Optimization patterns for OCL-based model transformations

Jesús Sánchez Cuadrado[1], Frédéric Jouault[2],
Jesús García Molina[1], and Jean Bézivin[2]

[1] Universidad de Murcia
{jesusc, jmolina}@um.es
[2] AtlanMod team, INRIA & EMN
{jean.bezivin, frederic.jouault}@inria.fr

Abstract. Writing queries and navigation expressions in OCL is an important part of the task of developing a model transformation definition. When such queries are complex and the size of the models is significant, performance issues cannot be neglected.
In this paper we present five patterns intended to optimize the performance of model transformations when OCL queries are involved. For each pattern we will give an example as well as several implementation alternatives. Experimental data gathered by running benchmarks is also shown to compare the alternatives.

1 Introduction

Rule-based model transformation languages usually rely on query or navigation languages for traversing the source models to feed transformation rules (e.g., checking a rule filter) with the required model elements. The Object Constraint Language (OCL) is the most common language for this task, and it is implemented in several languages such as: ATL [4], QVT [10], and Epsilon [5].

In complex transformation definitions a significant part of the transformation logic is devoted to model navigation. Thus, most of the complexity is typically related to OCL. From a performance point of view, writing OCL navigation expressions in an efficient way (e.g., avoiding bottlenecks) is therefore essential to transformations optimization.

We are currently working on the identification of common transformation problems related to performance. For each identified issue, we analyze several alternative solutions. In this work, we present some of our initial results in the form of idioms.

In particular, we describe five performance-related patterns in model transformations when OCL queries are involved. Each pattern is presented in three parts: i) a statement describing the problem, as well as a motivating example; ii) some experimental data gathered by running benchmarks, so that different implementation alternatives can be compared, and finally iii) some recommendations on the basis of this data.

The paper is organized as follows. Next section describes the five patterns. Section 3 presents some related work. Finally, Section 4 gives the conclusions.

2 Performance patterns

In this section we describe five OCL patterns and analyze them in order to improve the performance of OCL navigation expressions in model transformations. We rely on the experimental data obtained by running performance benchmarks to compare the different implementation strategies[3].

We identified these patterns by working on actual transformations in which we identified one or more bottlenecks. The corresponding expressions are then re-implemented in a more efficient way. Next, whenever a pattern is identified, a small, synthetic benchmark is created to isolate this particular problem and to compare several implementation options.

We will illustrate the patterns using the ATL language[4], but they are general for any rule-based transformation language using OCL, such as QVT [10]. Therefore, these optimization patterns can be considered as idioms or code patterns [3], since they provide recommendations about how to use OCL.

ATL is a rule-based model transformation language based on the notion of declarative rule that matches a source pattern and creates target elements according to a target pattern. It also provides imperative constructs to address those problems that cannot be completely solved in a declarative way. The navigation language of ATL is OCL. Helpers written in OCL can be attached to source metamodel types. Also, global attributes whose value is computed at the beginning of the transformation execution can be defined. The ATL virtual machine provides support for several model handlers, but in our experiments we have only considered EMF.

It should be noted that the patterns presented here suppose that no special optimization is performed by the compiler (i.e., straightforward implementation of the OCL constructs), and that all optimizations have to be done manually by the developer. This is the case with ATL 2006, but this may not be the case with all OCL implementations. These patterns could also probably be used by an OCL compiler to perform internal optimizations using expression rewriting, but this is out of the scope of this paper.

2.1 Short-circuit boolean expressions evaluation

Model transformations usually involve traversing a source model by means of a query generally containing boolean expressions. When such boolean expressions are complex and the source model is large, the order in which such operands are evaluated may have a strong impact on the performance if short-circuit evaluation [2] is used. Short-circuit evaluation means that the first condition is always evaluated but the second one is only evaluated if the first argument does not suffice to determine the value of the expression.

[3] The benchmarks have been executed in a machine with the following configuration: Intel Pentium Centrino 1.5Ghz, 1GB RAM. Java version 1.6.0 under Linux kernel 2.6.15.

[4] The ATL version used is: ATL 2006 compiler, using the EMFVM virtual machine on Eclipse 3.4.

Experiments The experiment carried out compares the possible performance impact of evaluating boolean expressions with and without short-circuit evaluation.

