Studienarbeit

Query Infrastructure and OCL within the SAP Project “Modeling Infrastructure”

Erik Burger

eingereicht am 31.10.2006 beim
Institut für Theoretische Informatik
der Universität Karlsruhe

Referent: Prof. Dr. Peter H. Schmitt
Betreuer: Dr. Stefan Kusterer, SAP AG

Studienanschrift:
Erzbergerstraße 10
76133 Karlsruhe
E-Mail: Erik.Burger@ira.uka.de
Contents

1 Introduction 5

2 Background 6
   2.1 Meta Object Facility (MOF)......................................................... 6
   2.2 Object Constraint Language (OCL).............................................. 7
   2.3 Java Metadata Interface (JMI).................................................... 9
   2.4 SAP’s Modeling Infrastructure (MOIN)........................................ 9

3 Principles of Mapping OCL to MQL 11
   3.1 Expressional Powers of OCL and MQL ........................................ 11
   3.2 Mapping MQL to OCL by example............................................... 11
   3.3 Mapping OCL to MQL................................................................. 18

4 Implementation 25
   4.1 Structure of OCL evaluation..................................................... 25
   4.2 Preparation................................................................. 25
   4.3 Optimizations............................................................... 28
   4.4 Evaluation of complex OCL expressions..................................... 30

5 Conclusion 31
   5.1 Alternate solutions............................................................. 31
   5.2 Related work................................................................. 31
   5.3 Future work/estimate of practicability....................................... 32

Glossary 33

Appendices

A MOIN Query Language syntax 36

B MQL mappings of MOF Constraints 39
List of Figures

1. An excerpt of the MOF model showing its key elements ........................................ 7
2. The OCL Expressions Package .............................................................................. 8
3. Small sample Metamodel ....................................................................................... 12
4. Categorisation of OCL expressions (from [DH99]) ............................................... 18
5. Optimization scheme .............................................................................................. 28
1 Introduction

In the course of the Modeling Infrastructure (MOIN) project at SAP\textsuperscript{1}, several components are being implemented, which together form a base for software modeling tools in Java. One of these components is a meta-model repository based on the Meta Object Facility (MOF). It features a query infrastructure with which model elements can be retrieved from the repository. The API of this query infrastructure is described in an abstract syntax, the MOIN Query Language (MQL).

The Object Constraint Language (OCL) represents a language with high expressional powers, which is also a feature of MOIN. With large meta-model repositories, however, the use of OCL imposes a problem in scalability: To evaluate an OCL expression, all elements in the expression’s context must be loaded before evaluation. This process is done by instantiating the model elements with the Java Metadata Interface (JMI). However, the costs of this instantiation can become high in terms of memory usage, depending on the repository’s size and the number of elements that have to be analyzed for the evaluation.

In order to minimize the number of elements to be instantiated, the query infrastructure of the meta-model repository can be used for evaluation: Subexpressions of OCL can be translated into queries whose result set is identical to the result of the expression’s evaluation. Thus, the number of elements that have to be instantiated and analyzed is reduced significantly. For such an integration of the query infrastructure into the OCL evaluation process, a mapping of OCL expressions to corresponding query language constructs has to be found. In this thesis, a mapping of OCL to MQL will be presented.

At first, an overview of the technologies used in MOIN will be given. After comparing the expressional powers of OCL and MQL, the principles of mapping OCL to the MOIN Query Language will be presented by defining the parts of OCL that can efficiently be mapped to a query language with limited expressional powers. To identify these parts, MQL structures will first be analysed and expressed in OCL by a set of examples. These examples will later be used for the desired mapping. An integration into an existing OCL parser/evaluator infrastructure will also be discussed, showing strategies to prepare the OCL constructs and to optimize the resulting queries, so that mappable and non-mappable parts can be individually evaluated either by the query infrastructure or the existing evaluator. Finally, in a conclusion, the practicability of the presented mapping will be estimated by applying it to the constraints of MOF itself.

\textsuperscript{1} SAP is a trademark or registered trademark of SAP AG in Germany and in several other countries.
2 Background

2.1 Meta Object Facility (MOF)

The Meta Object Facility specification [OMG02] defines a formal language and framework for specifying metamodels. MOF is an OMG standard. It was created in need of a language in which the UML metamodel can be described. A good introduction into MOF can be found in [Pro02].

Design of MOF

When speaking about metadata (which is often described as “data about data”), one could theoretically create an arbitrary number of meta-layers, since every kind of data can be described by a corresponding set of meta-data. (And thus the meta-data itself, and the meta-meta-data etc.) MOF is organized in a four-layered architecture in which the layers represent different levels of abstraction. The four layers are named $M_0$ to $M_3$, with $M_3$ being the highest level.

M3 Layer

The $M_3$ layer contains the MOF metamodel (also called MOF model). The MOF model is used to instantiate metamodels, which are then part of the $M_2$ layer. The only exception is the MOF model, which is an instance of itself. With this self-containedness property, MOF is a closed metamodelling architecture.

M2 Layer

The $M_2$ layer contains metamodels, which are instances of the MOF model. For example, the UML metamodel is part of the $M_2$ layer.

M1 Layer

In the $M_1$ layer (also called model layer) describes the format and semantics of data. It contains models. For example, UML models are part of the $M_1$ layer.

M0 Layer

The $M_0$ layer or information layer contains the actual objects or instances and thus, the data. For our considerations, the M0 layer is of little interest.

Internal Structure

MOF was originally derived from a subset of UML and shares many common modeling elements with UML. Figure 1 shows an excerpt of the MOF model (from [Het06]). The self-containedness property of MOF can be seen here: The MOF model is described in itself, using principles like classes, inheritance and associations in a diagram.
Import/Export with XMI

MOF defines an XML-based interface for metadata import and export called XML Metadata Interchange (XMI). It is intended to be used for any kind of metadata interchange between different tools. Today, this is mainly the case for UML-based modeling tools and MOF-based meta-data repositories.

2.2 Object Constraint Language (OCL)

OCL is a declarative, strongly typed, side-effect free language that was originally created as a constraint language for UML. In its current version, it can be used with any (meta-) model compliant to MOF. A complete definition of OCL will not be given here; an introduction can be found in [Sch05]. For further reading, see [WK04] or the specification [OMG06].

Some aspects of OCL that will be important in the course of this thesis shall be explained in the following sections.

**self and allInstances()**

The operator `self` is used to refer to the context of an OCL expression. Since an OCL expression is always evaluated for single objects\(^2\), the `self` operator represents an instance called the *contextual instance*. With the `self` operator, expressions can operate on a single element that represents all elements of a type. This is helpful for invariants and other constraints, but if a set operation has

---

\(^2\) see [WK04], chapter 3.1.3
to be performed on the instances of a context, the allInstances() operator is needed to provide a collection of all elements of a certain type.

The allInstances() operator works as a query operator on all object classes, i.e. types, and returns all elements that are instances of the referred type or one of its subtypes. It can only be used for classifiers that have a finite number of instances ([OMG06], 11.2.5). The OCL specifications document is still missing a formal definition of allInstances().

**Shorthand notation for collect()**

For navigation across several associations, a shorthand notation can be used. Instead of writing

```plaintext
element1.assoc1x2->collect(assoc2x3)->collect(assoc3x4)
```

the shorthand notation is

```plaintext
element1.assoc1x2.assoc2x3.assoc3x4
```

---

**The OCL expressions package**

The OCL expressions package defines the structure that OCL expressions can have. The inheritance relationships between the classes of the expressions package can be seen in Figure 2. An
OclExpression always has a type that can be statically determined by analyzing the expression and its context.

After the evaluation of an expression, a result value is returned. If the expression has a boolean result value, it can be used as a constraint. Expression with a result value of any type can be used for queries, iterator bodies etc.

The type FeatureCallExp will be most interesting for our considerations. A CallExp has exactly one source that is identified by an OclExpression. A FeatureCallExp evaluates to the type of the corresponding feature, which can be an operation, attribute or association.

