object-oriented foundation, but it must also align with the specific characteristics of MOF models. This means it must operate on MOF *objects* and MOF *links*, and also incorporate definitive characteristics of MOF objects such as reflection and coupling with meta-objects.

This leads to the conclusion that traditional query mechanisms do not have much to offer except on a high level. On this argument, MQL was based on OCL. The alignment between UML and the MOF makes OCL an ideal platform for extension, and OCL also has a strong mathematical formalisation, which is necessary as a foundation for any sound analytic techniques.

2.2 Motivating Example

An example MOF model is shown in Figure 1. Reference names have been omitted for brevity.

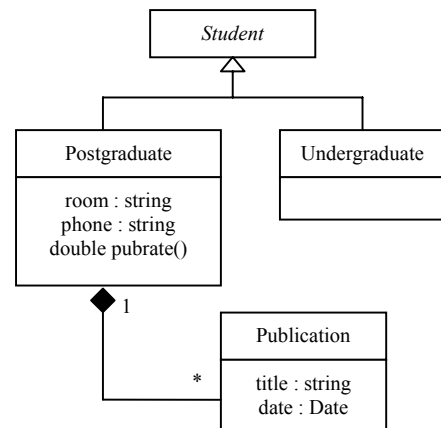
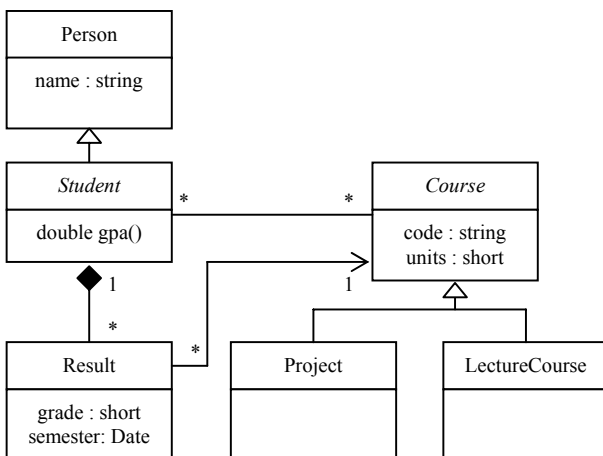


Figure 1. A sample MOF model.

The model describes a universe of Students and Courses with an enrolment relationship between them. Students have a past history of Results, from which a grade-point average (gpa) can be calculated. Postgraduate students also have a publication rate score. A Course is either a Project or a LectureCourse, and Postgraduates cannot enroll in Project courses.

Some example queries on such a model are:

- List all students in decreasing order of *rank*, where a student's rank is their gpa. For equal-rank students, list in the lexicographic order of their names.
- Repeat the previous query, except a Postgraduate's rank is the average of their gpa and their publication rate.
- List all students eligible to tutor a particular course, with eligibility determined by:
 - Postgraduates can tutor courses they have completed.
 - Undergraduates can tutor courses for which they achieved a grade >90%.
- List all the (Undergraduate, Project) enrolments.

The first query is essentially higher-order, since the query defines an ordering query (*rank*) that needs to be referred to by a sort query. However OCL does not have higher-order queries. The second query depends on the rank query being polymorphic, however OCL is not polymorphic. The final two queries can be written in OCL, however, as we shall see, some extensions to the language allow them to be expressed in a more intellectually satisfying manner.

2.3 Query style - functional queries

Two of the most common query styles are *selection* and *collection*. Selection is the process of selecting, from some input collection, elements that meet some criteria. Collection is the process of applying, to some input collection, some function, operation or transformation.

These two query styles are the way humans naturally express questions. For example, ‘Who are the students enrolled in course CS181?’ is clearly a selection operation, and ‘What is the age distribution of postgraduate students?’ is clearly a collection operation.

It is worth noting that a simple SQL query is the combination of a selection and collection. A general description of a simple SQL query can be written as:

```
SELECT f(x) FROM x WHERE p(x)
```

The query is a selection of all elements from an input collection x for which predicate $p(x)$ is true. To these remaining elements from x , the query *applies* the function $f(x)$, thus being a collection.

These two query styles are inherently *functional* in nature. Each of these operations requires two parameters – an input collection and a function. Since they need a function as an argument, they are higher-order. Selection and collection are very simply described by the functionals `filter` and `map` respectively, defined in Haskell [11] in Figure 2. `filter` can be described recursively as follows: if the predicate is true for the head of the collection, then the result is the head *cons*-ed with the filtered tail; otherwise the result is just the filtered tail. `map` can also be described recursively as follows: to map a function onto a collection, simply apply the function to the head of the collection and then recursively apply `map` to the tail of the collection.

```
filter p [] = []
filter p (h:t) =
  | p h = h:filter p t
  | otherwise = filter p t

map f [] = []
map f (h:t) = f h : map f t
```

Figure 2. filter and map defined in Haskell.

