OCL as a Specification Language for Business Rules in Database Applications

Birgit Demuth, Heinrich Hussmann, and Sten Loecher

Dresden University of Technology, Department of Computer Science

Abstract. Business rules are often specified only implicitly by applications to express user-defined constraints. OCL provides the chance to explicitly and automatically deal with business rules when building object-oriented applications. We investigate how OCL constraints can be handled in database applications as one of the most important kind of business applications. Based on our OCL toolset prototype and earlier research work we particularly experiment with various strategies for the evaluation of OCL constraints in object-oriented applications which use relational databases. For this work, a flexible SQL code generator is needed which can be used and adapted for different relational database systems and different object-to-table mappings. We implement such a database tool as an additional module for our OCL toolset using XML techniques.

1 Introduction

Business rules are not a new approach to define or constrain some aspects of the business. They represent business knowledge and govern how the business processes should execute [7]. There are many definitions at different abstraction levels of what business rules are. Within the context of data-intensive applications we only consider a subset of business rules: integrity constraints which are supported by database management systems (DBMS). We explicitly aim at non-trivial integrity constraints which may affect in relational DBMS tuples of several tables.

Business rules are often specified only implicitly by applications to express user-defined constraints. Facing this problem, an ECOOP'98 Workshop [12] discussed and listed requirements for tools and environments for business rules. Some of these requirements such as the use of a declarative language to express business rules or the support for maintaining software integrity are fullfilled by the UML Object Constraint Language (OCL) [14]. OCL provides the chance to explicitly and automatically deal with business rules when building UML-based object-oriented applications. Several authors already investigated the use of OCL for expressing business rules (e.g. [24] [7] [5]). Some OCL tools have also been built to allow experiments with formally specified business rules [18] [10], [14]. In first applications of OCL, these tools were particularly used during the software development itself such as for simulation, code generation and test automation

[10]. Using OCL as a language for business rule specification, extended investigations can be carried out which aim at maintaining the software integrity by checking business rules during the execution of a business process. This check can be done by two basic strategies:

- The theoretically best way is the *immediate constraint check*: The OCL expressions are immediately evaluated when objects are changed. In databases, the unit of consistency is the transaction. In object-oriented applications, check points are more difficult to find because most runtime systems like the Java Virtual Machine do not provide transaction support. In [25], this problem is addressed.
- A more realistic way in many cases is an *independent constraint check*: The OCL expression is only checked at selected points to achieve a trade-off between consistency maintenance and efficiency. As a consequence, inconsistent states of objects respectively of the database are possible.

According to the classification in [9] we basically distinguish active and passive constraints. Whereas *active integrity constraints* maintain consistency by executing actions, *passive integrity constraints* only prevent data manipulation operations which violate the consistency. Furthermore, there are two primary methods by which integrity constraints can be specified, applied and enforced in database applications [11]:

- through the application programs (*application enforced* constraints).
- through the DBMS itself (*database enforced* constraints)

Both methods are controversial [23], [13], [3]. Apriori we focus on database-enforced constraints because then we can exploit all advantages of the database technology [5]. The OMG proposal for a Data Modeling UML profile [19] includes constraints as stereotyped operations in a relational manner like `<<check>>` representing passive constraints and `<<trigger>>` representing active constraints. Based on this UML profile, an object-to-table mapping is given in [20]. In [5], we discussed the mapping of UML objects including OCL constraints to a relational database schema in a very similar way. However, we found many limitations both in the object-to-table mapping supported by most environments and in the maturity of current DBMS. In this paper, we address these problems and present some ideas to make object-to-table and OCL-to-SQL mappings more flexible and practicable.

Section 2 discusses an advanced approach for supporting OCL in object-relational applications by application- and database-enforced constraints. The basic idea is applying an independent constraint check by the evaluation of a database view representing invalid tuples. In Section 3 we describe a prototype implementation of an SQL tool including code generation for OCL constraints and its integration into our OCL toolset and CASE tools like Argo/UML. Finally, section 4 summarizes our results and gives an outlook for further research and tool development.

2 Approaches Supporting OCL Constraints in Object-Relational Applications

2.1 Discussion of different approaches

Up to now a number of different approaches and suggestions for the implementation of OCL specified business rules using the current database technology have been made. All these approaches support the immediate constraint check and share the problem of being not efficient for runtime evaluation or not practicable on current database systems.