Queries in OCL

In its current version 2.0, OCL can also be used as a query language for model objects. The following syntax is used (from [OMG06], ch. 7.3.6):

```
context Typename::operationName(param1 : Type1, ... ): ReturnType
body: -- some expression
```

Note that in contrast to the pre, post and inv parts, the return value of the expression in the body part is not restricted to boolean. It can be a single element of any type or a collection.

2.3 Java Metadata Interface (JMI)

The Java Metadata Interface (JMI) is the standard interface for access, manipulation and exchange of MOF-based metadata in Java. JMI defines a mapping between MOF and Java, which allows to create Java classes from MOF-based models.

JMI offers interfaces for instances, class proxies, associations and packages; furthermore, a reflective interface which can be used for introspection of objects and classes, and also for adding and removing features of classes. With reflection, information on the types of a model can be obtained, and thus, the metastructure of the model can be navigated.

For the complete JMI specification, see [Pro02].

2.4 SAP’s Modeling Infrastructure (MOIN)

SAP’s Modeling Infrastructure (MOIN) provides a MOF-based meta-model repository with a set of services built on top of it, using Java technology. These services will include model-to-model transformation and OCL. An OCL parser and evaluator have also been implemented (see [Het06]). MOIN complies to standards such as MOF 1.4, XMI 1.2, OCL 2.0 and JMI 1.1. Using these standards, MOF-based models can be imported and exported from and to XMI.

Query Infrastructure

The MOIN meta-model repository offers a query infrastructure that allows the retrieval of model data based on the MOIN Query Language (MQL). The query infrastructure is used as an entry point into the repository to obtain elements of a certain type and with certain properties.
MOIN Query Language (MQL)

The MOIN Query Language (MQL) provides an SQL-like syntax to query the MOIN meta-model repository. An MQL query is specified against a M2 meta model and queries M1 data. The MOIN Query Language is strongly typed against the MOF meta model. A concrete syntax is defined, see Appendix A for the full specification.

An MQL query always has a SELECT-clause and a FROM-clause. Optionally, it has multiple with-entries and where-entries.

Select-clause

The SELECT-clause specifies the “columns” of elements and attributes for the result set, using the aliases defined in the FROM-clause.

From-clause

The FROM-clause specifies the relevant meta-model types of which the cartesian product is constructed. For each type in the FROM-clause, an alias has to be defined.

Where-clause

A WHERE-clause specifies a local filter condition on one from-type. It operates on one alias only. For conditions on multiple aliases, multiple where-clauses can be formulated. A WHERE-clause is a boolean expression. Multiple WHERE-clauses are connected conjunctively.

With-clause

A WITH-clause specifies join conditions on a product of from-types or a nested query. It allows formulations of non-local filters that work on more than one alias defined in the FROM-entry. Multiple WITH-clauses are connected conjunctively.

Result Set

The result set of an MQL query consists of multiple result records, which contain elements of the type specified in the SELECT-clause.
3 Principles of Mapping OCL to MQL

In this section, the basic principles of an OCL-to-MQL mapping will be discussed. By a comparison of the expressional powers of both languages, MQL as the language with weaker expressional powers will first be analyzed in order to express its key features in OCL. These results will then be used to identify the parts of OCL which can be mapped to MQL.

3.1 Expressional Powers of OCL and MQL

As a Turing complete language, OCL offers a large number of features that cannot be expressed in MQL. Since MQL is a query language, it operates on a set of data and returns a subset of this data which complies to certain conditions. Unlike other query languages (e.g. SQL), MQL does not offer any computational features on the data. Aggregations cannot be expressed, so the return types of an MQL query are always a subset of the types in the FROM-clause.

Even more so, MQL has several limitations that depend on the types used and the clause in which an expression occurs. The main limitations of MQL are:

- No computational expressions on the result set (e.g. sum, size, isEmpty)
- Only primitive comparisons are possible (e.g. a.attr1 > a.attr2, but not a.attr1 > a.attr2 + 5)
- Boolean operators are not always available (e.g. not, and, or can be used in a WHERE-clause, but not in WITH-clauses)

The main problem with the mapping of OCL to a query-based language is the handling of procedural aspects. While concepts like types, attributes, associations and set operations can be transferred to MQL quite well, functions and procedural parts of OCL like the iterate operator do not have a counterpart.

It has been shown in [Sch98] that OCL can be mapped to SQL-92 using concepts like database assertions and stored procedures. Since these features are not available in the MOIN query infrastructure, we will mainly focus on the most common OCL constructs that are non-procedural and investigate the mapping capabilities to MQL.

3.2 Mapping MQL to OCL by example

In this section, the core concepts of MQL regarding MOF will be transferred to corresponding OCL expressions. For those parts of MQL that reflect implementation-specific MOIN specialties, no mapping will be done. The MQL-to-OCL mapping should be seen as a preparation for the
following section, where the desired OCL-to-MQL mapping will be described, using the results of this section.

For each example MQL query, we will offer two mappings to OCL: The first one will be the actual query, the second one will be an expression that evaluates to true for all and only for those elements in the query result set. The latter expression, called “characteristic predicate” in the following examples, is offered because MQL itself does not support boolean expressions over the result set. In the following section, it will however be useful to know the set of elements for which an invariant holds (or, more important, does not hold).

The queries and OCL expressions refer to the metamodel seen in Fig. 3. Since it is a MOF constraint that all model elements must have a containing package, we will assume that the metamodel lies in a package called package1.

<table>
<thead>
<tr>
<th>Type1</th>
<th>Type2</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute1:int</td>
<td>attribute1:int</td>
</tr>
<tr>
<td>attribute2:boolean</td>
<td>attribute2:boolean</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>assoc1x2</td>
<td>assoc1x2</td>
</tr>
<tr>
<td>assoc1x2EndA</td>
<td>assoc1x2EndB</td>
</tr>
</tbody>
</table>

Figure 3: Small sample Metamodel

Since a SELECT and FROM clause are mandatory for any MQL query, we will start with the simplest example:

**Example 1: Basic query**

MQL query

```ml
SELECT a1 FROM 'package1'::'Type1' AS a1
```

OCL query

```ocl
context: example1():Set(Type1)
body: Type1-->allInstances()-->select(s1|s1-->oclIsTypeOf(Type1))
```

OCL characteristic predicate

```ocl
context: Type1
inv: self-->oclIsTypeOf(Type1)
```

As we can see from the OCL query, an MQL expression in its basic form operates on all (M1-) instances of a (M2-) type. Since the allInstances() operator also returns all elements whose type is a subtype of Type1, the select() statement with type check is necessary.

Since an invariant is always an expression over all instances of a type, the allInstances() operator is omitted in the characteristic predicate expression (see section 4.2).
The situation becomes more difficult if we do not select from one type, but from several types, like in

```
SELECT a1, a2 FROM 'package1'::'Type1' AS a1, 'package1'::'Type2' AS a2
```

In this case, the result set is not just a Set of elements, but a set of tuples which together form the cartesian product of the types in the FROM-clause. Theoretically, this could be expressed by an iterator operation that collects those elements into corresponding tuples. But since we are interested in exactly mappable OCL expressions, this rather complicated case is omitted here, and the following example queries will only have one type in the SELECT-clause.

**Example 2: Subtypes**

MQL query

```
SELECT a1 FROM KIND 'package1'::'Type1' AS a1
```

OCL query

```
context: example2():Set(Type1)
body: Type1->allInstances()­>select(s1|s1­>oclIsKindOf(Type1))
```

OCL characteristic predicate

```
context: Type1
inv: self­>oclIsKindOf(Type1)
```

In this query, we extended the FROM-clause so that it includes subtypes. In this query, we extended the FROM-clause so that it includes subtypes. With examples 1 and 2, we have covered most of the functionality in the FROM-clause. One aspect that was not covered here are the scope-clauses which allow limiting the set of elements that are queried.