This leads to the hypothesis that queries are inherently functional in nature. Indeed, typical OCL expressions make heavy use of the collection operations `select` and `collect`. In fact, all OCL collection operations have simple functional equivalents. This follows immediately from the fact that, as part of the OCL semantics, all OCL collection operations must have a mapping to OCL’s `iterate` operation, and the `iterate` operation is equivalent to a left-fold function (`foldl`). That is, the OCL expression:

```
myCollection->iterate(x; acc=init | exp)
```

is equivalent to the function application:

```
foldl exp init myCollection
```

where left-fold is defined in Haskell in Figure 3 and can be expanded as shown in Figure 4.

```
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Figure 3. foldl defined in Haskell.

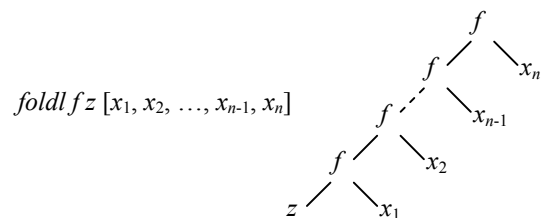


Figure 4. A left-fold shown as a tree.

OCL therefore has a functional flavour, but it is not functional. OCL does not treat functions (operations) as data values that are assignable to variables or passable as parameters. The power of OCL rests in the `iterate` operation, since this is the only primitive operation that exists on collections – all others are defined in terms of `iterate`. Therefore OCL is constrained by the fact that collection operations must be expressed in terms of `iterate`.

Indeed, most useful collection functions (such as `map` and `filter`) can be reduced and defined in terms of a left-fold; however there are other useful functions that cannot. Higher-order functions that use functions of more than one argument cannot be mapped to the `iterate` operation. A simple example is a `sort` function that requires, as a parameter, a comparison function of two arguments. There is no way to describe such a function in OCL, since OCL cannot treat functions as first-order values.

By breaking OCL’s `iterate` primitive into more low-level operations (such as `head` and `tail`, or `at`), and by allowing functions to be treated as first-order values, the query meta-model more closely matches the true functional nature of queries. It also allows higher-order functions and operations to be modelled explicitly, rather than depending on an *inbuilt* higher-order `iterate` function.

2.4 Functions and Operations

Since queries are naturally functional, it makes sense that a query should simply be a function. Therefore, a query may have input parameters, has a single returned result, and is not contained by any model element.

However, MOF models do not use the concept of functions. Instead, MOF models use operations. To query the MOF, we therefore need the ability to *define*

operations as well as queries. This approach gives MQL a strong object-oriented flavour, since operations are the heart of MQL queries and they are defined in an object-centric manner.

The difference between an operation and a function is subtle. An operation ω on a class-type t_c with parameter types $t_1 \dots t_n$ can be modelled by an equivalent function f with parameter types $t_c, t_1 \dots t_n$, i.e. the source object of an operation becomes an extra parameter in its equivalent function. The difference between operation ω and its equivalent function f is that ω has containment semantics (i.e. it is contained by class c), whereas f does not.

An operation can therefore be considered to be behaviourally equivalent to a function, and thus the terms ‘operation’ and ‘function’, when used in a behavioural context, may be used interchangeably.

2.5 Partial Evaluation

One of the characteristics of a functional language is the ability to create a new function by partially evaluating another. For example, $(+ 2)$ is a partial evaluation of the $+: \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R}$ function, and is a new, distinct function. MQL does not support this ability directly; however partial evaluation can be achieved through anonymous operations. For example, the equivalent of $(+ 2)$ would be a function $(\lambda x \bullet x + 2)$

So far, we have established that queries should be written in a functional style, with navigation being driven by encapsulation. This closely matches the style of OCL, and so OCL seems to be ideal for a basis for MQL.

2.6 Types in MQL

One of the benefits of a query model is that queries can be formally analysed. Proving correctness, deriving efficiency bounds, optimisations etc are all methods of analysis that are available once a formal query model is established. Clearly, analysis will be more effective (both easier and more efficient) with a more constrained model; however, as a consequence, queries become less flexible. A balance must be chosen between usefulness and generality.

A language’s type system is a prime example of this balance. Types essentially provide constraints within a language in order to prevent run-time errors. Strongly-typed languages (such as Pascal, Java) are often accused of being too constrained and not having the flexibility of weakly-typed languages (C, C++) and untyped languages (LISP, Prolog). Conversely, untyped languages are often more prone to error than typed languages because of the limited static analysis that can be performed.

Functional languages (such as Haskell and ML) are examples of languages with effective type-systems. They

are strongly-typed languages, thus preventing the vast majority of run-time errors, yet they are still highly flexible, and the amount of type information input by the programmer is minimal; types are derived through type inference.

The Hindley-Milner type system [12], the type system used in Haskell and ML, has an inference algorithm that allows the *principal* (i.e. the most general) type of any expression to be derived. It also turns out that the principal type has the smallest representation. However, the Hindley-Milner type system is not object-oriented. A number of attempts have been made to include type inheritance into this system (such as [13]); however these attempts usually result in the principal type containing much extra, often unused, information. This information describes all the extra situations that may arise with the existence of substitutable subtypes, and adds considerable complexity to principal types.

This leads to the conclusion that if a MOF query language is to be statically typed, then either the type information must be derived through type inference (yielding complex, convoluted types), or type information must be constrained in the language through type declarations. Since the MOF is an inherently typed universe (attributes, references, parameters etc all have types), it makes sense that MQL should be typed in some way.

OCL requires type declarations where an entity’s type would be otherwise unknown. However, since all compound expressions (expressions built from other expressions) have strong typing rules in OCL, the only unknown types arise from primitive expressions. Primitive expressions either refer to existing model elements (which have explicit or derivable types) or uninitialised variables or parameters. Uninitialised variables are meaningless in OCL; therefore programmer-input type information is only required for parameters. The task of assigning types to parameters is common practice in many programming languages, and is a small price to pay for the benefits of strong type checking.

Types in MQL are therefore declared where necessary (i.e. for parameters in operation signatures and query declarations), and all expressions have type inference rules, which allows strict and precise determination of their result type.

2.7 Higher-Order Queries

By choosing a functional style for querying, this immediately opens the door for higher-order queries and functions. Higher-order queries allow more generic queries to be written, promoting modular design and query re-use. A higher-order function (also called a *functional*) is simply a function that has a parameter that is also a function.

We have already seen a number of second-order functions (such as `map` and `filter`) that take a function of one argument. A useful second-order function that takes a function of two arguments is a `sort` function. A sort function needs a predicate that determines if two elements in a list are in the correct order. The sort function itself does not need to know the details of this predicate; it merely needs to call it. Such a function is easily written in a functional style, as seen in Figure 5.