The implementation of the OCL-to-SQL patterns from [5] shows that to the best of our knowledge no current relational DBMS supports the Full SQL92 level [11] as far as automatic integrity checks are concerned. The required check clause is provided by all advanced DBMS, but only for Intermediate SQL. That means that the search condition contained in the check clause shall not contain a subquery. Furthermore, standalone integrity constraints including an arbitrary number of tables (SQL92 assertions) are not supported at all. Therewith, only simple one-tuple related constraints can be checked in real DBMS.

In [22], procedural mapping patterns were investigated to translate an OCL expression into code executable on relational database systems. The resulting code of such a translation was a (proprietary) procedure such as a Sybase' Transact SQL stored procedure. The evaluation of the constraint can be done by calling this procedure. The right moment for doing this is at the end of a transaction that is before the transaction commit. One drawback of this approach is the dependence of database integrity from the applications instead of being part of the database constraints.

Another way to implement passive as well as active integrity constraints is to apply triggers to check respectively maintain the integrity after each data manipulation statement. One problem is that triggers are standardized foremost by SQL-99 [6], but are already implemented by different dialects.

2.2 The VIEW approach

Driven by our implementation experience and motivated by database literature [21], [17] we propose an approach to realise either an independent or an immediate constraint check for an SQL based implementation of OCL specified business rules. The basic element of our approach are SQL *view*s generated from OCL invariants. Each single OCL invariant is translated into a separate view definition. The result of a view evaluation is a set of tuples from the constrained table respectively the object which violate the specified business rule. This approach yields a number of advantages:

- The usage of SQL features available in most database systems makes the usage of OCL in database design practicable. A view allows to evaluate a complex search condition which is part of an integrity constraint. In this sense, a view can substitute the evaluation of the not supported assertions respectively "multiple table" check constraints in current DBMS.

- The generation of views from OCL invariants is basically not different from generating SQL92 assertions. For this reason we can use our already developed technique for SQL code generation.
- The views are based on declarative SQL code and, therefore, are subject of query optimization of the DBMS.
- The generation of declarative SQL code from OCL invariant specifications is simpler than the generation of procedural DBMS code.
- The views can be integrated into different constraint evaluation strategies. Then it can be decided when to evaluate the constraint and what to do if some constraint is violated.

The last mentioned item should be discussed in detail. Most real world applications have their own requirements according to performance, integrity, and other design issues. Some applications may prefer performance over integrity, others may demand full integrity at each database state[1]. For this reason, it is not realistic to always apply the typical immediate constraint check for any kind of application. Another question arises with respect to the handling of faulty data. Some applications may prefer to get notified about all constraint violations, others may prefer the handling of faulty data by the database system. The proposed view approach can be used to support various kinds of requirements. We consider the following three variations:

Application driven view evaluation. The evaluation of a view is not coupled to any database integrity mechanism like assertions or triggers. Instead, it is evaluated by an application. In the given context, the evaluation of the view representing invalid tuples can be seen as an independent constraint check. Furthermore, it is a hybrid method of checking application- and database-enforced constraints. For example, to access to a database, an object-relational middleware with an own transaction approach is often used. The middleware can evaluate the views just before the transaction commit is executed. If one of the views returns any tuples, the middleware is able to rollback the current transaction or do some treatment for the faulty data. This approach has the disadvantage of taking the integrity control away from the database system, but allows the middleware or the application itself to decide on the moment of constraint evaluation and thus leaves some space for the treatment of performance problems. Another possible way to use "integrity views" are database monitor applications which support database administrators to maintain huge data stores and keep them clean from faulty data.

Assertion replacement. The views can be used by triggers which evaluate the constraints after each critical data manipulation operation. When any constraint violation is found, the trigger should rollback the current transaction and send an appropriate error message to the invoking application. This approach can be used as a replacement for SQL92 assertions, if the database system does not support such a feature.

[1] In [17] these kinds of integrity are called *qualified* respectively *absolute* integrity.

ECA trigger template. To support the idea of active database systems a trigger template can be generated which must be edited by the application or database developer respectively. The template should support the ECA (Event-Condition-Action) rule paradigm [1]. Such a trigger is evaluated after each critical data manipulation operation (the *event*). If the *condition* holds, for instance a constraint is violated, the *action* part is executed. This way, faulty data can be treated before storing them in the database.