**Example 3: Attributes**

MQL query

```
SELECT a1.'attribute1' FROM 'package1'::'Type1' AS a1
```

OCL query

```
context: example3():Bag(int)
body: Type1->allInstances()­>select(s1|s1­>oclIsTypeOf(Type1))->collect(cl|cl.attribute1)
```

Note that the oclIsKind0f() construct in the characteristic predicate is sort of redundant here; we use it nevertheless to be consequent.
In this query, the desired result does not consist of the elements themselves, but their attributes. Since the result of this query is a Collection of the type of *attribute1*, the query must collect the attribute elements. The type of *attribute1* can be obtained from the metamodel. Constructing an characteristic predicate for all attribute values is not possible since attributes can also have primitive typed values. With object-valued attributes, one can formulate an predicate over all elements of the attribute type, which is not possible for primitives like integer or real, as *allInstances()* only operates on types with a finite number of elements (see section 2.2). And even with object valued attributes, navigation from the attribute element to its containing element is not possible.

**Example 4: Where-Entry**

MQL query

```mql
SELECT a1 FROM 'package1'::'Type1' AS a1
WHERE FOR a1 (condition1)
```

OCL query

```ocl
context: example4():Set(Type1)
body: Type1->allInstances()->select(s1|s1->oclIsTypeOf(Type1) and condition1(s1))
```

OCL characteristic predicate

```ocl
context: Type1
inv: self->oclIsTypeOf(Type1) and condition1
```

In the OCL query, the boolean condition used in the *where*-clause is reflected by a *select()* statement. In this example, the mapping is quite easy and straightforward because boolean conditions in the *where*-clause are limited to primitive boolean, numeric and String operations on attributes, which are all supported by OCL. In appendix A, the complete specification of possible operations is listed.

The boolean operators (*and, or, not*) and the primitive comparison operations can be mapped with the identical expressions. The boolean operations *isTrue* and *isFalse* must be translated into = true and = false.

The only MQL feature in the *where*-clause that is not supported by OCL is String comparison with the LIKE operator.

**Examples 5: With-Entry**

*With*-clauses can consist of three different types of predicates:

- association predicate
- link predicate
- comparison predicate
Example 5.1: association predicate

MQL query

\[
\text{SELECT } a1 \text{ FROM 'package1':'Type1' AS a1, 'package1':'Type2' AS a2 WITH (a1, a2) IN ASSOC (assoc1x2EndA, assoc1x2, assoc1x2EndB)}
\]

OCL query

\[
\text{context: example5_1():Set(Type1)}
\text{body: Type1->allInstances()->select(s1|s1->oclIsTypeOf(Type1) and s1.assoc1x2EndB->notEmpty())}
\]

OCL characteristic predicate

\[
\text{context: Type1}
\text{inv: self->oclIsTypeOf(Type1) and self.assoc1x2EndB.notEmpty()}
\]

Note that association predicates can also be references and attributes. This does not change the structure of the OCL expression.

Since multiple with- and where-clauses are always connected conjunctively, they can be expressed in OCL by combining the conditions in the select() expressions with and.

Example 5.2: link predicate

MQL query

\[
\text{SELECT a1 FROM 'package1':'Type1' AS a1 WITH (a1, assoc1x2, assoc1x2EndB) IN ( SELECT a2 FROM type2 AS a2 WHERE FOR a2 (condition1))}
\]

OCL query

\[
\text{context: example5_2():Set(Type1)}
\text{body: Type1->allInstances()->select(s1|s1.oclIsTypeOf(Type1) and s1.assoc1x2EndB->notEmpty and s1.assoc1x2EndB->exists(e1| e1->oclIsTypeOf(Type2) and e1.condition1))}
\]
OCL characteristic predicate

context: Type1
inv: self.oclIsOfType(Type1) 
  and self.assoc1x2EndB->notEmpty() 
  and self.assoc1x2EndB->exists(e|
    e->oclIsTypeOf(type2) 
    and e.condition1)

In this query it becomes obvious for the first time why we decided to translate MQL into boolean “characteristic predicate” expressions as well. The inner query in the MQL statement can be added to the select() clause in the OCL expression using the “characteristic predicate” mapping. (Of course, self has to be replaced by the iterator variable of the select statement.)

Since associations have multiplicities that can be greater than 1, we have to use the exists() operator to ensure that for at least one element in the association, the condition evaluates to true.

Note that there is a semantic difference that lies in the nature of queries: In the MQL result set, there is one entry for each match of an association between a1 and a2. So if one element has multiple links to elements that fulfill condition1, there will be one entry for each link. In OCL, the return type is a Set and thus, each element can occur only once.

For references and attributes, the mapping works analogously.

Example 5.3: comparison predicate

MQL query

SELECT a1 FROM 'package1'::'Type1' AS a1, 'package1'::'Type2' AS a2
WITH a1.'attribute1' = a2.'attribute1'

OCL query

context: example5_3():Set(Type1)
body: Type1->allInstances()->select(s1|s1->oclIsTypeOf(Type1) 
  and Type2->allInstances()->
    exists(e1|e1->oclIsTypeOf(Type2) 
    and s1.attribute1=e1.attribute2))

OCL characteristic predicate

context: Type1
inv: oclIsTypeOf(Type1) and Type2->allInstances()->
  exists(e1|e1->oclIsTypeOf(Type2) 
  and s1.attribute1=e1.attribute2)

A comparison predicate works on two aliases that do not have to be connected with an association. It therefore takes the cartesian product of the types in the from-clause and selects the tuples
that conform to the comparison predicate. An expression over classes that are not connected by an association does normally not occur in OCL.

**Example: compound query**

In this example, we will create a larger query to illustrate how the presented mappings can be combined.

```
SELECT a1.'attribute1' FROM 'package1':>'Type1' AS a1
WHERE FOR a1 ('attribute1' > 0)
WITH (a1, assoc1x2, assoc1x2EndB) IN
    (SELECT a2 FROM KIND type2 AS a2
     WHERE FOR a2 ('attribute2' isTrue)
    )
```

**OCL query**

```
context: example6():Bag(int)
body: Type1.allInstances() -> select(s1|
                 s1->oclIsTypeOf(Type1)
                 and s1.attribute1 > 0
                 and s1.assoc1x2EndB.notEmpty
                 and s1.assoc1x2EndB->exists(e1|
                     e1.isKindOf(type2)
                     and e1.attribute2 = true)
              )
        .collect(c1|c1.attribute1)
```
3.3 Mapping OCL to MQL

The first question that arises when investigating a possible mapping of OCL expressions to MQL queries is the question of types. As mentioned in section 2.2, an OclExpression always has a return type that can be statically determined. Thus, OCL expressions can be categorised by their result type and the type they operate on. The return type of an OCL expression can be completely different from the type that the expression works on. However, an MQL query always returns a result set that is a subset of the underlying elements, without any computation done.

In this approach, we will try to find OCL expressions that can be mapped exactly; this will mostly be possible for set operations.

For boolean expressions, the approach will be to find the elements in the context of the expression that evaluate to false, so that a later implementation can check that set for being empty.

A concept that will be very important for the evaluation of OclExpressions is the ability to provide scopes in an MQL query, since an OclExpression does not always operate on all instances of a type. This is however the entry point for an MQL query, so limiting the query to a fixed set of elements is necessary and is provided by the scope clauses.

In [DH99], OCL expressions are categorised using the scheme in Fig. 4. We are using this scheme, but the type of OclExpression from section 2.2 will also be taken into account.