Figure 1 shows the execution time of a query containing a boolean expression with the following form: `simpleCondition or complexCondition`, where *simpleCondition* is an operation with a constant execution time, while *complexCondition* is an operation whose execution time depends on the size of the source model. The query is executed once for each element of a given type in the model.

Three cases has been considered in the benchmark, each one tested with and without short-circuit evaluation:

- The *simpleCondition* operation returns true for half of the elements. This means that *complexCondition* must be executed, in any case, for the other half of the elements. This can be considered an average case.
- In the second case, *simpleCondition* is satisfied for all the elements. Thus, *complexCondition* may not be evaluated. This would be the best case.
- In the third case, *simpleCondition* is not satisfied for any element. Thus, *complexCondition* must always be evaluated. This would be the worst case.

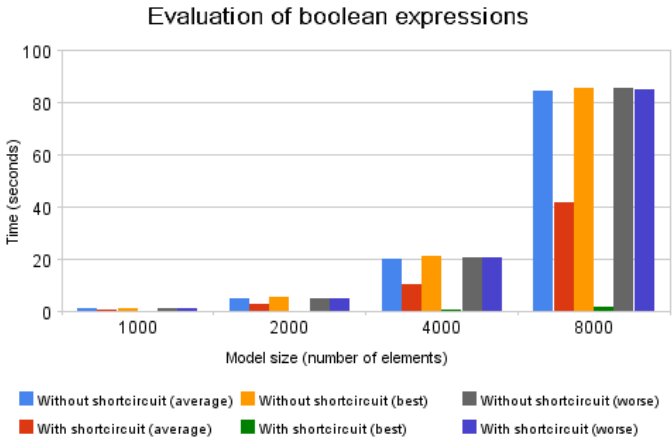

Fig. 1. Boolean expressions with and without short-circuit-evaluation

As expected, in the average case the performance improvement with short-circuit evaluation is directly proportional to the number of times the first condition is executed (i.e. its execution prevents the execution of the second one). In the best case the execution time with short-circuit evaluation is much lower because the complex condition is never executed. On the contrary, for the worst case the complex condition must always be executed, so there is no difference between having short-circuit evaluation or not.

Recommendations There are two well-known strategies to improve the performance of a boolean expression when short-circuit evaluation is considered, depending on whether it is "and" or "or".

- **And**. The most restrictive (and fastest) conditions must be placed first, that is, those conditions/queries more likely to return a "false" value. Thus, the slowest condition will be executed less often.
- **Or**. The less restrictive conditions (and fastest) must be placed first, that is, those conditions/queries more likely to return a "true" value. Again, the slowest condition will be executed less often.

If the implementation supports short-circuit evaluation then boolean expressions can be written in such a way that efficiency is considerably improved. If not, any expression can be rewritten using the rules of the following table, where the second column shows how to rewrite the expressions in OCL. It is important to notice that this table assumes that the results of the queries are defined values, i.e. true or false, but not OclUndefined. As a matter of fact, the current implementation of ATL uses a two-valued logic.

	With short-circuit	Without short-circuit
AND	`query1() and query2()`	`if query1() then query2() else false endif`
OR	`query1() or query2()`	`if query1() then true else query2() endif`

2.2 Determining an opposite relationship

Given a relationship from one model element to another, it is often necessary to navigate through the opposite relationship. For instance, if one is dealing with a tree defined by the *children* relationship it may be necessary to get a node's parent node (i.e., navigating the opposite relationship of *children*).

If the opposite relationship has been defined in the metamodel, then navigation in both directions can be efficiently achieved. However, such opposite relationship is not always available, so an algorithm to check all the possible opposite elements has to be worked out. This algorithm tends to be inefficient since it implies traversing all the instances of the opposite relationship's metaclass.

When the metametamodel supports *containment* relationships, and the reference we are considering has been defined as *containment*, then it is possible for a transformation language to take advantage of the unique relationship between an element and its container to efficiently compute the opposite one. For instance, ATL provides the `refImmediateComposite()` operation (defined in MOF 1.4) to get the container element.

Experiments The performance test has consisted in getting the owning package for all classifiers of a given class diagram. A package references its owned classifiers through a `classifiers` relationship. A helper for the `Classifier` metaclass has been created to compute the `owner` opposite relationship. The helper will be called once for each classifier of the model.

