

CODE GENERATION AND TRANSFORMATION FOR VISUAL QUERIES IN UML

Wiktor Filipowicz¹, Piotr Habela¹, Krzysztof Kaczmarek², Krzysztof Stencel¹

*¹Polish-Japanese Institute of Information Technology, Warsaw, Poland
{wiktor, habela, stencel}@pjwstk.edu.pl*

*²Warsaw University of Technology, Warsaw, Poland
k.kaczmarek@mini.pw.edu.pl*

Abstract: Declaring constraints or specifying precise behaviour in UML requires a standard language of expressive power comparable with today's query languages. OCL fulfils that role. However, its complicated syntax and inherent complexity of some expressions motivate the search for visual aids for expression creation. This paper presents an approach based on the Query by Example (QBE) concept and UML object diagrams. After a brief presentation of the visual language, we focus on describing OCL code generation and its subsequent transformation into a more readable and optimum form.

1 INTRODUCTION

In the VIDE project [1] we aimed at investigating the capability of executable modelling in the spirit of Model Driven Architecture (MDA) [2], following the standardized modelling languages: UML 2.1 [3], with its Actions and Structured Activities seamlessly integrated with OMG OCL 2.0 [4] expressions. This resulted in a textual programming language for precise UML behaviour specification; OCL expressions being its potentially most complex constituents. The visual nature of the high level UML constructs, inherent complexity of the expressions in some applications and the peculiarities of OCL syntax, motivated the search for a visual notation, especially for the expression part of the language. Several attempts and experimenting led us to a conclusion that the most promising solution would be exploiting the well-known concept of Query By Example (QBE) [5, 6] and applying it to UML models.

In this paper we describe a declarative, object-oriented, QBE-based approach to visualizing OCL expressions over a UML model instance, focusing on the underlying code generation and query rewriting issues.

2 RELATED WORK

The QBE language was developed in the mid 1970's at the IBM Research by Moshe Zloof [6]. QBE is available as a commercial IBM product as a part of Query Management Facility (QMF) interface option to DB2. Another well known implementation is the QBE part of the Paradox DBMS [7], and became one of the reasons of its great success. A visual interface similar to the original QBE is provided by Microsoft in its widespread MS-Access DBMS [8]. A simple and straightforward textual version of object QBE in pure Java/.NET is implemented in the db4o [9].

3 VISUAL EXPRESSIONS IN OBJECT QUERY BY EXAMPLE

The language design, called Object Query By Example (OQBE) went beyond the UML/OCL metamodel [3, 4], providing instead new, dedicated constructs based on the notion of *example of an object* (accompanied by *link example* and *attribute example*). The actual OCL expression is produced through the transformation of such an expression model. The functionality, syntax and motivations for design decisions for our OQBE have been presented in [10]. Here, we briefly summarize the essential features. OQBE introduces a number of new important solutions specific to the object-oriented data model and to OCL. These features include:

- Support for both object identity and value comparisons,
- Returning complex results of nested structures (*tuples* in the OCL terminology),
- Dedicated construct for the general quantifier (*->forAll* iterator operation in OCL),
- Querying class *extents* or from variables / attributes visible in the scope.

Object examples connected with link examples can represent respective data structure examples. An attribute example may play different roles:

- **Predicate** – if the attribute itself or the result of operation applied to it (including various kinds of comparison) provides a Boolean.
- **Output element** – if an attribute example has the *output* flag attached.
- **Sorting criterion** – if the *sort* flag with needed properties is attached.

Before we present a detailed algorithm in the subsequent section, let's describe the evaluation of an expression declared in OQBE as a sequence of the following steps (disregarding optimizations):

1. Retrieving objects represented by the entry points of a graph.
2. Performing joins according to the navigation through link examples.
3. Performing selections according to the constraints on the examples.
4. Sorting the result.
5. Building the final result by performing the projection into one or more fields indicated by the output flags.

Instead of an attribute name an arbitrary OCL code can be embedded.

Although the readability considerations limit the usage of connections between attribute examples, a simple construct called *Comparator* was introduced in order to represent comparisons between attribute or object examples. The figure below (based on a simple auction site example) illustrates the expression “return users who placed a bid in auctions whose seller has a lower rating than their own”.

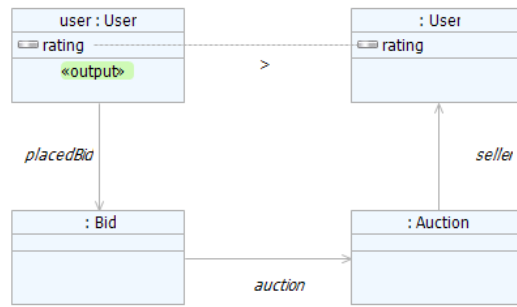


Figure 1. Comparator, navigation links and object-typed output example

One of the features that make QQBE different from the traditional QBE solutions based on the relational model is the availability of nested structures in expression result construction. In the textual OCL this is realized by nesting subexpressions inside an explicit Tuple constructor. We call its visual counterpart the *nested region*.

