OCL as the Query Language for UML Model Execution*

Piotr Habela[3], Krzysztof Kaczmarski[3,4], Krzysztof Stencel[2,3], and Kazimierz Subieta[1,3]

[1] Institute of Computer Sciences of the Polish Academy of Sciences, Warsaw, Poland
[2] Institute of Informatics, Warsaw University, Warsaw, Poland
[3] Polish-Japanese Institute of Information Technology, Warsaw, Poland
[4] Faculty of Mathematics and Information Science, Warsaw University of Technology, Warsaw, Poland
{habela, stencel, subieta}@pjwstk.edu.pl,
k.kaczmarski@mini.pw.edu.pl

Abstract. Despite the specification of OCL mentions "query language" as one of its possible applications, there are rather few efforts in that direction. However, the problem becomes central where applying MDA to data intensive application modelling is considered. Recently added UML elements of Actions and Structured Activities make it possible to represent a level of detail similar to the one of common programming languages, but data processing requires adequate querying capability as well. As the OMG specification of the UML family, the Object Constraint Language becomes the most obvious candidate to serve this purpose. In this paper we research this role of OCL. Especially, we address the issues of seamless integration with UML metamodel and the useful features of query languages that are missing from OCL.

Key words: OCL, query language, MDA, UML

1 Introduction

The approach of model-driven software development and the MDA initiative in particular, sketch the vision of the next big step in raising the level of abstraction and flexibility of programming tools. While any method which focuses on modelling activities can be considered "model-driven", the key expectation behind MDA is achieving a productivity gain through the automating software construction based on models. This results in a significant shift of expectations regarding modelling constructs – from being merely a semi-formal mean for outlining and communicating project ideas, to machine-readable specification demanding precise semantics. Thus MDA creates a spectrum of model applications. In this paper we focus on one of them, namely executable models. If models are to be executable, precise semantics is inevitable. Furthermore, executable models could blur the distinction between modelling and programming, since they would facilitate automatic production of executable code.

* Supported by the EC 6-th FP, Project VIDE, IST 033606 STP

In our approach to this problem we strive to combine available standards in order to ease the adoption of the developed solution. If most things to be learned by a prospective user are relatively well-known (which should concern OO standards), the user will be more eager to accept and employ such a solution. In our opinion UML Structures unit seems to be rich and versatile enough to be considered as a foundation for a data model used in the platform-independent development. A number of semantic details need to be clarified to achieve that aim though. To make the model complete, the means of imperative programming need to be available at the PIM level. To raise the intuitiveness and productivity compared to the mainstream platform-specific technologies, the statements and queries should be integrated into a single language in a truly seamless way. We will also provide an execution engine for PIMs as a reference implementation. It is essential as a modelling tool component serving for platform-neutral model validation.

Within the VIDE project [14], as a part of visual development toolset we have developed VIDE-L, which stands for VIDE language. VIDE-L is a textual language for PIM level representation of behaviour in terms of UML Actions and Activities with the Object Constraint Language (OCL) as an expression/query language. Most applications developed nowadays are database applications in the sense that their functionality highly depends on persistent and shared business data. On the PIM level we have a chance to avoid the impedance mismatch and provide a query language seamlessly integrated with the host language. We have chosen to use OCL as this query language, since its specification claims that it is suitable for this purpose. In this paper we will describe our reference implementation of OCL as a database query language and a textual language for UML Actions and Activities as the host language. This reference implementation of VIDE-L is based on the Stack-Based Approach to query languages [8–10]. We use its query language (SBQL) as the assembly language underlying OCL executions. OCL queries are mapped onto SBQL queries which are eventually executed. SBA/SBQL is a well-elaborated execution framework with very general data store model, strong typing [11] and query optimisation [12],[13]. This solves the issue of type checking and optimisation of OCL queries. Those features also prove that the idea of the language is realistic in terms of the requirements the code generated from it needs to meet at the execution platforms. As a small off-topic remark we note that SBA has also a powerful updateable view mechanism, which makes OCL views stand just behind the corner.