Four ways of computing an opposite relationship have been compared:

– Using an iterative algorithm such as the following (all examples are given in ATL syntax[5]):

```
helper context CD!Class def : parent : CD!Package =
    CD!Package.allInstances()->any(p |
       p.classifiers->includes(self) );
```

– Using the *refImmediateComposite()* operation provided by ATL.
– Precomputing, before starting the transformation, a map (dictionary in QVT terminology) associating elements with their parent. In the case of ATL, maps are immutable data structures, and as we will see this issue affects performance.

```
helper def : pkgMap : Map(CD!Class, CD!Package) =
  CD!Package.allInstances()->iterate(p;
                       acc  : Map(CD!Class,CD!Package) = Map{} |
    p.classifiers->iterate(c;
                       acc2 : Map(CD!Class, CD!Package) = acc |
      acc2.including(c, p)
    )
  );

helper context CD!Class def : owner : CD!Package =
  thisModule.pkgMap.get(self);
```

– The same as the previous strategy but using a special mutable operation to add elements to the map.

The results of this benchmark are shown in Figure 2. Three class diagrams with $n \times m$ elements, where n is the number of packages and m is the number of classes per package, have been considered at this time: (1) a small model with 10 packages and 250 classes per package, (2) a second model with 25 packages and 500 classes per package, and (3) a third model with 500 packages and 25 classes per package. This last model has been introduced to test how the "shape" of the model may affect the performance.

The `refImmediateComposite` operation proves to be the best option, however the performance of the "mutable map version" is comparable. The iterative algorithm is more efficient than the "immutable map version" when the number of packages is smaller than the number of classes, which will be probably the common case. The reason is that such an algorithm only iterates over the packages, while the "map version" also iterates over all the classes. The main reason for the poor numbers is that, since it is immutable, the cost of copying a map each time a new element is added is too high.

[5] The *helper* keyword and the terminal semicolon are required by ATL to syntactically identify OCL helpers. ATL also requires that type names be prefixed by the name of the metamodel defining them (e.g., *CD* here), separated by an exclamation mark.

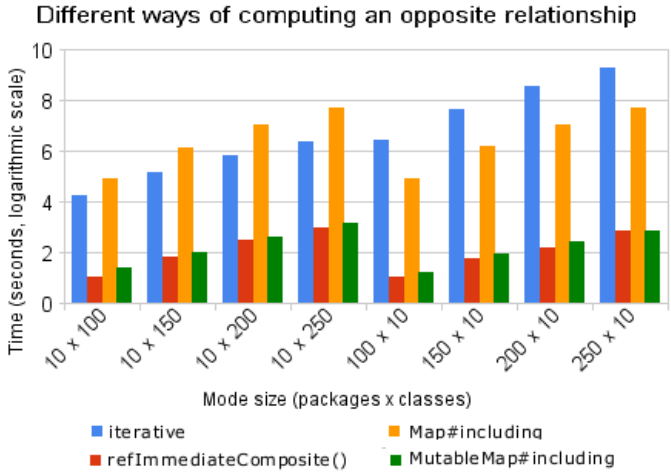

Fig. 2. Comparison of different ways of computing an opposite relationship. The logarithmic scale used for the time axis corresponds to the following formula: $1.59 \times ln(time) + 8.46$.

Recommendations According to this data, to compute the opposite of a containment relationship the `refImmediateComposite` operation should be used. If the reference is not containment or just the metametamodel does not provides this feature, using a mutable map proved to be the best option.

If the transformation language does not provide a mutable map data type, but an immutable one, the iterative algorithm or the map strategy has to be chosen according to the most usual shape of the models.

As a final remark, although maps or dictionaries are not natively supported by OCL, transformation languages usually extends OCL to implement them. For instance, ATL provides an immutable Map[6] data type (usable in side-effect free OCL expressions), and QVT provides a mutable Dictionary data type.

2.3 Collections

OCL provides different collection data types. Each type is more efficient for certain operations and less efficient for others. It is important to choose the proper collection data type according to the operations to be applied, otherwise performance may be compromised.

In this section we compare the implementation of the `including` (adds an element to a collection), and `includes` (check the existence of some element) operations for three collection data types, `Sequence`, `Set`, and `OrderedSet`. The performance results are applicable to other operations, such as `union`.

[6] In order to measure the performance using a mutable Map, we had to implement it in a test version of the ATL engine.

Experiments The benchmark for the `including` operation consists of iterating over a list of n elements, adding the current element to another list in each iteration step. The execution time for different input sizes, as well as for the three collections data types is shown in Figure 3.