A similar visual construct was introduced for the general quantifier (*forall*). From the example graph outside the *forall* region that is connected with the examples inside the region, only those instances will be included in the result for whose all navigations through the link passing the region boundary the condition is met.

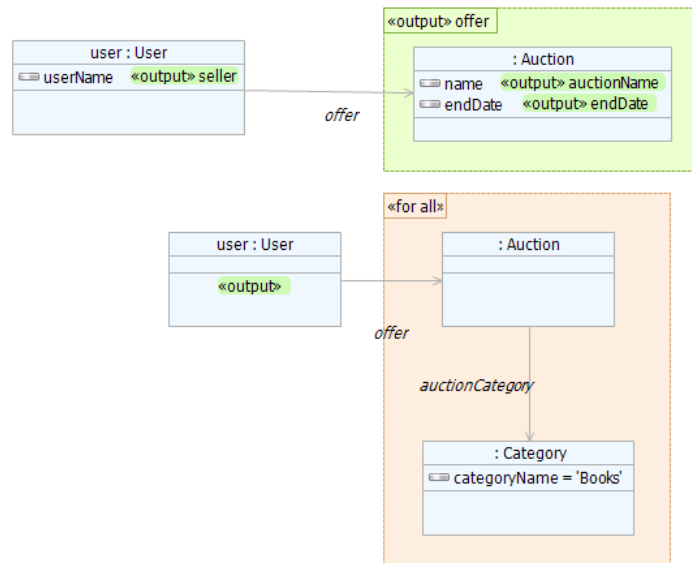


Figure 2. Nested result and forall region examples (“seller with their offer list nested” and “user who sells books only”)

4 OCL CODE GENERATING PROCEDURE

QQBE can be mapped to a number of query/programming languages. In this project OCL is used and thus QQBE will be mapped to and executed as OCL. Each QQBE query can be mapped to OCL in more than one way. Furthermore, not all OCL constructs can be mapped to QQBE, since it covers only a subset of OCL querying capabilities. Therefore, the reverse mapping (from OCL to QQBE) was not considered in this project. The mapping procedure is quite complex mainly due to the lack of the join operator in OCL. In the following we show a workaround, however the generated queries would be much simpler, if the join were present in OCL.

The mapping is constructed in several steps. An OQBE query is a directed graph with vertices representing examples and edges representing navigation links and comparators between examples.

1. We construct the set S being the minimal set of examples such that all examples are reachable by navigation links from some examples of this set.
2. For each example E from the set S we construct a query selecting database objects described by this example. It has a form:

```
generator->select(condition)
```

where the *generator* is either the extent of the class of the example E (if E is not named) or a variable with the same name as the name of example E . The *condition* is the filtering predicate for the example E .

3. Next we combine all queries constructed in point 2 into $query_n$ which returns all possible tuples built from the results of queries from the step 2. The number n is the cardinality of S . The stepwise construction of this query is performed as follows:

```
query1 = generator1->select(condition1)->collect ( it | Tuple { v1 = it } )
```

```
query2 = query1->collect(it | generator2 ->select(condition2)
->collect(vTemp | Tuple {v1 = it.v1, v2 = vTemp}))
```

...

```
queryk = queryk-1->collect(it | generatork ->select(conditionk)
->collect(vTemp | Tuple {v1 = it.v1, v2 = it.v2, ..., vk-1 = it.vk-1, vk = vTemp}))
```

...

```
queryn = queryn-1->collect(it | generatorn ->select(conditionn)
->collect(vTemp | Tuple {v1 = it.v1, v2 = it.v2, ..., vn-1 = it.vn-1, vn = vTemp}))
```

4. Then $query_n$ is enriched with fields corresponding to subsequent examples which can be reached from currently available (already reached examples). The $query_m$ always returns tuples with m fields representing m examples. Below we will show how to add the example E_{m+1} . Since it is not a member of S , it must be reachable from S by navigation links. Let us assume that E_{m+1} is reachable from example E_i by link name *linkName*. Let us also denote by $condition_{m+1}$ the filtering condition for example E_{m+1} . In this step we construct the following query:

```
querym+1 = querym->collect(it | vi.linkName->select(conditionm+1)
->collect(vTemp | Tuple {v1 = it.v1, v2 = it.v2, ..., vm = it.vm, vm+1 = vTemp}))
```

5. We repeat the step 4 until all examples are added to the constructed query.
6. Then we collect the sorting criteria and add them at the end of the query obtained after finishing the steps 4-5. The criteria are added from left to right starting from the least significant criterion. The trail added to the query is the following (the $criterion_k$ is most significant, while the $criterion_1$ is least *significant*):

```
->sortedBy(criterion1)->...->sortedBy(criterionk)
```

7. Finally we add the collect operator to limit the output to only these items which are marked as query outputs. The result will be a tuple if the number of outputs is greater than 1. Let us assume that $output_1, \dots, output_r$ specify the outputs of the

constructed query; each of them is either an example of one of its attributes). The trail added in this step is one of the two:

```
->collect(output1)
```

if the number of query outputs is one.

```
->collect(Tuple{name1=output1, ..., namer=outputr} )
```

if the number of query outputs is greater than one, where $name_1, \dots, name_r$ are the names of items of the output tuple. The name $name_i$ is either the name given to the «output» stereotype or the name of the output element (attribute or example), if the «output» stereotype is not named.