The rest of the paper is organized as follows. In Sec. 2 ODRA DBMS is presented. Sec. 3 and Sec. 4 describe the problems solved during the implementation of OCL. Sec. 5 gives some examples of using OCL in method bodies, ad-hoc queries and programs. Sec. 6 outlines open details and some of standards' refinements needed. Sec. 7 shows related works while Sec. 8 concludes.

2 ODRA DBMS

ODRA (Object Database for Rapid Application development) is a database management system which implements SBA and OCL on top of it. Users can post queries either in SBQL or OCL. SBQL is treated as the assembly language, so OCL

queries are compiled to and executed as SBQL queries. If the client query is formulated in OCL, it will be mapped onto SBQL during parsing. OCL-SBQL mapper is a component of the ODRA parser. Since SBQL is very powerful, there is no limitation on query languages which can be implemented this way. Soon, we plan to add to ODRA mapper components for other query languages: XQuery and RDQL. ODRA implements type checking and query optimization. Mapping any query language onto SBQL makes it possible to employ its optimization and strong typing capabilities. After mapping onto SBQL the query is type checked, and optimized by rewriting (also view expansion if possible) and exploitation of indices. Then it gets compiled to bytecode. The compiled query is then executed by the bytecode interpreter.

3 OCL Grammar Disambiguation

The main problem with the implementation of OCL was implied by its ambiguous syntax. The specification [3] contains the whole chapter titled *Concrete syntax*. Unfortunately this syntax is (probably intentionally) ambiguous. Let us take a look at the following obvious example fragments of rules taken from the specification:

```
OclExpressionCS ::= PropertyCallExpCS
OclExpressionCS ::= VariableExpCS
...
PropertyCallExpCS ::= ModelPropertyCallExpCS
...
ModelPropertyCallExpCS ::= AttributeCallExpCS
...
AttributeCallExpCS ::= simpleNameCS isMarkedPreCS?
...
VariableExpCS ::= simpleNameCS
```

If the parser has to analyze an `OclExpressionCS` and sees an identifier token (i.e. `simpleNameCS` in the terminology of OCL) on the input, it cannot make a choice which rule to reduce. Among the rules shown above it could be `VariableExpCS` or `PropertyCallExpCS`. However, there are a lot of other possibilities in the grammar for syntactical analysis of an identifier. . It is a well known fact, that the OCL grammar as defined in the language specification is ambiguous [15]. The specification uses contextual information, which is not available during a purely syntax based analysis (such as parsing).

Since ODRA DBMS is written in Java, we have chosen Cup from many available LALR(1) parser generators. When run for the first time on the OCL grammar, Cup reported 104 shift/reduce conflicts and 164 reduce/reduce conflicts. Moreover, 23 rules were never reduced, mainly due to conflicts. We strived to disambiguate this grammar and eventually succeeded. Most of the branches of `OclExpressionCS` had to be deleted or factored upwards to the rule for `OclExpressionCS`. We faced the last mile problem. 20% of the effort was devoted to the first 80% conflicts, but 80% of the effort was devoted to the last 20% conflicts. The less the number of

conflicts was, the harder was to remove of remaining conflicts. We managed to retain the weirdest part of the syntax, i.e. Smalltalk-like prefix and infix method calls:

```
OperationCallExpCS ::=
            OclExpressionCS simpleNameCS OclExpressionCS
...
OperationCallExpCS ::= simpleNameCS OclExpressionCS
```

The question is why the specification [3] calls it a concrete syntax, if it is not suitable for parsing? This rather defines an abstract syntax. The so called 'disambiguating rules' require the parser to consult some meta information in a database. This is not the way how generated LALR(1) parser make their choices.

4 Improvements Towards Data Intensive Operations

First of all, we have to agree with [18] that OCL's syntax is not the best for database programmers. It is not intuitive, too complicated and too elaborate. However, for the reasons mentioned in the introduction we follow the standard.