<table>
<thead>
<tr>
<th>OCL expression over</th>
<th>Result type of an OCL expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>BASIC TYPE</td>
</tr>
<tr>
<td>Model Types:</td>
<td></td>
</tr>
<tr>
<td>- Class</td>
<td>CLASS</td>
</tr>
<tr>
<td>- Association Class</td>
<td>AND</td>
</tr>
<tr>
<td>- Attribute</td>
<td>ATTRIBUTE</td>
</tr>
<tr>
<td>- Association End</td>
<td>NAVIGATION</td>
</tr>
<tr>
<td>- Operation</td>
<td>OPERATION</td>
</tr>
<tr>
<td>Collection types</td>
<td>COMPLEX</td>
</tr>
<tr>
<td></td>
<td>PREDICATE</td>
</tr>
</tbody>
</table>

Figure 4: Categorisation of OCL expressions (from [DH99])

Pattern: BASIC TYPE

The basic types used in OCL (Integer, Real, String, Boolean) are available in MQL. The only computation that can be done on numeric types is comparison. This means that operations on basic types that have a basic value as result type cannot be mapped to MQL, including most simple expressions like a+1. Comparisons can be performed on all primitive types (see App. A).

4Fields that are blank indicate that for this combination, no expressions exist. For example, there are no expressions on basic types that result in a collection.
Pattern: CLASS AND ATTRIBUTE

An OCL expression referring to a class or an attribute can basically be mapped to MQL.

Constraints, after their preparation with the rules of section 4.2, start with

\[ \text{<Type>\text{.allInstances()}} \]

This is translated into

\[ \text{SELECT } \text{a1 FROM KIND <Package>::<Type> AS a1} \]

Attributes (which are called by an \text{AttributeCallExp}) are simply referenced using the same notation as in OCL.

\[ \text{element1.attribute1} \]

becomes

\[ \text{a1.attribute1} \]

depending on the alias that is used in MQL for the type of \text{element1}.

Pattern: NAVIGATION

OCL

\[ \text{element1.assoc1x2EndB} \]

MQL

\[ \text{SELECT a2 FROM 'package1':'Type1' AS a1 IN element1.mri, KIND 'package1':'Type2' AS a2 WITH (a1, a2) IN ('assoc1x2EndA', 'assoc1x2', 'assoc1x2EndB')} \]

OCL

\[ \text{element1.reference1x2} \]

MQL

\[ \text{SELECT a2 FROM 'package1':'Type1' AS a1 IN element1.mri, KIND 'package1':'Type2' AS a2 WITH a1.'reference1x2' = a2} \]

In OCL, a navigation expression (\text{NavigationCallExp}) over an association uses the name of an association’s referenced end. From the metamodel, the association’s name and the name of the exposed end can be obtained. These names are needed in the MQL query. The same is true for the type of the referenced end. For reference, the name of the reference is simply used. Depending
on whether the expression consists of an association or a reference, the proper format of the <assoc-predicate> has to be chosen.

Whenever an association is traversed, the type of its referenced end must be added to the FROM-clause. Since the element at the referenced end of the association can also be of a subtype of the association’s type, the KIND expression has to be used. If the association describes a query result, the type of its referenced end must also be added to the SELECT-clause, and the type of its exposed and has to be removed.

The scope-clause used here is kind of a pseudocode notation; the actual query must specify the MOIN Resource Identifier (MRI) for all the elements in the query’s scope. However, since for the patterns, generic transformation rules have to be specified, we will use the notation element1.mri to indicate that the MRI of the element is calculated.

Since the NAVIGATION pattern can occur within other operations, the optimizations in section 4.3 can be applied on an expression that contains navigation.

**Pattern: OPERATION**

An operation in OCL can not be mapped to an MQL query due to the limitations mentioned in section 3.1.

**Pattern: COMPLEX PREDICATE**

Predicates cannot be mapped, since MQL queries do not allow boolean expressions over the result set. However, some operators can be transformed into more suitable, semantically equal expressions. See section 4.2.

For example, the following common functions cannot be transformed or mapped:

- isEmpty()/notEmpty()
- includes()/includesAll()
- excludes()/excludesAll()

The only exception are the predicates isTypeOf()/isKindOf(), if they appear inside a select() statement. This is described in the pattern QUERY.

**Pattern: BASIC VALUE**

An expression that operates on a collection and returns a basic value cannot be expressed in MQL due to the limitations mentioned in section 3.1. This includes all aggregation functions like sum(), size(), but also mathematical expressions like min(), max().

**Pattern: QUERY**

This pattern contains all collection operations that have a collection as their return type. Hence, it is basically possible to express them using an MQL query. The operator that resembles a query most is the select() operator.
select()-operation

For the following mapping, it is assumed that \texttt{collection1} is of type \texttt{Type1}.

\textbf{OCL}

\begin{verbatim}
\texttt{collection1->select(s1|condition1(s1))}
\end{verbatim}

\textbf{MQL}

\begin{verbatim}
SELECT s1 FROM 'package1'::'Type1' AS a1 IN collection1.mri
WHERE FOR a1 condition1(a1)
\end{verbatim}

Here, the ‘IN’ operator (scope-clause) is used to restrict the execution of the query to the elements of \texttt{collection1}. Again, a pseudocode notation is used to indicate the list of the MRIs of the collection’s elements.

The aspect that has to be taken in account here is the nature of \texttt{condition1}. As long as it is a simple expression that only works on the iterator variable \texttt{s1} and conforms to \texttt{<where-condition>} from App. A, it can directly be translated into MQL.

The problem with boolean conditions like \texttt{condition1} is that they can be arbitrarily complex. If \texttt{condition1} is the return value of a \textit{COMPLEX PREDICATE} expression, the statement cannot be transformed, since such an expression can never be completely evaluated by a query. A final \texttt{isEmtpy()} check or similar operations always have to be executed to get the boolean result, and thus, it is not possible to map the \texttt{select()} statement if its body consists of such a predicate. The only exception are the \texttt{oclIsKindOf()}/\texttt{oclIsTypeOf()} operators mentioned below.

Multiple iterator variables

It is also possible for a \texttt{select()} statement to have multiple iterator variables, which is typically done for equality or uniqueness tests. Since the result type of such a select statement is a collection of tuples, the mapping in MQL is the following:

\textbf{OCL}

\begin{verbatim}
\texttt{collection1->select(s1,s2|condition2(s1,s2))}
\end{verbatim}

\textbf{MQL}

\begin{verbatim}
SELECT a1,a2 FROM 'package1'::'Type1' AS a1 IN collection1.mri,
    'package1'::'Type1' AS a2 IN collection1.mri
WITH condition2(a1,a2)
\end{verbatim}

Navigation

If navigation occurs within a \texttt{select()} expression, a nested query is used for the mapping. This is only possible if \texttt{condition1} conforms to any of the other mappings for \texttt{select()} in this section. The inner query is then the mapping of a the \texttt{select()} clause as if contained no navigation.
3 Principles of Mapping OCL to MQL

OCL

\[
\text{collection1}\rightarrow\text{select}(s1|[\text{not}]\ s1.assoc1x2EndB.\exists(e|\text{condition1}(e)))
\]

MQL

\[
\text{SELECT } a1 \text{ FROM } 'package1'::'Type1' \text{ AS } a1 \text{ IN collection1.mri}
\]

\[
\text{WITH } (a1, 'assoc1x2', 'assoc1x2EndB') [\text{NOT}] \text{ IN }
\]

\[
( \text{SELECT } a2 \text{ FROM } 'package1'::'Type2' \text{ AS } a2
\]

\[
\text{WHERE condition1}(a2)
\]

If the multiplicity of \text{assoc1x2EndB} is 1, the \text{exists()} operator can also be missing. If multiple conditions are expressed over the same association for navigation, these conditions are combined in the \text{WHERE}-clause of the inner query, using the boolean operators with which the conditions are concatenated in the OCL statement.

Comparison with navigation

If the boolean expression is a comparison between attributes using navigation, the \text{NAVIGATION} pattern can be integrated:

OCL

\[
\text{collection1}\rightarrow\text{select}(s1|s1.\text{attribute1} > s1.assoc1x2EndB.\text{attribute1})
\]

MQL

\[
\text{SELECT } a1 \text{ FROM } 'package1'::'Type1' \text{ AS } a1 \text{ IN collection1.mri,}
\]

\[
'package1'::'Type2' \text{ AS } a2
\]

\[
\text{WITH } (a1,a2) \text{ IN ASSOC (assoc1x2EndA, assoc1x2, assoc1x2EndB)}
\]

\[
\text{WITH a1.'attribute1'} > a2.'attribute1'
\]

Comparison without navigation

Since MQL only supports comparisons with numerical constants directly, a comparison of two attributes of an element has to be seen as a self-join, and hence, it has to be expressed in a \text{WITH}-clause. Also, two aliases have to be defined for the type of the element.