5 REWRITING GENERATED OCL QUERIES

Queries generated by the algorithm described in the previous section are verbose and unreadable. They are not intended to be further developed textually by a programmer. Even, if it is intended only to be directly executed, it can be a challenge for the query engine. In this section we will list query optimization methods which can be used. The rewriting strategies are presented in [11, 12, 13, 14]. In case of generated OCL queries the following methods are iteratively performed:

1. removal of dead subqueries,
2. removal of unused variables,
3. exploiting associativity of the OCL *collect* operator:

```
a->collect(b|c)->collect(d) ≡  
a->collect(b|c->collect(d))
```

4. flattening of vain collect-Tuple-collect chain:

```
a->collect(Tuple{v1=v})->collect(v)≡  
a->collect(v)
```

5. flattening vain collect-select-collect chain:

```
a->collect(v|b->select(e)->collect(v)) ≡ a->select(e)
```

6. and a rather technical issue like the alpha-conversion of iterator variable names.

The third, fourth and fifth optimisation methods above were necessary, because of some non-optimal syntax decisions made during the design of OCL. If OCL were improved to be more programmer-friendly, no one would ever think about those methods. Usually, as the result of optimisation we get a much simpler query, which can be further sped up e.g. by using path or classic field indices. Most of optimisation steps are devoted to removal of subqueries which are introduced by the general algorithm but are redundant in the query at hand. The presence of those subqueries is unavoidable so that we can make the generation algorithm universal and powerful. This must not be regarded as a disadvantage of the generation algorithm, since we have tools which remove unnecessary parts of generated queries.

5 CONCLUSIONS

In this paper we have outlined the concept of the *object query by example* for UML. Contrary to UML as a whole however, OCL is known to relatively few professionals, and may pose a barrier for a broader adoption for a UML based

programming language. Hence our proposal of a visual language that offers yet higher level of abstraction compared to OCL. The prototype has been implemented atop of our experimental ODBMS “ODRA” [15]. The core challenge of this work, outlined in this paper, is the code generation and its further transformation that requires applying optimization methods based on rewriting.

LITERATURE

1. Falda, G., et al, 2008.b Executable Platform Independent Models for Data Intensive Applications. In *Computational Science - ICCS 2008, 8th International Conference, Kraków, Poland, June 23-25, 2008, Proceedings, Part III*. Springer, pp. 301-310.
2. Object Management Group: *OMG Model Driven Architecture*, <http://www.omg.org/mda/>
3. Object Management Group, 2007: *Unified Modeling Language: Superstructure version 2.1.1*. formal/2007-02-05.
4. Object Management Group, 2006: *Object Constraint Language version 2.0*. formal/06-05-01.
5. Elmasri, R., Navathe, S.B. 2004. *Fundamentals of Database Systems*, Addison Wesley, Fourth Edition.
6. Zloof, M., 1975. Query By Example. NCC, AFIPS, 44.
7. Business Wire, 1995a. *Paradox 5.0 for Windows Voted Best Buy*, Feb 6, 1995 http://findarticles.com/p/articles/mi_m0EIN/is_1995_Feb_6/ai_16407965
8. Microsoft *Microsoft Access Home Page*, <http://office.microsoft.com/en-us/access>
9. Paterson, J., Edlich, S., Hörning, H.R., 2006. *The Definitive Guide to db4o*, Apress, Berkeley, California
10. Habela, P., Charfi, A., Spriestersbach, A., 2008. *Visual modelling of behaviour in data-intensive business applications*. <http://www.vide-ist.eu/publications.html>
11. Płodzień, J., 2000. *Optimization Methods in Object Query Languages*. Ph.D. Thesis. Institute of Computer Science, Polish Academy of Sciences. <http://www.sbql.pl/phds/PhD%20Jacek%20Plodzien.pdf>
12. Plodzien, J., Subieta, K., 2001 a. Applying Low-Level Query Optimization Techniques by Rewriting. In *DEXA 2001*. pp. 867-876
13. Plodzien, J., Subieta, K. 2001 b. Query Optimization through Removing Dead Subqueries. In *ADBIS 2001*. pp. 27-40.
14. Subieta, K., et al., 2008. *Stack-Based Approach (SBA) and Stack-Based Query Language (SBQL)*. <http://www.sbql.pl>
15. Adamus, R., et al, 2008. Overview of the Project ODRA. In *Proceedings of the First International Conference on Object Databases, ICOODB 2008*. pp.179-197.