We are also conscious that database community would welcome modification of several operations. One of them would be a Cartesian product operator, which is too limited in its abilities. Another candidate for extension would be introduction of a transitive closure operation, which is as important for modern databases as recursion for programming languages. Its lack, even if justified in OCL, certainly limits queries possible to be expressed.

For basic consistency with database systems we added some features, which are not modifying the language syntax and would be obvious for any programmer. OCL defines only two aggregation functions: `size` and `sum`. Using them we can compute the number of employees of a department or the total salary in a department, e.g.:

```
Dept->allInstances()->select(name='toys').employs
  ->size()

Dept->allInstances()
  ->select(name='toys').employs.salary->sum()
```

We added `min`, `max` and `avg` aggregation, so that more statistics can be computed:

```
Dept->allInstances()
  ->select(name='toys').employs.salary->avg()

Dept->allInstances()
  ->select(name='toys').employs.salary->min()

Dept->allInstances()
  ->select(name='toys').employs.salary->max()
```

The rest of the implementation of OCL was relatively easy, since SBQL (the query language we mapped OCL onto) is quite powerful compared to OCL. There were almost no problems in finding this mapping.

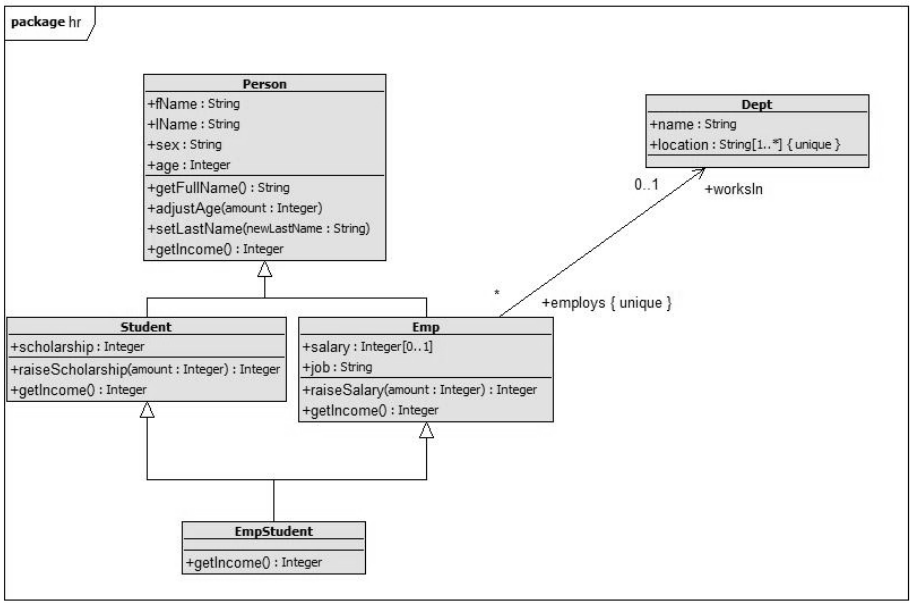

Fig. 1. Example database schema.

5 Examples

In this section we will show some examples of the code written in VIDE-L (OCL is seamlessly embedded in VIDE-L). We use the database schema from Fig. 1. The schema also contains multiple inheritance which is smoothly implemented in ODRA. VIDE-L with OCL will be used in method bodies and ad-hoc queries.

Let us start from definitions of some methods. Of course we have to augment the OCL `context` phrase, since it was not originally intended to define arbitrary method bodies. We will use the keyword `body` to indicate that a particular `context` phrase introduced a method body. Here are the methods of the class Person (apart from the method *getIncome* which will be discussed later):

```
context Person::getFullName() : String
  body { return fName + ' ' + lName; }

context Person::adjustAge(amount : Integer)
  body { age += amount; }

context Person::setLastName(newLastName : String)
  body { lName := newLastName; }
```