```
insertionsort p [] = []
insertionsort p (h:t) =
    insert p h (insertionsort p t)

insert p e [] = [h]
insert p e (h:t) =
    | p e h      = e:(h:t)
    | otherwise = h:(insert p e t)
```

Figure 5. An insertion-sort written in Haskell.

Higher-order functions provide a great deal of abstraction, which allows re-useable queries to be written on a very high level.

MQL allows the use of higher-order queries by treating functions/operations as first-class entities that can be assigned to variables and passed as parameters.

2.8 Polymorphism

Polymorphism enhances a language's flexibility and promotes modular design. The exact meaning of the term varies across paradigms, but in general refers to the ability of a single entity to alter its behaviour or form based on context. In an object-oriented framework, polymorphism specifically refers to the ability of the *behaviour* of a supertype to be modified by a subtype. Typically, this refers to a method. This allows a generic description of something involving supertypes (e.g. an operation) to be refined *outside* that description (i.e. inside the subtype), thus promoting re-use and modularity.

Polymorphism is an important component of object orientation, and thus also for an object-oriented query mechanism. Polymorphism is typically implemented through dynamic binding - the binding of an operation call to a specific operation is not done until run-time, and the binding is made on the basis of the run-time types of the entities involved.

Polymorphism poses an obstacle for a query model. The essence of polymorphism is that an operation call must be considered to be statically non-deterministic, but dynamically deterministic. In other words, the operation that is actually called is undetermined at compile time, and only determined at run-time. Since a query model can only represent static information (dynamic

information can only be produced through evaluation), there is a real problem of how to model polymorphism. The three logical options for modelling an operation call are:

- do not include binding information at all,
- include only a single static (monomorphic) binding, or
- include all possible bindings.

The first option would seem to be the most natural choice, since polymorphism is associated with a sense of freedom and dynamic (rather than static) behaviour. The operation to be called must be found at run-time, involving some form of search. Apart from the significant performance cost this would incur, this solution is not very useful from a modelling perspective, since there is no binding information available at all, and if this information is required for query analysis (such as type-checking) then it must be derived when needed.

The second option is the approach adopted by the OCL meta-model (which has a single binding between an *OperationCallExp* and an *Operation*). This is possibly the simplest solution, but again, if information on run-time behaviour is necessary (e.g. for proving correctness) then the other possible bindings must be derived.

The third option involves representing all possible bindings that could be made, and then at run-time selecting the 'best' operation based on some deterministic rules. This solution captures the most information, and thus is the preferred option in the modelling sense; however this comes at the price of being able to compute all possible bindings. This necessarily implies that all possible bindings are known at parse-time (i.e. the target model exists in a *closed world*).

MQL adopts the third option, considering the modelling advantages to outweigh the computation and closed-world disadvantages.

2.9 Parametric Polymorphism

Parametric polymorphism is a technique that permits the use of entities with unknown types. These entities are typed by *type variables*, also known as *generics* or *templates types*. This allows the actual type of an entity to be abstracted away when not important, again promoting re-usable queries and modularisation. This feature already exists in many languages, such as Ada (generics) and C++ (templates).

Collection operations are an example that illustrates the benefit and importance of parametric polymorphism. Collections often require operations that are behaviourally invariant with respect to the *type* of elements within the collection (such as *select*, *collect*, *head*, *tail* etc). Since collections play an important part in queries, then operations to manipulate collections become quite important. There is no sound method in the MOF

type-system to *generically* model these important collection operations. The reason for this is that all collections should have certain operations (such as *select*, *collect* etc), but since each collection is modelled as a separate type, then either these operations have to be replicated for each collection or all collections need to inherit some general collection. The latter solution would involve having a collection of some general type (e.g. a CORBA Any), however this reduces the type information held in a query model, since the return type of collection operations becomes generic and can only be derived by semantic information defined for each collection operation. Since this information is not easily modelled, this is not an ideal solution.

The former of the two solutions (replicating generic operations for each collection type) is the solution adopted by OCL. It would be preferable if these operations could be modelled as a single instance, as can be done with parametric polymorphism. By defining these operations on some *generic* type (i.e. a collection of unknown type), the specific type of the elements can be abstracted away, and the information in the semantic rules above can be modelled explicitly. A type (constructed or not) that contains type variables is called a *generic* type, and a type with no type variables is called a *simple* type.

The cost of parametric polymorphism is the introduction of generics into the type system and also a *generic-matching* process. At parse-time, when an operation of a generic type T is used on an entity of type R, the real type R must be matched with the generic type T to create a set of *type-variable substitutions*, such that all type variables in T can be substituted for real types from R. This matching process is a unification problem in a type system with inheritance, as illustrated in Table 2, where Student inherits Person, Integer inherits Real, and Set(T) and Bag(T) inherit Collection(T).

Table 2. Unification of generic types

<i>Generic type</i>	<i>Simple type</i>	<i>Bindings</i>
Set(T)	Set(Student)	T=Student
Operation(T, T)	Operation (Postgraduate, Student)	T=Student
Operation (Collection(T), T)	Operation (Set (Bag (Real)), Bag (Integer))	T= Bag (Real)

MQL supports parametric polymorphism by including generic types in its type system. Operations can be defined on generic types and generic types can be used to make constructed types.

2.10 Argument Polymorphism

In the traditional object-oriented form of polymorphism, a subtype can *redefine* (or *override*) features that exist in one of its supertypes. The feature that is used at a particular time is decided based on the run-time type of the source object.

The benefit of polymorphism is that subtypes may have different behaviour for a given operation, and this behaviour is defined within the subtype. However, sometimes it makes more sense for the type-specific behaviour to not be associated with the specialised types, but rather with the operation itself. For example, consider adding a `canBeTutoredBy(Person)` operation to the Course class. The university may impose eligibility criteria that are specialised for particular types of people. The specialized criteria, although it exists because of the subtypes of Person, is not something that Person objects should be concerned with. Rather, the specialised rules make more sense if they exist within the Course class. Although this can be achieved by testing the run-time type of the Person parameter within the operation itself, it would be better if the binding mechanism could do it instead, as illustrated in Figure 6. This creates a more declarative style for specialising behaviour based on parameter subtypes. It also means that subclasses of Course may define their own specialised criteria for specialized argument subtypes. For example, a Project might add an operation for an argument of type Professional (subtype of Person).