For a better understanding of how our concept works, we give a simple example. Suppose there is a class called `PERSON` with two attributes: the `age` of a person and a Boolean flag which indicates whether this person `isMarried` or not. In our case, this class is mapped to a single table with the according columns `AGE` and `ISMARRIED` respectively. A simple business rule is that all persons which are married should be at least 18 years of age. The respective OCL expression is given below:

```
context Person
inv ageOfMarriage: (isMarried = true) implies (age>=18)
```

The first step is the translation of this invariant into a corresponding SQL view evaluating all tuples of the table `PERSON` which violate the specified business rule. We use an adapted version of the OCL INVARIANT pattern from [5]:

```
create view AGEOFMARRIAGE as
(select * from PERSON SELF
where not (not (ISMARRIED = true) or (AGE>=18)))
```

According to the three approaches for the use of integrity views, the evaluation of the given business rule is as follows:

Application driven view evaluation. Invoking the evaluation of views by an application can be done in various ways. Using Java and JDBC to access to the database, the following code can be used:

```
ResultSet rs = theStatement.executeQuery(
  "select nvl(count(*),0) from AGEOFMARRIAGE "
);
if (rs.next().getInt(0) > 0) {
  // integrity error handling
}
```

Note that `nvl()` is an Oracle specific function which, in the above example, returns `0` if `count(*)` is `null`, otherwise it returns the number of selected tuples.

Assertion replacement. The trigger template for the assertion replacement would look like this:

```
create trigger TR_AGEOFMARRIAGE
after insert or update or delete on PERSON
begin
  if (select nvl(count(*),0) from AGEOFMARRIAGE) > 0 then
    raise_application_error("Integrity error !", 20900);
  end if;
end;
```

ECA trigger template. If the treatment of faulty data is necessary, the use of the according trigger template should be preferred. In this example, the action code would be implemented directly by the trigger body:

```
create trigger TR_AGEOFMARRIAGE
after insert or update or delete on PERSON
begin
  if (select nvl(count(*),0) from AGEOFMARRIAGE) > 0 then
    // todo: add action code here
  end if;
end;
```

Due to the simplicity of the example, an important fact is not shown which must be considered in more detail. If an integrity view uses more than one table of the database to evaluate a business rule, the constraint evaluation must be done after manipulation of all of these tables.

For instance, a view can be generated from an OCL expression which uses navigation to express constraints on a model. Suppose we add a second class to our example stated above. This class is called `Car` and is also mapped to a single table. Between class `Person` and class `Car` exists an one-to-many association which describes the ownership between persons and cars. This association will be mapped by inserting the primary key of the `PERSON` table (`PID`) to the `CAR` table. A constraint is that each person can be the owner of at most two cars:

```
context Person
inv maxCars: self.car->size <= 2
```

This OCL expression is mapped using the NAVIGATION pattern from [5] into the following view:

```
create view MAXCARS as
(select * from PERSON SELF
where not ((select count(CID)
            from CAR
            where PID = self.PID) <= 2)
```

As one can see, the view uses both the table `PERSON` and the table `CAR` to evaluate the constraint. It is important to evaluate the view if any of the two tables is modified. If the view MAXCARS is integrated into any trigger evaluation mechanism, this means that a trigger must be created for each table to evaluate the according view.

3 Extended OCL toolset

The implementation of the OCL constraint translation to SQL (called the **OCL2SQL tool**) follows the above explained VIEW approach and is done by a modular extension of the OCL toolset [10]. Our toolset has already proven to be a stable and flexible environment for the development of OCL tools [25]. In the following subsections, we describe the design and outline the implementation of the OCL2SQL tool based on a first prototype implementation.

3.1 Experience with the OCL-to-SQL pattern catalogue

We use the OCL-to-SQL pattern catalogue from [5] to translate OCL invariants to SQL code. Our first prototype implementation of an SQL code generator implementing these patterns has shown the following problems:

Object-relational mapping. The translation of OCL expressions to SQL code is dependent from the underlying object-to-table mapping. Therefore, one requirement for the SQL code generator should be a flexible interface that allows the integration of different object-to-table mappings. Unfortunately, the patterns are described only based on the commonly used one-object-to-one-tuple mapping.

Metadata. The mapping of operations on metadata is not considered and thus should be added by new patterns.

Full SQL92 level. The mapping patterns take advantage of the SQL92 full level specification. Assertions over any number of tables, and derived tables in the from clause of select statements are important SQL features for the patterns. How explained above, current database systems lack the support for these features, especially the first one.

The iterate problem and sequences. The iterate operator and OCL sequences are two features which have to be discussed in more detail.