The method *raiseScholarship* is rather straightforward:

```
context Student::raiseScholarship(amount : Integer) :
Integer   body {     scholarship += amount;
                     return scholarship; }
```

The method *raiseSalary* is more complex since we have check if a subobject *salary* exists, because it is optional. If it does not exist, it must be created and inserted by appropriate UML Action represented in textual syntax. Note the smooth integration of the OCL query and VIDE-L.

```
context Emp::raiseSalary(amount : Integer) : Integer
  body {    if (self.salary->size() = 0)
              self.salary insert amount;
            else self.salary := amount;
            return self.salary; }
```

The *getIncome* method is overridden in all classes in the hierarchy of class *Person*:

```
context Person::getIncome() : Integer
  body { return 0; }

context Student::getIncome() : Integer
  body { return scholarship; }

context Emp:: getIncome() : Integer
  body { return
      if salary->size() = 0 then 0 else salary endif; }

context EmpStudent:: getIncome() : Integer
  body {
    return scholarship
     + if salary->size() = 0 then 0 else salary endif;}
```

These methods and multiple inheritance are properly handled by the ODRA run-time, so to compute the total income of all persons in the database we issue the query:

```
Person->allInstances()->collect(getIncome())->sum()
```

We can also find the average salary for each department by means of the following query which reminds a dependent join:

```
Dept->allInstances()->collect(d |
  Tuple { dept = d,
          totalSal = d.employs.getSalary()->avg() } )
```

Now we will show some examples of ad-hoc imperative statements which can be executed against such a database. The first one assigns minimal salary in a department to all employees of this department who do not have established salaries yet:

```
Emp->allInstances()->select(salary->size()=0) foreach
  { e | e.raiseSalary(e.worksIn.getSalary()->min()); }
```

The next one moves all employees from the Toys department to the Research department. We will show two ways of finding all subject employees. The first one starts from the Toy department (*Dept*) and collects all its employees:

```
Dept->allInstances()->select(name='Toys')
  ->collect(employees) foreach { e |
      e unlink worksIn;
      e link worksIn to Dept->allInstances()
        ->select(name = 'Research'); }
```

The second one starts from employees (*Emp*) and selects those who work in the toy department. The body of the loop is the same.

```
Emp->allInstances()->select(worksIn.name = 'Toys')
foreach { e |
  e unlink worksIn;
  e link worksIn to
    Dept->allInstances()->select(name = 'Research'); }
```

The last example program gives 10% raise to all students which are also employees of departments located in Warsaw. Note that the selection of departments by location is slightly more complex because it is a multi-valued attribute.

```
EmpStudent->allInstances()
  ->select(worksIn.location->exists(l | l = 'Warsaw') )
foreach { es | es.raiseSalary(es.getSalary * 0.1)); }
```

As we can see even complex tasks can be solved by relatively simple OCL queries. Furthermore, imperative constructs of VIDE-L embed OCL queries in a very natural way. VIDE-L has statements which handle collections (like the **foreach** statement) returned by OCL queries. Each expression in this language is a query and vice versa. The impedance mismatch is mostly eliminated this way.

6 Improving the Integration with the Imperative Part

Despite the obvious benefit of reusing popular specification (that is, OCL) for the purpose of a model-level query language, a number of issues arise resulting from the fact this purpose was not fully foreseen at the time OCL was designed. Some of the problems can be removed by updating UML and OCL specifications to fully integrate them and to reduce redundancy between UML Actions and OCL expressions. To this extent the postulates would include:

- Completing the UML expressions metamodel part with the means of accessing local variables defined by UML – e.g. inside method bodies.
- Reducing the number of UML actions by those that overlap with OCL (various "read" actions dealing with: properties, variables, extents, links, and *self* variable).

- Unifying a type system between OCL and UML to assure bidirectional interoperability (so that not only OCL can read any UML-defined features, but also that OCL expression results can be consumed by UML actions and activities).

To illustrate, how the pragmatic features of the language are dependent on unifying the types between UML and OCL, consider the case of tuple results. The example below illustrates, how the style of coding changes (for a pure query method), depending whether the tuple results are allowed for UML methods.