```
Course::canBeTutoredBy(Person p)
Course::canBeTutoredBy(Undergraduate l)
Course::canBeTutoredBy(Postgraduate s)
```

Figure 6. Specialising behaviour based on parameter subtypes.

MQL supports this idea by using the concept of dynamic binding based on *all* the run-time types of arguments (prioritised left to right), rather than just the first. This extends the traditional notion of polymorphism, where dynamic binding is based solely on the source object's run-time type, to achieve *argument polymorphism*. This allows operations within the same class to *specialise* their behaviour based on the types of their arguments. This tends slightly towards the style of logic programming.

3 MQL meta-model

In order to achieve the advanced query techniques discussed in Section 2, the OCL 2.0 and MOF 1.4 meta-models were extended. Migration to a MOF 2.0 universe

should be largely trivial. The additions to the abstract syntax are discussed first, followed by additions and modifications to the semantics. Note that containment relationships are modelled by inheriting *Namespace* rather than introducing new composite-end associations.

3.1 Extensions to MOF Core

Figure 7 illustrates the two most significant additions to MOF meta-classes: *TemplateType* and *OperationType*.

The *TemplateType* class represents a template (or generic) type. It is used as a type variable, which provides the basis for parametric polymorphism.

The *OperationType* class represents a functional/operational type. It contains (via the Contains association inherited from *Namespace*) a number of Parameters. There is an optional parameter with a direction of *context_dir*, representing a source object for an operation (the presence of this parameter distinguishes a function from an operation). There is also a single return Parameter and other input/output Parameters.

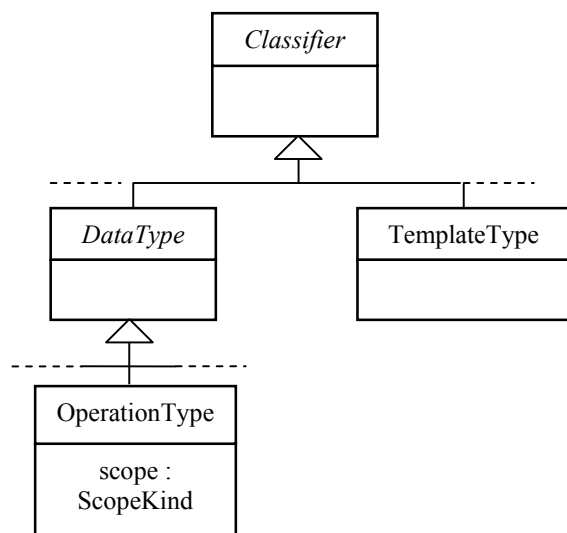


Figure 7. Meta-classes added to MOF Core

3.2 Extensions to OCL Expressions

Figure 8 illustrates the most significant extensions to the OCL Expressions package.

FeatureExp represents a reference to Features of the source model. It refers to a set of features, called the *feature-set*, from which a single feature is chosen at run-time for binding.

QueryExp represents a query call. It has a reference to a set of queries called the *query-set*. One of the queries in the query-set is chosen at run-time for binding. A

QueryExp also contains Expressions that act as arguments passed to the query.

An *ApplicationExp* models the *application* of some feature to some arguments, in parallel with the lambda-calculus concept of the application of a function to some arguments. However to achieve polymorphism, an *ApplicationExp* models the application of a *set* of features, from which one feature is chosen at evaluation-time.

An *ApplicationExp* contains an ordered list of Expressions; the first Expression evaluates to a function-set, and the remaining Expressions evaluate to the arguments for the function-set. If the function-set is a set of operations, then the second Expression evaluates to the source-object. Otherwise, if the function-set contains queries, then the second Expression is simply the first argument to the query.

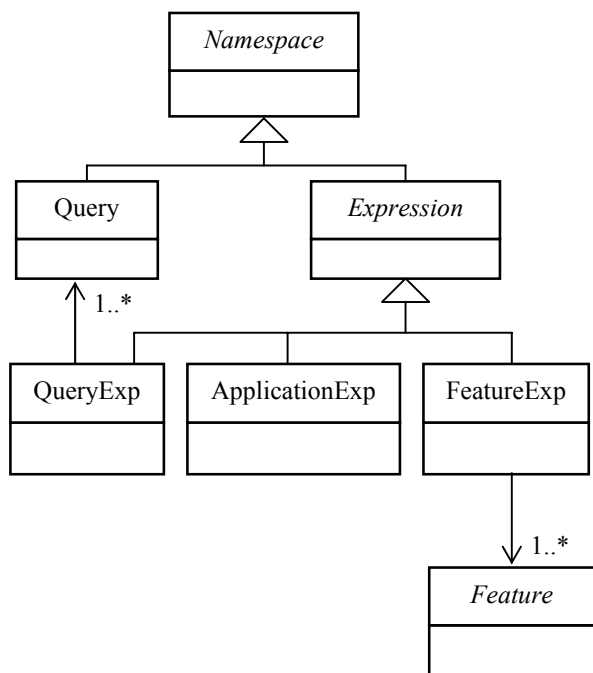


Figure 8. Meta-classes added to OCL Expressions

An example of an *ApplicationExp* and a *FeatureExp* is shown in Figure 9. The figure represents an expression written against a model of Students and Courses. The Student class defines an operation *canEnrollIn(Course)*, that tests if a given student can enroll in a given course. The enrolment rules are different for postgraduate student and undergraduate students, and so this operation has been specialised. The expression in Figure 9 is *s.canEnrollIn(c)*, where *s* is a variable of type Student and *c* is a variable of type Course. The *FeatureExp* has

references to all possible polymorphic bindings of the canEnrollIn operation.

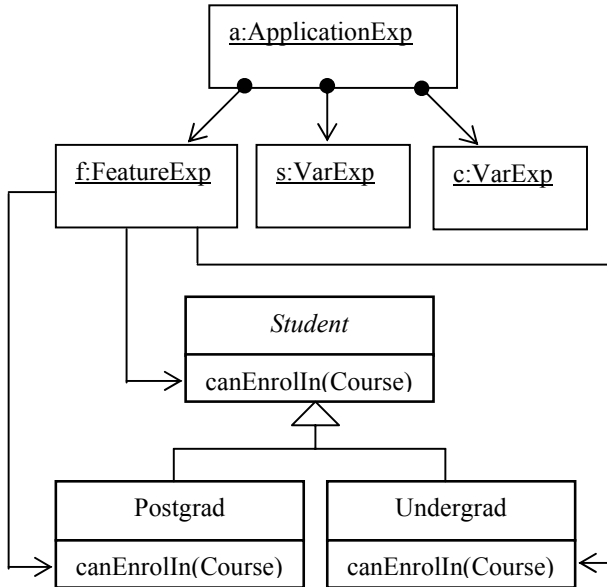


Figure 9. An example feature application.

Figure 10 shows the new Literal meta-classes: OperationLiteral and ClassifierLiteral. An OperationLiteral represents an anonymous (un-named) operation on some Classifier. It contains Parameters, VariableDeclarations and an Expression that defines the operation. An anonymous operation allows the arbitrary definition of an operation-value, analogous to other Literals for non-functional values.

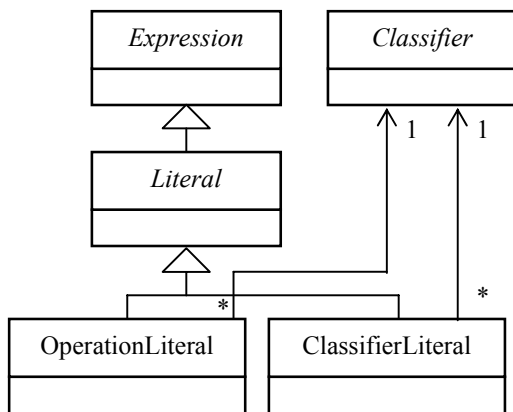


Figure 10. Literal meta-classes added to MQL.

In MQL, classifier-level features are accessed by applying them to a type-expression, which is built from ClassifierLiterals. For example, the MQL standard

library defines an operation allInstances() that, when called on a classifier, returns the collection of all the instances of that classifier.