3.2 Design

The current version of the OCL2SQL tool takes a static UML model and a number of OCL invariants as input and generates an according DDL script including a database schema as well as view definitions and trigger templates representing the constraints as output. It consists of the following components:

Model repository As stated above, the OCL2SQL tool needs information about the used static UML model and the number of OCL invariants specified on this model. Since we aim at the integration of the OCL2SQL tool into different environments like UML CASE tools, it is mandatory to provide appropriate interfaces which can be implemented for different use cases. A more loose integration is the use of XMI files [26] for static UML model information. The OCL toolset already provides the necessary component to use this technology. For a tight integration as it has been realised for Argo/UML, the "model interfaces" must be implemented by a CASE tool integration component accessing whose repository.

SQL code generator The core component is the SQL code generator which generates the SQL code for an OCL invariant based on the parsed, type-checked and normalised OCL expression given as an abstract syntax tree. The "SQL code" generated by this component is a view definition such as the AGEOFMARRIAGE example in subsection 2.2. The implementation of the SQL code generator is explained in the following subsections in more detail. To make such a SQL code generator work, we need some additional information about the underlying object-to-table mapping. Since there exists a great number of different object-to-table mappings, an interface is provided for the integration of various strategies.

Schema generator The first idea was to integrate the SQL code generator with the database schema generation functions available by most UML-CASE tools. Unfortunately, we recognized that the quality of the generated schemas does not match flexible requirements. Therefore, we had to implement an own object-to-table mapping and to provide an interface for later CASE tool integration efforts.

Trigger template generator In contrast to the SQL code generator, the trigger template generator is rather simple. It takes the output of the SQL code generator and produces a number of triggers according to the view specifications and user requirements.

3.3 XML coded pattern refinement

The implementation of the eight rather general OCL-to-SQL patterns requires a further refinement of the patterns to make them applicable in code generation. Beyond the refinement, a flexible specification of the patterns is needed to adapt them to different SQL dialects for their practical use in DBMS. We decided to use XML [27] to describe the refined patterns because of the following reasons:

- The XML files can be comfortably edited by XML editors and thus are easily adaptable to a certain SQL dialect.
- An XML file containing a set of refined patterns can be directly interpreted by the SQL code generator written in Java.
- XML technology supports well defined structures of documents.

The structure of an XML document is specified by a Document Type Definition (DTD). An adequate DTD called **CODEGEN** describing OCL-to-SQL patterns

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT  catalog      (description?, pattern*)>

<!ATTLIST  catalog name CDATA #REQUIRED>
<!ELEMENT  description  (#PCDATA)>
<!ELEMENT  pattern      (template)*>
<!ATTLIST  pattern rule ID #REQUIRED>
<!ELEMENT  template     (li)+>
<!ATTLIST  template     spec CDATA #REQUIRED
                        rem  CDATA #IMPLIED>
<!ELEMENT  li           (#PCDATA | param)*>
<!ATTLIST  li connector (true|false) "false">
<!ELEMENT  param        EMPTY>
<!ATTLIST  param name   CDATA #REQUIRED>
```

Fig. 1. The CODEGEN DTD

is shown in fig. 1. According to this DTD, the XML file is structured as a catalogue which consists of a description and an arbitrary number of patterns. Each pattern is associated to an OCL grammar rule. For example, the patterns that describe the mapping of OCL features like operations over collections are associated to the `featureCall` grammar rule. Since a single pattern usually consists of several mapping descriptions, a pattern consists of at least one SQL code template. Each SQL code template again is described by lines of code. The templates are identified by the `spec` parameter to distinguish them from each other. Let us consider for example the OCL grammar rule `featureCall`. There are several mapping templates for collection operators like `collect`, `select`, `reject`, and `forAll`. The templates can contain parameters respectivly placeholders which must be replaced by further templates. One example for such a pattern rule is represented in fig. 2. The example shows the template which maps the OCL operator `collect` to an SQL query block and refines the general QUERY pattern [5] (indicated by the `rem` parameter). As explained above, from a translation point of view, the template belongs to the grammar rule `featureCall` and needs two further templates for the complete code generation. That is, the parameters `column` and `table` must be replaced with appropriate SQL code.

3.4 SQL code generator

In our first prototype implementation based on the OCL-to-SQL92 patterns we already got some experience with the implementation of an SQL code generator. The recent SQL code generator represents the adapted prototype and realises the generation of view definitions. The design of the SQL code generator is done under consideration of two primary aspects:

```
<?xml version="1.0">
<!DOCTYPE catalogue SYSTEM "CODEGEN.dtd">
<catalog name="EntrySQL">
  ...
  <pattern rule="feature_call">
    ...
    <template spec="collect" rem="QUERY">
      <li>select <param name="column"></li>
      <li>from <param name="table"></li>
    </template>
    ...
  </pattern>
  ...
</catalogue>
```

Fig. 2. Example of the *feature call* pattern rule

- The SQL code generator should be completely based on the OCL toolset because the toolset provides a quite reliable platform for syntactical analysis, typechecking and normalisation of OCL expressions.
- The XML based approach for the translation of OCL expressions to SQL statements has been proven and therefore should be used.

For the integration of different code generators like for Java or SQL, the OCL toolset provides two interfaces [8]. At first a code generator must implement the interface `CodeGenerator`. Such a code generator produces a certain number of code fragments. These code fragments are stored in objects of classes which implement the interface `CodeFragment` (see fig. 3).

The patterns described in [5] aim at the generation of declarative SQL code only. As already described in the preceeding subsection, each pattern was designed to encapsulate one OCL language concept. The equivalent SQL code allows the nesting of SQL expressions according to the structure of the given OCL expression. To implement this concept, we chose a general syntax driven approach.

The feature of being rather general than specific to SQL code generation led to the following design decision. The core functionality of the strategy is encapsulated in a separate class called `DeclarativeCodeGenerator` since the strategy seems to be suitable for the generation of declarative target code preferably. Subclasses of this class generate code fragments that will be stored in objects of the class `DeclarativeCodeFragment`.

The actual generation of SQL code is realised by the class `SQLCodeGenerator` which implements only features specific to the SQL code generation problem. The `SQLCodeGenerator` uses an XML file based on the CODEGEN DTD. The `CodeAgent` parses the XML file and fetches the appropriate code template for the code generator. Therefore it is prepared with a number of parameters which

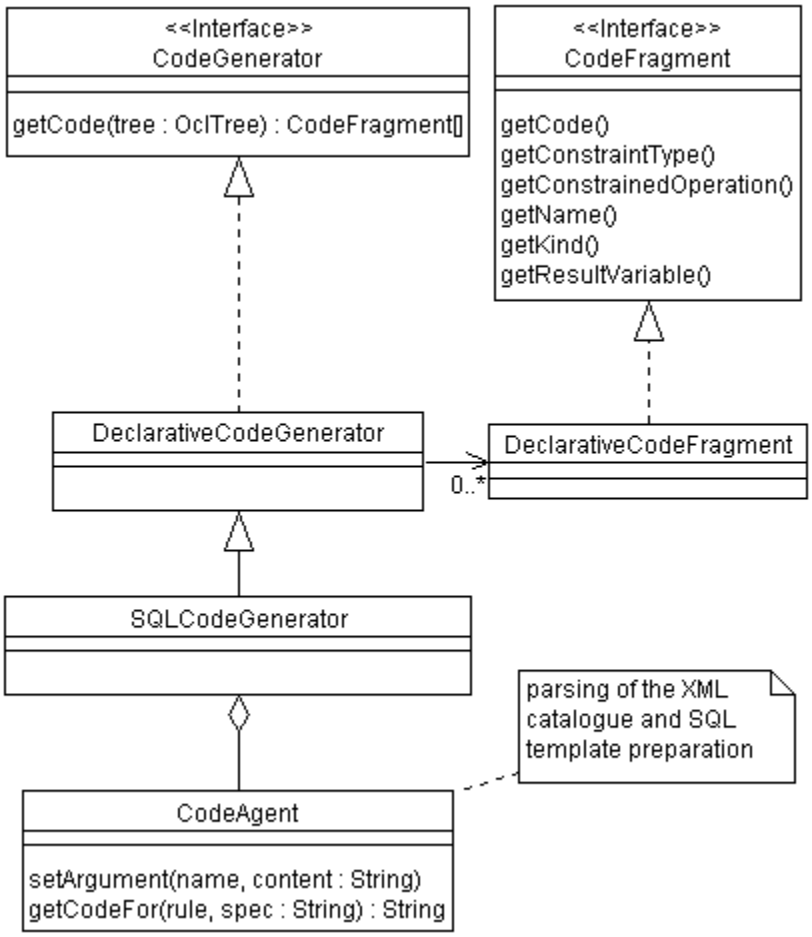

Fig. 3. SQL code generator

are supposed to replace the placeholders in the templates. Then the `getCode()` method is called with the pattern rule ID for the requested OCL grammar rule and the accurate template specification (`spec` parameter). The placeholders allow an arbitrary nesting of templates. Then the `getCode()` method returns a prepared SQL code template which can be further handled by the SQL code generator.

Experiments with the first SQL code generator prototype have already shown the applicability of the XML based approach for code generation. We used two XML coded pattern catalogues, one for an "ideal" DBMS supporting SQL92 full level and another one for Oracle 8i. Oracle8i is an example for an advanced DBMS where we could demonstrate our mapping approach and its limitations on current database management systems. Now, we reuse the code generator prototype for the implementation of the above described OCL2SQL tool.