The example (see Fig. 2 for its schema) assumes producing a nested data structure retrieved from objects of several different classes. This may be needed for constructing report or e.g. for feeding a GUI forms. The first version (*getOrderDetailsObjects* operation) uses objects being created inside a method. The second version (*getOrderDetailsTuples*) attempts to take full advantage of the OCL, and hence is implemented by a single OCL expression.

Although, due to the way UML class diagrams can describe nested structures, the both approaches are similarly complex in terms of their static model, the difference of method behaviour code is significant.

Consider the first variant that assumes that Tuple types are supported only inside OCL expressions and (accordingly to straightforward understanding of UML Actions validity constraints) each assignment deals with a single value.

```
ShopModule.getOrderDetailsObjects(in cName : String) :
OrderDetailsClass [0..*] {
  oList : Bag [0..*] (OrderDetailsClass);
  order->select(customer.name=cName) foreach { o |
    oDetail : OrderDetails =
     OrderDetailsClass create { id := o.ID;
                          custName := o.customer.name;
                          comments := o.comments };
    o.items foreach oDetail.item insert
      ItemInfoClass create { prodName := product.name;
                             prodQuantity := quantity};
    oList insert oDetail; }
  return oList;}
```

Now we can compare it with the variant in which tuples are allowed to be used in operation result declarations.

```
ShopModule.getOrderDetailsTuples(in cName : String) :
OrderDetailsTuple [0..*] {
  return order->select(customer.name=cName)
   ->collect(o | Tuple {    id = o.ID,
     custName = o.customer.name, comments = o.comments,
     item = o.item->collect( Tuple { prodName =
          product.name, prodQuantity = quantity}) }); }
```

As can be seen, allowing the use of Tuple types in method signatures can spare us a number of statements. In this particular example, two variable declarations, two foreach loops (realized by UML's *ExpansionRegion* construct), two object creation actions and several respective assignment actions are avoided.

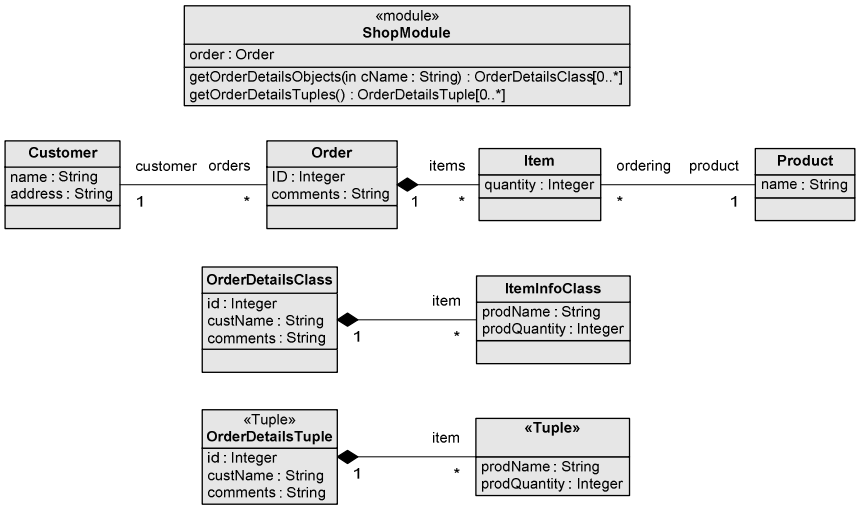

Fig. 2. Exemplary schema involving tuple results.