3.3 Extensions to OCL Values

In OCL, references can be made to values before and after an operation. This requires values to have some form of temporal semantics. MQL does not currently have any synchronization semantics; it is assumed that a query is performed in a single transaction, and a query can not make any state changes. As such, there is no need for OCL's notion of snapshots and histories. This allows MQL to greatly simplify the Values package.

A FeatureValue has a reference to a set of Features, such that a single feature is chosen from this set whenever it is to be used.

A QueryValue represents a query. As with a FeatureValue, it contains references to a number of Query objects, and a single reference is chosen for binding at run-time.

A ClassifierValue is used to represent the value of type expressions.

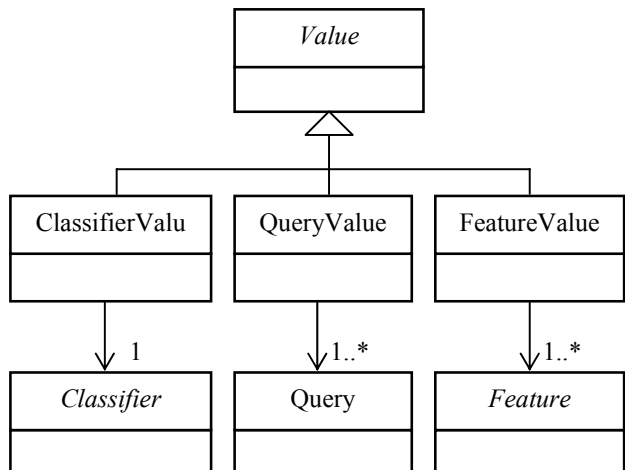


Figure 11. New Values meta-classes.

4 Semantics

The semantics described here use an object model of the MOF that is almost identical to the object model for UML defined in OCL. Important definitions include the set of all un-constructed types, T , the set of type-expressions (including constructed types), $T_{Expr}(T)$, the set of expressions of type t , $Expr_t$, the interpretation of a type, $I(T)$ and the interpretation of an expression e in an environment τ , $I[e](\tau)$. τ includes all variable bindings and object state information, and in MQL also includes

type-variable bindings. For more details, please refer to the Semantics chapter of the OCL document [9].

MQL modifies OCL's semantics in order to include higher-order operations, parametric polymorphism and argument polymorphism.

Higher-order operations are achieved by allowing functions and operations to be considered first-class entities, meaning that an expression can evaluate to a function, a function can be assigned to variables and a variable can be used to make a function call.

Parametric polymorphism is achieved by including type variables in the type system and by replacing the type-hierarchy operator \leq with a type-hierarchy unification operator \leq_ρ , where ρ is a set of type-variable

bindings. For two types t_1 and t_2 , we say $t_1 \leq_\rho t_2$ if and

only if t_1 matches t_2 with most general unifier ρ . If t_1 and t_2 are simple types (i.e. they have no type variables) then ρ is empty.

MQL supports polymorphism by defining rules for dynamic feature selection. Whenever a query or feature is applied, an MQL evaluator must select the 'best' match from the feature-set. The rules for selecting the best match have been defined so that MQL's form of polymorphism extends the traditional sense of object-oriented polymorphism to achieve *argument polymorphism*, where operations can specialize their behaviour based on the run-time types of their arguments.

4.1 Higher-order operations

Higher-order operations are achieved by treating functions as first-order entities and by modeling function application as the combination of a function-valued term and argument terms, rather than as the combination of an explicit functor and argument terms. This allows variables to be function-valued and allows functions to be passed as parameters to other functions.

MQL makes three extensions to OCL semantics in order to achieve this: the introduction of *operation-types*, the introduction of *lambda-terms* and the replacement of property calls with *application terms*.

An *operation-type* is built from existing types, and thus exists within the set of type expressions. An operation-type $\text{Operation}(t_1, \dots, t_n, t_r)$ represents the type of a function from types t_1, \dots, t_n to t_r , and maps to the OperationType meta-class.

$$t_1, \dots, t_n, t_r \in T_{\text{Expr}}(\text{T}) \Leftrightarrow \text{Operation}(t_1, \dots, t_n, t_r) \in T_{\text{Expr}}(\text{T})$$

$$I(\text{Operation}(t_1, \dots, t_n, t_r)) = I(t_1) \times I(t_2) \times \dots \times I(t_n) \rightarrow I(t_r)$$

Lambda-terms are used to defined anonymous operations. A lambda-term $\lambda p_1 : t_1, \dots, p_n : t_n \cdot e$

represents a function with parameter-types t_1, \dots, t_n that is defined by the expression e , and maps to the OperationLiteral meta-class.

$$e \in \text{Expr}_t \wedge \forall i \in \{1..n\} \bullet p_i \in \text{Var}_{t_i}$$

$$\Leftrightarrow \lambda p_1 : t_1, \dots, p_n : t_n \bullet e \in \text{Expr}_t$$

$$I[\lambda p_1 : t_1, \dots, p_n : t_n \bullet e](\tau) = \{\omega\}$$

$$\text{where } \omega(p_1, \dots, p_n) = I[e](\tau')$$

