Incremental Checking of OCL Constraints through SQL Queries

Xavier Oriol and Ernest Teniente

Department of Service and Information System Engineering
Universitat Politècnica de Catalunya – BarcelonaTech
{xoriol,teniente}@essi.upc.edu

Abstract. We propose a new method for efficiently checking OCL constraints by means of SQL queries. That is, an OCL constraint is satisfied if its corresponding SQL query returns the empty set. Such queries are computed in an incremental way since, whenever a change in the data occurs, only the constraints that may be violated because of such change are checked and only the relevant values given by the change are taken into account. Moreover, the queries we generate do not contain nested subqueries nor procedures. In this way, we take advantage of relational DBMS capabilities and we get an efficient check of OCL constraints.

Keywords: OCL, Constraints Checking, SQL

1 Introduction

A conceptual schema is the description of an Information System in terms of the data it should contain and the operations available to users to modify such data [1]. To define conceptual schemas, the Object Management Group (OMG) has defined the UML/OCL standards [2,3]. Broadly speaking, UML is used for specifying a class diagram, i.e. the structure of the data, and OCL for stating the conditions (i.e. the constraints) that should always be satisfied by the data.

We aim at defining an efficient method to perform integrity checking of OCL constraints. That is, to efficiently check whether the OCL constraints of the schema are satisfied by the data contents. Such problem arises in several situations like conceptual schema execution in animation tools (like USE [4]) or checking whether a model satisfies the constraints defined in its metamodel in MDA [5]. Unfortunately, there are not efficient OCL checkers able to deal with medium-large scenarios [6].

One way to efficiently check OCL constraints is aimed at reducing such problem to check the emptiness of some SQL query [7]. Intuitively, given an OCL constraint, we can build an SQL query that returns all instances that violate it. Thus, the OCL constraint is satisfied if and only if the SQL query is empty. In this way, we benefit from all optimization techniques of current DBMS for query answering. Moreover, since SQL is widely used for storing data from constrained domains, bringing an efficient method for checking constraints based on SQL might be integrated in current industrial systems without crossing technologies.

To our knowledge, there are two implemented tools that perform such translation: OCL2SQL [8] and MySQL4OCL [9]. However, their translation should be further optimized to scale up for efficient integrity checking in large scenarios. Mainly, because whenever a change in the data occurs (e.g. an insertion of a new instance), the whole query is recomputed, when it probably just need to check whether the updated data should appear as a result of the query.

The main goal of this paper is to overcome the previous drawback by proposing a translation from OCL constraints to SQL queries which allows to compute them incrementally by a relational DBMS. This is achieved by generating SQL queries which are only recomputed when the change applied to the data may violate their associated constraint and such that, whenever the computation is performed, only the relevant values given by the update are taken into account. In addition, our generated queries do not nest subqueries nor procedures.

Our method starts by applying the automatic translation of OCL Constraints into Event Dependency Constraints (EDCs), as defined in [10]. An EDC is a logic formula which states when some structural events (i.e. insertions or deletions of data) may cause the violation of a constraint. In terms of logics, an EDC is a conjunctive query with negated base atoms and built-in literals (i.e. arithmetic comparisons). So, an EDC has the form: $l_1 \wedge ... \wedge l_n \wedge bil_1 \wedge ... \wedge bil_m \rightarrow \bot$, where each l_i is a literal representing an instance, an instance insertion or an instance deletion; and bil_j is a built-in literal. For example, the EDC: $user(X) \wedge \iota userAge(X, Age) \wedge Age < 18 \rightarrow \bot$ states that a constraint is violated if there is a user X in the data and we insert some Age below 18 to this user X.

From this point, we define an inductive translation from EDCs to SQL. Broadly speaking, l_1 gives the initial table to start the *FROM* clause, then, each l_i is joined with the other tables by a cross join (i.e. a Cartesian product), inner join or anti join depending on the binding of the variables of l_i and its positive/negative sign. Any bil_j is directly placed in the *WHERE* clause.

Literals l_i representing an instance insertion or deletion, i.e. a change taking place over the data, provide the key for incrementality. Each such l_i is translated as an SQL join from a table containing such insertions/deletions to the rest of tables translated from the other EDC literals. When a user wants to apply an update, this update is first inserted in the auxiliary tables and, once the method has ensured that it does not cause any integrity constraint violation, the update is applied to the real tables and the auxiliary tables are emptied. Such process may be automatized by means of a DB controller.