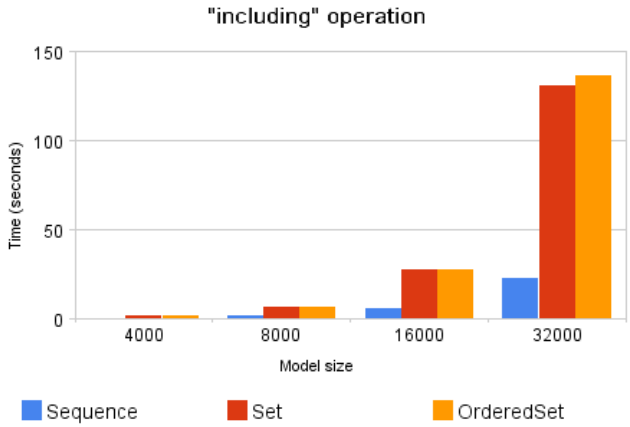

Fig. 3. Comparison of the `including` operation for different collection data types

As can be seen, `including` is more efficient for sequences than for sets. This is what one would expect, since in a sequence a new element is inserted at the tail, and there is no need to check if the element was already added. The performance of ordered sets is slightly worse than sets, basically because it is internally implemented in ATL using a `LinkedHashSet`, that is, both a hash map and a linked list must be updated in each insertion.

The benchmark for the `includes` operation consists of finding an element which is in the middle of a list of n elements. The same code is executed 2000 times. The execution time for different input sizes, as well as for the three collections data types is shown in Figure 4.

As expected the cost of `includes` is greater for sequences. However, if it executed less times, for instance 100 times, the execution time is similar in all cases, and there is not difference in using a sequence or a set. This shows that it is not worth converting a collection to a set (using `asSet`) if the number of query operations (such as `includes`) is not large.

Recommendations The decision about which collection data type to use should be based on which will be the most frequent operations. In particular, these tests show that unless one needs to make sure that there is no duplicated elements into the collection (or if the transformation logic cannot enforce it),

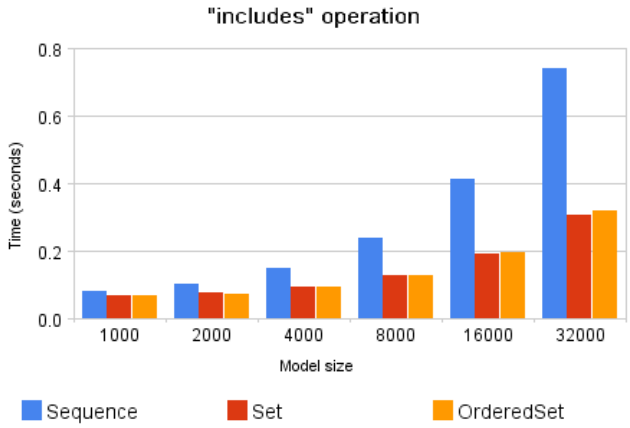

Fig. 4. Comparison of the `includes` operation for different collection data types

then the sequence type should be used, in particular when operations to add new elements are frequently called.

2.4 Usage of iterators

OCL encourages a "functional" style of navigating through a model by promoting iterators to deal with collections. Thus, queries are often written without taking into account the efficiency of the expressions, but just trying to find out a readable, easy or more obvious solution.

For instance, it is common to come across OCL code like expression (a) shown below, which obtains the first element of a collection satisfying a condition. However, expression (b) may be more efficient since the iteration can finish as soon as the condition is satisfied. Of course, an optimizing compiler could rewrite (a) into (b).

```
(a) collection->select(e | condition)->first()
(b) collection->any(e | condition)
```

Thus, it is important to take into account the contract of each iterator and operation to choose the most appropriate one, depending on the computation to be performed.

Experiments To assess whether it is really important, from a performance point of view, to be careful when choosing an iterator we have compared these two ways of finding the first element satisfying a condition in a list of n elements. In this benchmark the condition is satisfied by all elements after the middle of the list. This means that option (a) will return a list of $n/2$ elements.

The first time we ran this benchmark, the execution time for both cases (a) and (b) was the same. The reason is that the current implementation of `any` in ATL does not finish the iteration as soon as possible, but it is equivalent to "select()->first". Thus, a new optimized version was implemented and its performance is also compared.

Figure 5 shows the execution time for the three cases: using the original version of `any`, using a fixed version and with "select()->first". It also shows another case which is explained below.