In OCL, property-call semantics involve an operation ω being applied to argument expressions, $e_1 \dots e_n$. In MQL a property-call is replaced with a function application $e_f(e_1, \dots, e_n)$. An application-expression has a function-expression and argument expressions. The argument expressions are first evaluated to give a sequence of values, v_i , with their run-time types, τ_i . The function expression is then evaluated to give a function-set, F . The run-time types of the arguments are used to select the best-fit function from F , using the selectFunction function. This selection process returns a single operation ω with a set of type-variable substitutions, δ . If ω is undefined then there was no function in F that matched the run-time types of the arguments, and thus the result of the application is undefined. This only happens if the function-set F is not well formed. If ω is defined then it is evaluated in a new environment τ' by adding in variable bindings from ω 's parameters p_i to the argument values v_i , and also adding the type-variable bindings δ returned from the matching process in selectFunction .

$$e_f \in \text{Expr}_{\text{Operation}(t_1, \dots, t_n, t_r)} \wedge \forall i \in \{1..n\} \bullet e_i \in \text{Expr}_{t_i}$$

$$\Leftrightarrow e_f(e_1, e_2, \dots, e_n) \in \text{Expr}_{t_r}$$

$$I[e_f(e_1, e_2, \dots, e_n)](\tau) = I[\text{exp}(\omega)](\tau')$$

$$\text{where } \forall i \in \{1..n\} \bullet v_i = I[e_i](\tau) \wedge t_i = \text{typeof}(v_i)$$

$$\text{and } F = I[e_f](\tau)$$

$$\text{and } (\omega, \rho) = \text{selectFunction}(F, (t_1, \dots, t_n))$$

4.2 Type hierarchy

MQL replaces OCL's type hierarchy \leq with a hierarchy that includes matching against generics, \leq_ρ .

For two types t_1 and t_2 , we say $t_1 \leq_\rho t_2$ if and only if t_1

matches t_2 with most general unifier ρ .

$\rho : T_T \rightarrow \mathcal{P}(T)$ is a set of substitutions (i.e. a function) of type-variables for types. It is a little different from a normal unifier in that the each type variable has a *set* of substituting types, rather than a single one. This is

because, in some cases, there is no single substitution that is most general. This is a consequence of multiple inheritance.

MQL also defines a function $\uparrow(S)$ that takes a set of types, S , and returns the set of *most specific common conformees*, T . In other words, the set of types T such that every element of S conforms to every element of T , and there is no type u such that every element of S conforms to u and u conforms to any element of T .

Unifiers can be combined using the combination operator, $\rho_1 \circ \rho_2$. This infix operator, which is both commutative and associative, combines two most general unifiers into a single most general unifier. The combination of two unifiers is the union of all their mappings where the domains do not intersect, and the most specific common conformees of the mappings for when the domains do intersect.

4.3 Polymorphism

MQL extends OCL by incorporating polymorphism into the language semantics. This is achieved by working with operation sets rather than individual operations. A feature-expression has links to all features that could possibly be polymorphically bound, including features defined in supertypes and subtypes. When evaluated, the dynamic function selection of MQL will select the ‘best’ feature to use.

4.4 Dynamic Function Selection

In MQL, function-valued entities are modelled by a *set* of functions, each function being a possible polymorphic binding. When a function-set is applied to some arguments during evaluation, a single function from that set must be selected. This selection is equivalent to dynamic binding.

The `selectFunction` function takes a set of functions F and an ordered tuple of actual parameter types (t_1, \dots, t_n) , and selects the function whose signature is a best-fit for those types.

A non-generic function will always be preferred over a generic one. Thus the search for a best-fit must first search the non-generic functions of F , and then if no functions were found, search the generic functions of F .

The selection process uses a set of pairs, each pair being a function and the type-variable substitutions that would be necessary should that function be selected. This set of pairs is called F_i , and represents the set of functions that are still eligible for binding after looking at the parameter types of all the i th parameters. Thus F_0 is the set of all functions of F that match the run-time types t_1, \dots, t_n . F_i is then defined as the set of functions from F_{i-1}

for which there are no other functions in F_{i-1} that have a more specific type for their i th parameter.

This is almost saying that the functions of F_i are the functions whose i th parameter type is one of the minimal elements (under \leq_ρ) of all the i th parameter types of F_{i-1} .

However, since \leq_ρ is only a partial order, we must instead use those elements for which there is nothing greater.