3.5 CASE tool integration

An important requirement of tools supporting OCL is the tight integration with UML-CASE tools. As explained in the preceding subsections, our OCL2SQL tool has a number of well defined interfaces which allow the integration of the tool into different environments. A new module of the OCL toolset is a comfortable OCL editor which includes besides editing of constraints features like a toolbar and adequate error messages. The according user interface is designed to integrate the OCL editor not into a specific CASE tool, but into various environments. Currently it is tested with the Open Source CASE Tool Argo/UML and also serves as test environment for the OCL2SQL tool. The screenshot in fig. 4 gives an impression of the new OCL editor integrated into Argo/UML.

4 Conclusion

In this paper, we reported on our recent research results using OCL constraints as business rules in object-relational database applications [5] as well as extending the OCL toolset [10] for SQL code generation. The automatic translation of complex OCL expressions to database integrity constraints specified by SQL rises some serious problems which we try to solve by trigger-based techniques or, especially if performance plays an important role, by an unusual approach which we call independent constraint check. All these techniques use an integrity view which is defined as the set of tuples from the constrained tables respectively objects violating the according OCL invariant. We design an SQL tool extending the OCL toolset which supports in a flexible way both different constraint evaluation strategies and different SQL dialects of the DBMS vendors. Furthermore, different object-to-table mappings can be handled by the design of flexible interfaces.

Our plans for further practical and theoretical investigations as well as OCL toolset development are the following:

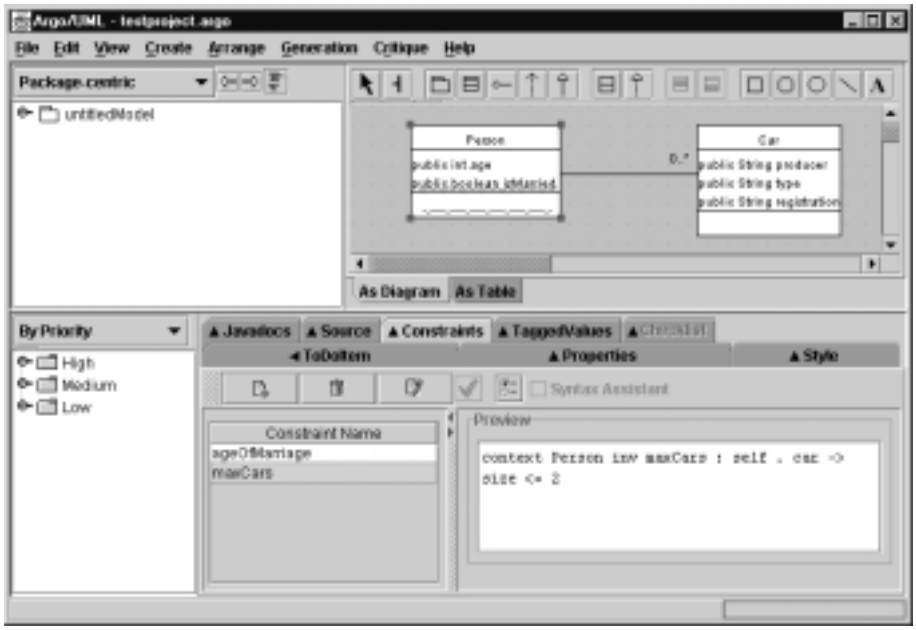

Fig. 4. OCL editor integrated into Argo/UML

- So far we only considered the use of OCL invariants for the specification of business rules in database applications. However, which role play pre and post conditions for methods handling persistent objects?
- The OCL toolset should realise important requirements for the handling of business rules [12]. According these requirements it seems desirable to extend the OCL toolset by modules such as
  - a repository for business rules
  - dedicated browsers
  - supporting different scopes of business rules (application and database enforced constraints) including different constraint evaluation strategies
  - adaptable for changing, refining and removing existing rules
  - a debugger for systems containing lots of business rules
  - a conflict detection
  - a reasoning engine

For the future, an important objective of our work is the practical use of the OCL toolset in case studies to gain further experience with application- and database-enforced constraints.

Acknowledgment: The authors would like to thank Frank Finger, Ralf Wiebicke and Steffen Zschaler for their contributions to the OCL toolset.

References

1. ACT-NET Consortium, The Active Database Management System Manifesto: A Rulebase of ADBMS Features. SIGMOD Record 25(1996)3:40-49