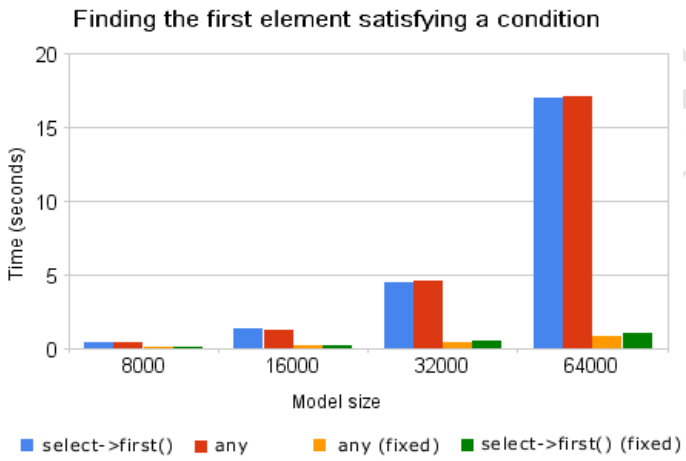

Fig. 5. Finding the first element satisfying a condition.

According to the proposed benchmark, the execution time of "select()->first" should be worse than using `any`, but not so much. We looked into this issue and the reason is related to the implementation of the `select` operation in ATL. It internally uses the standard OCL `including` operation to add an element to the result each time the condition is satisfied. Since `including` is an immutable operation the whole partial result is copied for each selected element. That is why as the size of the list grows the execution time grows exponentially.

We implemented an optimized version of `select` which uses a mutable operation to add elements to the result. As can be seen in Figure 5, its performance is greater than the original, but it is also comparable to `any`. The main reason for this result is that the transformation execution involves a constant time which is independent of the iterator execution time. When such constant time is removed, the `any` iterator is around 150% faster.

Recommendations The `select` iterator should be used only when it is strictly needed to visit all elements of the collections. Iterators such as *any*, *exists*, *in-*

cludes, etc. should be used to avoid iterating the whole collection. In any case, the benchmark results show that if the `select` iterator is properly implemented then it can provide a performance comparable to other iterators.

2.5 Finding constant expressions

In rule-based transformation languages, rules are executed at least once for each instance matched against the source pattern, so all expressions within the rule may be executed once for each rule application. When such expressions depend on the rule's source element, then it is inevitable to execute them each time. However, those parts of an expression which are independent of variables bound to the source element can be factorized in a "constant", so that they are executed only once when the transformation starts. Some transformation engines do not support this kind of optimization, so it has to be done manually.

As an example, let us consider the following transformation rule, which transforms a classifier into a graphical node. The condition to apply the rule is that the classifier (instance of `Classifier`) must be referenced by another element which establishes whether or not it is drawable (`Drawable`). Since the filter is checked for each classifier, all elements of type `Drawable` are traversed each time the engine tries to find a match.

```
rule Classifier2Node {
  from source : CD!Classifier (
     DrawModel!Drawable.allInstances()->exists(e | e.element = source)
  )
  to Graphic!GraphNode ...
}
```

A more efficient strategy is to compute in a constant attribute all the drawable elements, so that the transformation can be rewritten in the following way:

```
helper def : drawableElements : Set(CD!Classifier) =
  CD!Drawable.allInstances()->collect(e | e.element);

rule Classifier2Node {
  from source : CD!Classifier (
    thisModule.drawableElements->includes(source)
  ) ...
}
```

Experiments The rule shown above has been executed for several input models (the same number of elements of type `Classifier` and `Drawable` is assumed). Figure 6 shows the results for the following three cases: (1) without pre-computing the common query code into a constant, and using the original ATL version of `exists`, (2) the same but using an optimized version of `exists` which finishes the iteration as soon as it finds the required element, and (3) using the strategy of pre-computing a constant.

As can be seen the third strategy has a cost which is significantly lower than the two others, and does not grow as fast (the algorithm complexity is $O(n+m)$), while the first one has a cost of $O(n \cdot m)$, where n is the number of elements of type `Classifier` and m is the number of elements of type `Drawable`.