Since each SQL query contains at least one of such literals, such query computations will always be tied to the data changes. For instance, in the previous EDC, $\iota userAge(X, Age)$ joins the query with the insertion of an age to a user. Thus, if we do not insert any age for any user, the join will return the empty set and no more evaluation will be needed. On the contrary, if there is some insertion of an age to a user, the join will retrieve only the affected user(s) from which to continue the query computation. Note that such evaluation is better than retrieving all the users of the database and then perform the convenient checks for each one for any kind of data update.

As a result, we get some SQL queries that are empty if and only if the OCL constraints are satisfied. Such queries perform incrementally and, in addition, avoid the use of nested queries/procedures. As a trade-off, the expressiveness of OCL is limited to the fragment of OCL translatable to EDCs.

To show the benefits of our method, we have performed some experiments to compare the efficiency of our translation with the translations given by OCL2SQL and MySQL4OCL. In such empirical study, we show that whereas our approach is capable to check the integrity of a set of constraints in at most 3 seconds per constraint regarding scenarios with $5{\cdot}10^6$ instances and $5{\cdot}10^4$ updates, OCL2SQL and MySQL4OCL could not check some invariants after one hour of execution.

2 Related Work

To our knowledge, there are two tools implementing an OCL to SQL translation: OCL2SQL[8] and MySQL4OCL[9]. We review both of them separately. In addition, we review the research on incremental OCL integrity checking, some work based on translating OCL to graph patterns, and some other based on translating FOL based constraints to incremental SQL queries.

OCL2SQL. OCL2SQL is a component of the OCL Dresden Toolkit for translating OCL constraints to SQL views [8]. The idea is that such SQL views are empty if and only if all the constraints are satisfied. The bases for such translation were early established in [7], however, since such bases were not defined inductively, it is hard to realize which OCL subset does it deal with. In any case, the main drawback that we find in it is the non-incrementality of their checks.

MySQL4OCL. MySQL4OCL is a tool for translating OCL expressions to MySQL queries [9]. The translation is tied to MySQL because it uses some procedures for translating iterator expressions of OCL (e.g. *forAll*, *exists*, *select*, etc). In this way, they deal with a broad subset of OCL. Moreover, they can also deal with the three valuated logic of OCL (i.e. true, false, null). However, and similarly to OCL2SQL, the translation of the constraints is not incremental and thus, we do not know which queries should be recomputed and which is the relevant data to take into account to check the data integrity after some update.

Incremental OCL Integrity Checking. There are already some proposals on how to incrementally check OCL constraints [11,12,13]. Applying such methods it is possible to realize which constraints should be checked and which is the relevant data to analyse after some data updates. Nevertheless, as far as we know, none of them have been adapted to work with databases on its behind.

OCL translation to graph patterns. The work of [5] consists in translating OCL constraints into graph patterns to benefit from its incremental capabilities. Such proposal relies on the intensive usage of memory since it is not intended to work with databases, but from data kept in memory. To overcome such difficulty, in [14] there is a proposal implementing the incremental graph patterns approach using some distributed databases. However, it seems that such databases are just implementing the persistence and the data is still kept (and queried) in memory.

FOL to incremental SQL queries. Decker proposed a method to translate constraints specified in a logic notation (different from OCL) into incremental SQL queries [15]. In his proposal, constraints are translated into queries that are invoked by triggers when a change over the data may cause a violation of a constraint. This is the way efficiency is achieved. However, and despite the difference on the language used to specify the constraints, we are not aware of any implementation of this approach that allow us to compare its efficiency with ours in the experiments we have performed.

3 Translating OCL Constraints to SQL Queries

In the following, we first define a conceptual schema with some OCL constraints that will be used as a running example to illustrate our method. Then, we show the representation of the OCL constraints in the form of EDCs. Such EDCs adds the incrementality capabilities to constraint checking. Finally, we give an inductive translation from such EDCs to SQL queries to perform the incremental checks of the constraints by means of relational DBMS technology.

3.1 An Illustrative Example

Consider the example in Fig. 1 of a message service application. In this class diagram, a *user* sends *messages* to some *conversation groups*. There are two kinds of conversation groups: *pairs*, that is, two simple users sending messages to each other; and *groups*, that is, two or more users formally grouped since some creation date. As expected, a group has some users as *members* and also one user as *owner*.