OCL

\[
\text{collection}\rightarrow\text{select}(s1|s1.\text{attribute1} > s1.\text{attribute4})
\]

MQL

\[
\text{SELECT } a1 \text{ FROM } 'package1'::'Type1' \text{ AS } a1, 'package1'::'Type1' \text{ AS } a2
\]

\[
\text{WITH a1.'attribute1'} > a2.'attribute4'
\]

\[
\text{WITH a1=a4}
\]
For the two type operators, a mapping can be done, since the selection of types and their subtypes is directly supported by MQL.

\[
\text{OCL: } \text{collection1} \rightarrow \text{select}(s1 | s1.oclIsTypeOf(\text{Type1}))
\]

\[
\text{MQL: } \text{SELECT } a1 \text{ FROM } 'package1'::'\text{Type1}' \text{ AS } a1 \text{ IN collection1.mri}
\]

Here, the type in the \textit{FROM} clause does not depend on the type of \textit{collection1}, but on the parameter of the \texttt{oclIsTypeOf()} expression.

\textbf{collect()-operation}

**Attributes**

The \texttt{collect()} operator is used on collection types to create a collection of a specific feature, which can be an association, an attribute or an operation. If the collected element is an attribute (meaning that the body of the operation is a \texttt{AttributeCallExp}, the mapping is straightforward and is done the following way:

\[
\text{OCL: } \text{collection1} \rightarrow \text{collect}(c1 | c1.\text{attribute1})
\]

\[
\text{MQL: } \text{SELECT } c1.\text{attribute1} \text{ FROM } 'package1'::'\text{Type1}' \text{ AS } c1 \text{ IN collection1.mri}
\]

**Navigation**

If an association is traversed (\texttt{NavigationCallExp}) the \textit{NAVIGATION} pattern has to be used. The types of the association ends and the association name have to be taken from the metamodel in the same way as in section 3.3:

\[
\text{OCL: } \text{collection1} \rightarrow \text{collect(assoc1x2EndB)}
\]
MQL

```
SELECT a2 FROM 'package1'::'Type1' AS a1 IN collection1.mri,
     'package1'::'Type2' AS a2
WITH (a1,a2) IN ASSOC(assoc1x2EndA, assoc1x2, assoc1x2EndB)
```

If the body of `collect()` is of the type `OperationCallExp`, the expression cannot be mapped. See pattern `OPERATION`.
4 Implementation

4.1 Structure of OCL evaluation

After an OCL expression has been parsed, an abstract syntax tree (AST) is created. The evaluator operates on this tree. To integrate the query infrastructure, the evaluator has to perform the following steps:

1. Prepare OCL statement
2. Match subexpressions to patterns
3. Distinguish mappable and non-mappable subexpressions
4. Translate mappable subexpressions to MQL
5. Perform optimization
6. Evaluate expression (with execution of queries)

4.2 Preparation

During the preparation of an OCL statement, transformations are performed that do not change the semantics of the statement. These transformations change the OCL statements so that greater parts of them can be mapped.

The preparation rules must be applied in the order given here.

Constraints

Constraints always have a context that defines the elements to which self refers. They can be transformed into OCL expressions without a context that can be evaluated using the patterns from section 3.3, since those patterns do not specify a context. The transformation is the same for all constraints (pre, post, inv). In this example, an invariant is used.

```
context: Type1
inv: expression1(self)
```

Transformation

```
body: Type1.allInstances() ->forall(s1 | expression1(s1))
```
OCL queries

Since an OCL query can have any type of expression in its body part, it does not need a special preparation.

Quantifiers and navigation

If an expression with a quantifier (forall(), exists()) contains another quantifier in its scope, it can be transformed in the following cases:

forall()

\[ \text{collection1} \rightarrow \forall(c1|c1.association \rightarrow \forall(c2|\text{booleanExp}(c2))) \]

Transformation

\[ \text{collection1} \rightarrow \text{association} \rightarrow \forall(c2|\text{booleanExp}(c2)) \]

eexists()

\[ \text{collection1} \rightarrow \exists(c1|c1.association \rightarrow \exists(c2|\text{booleanExp}(c2))) \]

Transformation

\[ \text{collection1} \rightarrow \text{association} \rightarrow \exists(c2|\text{booleanExp}(c2)) \]

This is only possible if two identical quantifiers are used. A “mixed” statement cannot be simplified. Also, the expression booleanExp must be over the variable c2 alone. If it refers to another variable (like c1), the transformation cannot be used. A universal quantifier within the scope of another universal quantifier, as in this preparation pattern, occurs every time an invariant with a forall() statement is converted into a context free form.

select() and isEmpty()

The pattern from the preceding section can be generalized to all expressions where notEmpty() occurs within an select() statement that is checked for emptiness afterwards. The operation() statement must have Collection as its return type, like e.g. select() and collect(), but all other operations are possible also. If no operation() statement is present, the mapping can still be used to for the navigation part only.

If an expression has the following form, it can be transformed:

Expression

\[ \text{collection1} \rightarrow \text{select}(s1|s1.association.operation() \rightarrow \text{notEmpty()} \rightarrow \text{isEmpty()} \]
transformation

This mapping corresponds to the mapping for the universal quantifiers from section 4.2, after they have been split up into \texttt{select}() and \texttt{isEmpty()} statements. The existential quantifier mapping, in its generalized form, is the following:

expression

\[
\text{collection1->select(s1|s1.association.operation()->notEmpty())->notEmpty()}
\]

transformation

\[
\text{collection1->association->operation()->notEmpty()}
\]

**Complex predicates**

As mentioned in the pattern \textit{COMPLEX PREDICATES} in section 3.3, these expressions can not be mapped. Therefore, we must transform them into a combination of basic OCL expressions, of which some can be mapped.

\texttt{forall()}

\[
\text{forall(a1|expression1(a1))}
\]

transformation

\[
\text{select(a1|not expression1(a1))->isEmpty()}
\]

This expression can in the best case be mapped completely except for the trailing \texttt{isEmpty()} statement. The check whether the result set that is returned by the MQL query is empty has to be done otherwise by the evaluator, meaning in JMI.

\texttt{exists()}

\[
\text{exists(a1|expression1(a1))}
\]

transformation

\[
\text{select(a1|expression1(a1))->notEmpty()}
\]

**Boolean operators**

In general, all boolean operators (e.g. \texttt{implies}) have to be reduced to the basic operators \texttt{and}, \texttt{or} and \texttt{not}.  


select() with ‘and’

A select() statement with multiple entries that are concatenated conjunctively can be split up:

\[
\text{collection1} \rightarrow \text{select}(s1|boolean1(s1) \text{ and boolean2}(s1))
\]

Transformation

\[
\text{collection1} \rightarrow \text{select}(s1|boolean1(s1)) \rightarrow \text{select}(s2|boolean2(s2))
\]

The purpose of this transformation is the splitting of mappable and non-mappable expressions, e.g. if \(\text{boolean1}\) is a simple boolean expression that allows the first select() statement to be mapped, but \(\text{boolean2}\) is a predicate or function that can not be mapped. Furthermore larger groups of select() statements can be arranged in a way so that the mappable statements are next to each other and can be combined to one query. (see section 4.3)

4.3 Optimizations

If two or more OCL expressions that are mappable are called after another, i.e. if a mappable expression operates on the result set of another mappable expression, the resulting query mappings can be combined to one query. Thus, instantiating the query results and re-calculating the next query’s scope from the instances is avoided.