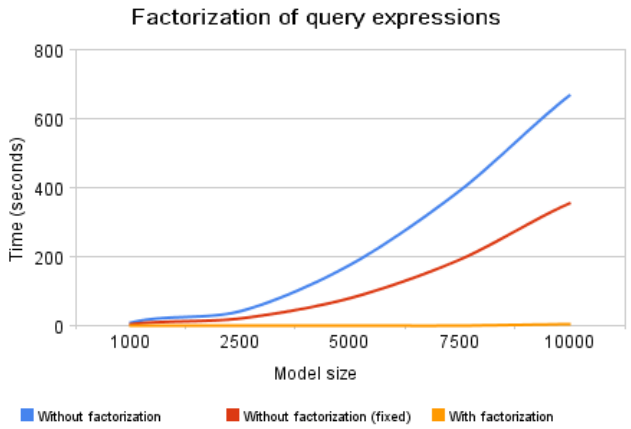

Fig. 6. Performance impact of the factorization a common expression into a constant.

Recommendations It is possible to easily identify expressions within rules and helpers which can be factorized into some constant because they usually rely on `allInstances()` to get model elements without navigating from the rule's source element. Therefore, the transformation developer should be aware of this kind of optimization and apply it whenever possible.

Also, it is worth noting that using a `let` statement (or similar) is a good practice to factorize expressions at the local level.

3 Related work

In [6] the need for developing benchmarks to compare different OCL engines is mentioned. The authors have developed several benchmarks that can be found in [1]. However, they are intended to compare features of OCL engines, rather than performance.

In [8] and [9] the authors present several algorithms to optimize the compilation of OCL expressions. They argue that its optimizing OCL compiler for the VMTS tool can improve the performance of a validation process by 10-12%.

Regarding performance of model transformation languages few work has been been done. In [11] a benchmark to compare graph-based transformation languages is proposed. In [7] some general recommendations about how to improve performance of model driven development tools are presented, but neither concrete examples or experimental data are given.

4 Conclusions and future work

In this paper we have presented several optimization patterns for OCL-based transformation languages. These patterns address common navigation problems in model transformations from a performance point of view. For each pattern we have provided several implementation options along with performance comparison data gathered from running benchmarks.

The contribution of this work is twofold, on the one hand these patterns may be useful as a reference for model transformation programmers to choose between different implementation alternatives. On the other hand, they provide some empirical data which is valuable for tool implementors to focus on the optimization of some common performance problems.

As future works we will continue defining benchmarks for model transformations in order to identify more patterns related to performance. We are also improving our framework for benchmarking to consider other transformation languages. Beyond the individual patterns that are being identified, we are also looking at improving and generalizing a method for finding, identifying and classifying transformation patterns.

References

1. OCL benchmarks, http://muse.informatik.uni-bremen.de/wiki/index.php/ocl_benchmark_-_core.
2. A. V. Aho and J. D. Ullman. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley, 1977.
3. K. Beck. *Implementation Patterns*. Addison-Wesley Professional, 2006.
4. F. Jouault and I. Kurtev. Transforming Models with ATL. In *MoDELS 2005: Proceedings of the Model Transformations in Practice Workshop*, Oct 2005.
5. D. S. Kolovos, R. F. Paige, and F. A. Polack. The epsilon transformation language. In *1st International Conference on Model Transformation, ICMT'08*, pages 46–60, 2008.
6. M. Kuhlmann and M. Gogolla. Analyzing semantic properties of ocl operations by uncovering interoperational relationships. In *Ocl4All: Modelling Systems with OCL, MoDELS 2007 Workshop*, Nashville, Tennessee, 2007.
7. B. Langlois, D. Exertier, and S. Bonnet. Performance improvement of mdd tools. In *EDOCW '06: Proceedings of the 10th IEEE on International Enterprise Distributed Object Computing Conference Workshops*, page 19. IEEE Computer Society, 2006.
8. G. Mezei, T. Levendovszky, and H. Charaf. Restrictions for ocl constraint optimization algorithms. Technical report, Technische Universitat Dresden, Genova, Italy, October 2006.
9. G. Mezei, T. Levendovszky, and H. Charaf. An optimizing ocl compiler for metamodeling and model transformation environments. In *Software Engineering Techniques: Design for Quality*, pages 61–71. Springer, 2007.
10. OMG. Final adopted specification for MOF 2.0 Query/View/Transformation, 2005. www.omg.org/docs/ptc/05-11-01.pdf.
11. G. Varro, A. Schurr, and D. Varro. Benchmarking for graph transformation. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 79–88. IEEE Computer Society, 2005.