7 Related Work

Although our VIDE-L seems to be the first application of OCL as a database query language, there were other OCL based tools which should be mentioned. Most of them implemented OCL in version 1.4 or 1.5. We know of only two that are compliant with the latest specification. Dresden OCL Toolkit [16] provides an OCL 2.0 parser and interpreter but is a metamodel based solution. It provides an API to define and execute constraints of UML models in version 1.5. It is not intended to be a database query engine, thus it does not provide any kind of optimizations and it does not define any database specific operators. Another implementation of OCL is MDT-OCL (Eclipse Model Development Tools) plug-in [17]. It provides a single metamodel integrated from UML 2.1 and OCL 2.0 plus OCL parser and interpreter. Its main purpose is to evaluate UML model constraints, thus to work on M2 meta-level, but may also be used to query data stored as a model instance. It is probably one of the best OCL based tools but still cannot be accepted as a database solution.

8 Conclusions

In this paper we described the implementation of OCL as a database query language. We presented the problems which were solved during this work (especially with ambiguous grammar) and augments which have been added to OCL so that it could be called a query language. Our implementation efforts have a wider purpose. OCL was embedded into a high level programming language VIDE-L. In our opinion this

embedding is perfectly smooth and allows formulating even complex queries and program quite compactly. This creates a good starting point for further research which aims at development of modeling tools which follow MDA philosophy of creating machine-readable executable specifications to be executable on PIM level. Since these tools will use standards like OCL and UML Actions and Activities, it can be easier adopted by the community of developers and modellers.

References

1. Object Management Group: Unified Modeling Language: Superstructure version 2.1.1, February 2007. www.omg.org/cgi-bin/doc?formal/2007-02-05
2. Mellor, S.J., Scott, K., Uhl, A. , Weise, D.: MDA Distilled: Principles of Model-Driven Architecture. Addison Wesley (2004)
3. Object Management Group: Object Constraint Language version 2.0, May 2006. www.omg.org/cgi-bin/doc?formal/2006-05-01
4. Hailpern, B., Tarr, P.: Model-driven development: The good, the bad, and the ugly. IBM Systems Journal: Model-Driven Software Development. 45(3), (2006)
5. Ambler, S.W.: A Roadmap for Agile MDA. Ambysoft, (2007), www.agilemodeling.com/essays/agileMDA.htm
6. Thomas, D. A.: MDA: Revenge of the Modelers or UML Utopia?, IEEE Software 21 (3), 15–17 (2004)
7. Warmer, J., Kleppe, A.: Object Constraint Language, The: Getting Your Models Ready for MDA. Addison Wesley (2003).
8. Subieta, K.: Stack-Based Approach (SBA) and Stack-Based Query Language (SBQL). sbql.pl (2008)
9. Subieta, K., Kambayashi, Y., Leszczyłowski, J.: Procedures in Object-Oriented Query Languages. In: Proc. VLDB Conf., pp. 182-193. Morgan Kaufmann (1995)
10. Subieta, K.: Theory and Construction of Object-Oriented Query Languages. Editors of the Polish-Japanese Institute of Information Technology, in Polish (2004)
11. Lentner, M., Stencel, K., Subieta, K.: Semi-strong Static Type Checking of Object-Oriented Query Languages. In: SOFSEM, pp. 399-408 (2006)
12. Płodzień, J., Kraken, A.: Object Query Optimization through Detecting Independent Subqueries. Inf. Syst. 25(8). 467-490 (2000)
13. Płodzień, J., Subieta, K.: Query Optimization through Removing Dead Subqueries. In: ADBIS, pp. 27-40 (2001)
14. Visualize all moDel drivEn programming, www.vide-ist.eu/
15. Akehurst, D., Patrascoiu, O.: OCL 2.0 - Implementing the Standard for Multiple Metamodels, In: Electronic Notes in Theoretical Computer Science, Volume 102, 2 November, pp. 21-41, Proceedings of the Workshop OCL 2.0 - Industry Standard or Scientific Playground? (2004)
16. Dresden OCL Toolkit, dresden-ocl.sourceforge.net
17. Model Development Tools OCL, Eclipse Foundation, wiki.eclipse.org/MDT,
18. Vaziri, M., Jackson, D.: Some Shortcomings of OCL, the Object Constraint Language of UML, Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00), pp. 555-562. IEEE Computer Society (2000)