Two mappable expressions can be joined using scheme seen in Fig. 5.

<table>
<thead>
<tr>
<th>expression1</th>
<th>expression2</th>
<th>expression1.expression2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S_1)</td>
<td>(S_2)</td>
<td>(S_2)</td>
</tr>
<tr>
<td>(F_1)</td>
<td>(F_2)</td>
<td>(F_1 \cup F_2)</td>
</tr>
<tr>
<td>(I_1)</td>
<td>(I_2)</td>
<td>(I_1)</td>
</tr>
<tr>
<td>(W_1)</td>
<td>(W_2)</td>
<td>(W_1 \cup W_2)</td>
</tr>
<tr>
<td>(W_{h1})</td>
<td>(W_{h2})</td>
<td>(W_{h1} \cup W_{h2})</td>
</tr>
</tbody>
</table>

Figure 5: Optimization scheme

Select-clause

The SELECT clause of the second expression determines the types that remain in the final result set; hence, it is used for the final query.

From-clause

Since WITH- and WHERE-clauses operate on the types in the FROM-clauses, all FROM-clauses have to be included in the final query. When constructing the unions of the FROM-clauses, aliases may have to be adapted so that no collisions occur, and aliases referring to equal types are merged to one alias. Since the scope-clause of the second expression (\(I_2\)) refers to the result set of the first expression, it is not needed in the resulting query.
With- and where-clause

All conditions of both queries can be combined, since WITH- and WHERE-clauses are connected conjunctively. Executing a query and evaluating another query on the result set is equal to forming one query that joins the conditions conjunctively.

Example

A simple OCL example that can be transformed into one single query:

```
Type1.allInstances().assoc1x2EndB->select(s1|s1.attribute1>0)
```

The expression can be split in three mappable parts:

<table>
<thead>
<tr>
<th>part 1</th>
<th>part 2</th>
<th>part 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type1.allInstances()</td>
<td>assoc1x2EndB</td>
<td>select(s1</td>
</tr>
</tbody>
</table>

Using the CLASS AND ATTRIBUTE pattern, the first part can be mapped to:

```
SELECT a1 FROM KIND 'package1'::'Type1' AS a1
```

Part 2 is mapped using the NAVIGATION pattern. Here resultSet1 is used as a pointer to the result set of the preceding query:

```
SELECT a2 FROM KIND 'package1'::'Type1' AS a1 IN resultSet1.mri, KIND 'package1'::'Type2' AS a2
WITH (a1, a2) IN ('assoc1x2EndA', 'assoc1x2', 'assoc1x2EndB')
```

Finally, part 3 is mapped using the QUERY pattern:

```
SELECT s1 FROM KIND 'package1'::'Type2' IN resultSet2
WHERE s1.attribute1>0
```

It is not important in which order the queries are combined. We use a left-to-right order. Combination of part 1 and part 2:

```
SELECT a2 FROM KIND 'package1'::'Type1' AS a1, KIND 'package1'::'Type2' AS a2
WITH (a1, a2) IN ('assoc1x2EndA', 'assoc1x2', 'assoc1x2EndB')
```

Since part 1 did not have a scope (in-clause), the resulting query does not have a scope either. If we combine this query with part 3, we get

```
SELECT a2 FROM KIND 'package1'::'Type1' AS a1, KIND 'package1'::'Type2' AS a2
WITH (a1, a2) IN ('assoc1x2EndA', 'assoc1x2', 'assoc1x2EndB')
WHERE a2.attribute1>0
```

Note that s1 and a2 have been resolved to a2, since they referenced the same type.
4.4 Evaluation of complex OCL expressions

A complex OCL expression will most likely have one or more parts that conform to a pattern from section 3.3 that is not mappable. As a consequence, an implementation of an OCL-to-MQL mapping has to provide a strategy with which the mappable parts of an OCL expression can be evaluated using the query engine, while the non-mappable parts have to be evaluated in the classical way, using JMI. This requires splitting of the OCL expression into subexpressions, which can then be compared with the mapping patterns.

If a subexpression has been identified as mappable, the following two parameters are needed to integrate the query engine into the evaluation process:

- the source of the expression
- the return type of the expression

Both can be determined by the OCL parser.

The source of the expression is an element or a collection of elements. It has to be passed on to the query engine, which uses it in the scope clause of the generated query. For each element, the MOIN Resource Identifier (MRI) has to be determined.

After the query has been executed, the result set consists of MRIs for which the corresponding JMI instances have to be created. The evaluator can then operate on these instances. If an optimization can be performed that combines queries, the instantiation can be avoided for those cases.

The final check whether result sets are empty can be done by analyzing the size of the result set. The implementation of the query infrastructure provides a method to obtain this size.
5 Conclusion

In order to use the query infrastructure for the evaluation of OCL expressions in MOIN, this thesis introduced a mapping of OCL to MQL. Together with the concepts for implementation, including preprocessing and optimization, certain OCL subexpressions can be evaluated using the query infrastructure. The integration into the existing evaluation structure has been described so that the evaluation of expressions that can only be mapped in parts can also profit from the query infrastructure.

For an implementation of the presented concepts, it has to be estimated for which cases the usage of the query infrastructure makes sense, since the evaluation in JMI may be faster for many cases. With very large numbers of elements, however, this effect will be marginal due to memory consumption which, in the end, will slow down the process. For this reason, mappings were presented for all patterns where possible.

5.1 Alternate solutions

A different idea that was not worked out in this thesis was the calculation of minimal supersets for OCL expressions.

For expressions that are not mappable completely, it is imaginable to use the query infrastructure as a preprocessor to limit the elements on which the evaluation works. In these cases, the query infrastructure is not used to calculate the exact set of results of an expression, but to return a set of elements that is smaller than the set of all instances of a type, but still a superset of the desired result set, ideally a minimal superset. The full evaluation still has to be performed on this set, but the improvement in memory consumption can be substantial nevertheless.

This idea was not pursued because it does not solve the main limitations of the query infrastructure, the lack of aggregation functions - the calculated sets can still only be subsets of the types in the repository, without any calculations performed on them. For the determination of the elements which are influenced by an OCL expression, an impact analysis was introduced in [Het06].

5.2 Related work

The use of OCL as a query language is a relatively new development, so there are only few publications dealing with a language transformation of OCL to other query languages. The use of SQL is different from MQL in many ways when applying OCL features to it. Since SQL operates on a relational database, but not on a model repository, database schemata and table definitions
have to be specified for an element-to-table mapping. Depending on the schema, the mapping rules for OCL are individually different.

In [Sch98], a complete mapping of OCL to SQL-92 has been performed using more advanced features of SQL, like stored procedures. A database schema for UML models is also introduced. These concepts are used in [DH99] and [DHL01] to define integrity constraints on relational databases and to formulate business rules in a database system.

The use of OCL as a query language for UML data models is shown in [AB01], where the opposite way is investigated, applying the concepts of relational databases to OCL. Thus, an object-oriented database querying concept is created.

The integration of OCL in MOIN is described in [Het06], where an impact analysis for OCL expressions is discussed.

5.3 Future work/estimate of practicability

The integration into the existing parser/evaluator structure of MOIN can be performed using the techniques presented; since the MOIN Query Language is still under development, future enhancements of MQL like aggregations would then have to be regarded. Nevertheless, they can be integrated into the scheme of this thesis.

The question whether query infrastructure integration offers a substantial improvement in the evaluation process of OCL expressions is yet to be answered. Since only certain subexpressions—those conforming to the patterns presented—can profit from the query infrastructure, it is hard to predict which kinds of OCL expressions will be mappable. Often, small changes in the expressions result in great changes of mappability, e.g., if a constraint is changed from \( a > b \) to \( a > b+1 \) the latter cannot be mapped. The same is true for boolean operators: Expressions that are concatenated with `and` can be mapped to one query, while those concatenated with `or` cannot. Hence, it is not transparent to the user why evaluation of one expression consumes much more time than the evaluation of another.