In a well-formed model, the final set of best-fit functions F_n should be a singleton set, thus allowing `selectFunction` to select any (i.e. the only) element from it. However it is possible that in a complex or poorly designed model, it may contain more than one function, resulting in an ambiguity in the selection process. In this rare case, MQL assumes that the behaviour of those operations in F_n should not be substantially different, and thus selects any operation from that set. An evaluation tool should output a warning that there was some non-determinism in the query.

The selection process could be adapted to take ordered inheritance into account, since the MOF Generalizes association is ordered. If this ordering were somehow incorporated into the type hierarchy, then it would be possible to *always* establish a single, best-fit function reference. However, this would presumably complicate the selection process quite significantly, and therefore MQL does not currently support this since the need for such a rigorous selection process is rare.

5 Other Extensions

MQL includes some other extensions that are made possible through the MQL standard library.

5.1 Meta-queries

One of the characteristics of the MOF is the coupling between an instance (at the M1 level) and its model (at the M2 level). This binding exists through the Reflective interfaces that every M1 level object implements. For example, every M1 object has the ability to navigate to its meta-object, i.e. the object in the model that describes its type.

By allowing queries to access the meta-model elements (i.e. the MOF model elements) that describe a particular entity’s type, queries can search for and use meta-information. Applications of this are

- Versioning (queries that have different behaviour for different versions of a source model). and
- Generic queries (covered in Section 5.2).

An example of a query that incorporates versioning is if a later version of a model has a different name for some

piece of information, for example: “*If this is version 1.6 of the University model then rank students according to grade-point average. If this is version 1.7 of the model, then rank students according to weighted-mean average.*”

If the version information is only contained in the University model (and not instances of it), then navigation to meta-data is the only way to implement this query. As a side note, model versioning is the subject of a separate Versioning and Development Lifecycle RFP for MOF 2.0 [14] issued by the OMG.

5.2 Generic Queries

In the MOF, all objects implement the Reflective interface. This interface allows reflective programming, which includes the dynamic discovery of an object’s attributes, references and associations. This allows *generic* code to be written that makes use of an object’s features without having *a priori* knowledge of those features. In MQL, all objects have access to operations in the reflective interface.

One of the problems with allowing access to reflective operations is that the specific types returned are unknown until run-time. This makes it impossible to model the actual types involved. The only solution is to introduce some general type (such as CORBA’s Any) to model the returned values. However, this is a consequence of using reflection, rather than a weakness in expressive power.

6 Discussion

Although MQL has achieved significant expressive power, there are two main factors that hinder the usability of MQL: its concrete syntax and its execution speed.

Preliminary feedback from users of MQL indicates that an SQL-style syntax would be preferable to the current OCL-style. This could be due to the large amount of syntactic sugar in SQL, thus making queries more human-friendly, or it could be that SQL is, for most system modellers, a more familiar syntax.

The execution speed of MQL queries in the current implementation is currently unacceptably slow. This is due to a number of reasons:

- the interpreter runs in a CORBA environment (thus requiring remote method invocation and object creation),
- the prototype interpreter is written in Java (thus incurring slow execution), and
- the interpreter has very few optimisations (such as caching).

The first three points cannot be attributed to the MQL language, but are rather faults of the MQL interpreter, and all of these faults can be remedied. Firstly, a non-CORBA implementation of the MOF standard (such as

the Java Metadata Interface (JMI) [15]) would remove the overhead associated with the distribution of objects, significantly improving performance. Secondly, a faster JVM would significantly enhance execution speed. Finally, simple optimization techniques such as caching and multithreading would significantly increase evaluation speed.

7 Revisited Examples

The example queries from 2.2 are now shown written in MQL’s concrete syntax, which is based heavily on the OCL syntax.