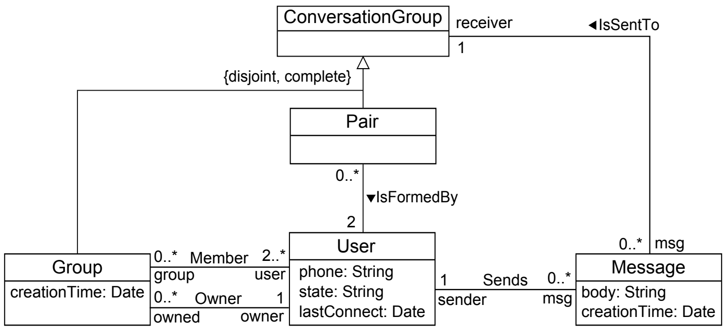

Fig. 1. Class diagram of a instant message service

OCL Constraints in Fig. 2 states some constraints that should always be satisfied by the data of such schema. Concretely, *MessagesInAPairBelongToPair* and *MessagesInAGroupBelongToGroup* state that the messages received by any pair/group are sent by the members of such pair/group. *UserIsMemberOfOwnedGroups* states that the owner of a group is also a member of such group and finally, *MessagesOfGroupAreSentAfterItsCreation* states that any messages sent to a group has to be created after the creation of the group.

Note that we have deliberately omitted the identifier constraints of the classes (e.g. *User*.`allInstances()->isUnique`(*phone*)) because they can be easily well treated by SQL primary keys on tables.

```
context Pair inv MessagesInAPairBelongToPair:
self.user->includesAll(self.msg.sender)

context Group inv MessagesInAGroupBelongToGroup:
self.user->includesAll(self.msg.sender)

context User inv UserIsMemberOfOwnedGroups:
self.group->includesAll(self.owned)

context Group inv MessagesOfGroupAreSentAfterItsCreation:
self.msg->forAll(e|e.creationTime > self.creationTime)
```

Fig. 2. OCL Constraints for the previous class diagram

Given a translation of the UML class diagram into SQL tables, our goal is to translate each OCL constraint into an SQL query in such a way that the constraint is not violated by an update (i.e. a change over the data) if and only if its corresponding SQL query is empty. For this purpose, we will first represent the violations of OCL constraints by means of EDCs and, then, obtain the SQL queries from this intermediate representation.

3.2 The EDC Representation of an OCL Constraint

The starting point of our work is the Event-Dependency Constraints (EDC) representation of the OCL constraint violations. The translation from OCL to EDCs can be fully automatized using the method described in [10]. For the sake of self-containment we review the formal notion of EDC below.

An EDC is a logic formula that states when some structural events (i.e. insertions or deletions of data) may cause the violation of a constraint. In terms of logics, an EDC is a conjunctive query with negated base atoms and built-in literals (i.e. arithmetic comparisons). That is, it has the form $l_1 \wedge ... \wedge l_n \wedge bil_1 \wedge ... \wedge bil_m \rightarrow \bot$ where each l_i is a literal representing either an instance, an instance insertion or an instance deletion; and bil_j is a built-in literal. Such built-in-literals are optional. In addition, EDCs are *safe clauses*, that is, any variable appearing in a negated or built-in literal, also appears in a positive literal.

For instance, the EDCs representation of the *UserIsMemberOfOwnedGroups* OCL constraint is:

$$\neg owner(G,U), \iota owner(G,U), \neg member(G,U), \neg\iota member(G,U) \rightarrow \bot \tag{1}$$

$$\neg owner(G,U), \iota owner(G,U), member(G,U), \delta member(G,U) \rightarrow \bot \tag{2}$$

$$owner(G,U), \neg\delta owner(G,U), member(G,U), \delta member(G,U) \rightarrow \bot \tag{3}$$

Note that an OCL constraint is written into more than one EDC since each EDC defines a different combination of events that may lead to the violation of the constraint. Concretely, Rule 1 above states that *UserIsMemberOfOwnedGroups* will be violated if we insert a user U as the owner of group G, where U is not a member of G and without inserting U as a member of G. Rule 2 identifies a violation of the same constraint when we insert U as the owner of G while deleting U as a member of G. Finally, Rule 3 indicates a violation when we delete U as a member of G without deleting U as the owner of G.