As a case study, the MOF constraints in App. B have been mapped to MQL queries. Of 59 constraints, 21 could be mapped completely, which is is a percentage of about 35%.
Glossary

**Application Programming Interface (API)**  The interface provided by a system, library or application in order to exchange data or provide services.

**Extended Backus-Naur Form (EBNF)**  An Extension of the basic Backus-Naur form metasyntax notation in which the syntax of formal languages can be expressed. Like MOF, EBNF can be described by itself.

**Java Metadata Interface (JMI)**  A mapping between Java and MOF. It provides a standard interface for access, manipulation and exchange of MOF-based metadata in Java.

**Meta Object Facility (MOF)**  Formal language and framework for specifying metamodels. OMG standard. It is used to define how meta data is organized.

**Modeling Infrastructure (MOIN)**  SAP technology which provides a MOF-based meta-model repository and OCL integration.

**MOIN Query Language (MQL)**  The API of MOIN’s query infrastructure, which describes the abstract syntax of the query language supported by MOIN.

**MOIN Resource Identifier (MRI)**  The unique identifier for model elements in MOIN.

**Object Constraint Language (OCL)**  Formal, side effect free language to describe constraints on MOF-based models. These constraints can be invariants, pre- and post conditions and also query result definitions.

**Object Management Group (OMG)**  A consortium that defines standards for modeling of programs, systems and business processes. Most prominent OMG standards are CORBA, UML and MOF.

**Structured Query Language (SQL)**  A standard language to create, modify, retrieve and manipulate data from relational database management systems.

**Unified Modeling Language (UML)**  A widely used graphical language for object modeling and specification. OMG standard.

**XML Metadata Interchange (XMI)**  XML-based interface for serialization of metadata, mainly used for import and export of metamodels.
Bibliography


Appendix A  MOIN Query Language syntax

MQL Query Format

An MQL query is syntactically the result of the following EBNF grammar rule:

```xml
<MQLquery> ::= "select" <select-clause>
"from" <from-clause>
{"with" <with-clause>}
{"where" <where-clause>}

Select Clause

<select-clause> ::= <select-entry> [<rem-select-clause>]
<rem-select-clause> ::= "," <select-clause>
<select-entry> ::= <alias-identifier> [."" <attr-identifier> "" ]

From Clause

[from-clause] ::= <from-entry> [rem-from-clause]
<rem-from-clause> ::= "," <from-clause>
[from-entry] ::= (<type-clause> | <mri-type-clause>)
[] <scope-clause> "as" <alias-identifier>
[type-clause] ::= \("" <fqualified-type-name> [kind \("" <fqualified-type-name> \)"
[mrit-type-clause] ::= \("" <mri-identifier> [kind \("" <mri-identifier> \)"
[fqualified-type-name] ::= \("" <container-identifier> [::] <qualified-type-name>
[qualified-type-name] ::= \("" <type-path-identifier> [::] <qualified-type-name>
[scope-clause] ::= \("" not \) in \("" <mri-scope-list> \)" | \("" <container-scope-list> \)"
[mri-scope-list] ::= \("" <mri-identifier> \)" ["," <mri-scope-list>]
[container-scope-list] ::= \("" <container-identifier> \)" ["," <container-scope-list>]
```
APPENDIX A MOIN QUERY LANGUAGE SYNTAX

Where Clause

```
<where-clause> ::= "for" <alias-identifier>
                 "(" <where-condition> ")"
<where-condition> ::= "not" "(" <where-condition> ")"
                     | "(" <where-condition> ")" "or" "(" <where-condition> ")"
                     | 
                     | "(" <where-condition> ")" "and" "(" <where-condition> ")"
                     | "" <attr-identifier> ""
<operation-part> ::= <numeric-operation-part>
                     | <boolean-operation-part>
                     | <string-operation-part>
<numeric-operation-part> ::= <primitive-operation> <numeric.literal>
<primitive-operation> ::= ":=" | ":" | "<" | "=" | "<=" | ">=" | "!="
<boolean-operation-part> ::= "isTrue" | "isFalse"
<string-operation-part> ::= <numeric-operation> "" <string.literal> ""
<string-operation> ::= ":=" | "!=" | "like"

With Clause

```
<with-clause> ::= <assoc-predicate>
                | <link-predicate>
                | <comparison-predicate>
<assoc-predicate> ::= "(" <alias-identifier> ","
                 <alias-identifier> ")"
                 "in" "assoc" "(" 
                 "" <assoc-end-identifier> ")"
                 "" <fqualified-assoc-name> ")"
                 "" <alias-identifier> ")"
                 "" <reference-identifier>
                 "" <attr-identifier> ""
                 "=" <alias-identifier>
<fqualified-assoc-name> ::= [""""<container-identifier>"""" "::""]
<qualified-assoc-name> ::= """"<assoc-path-identifier>"""
                         ["""" <qualified-assoc-name>"
<link-predicate> ::= "(" <alias-identifier> ","
<link-predicate-prefix> ")
["not"] "in" "(" <MQLquery> ")"

<link-predicate-prefix> ::= "'" <fqualified-assoc-name> ",",
"'" <assoc-end-identifier> ","
| "'" <reference-identifier> ","
| "'" <attr-identifier> ","

<comparison-predicate> ::= <alias-identifier> ".",
"'" <attr-identifier> ","
<primitive-operator>
<alias-identifier> ".",
"'" <attr-identifier> ","
| <alias-identifier> "="
<alias-identifier>
Appendix B  MQL mappings of MOF Constraints

These constraints are taken from the MOF specification ([OMG02]) and are used as a case study and examples for how the presented mapping can be used. They refer to the MOF metamodel, which can be found in [OMG02].

All constraints that are mentioned here are completely mappable except for the final check for emptiness. This check has to be done to all MQL queries presented here; if it evaluates to true, all elements comply to the respective constraint.

[C-12]

context GeneralizableElement
inv: self.supertypes -> forall(s | not s.isLeaf)

After preparation:

body: GeneralizableElement.allInstances() -> supertypes ->
select(s | s.isLeaf).isEmpty()

MQL query after optimization:

SELECT a2 FROM KIND 'GeneralizableElement' AS a1,
    KIND 'GeneralizableElement' AS a2
WITH (a1, a2) IN ASSOC (subtype, Generalizes, supertype)
WHERE FOR a2 ('isLeaf' isTrue)

[C-13]

context TypedElement
inv: not self.type.oclIsKindOf(Association)

After preparation:

body: TypedElement.allInstances() -> type ->
select(s | s.oclIsKindOf(Association)) -> isEmpty()

MQL query after optimization:

SELECT a2 FROM KIND 'TypedElement' AS a1, KIND 'Association' AS a2
WITH (a1, a2) IN ASSOC (typedElements, IsOf_Type, type)
[C-16]

context Class
inv: self.isAbstract implies not self.isSingleton

After preparation:

body: Class.allInstances() -> select(s1 | s1.isAbstract) -> select(s2 | s2.isSingleton)

MQL query after optimization:

```
SELECT a1 FROM KIND 'Class' AS a1
WHERE FOR a1 ('isAbstract' isTrue)
WHERE FOR a1 ('isSingleton' isTrue)
```

[C-19]

context DataType
inv: self.supertypes -> isEmpty()

After preparation:

body: DataType.allInstances() -> supertypes -> isEmpty()

MQL query after optimization:

```
SELECT a2 FROM KIND 'DataType' AS a1, KIND 'GeneralizableElement' AS a2
WITH (a1, a2) IN ASSOC (subtype, Generalizes, supertype)
```

[C-20]

context DataType
inv: not self.isAbstract

After preparation:

body: DataType.allInstances() -> select(s | s.isAbstract) -> isEmpty()

MQL query after optimization:

```
SELECT a1 FROM KIND 'DataType' AS a1
WHERE FOR a1 ('isAbstract' isTrue)
```
### [C-23]