```
context Student
def: oper rank() = gpa()

context PostgraduateStudent
def: oper rank() = (gpa() + pubrate()) / 2

context Course
def: oper canBeTutoredBy(Person p) = false
def: oper canBeTutoredBy(Postgraduate s) =
    s.results->exists(r |
        r.course = this)
def: oper canBeTutoredBy(Undergraduate s) =
    s.results->exists(r |
        r.course = this
        and r.result > 0.9)

query rankedStudents =
    Student::allInstances()->sort(
        s1 : Student, s2 : Student |
        s1.rank() < s2.rank()
        or
        (s1.rank() = s2.rank()
        and s1.name < s2.name)

query possibleTutors(Course c) =
    Person::allInstances()->select(p |
        c.canBeTutoredBy(p))

query studentproject =
    Enrols::allLinks()->collect(t |
        t.student.type = Undergraduate
        and course.type = Project)
```

8 Future Work

Three areas for future work are proposed: pattern-matching, an SQL-like syntax and query tools.

MQL currently requires unification in order to achieve parametric polymorphism. Making unification available for use in expressions can, in many cases, result in more concise query definitions. It should be possible to extend MQL with a variety of expression classes for the sole purpose of pattern matching. These expressions would instantiate variables that could be used in other expressions, and could be Boolean-typed to indicate a successful or unsuccessful match.

A binding from the MQL abstract syntax to an SQL-like concrete syntax would make queries more usable for people with an SQL background. However, there may be problems with expressing more advanced queries (such as higher-order or parametrically polymorphic queries) in such a syntax.

Finally, tools to implement query optimization, query verification and code generation would be valuable additions to MQL.

9 Conclusion

The MQL project has achieved the goals listed in Section 1.2. These were:

- to create a PIM defined in the MOF for querying the MOF,
- to create a concrete syntax for the language described by the PIM,
- to formally define the semantics of this language, and
- to create an interpreter to evaluate queries written in this language.

A PIM (defined in MOF) was created to express queries in the MOF (Section 3) based on the design principles discussed in Section 2. An overview of the semantics of the MQL language were presented in Section 4. A prototype interpreter has been written in Java that allows the definition and evaluation of MQL queries.

10 Acknowledgements

The work reported in this paper has been funded in part by the Co-operative Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programs (Department of Education, Science and Training).

11 References

- [1]. "Model-Driven Architecture (MDA)", OMG document ormsc/2001-07-01, July 2001.
- [2]. "Meta-Object Facility (MOF) Specification", OMG document 02-04-03, April 2002.
- [3]. "OMG Unified Modeling Language Specification", Version 1.4, OMG document/01-09-67, September 2001.
- [4]. "SQL", ISO/IEC 9075:1999.
- [5]. "OMG XML Metadata Interchange (XMI) Specification", Version 1.2, OMG document/02-01-01, January 2002.
- [6]. "XQuery 1.0: An XML Query Language", <http://www.w3.org/TR/xquery>, November 2002.
- [7]. "XML Path Language (XPath) Version 1.0", <http://www.w3.org/TR/xpath>, November 1999.

- [8]. T. Attwood et al. "The Object database standard / ODMG-93", Morgan-Kaufmann, San Mateo, 1994.

- [9]. "Response to the UML 2.0 OCL RFP, Version 1.6", OMG document ad/03-01-07, January 2003.

- [10]. D. Hearnden, "Querying in the Model Driven Architecture™", Department of Information Technology and Electrical Engineering, University of Queensland, October 2002.

- [11]. S. Peyton Jones, J. Hughes et al, "Haskell 98: A Non-strict, Purely Functional Language", February 1999.

- [12]. R. Milner, "A Theory of Type Polymorphism in Programming", Journal of Computer and System Sciences, 17(3):348-375, December 1978.

- [13]. J. Nordlander, "Polymorphic subtyping in O'Haskell", *Proceedings of the APPSEM Workshop on Subtyping and Dependent Types in Programming*, Ponte de Lima, Portugal, 2000.

- [14]. "MOF 2.0 Versioning and Development Lifecycle RFP", OMG document ad/02-06-23, June 2002.

- [15]. "Java Metadata Interface (JMI) Specification", JSR 040, <http://www.jcp.org/aboutJava/communityprocess/review/jsr040/>