2. Argo/UML Page, http://www.ArgoUML.com
3. Blaha, M., Premerlani, W.: Object-Oriented Modeling and Design for Database Applications. Prentice Hall, 1998
4. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, 1999
5. Demuth, B., Hussmann, H.: Using OCL Constraints for Relational Database Design. in: UML'99 The Unified Modeling Language, Second Int. Conference Fort Collins, CO, USA, October 1999, Springer, 1999
6. Eisenberg, A., Melton, J.: SQL: 1999, formerly known as SQL-3. ACM SIGMOD Record, 22(1999)1, 131-138
7. Eriksson, H.-E., Penker, M. Business Modeling with UML. Business Patterns at Work, John Wiley & Sons, Inc., New York, 2000
8. Finger, F., Design and Implementation of a Modular OCL Compiler. diploma thesis, Dresden University of Technology, 2000
9. Herbst, H. et al, The specification of business rules: a comparison of selected methodologies. in: Methods and Associated Tools for the Information System Life Cycle. Elsevier, Amsterdam, 1994
10. Hussmann, H., Demuth, B., Finger, F.: Modular Architecture for a Toolset Supporting OCL. in: UML'2000 - The Unified Modeling Language. Advancing the Standard, Third Int. Conference York, UK, October 2000, Springer, 2000
11. Melton, J., Simon, A.: Understanding the New SQL: A Complete Guide. Morgan Kaufmann, 1993
12. Mens, K. et al, Workshop Report - ECOOP'98 Workshop 7 Tools and Environments for Business Rules. in: Object oriented technology: ECOOP'98 Workshop Reader. Springer, 1998
13. O'Neil, P., Database - principles, programming, performance. Morgan Kaufmann, 1994
14. OCL Center, Klasse Objecten, http://www.klasse.nl/ocl/index.htm
15. OCL Page, Dresden University of Technology, http://dresden-ocl.sourceforge.net/
16. OMG UML v. 1.3 specification, http://www.omg.org/cgi-bin/doc?ad/99-06-08
17. Motro, A., Integrity= validity + completeness. ACM Transactions on Database Systems, 14(1989)4,480-502
18. Richters, M., Gogolla, M., Validating UML Models and OCL Constraints. in: UML'2000 - The Unified Modeling Language. Advancing the Standard, Third Int. Conference York, UK, October 2000, Springer, 2000
19. Rational. The UML and Data Modeling. Whitepaper TP-180, 2000, http://www.rational.com
20. Rational. Mapping Objects to Data Models with the UML. Whitepaper TP-185, 2000, http://www.rational.com
21. Ross, K., Srivastava, D., Sudarshan, S., Materialized view maintenance and integrity constraint checking: Trading space for time. in: Proc. of the ACM SIGMOD Int. Conference on Management of Data, Montreal, Canada, 1996, ACM Press, 1996
22. Schmidt, A.: Untersuchungen zur Abbildung von OCL-Ausdruecken auf SQL. Dresden University of Technology, diploma thesis, 1998
23. Spencer, B., Business Rules vs. Database Rules. A Position Statement. in: Object oriented technology: ECOOP'98 Workshop Reader. Springer, 1998

24. Warmer, J., Kleppe, A.: The Object Constraint Language. Precise Modeling with UML. Addison-Wesley, 1999
25. Wiebicke, R., Utility Support for Checking OCL Business Rules in Java Programs. diploma thesis, Dresden University of Technology, 2001
26. OMG, XMI SMIF Revised Submission (ad/98-10-06). http://www.omg.org
27. W3C, Extensible Markup Language (XML). http://www.w3.org