**context** Reference

**inv:** self.isChangeable = self.referencedEnd.isChangeable

After preparation:

```plaintext
body: Reference.allInstances ->
    select(s | s.isChangeable <> s.referencedEnd.isChangeable)
```

MQL query after optimization:

```sql
SELECT a2 FROM KIND 'Reference' AS a1, KIND 'AssociationEnd' AS a2
WITH (a1, a2) IN ASSOC (referent, RefersTo, referencedEnd)
WITH a1.'isChangeable' != a2.'isChangeable'
```

### [C-25]

**context** Reference

**inv:** self.referencedEnd.isNavigable

After preparation:

```plaintext
body: Reference.allInstances() -> referencedEnd ->
    select(s | not s.isNavigable) -> isEmpty()
```

MQL query after optimization:

```sql
SELECT a2 FROM KIND 'Reference' AS a1, KIND 'AssociationEnd' AS a2
WITH (a1, a2) IN ASSOC (referent, RefersTo, referencedEnd)
WHERE FOR a2 ('isNavigable' isFalse)
```

### [C-34]

**context** Association

**inv:** self.supertypes -> isEmpty()

After preparation:

```plaintext
body: Association.allInstances() -> supertypes -> isEmpty()
```

MQL query after optimization:

```sql
SELECT a2 FROM KIND 'Association' AS a1, KIND 'GeneralizableElement' AS a2
WITH (a1, a2) IN ASSOC (subtype, Generalizes, supertype)
```
[C-35]

```plaintext
context Association
inv: self.isRoot AND self.isLeaf
```

After preparation:

```plaintext
body: Association.allInstances()
    -> select (s | not s.isRoot OR not s.isLeaf) -> isEmpty()
```

MQL query after optimization:

```sql
SELECT a1 FROM KIND 'Association' AS a1
WHERE FOR a1 ('isRoot' isFalse or ('isLeaf' isFalse))
```

[C-36]

```plaintext
context Association
inv: not self.isAbstract
```

After preparation:

```plaintext
body: Association.allInstances() -> select(s | s.isAbstract) -> isEmpty()
```

MQL query after optimization:

```sql
SELECT a1 FROM KIND 'Association'
WHERE FOR a1 ('isAbstract' isTrue)
```

[C-39]

```plaintext
context AssociationEnd
inv: self.type.oclIsTypeOf(Class)
```

After preparation:

```plaintext
body: AssociationEnd.allInstances()
    -> select(s | not s.type.oclIsTypeOf(Class)) -> isEmpty()
```

MQL query after optimization:

```sql
SELECT a1 FROM KIND 'AssociationEnd' AS a1
WITH (a1.'type') NOT IN (SELECT a2 FROM 'Class' AS a2)
```
APPENDIX B  MQL MAPPINGS OF MOF CONSTRAINTS

[C-40]

context AssociationEnd
inv: (self.multiplicity.upper > 1 or self.multiplicity.upper = unbounded) implies self.multiplicity.isUnique

After preparation:

body: AssociationEnd.allInstances
   -> select(s1 | (s1.multiplicity.upper >1 or s1.multiplicity.upper = unbounded) and (not s1.multiplicity.isUnique))

MQL query after optimization:

SELECT a1 FROM KIND 'AssociationEnd' AS a1
WITH (a1.'multiplicity') IN (
   SELECT a2 FROM KIND 'MultiplicityType' AS a2
   WHERE FOR a2 ((('upper'>1) or ('upper'==unbounded)) and ('isUnique' isTrue))
)

[C-41]

context AssociationEnd
inv: self.multiplicity.isOrdered implies self.otherEnd.multiplicity.isOrdered

After preparation:

body: AssociationEnd.allInstances
   -> select(s1|s1.multiplicity.isOrdered)
   -> select(s2|not s2.otherEnd.multiplicity.isOrdered) -> isEmpty()

MQL query after optimization:

SELECT a1 FROM KIND 'AssociationEnd' AS a1
WITH (a1.'multiplicity') IN (
   SELECT a2 FROM KIND 'MultiplicityType' AS a2
   WHERE FOR a2 ('isOrdered' isTrue)
)
WITH (a1.'otherEnd') IN (
   SELECT a3 FROM KIND 'AssociationEnd' AS a3
Appendix B  MQL Mappings of MOF Constraints

[C-44]

context Package
inv: not self.isAbstract

After preparation:

body: Package.allInstances() -> select(s | s.isAbstract) -> isEmpty()

MQL query after optimization:

SELECT a1 FROM KIND 'Package' AS a1
WHERE FOR a1 ('isAbstract' isTrue)

[C-46]

context Import
inv:
self.imported.oclIsTypeOf(Class) or
self.imported.oclIsTypeOf(Package)

After preparation:

body: Import.allInstances()
    -> select(s1 | not s1.imported.oclIsTypeOf(Class))
    -> select(s2 | not s2.imported.oclIsTypeOf(Package))
    -> isEmpty()

MQL query after optimization:

SELECT a1 FROM KIND 'Import' AS a1
WITH (a1,Aliases,imported) NOT IN (  
    SELECT a2 FROM 'Class' AS a2  
)
WITH (a1,Aliases,imported) NOT IN (  
    SELECT a3 FROM 'Package' AS a3  
)
[C-47]

context Import
inv: self.container <> self.imported

After preparation:
body: Import.allInstances() -> select(s | s.container = s.imported) -> isEmpty()

MQL query after optimization:

SELECT a1 FROM KIND 'Import' AS a1, 'Namespace' AS a2, 'Namespace' AS a3
WITH a1.'container'=a2
WITH (a1,a3) IN ASSOC (importer, Aliases, imported)
WITH a2 = a3

[C-53]

context Constant
inv: self.type.oclIsTypeOf(PrimitiveType)

After preparation:
body: Constant.allInstances()
  -> select(s | not s.type.oclIsTypeOf(PrimitiveType)) -> isEmpty()

MQL query after optimization:

SELECT a1 FROM KIND 'Constant' AS a1
WITH (a1.'type') NOT IN (SELECT a2 FROM 'PrimitiveType' AS a2)

[C-54]

context Multiplicity
inv: self.lower>=0 and self.upper=Unbounded

After preparation:
body: Multiplicity.allInstances()
  -> select(s | s.lower<0 or self.upper<>Unbounded) -> isEmpty()

MQL query after optimization:

SELECT a1 FROM KIND 'Multiplicity' AS a1
WHERE FOR a1 (('lower'<0) or ('upper'!=Unbounded))
[C-55]

context MultiplicityType
inv: self.lower <= self.upper or self.upper = Unbounded

After preparation:

body: MultiplicityType.allInstances()
  -> select(s1|s1.lower>s1.upper)
  -> select(s2|s2.upper<Unbounded) -> isEmpty()

MQL query after optimization:

SELECT a1 FROM KIND 'MultiplicityType' AS a1, KIND 'MultiplicityType' AS a2
WHERE FOR a1 ('upper'!=Unbounded)
WITH a1.'lower' > a2.'upper'
WITH a1=a2

[C-56]

context MultiplicityType
inv: self.upper >= 1 or self.upper=Unbounded

After preparation:

body: MultiplicityType.allInstances()
  -> select(s1|s1.upper < 1)
  -> select(s2|s2.upper<Unbounded) -> isEmpty()

MQL query after optimization:

SELECT a1 FROM KIND 'MultiplicityType' AS a1
WHERE FOR a1 ('upper'<1)
WHERE FOR a1 ('upper'!=unbounded)

[C-57]

context MultiplicityType
inv:
self.upper = 1 implies
(not self.isOrdered and not self.isUnique)

After preparation:
body: MultiplicityType.allInstances()
   -> select(s1|s1.upper = 1)
   -> select(s2|s2.isOrdered or s2.isUnique) -> isEmpty()

MQL query after optimization:

```
SELECT a1 FROM KIND 'MultiplicityType' AS a1
WHERE FOR a1 ('upper'==1)
WHERE FOR a1 ('isOrdered' isTrue) or ('isUnique' isTrue))
```