The length of an EDC is directly proportional to the length of the OCL constraint, but the number of EDC rules we get for an OCL constraint is exponential to the number of navigation steps of it since there is an exponential number of different ways to violate a constraint and each EDC captures one. To avoid such situation, we could take out the common factor of EDCs (e.g. rules 2 and 3 could be summarized in one rule using disjunctions). In this way, we would achieve a behavior similar to the TREAT algorithm [16] where joins are performed using the union between a table and its new insertions. However, in TREAT, each constraint is evaluated as many times as tables containing new instances are accessed in the definition of the constraint. The efficiency comparison of our proposal and TREAT is left out for further work since, as far as we know, there is no tool implementing TREAT in SQL for OCL constraints.

It is worth saying that the current translation from OCL to EDCs only deals with a fragment of OCL. However, such fragment is expressive enough to deal with a superset of the constraints that can be specified with the constraint patterns defined in [17], which have been shown to be useful for defining around the 60% of the integrity constraints found in real schemas. The grammar of the OCL constraints translatable into EDCs can be found in [10].

3.3 From the UML Class Diagram to SQL Tables

Before translating the EDCs into SQL queries, we need to translate the UML class diagram into SQL tables. In our example, we will suppose that each UML class/association has been translated into a different SQL table. Thus, the signatures of the tables obtained from the class diagram of Fig. 1 are the following:

Pair(id)	*ConversationGroup(id)*	*Group(id,creationTime)*
Owner(user, group)	*Member(user, group)*	*User(id,phone,state,lastConnect)*
Sends(user, message)	*Message(id, body, time)*	*IsSentTo(conversgroup,message)*
IsFormedBy(pair,user)		

3.4 Translating EDCs to SQL Queries

We define now an inductive translation from EDCs to SQL queries based on the EDC length. The translation is composed by two functions: one for computing the *from* clause and another for computing the *where* clause. Regarding the *select* clause, we select the column that represents the id of the instances violating the constraint (i.e. the *self* OCL instances for which the invariant evaluates to false).

Therefore, the translation of an EDC to an SQL query is given by the pattern:

$$\texttt{SELECT}\ \mathit{sqlColumn}(\mathit{edcVariable}(\text{"}self\text{"}))$$
$$\texttt{FROM}\ \mathit{fromTransl}(EDC)$$
$$\texttt{WHERE}\ \mathit{whereTransl}(EDC)$$

Where *sqlColumn* returns the SQL column name corresponding to the given EDC variable. The EDC variable in which we are interested is the one corresponding to the OCL *self* variable, so, we use the function *edcVariable* with the parameter "self" to obtain it. In case that there is no "self" variable in the OCL constraint (e.g. the constraint may be like *Class.allInstances()->forAll(e|...)*), other options should be considered like selecting the column/s corresponding to the OCL iterator variable/s (e.g. the previous e variable).

Since EDCs use some literals to represent the insertion/deletion of instances, we assume that our database schema contains also some public auxiliary tables in which we temporally store the instances that are being inserted/deleted. Thus, such literals are mapped to those auxiliary tables.

For instance, EDC 2 uses the literal $\iota owner$. Such literal is mapped to a new auxiliary SQL table *ins_owner* where we temporary write the insertions of instances of *owner* we are applying to the data.

Finally, recall that an EDC has the form $l_1 \wedge ... \wedge l_n \wedge bil_1 \wedge ... \wedge bil_m \rightarrow \bot$. Without loss of generality, we assume that all negated literals are placed in the end of the formula (i.e. from some l_i to l_n); and that all terms of a literal l_j are variables with different names. Such condition can be ensured by replacing some terms for new fresh variables and binding such variables to its actual terms with new built-in literals (e.g. *P(X,1)* would be translated to *P(X,Y)* $\wedge$ *Y = 1*).

Translation Base Case In the base case, the EDC is composed by just one literal l_1. Such literal is necessarily positive to ensure the safeness of the clause. Then, the translation is as follows:

$$\begin{aligned} &\mathit{fromTransl}(l_1) = \mathit{sqlTable}(l_1) \\ &\mathit{whereTransl}(l_1) = \emptyset \end{aligned}$$

Translation Inductive Case In the inductive case, the EDC has the form $L \wedge l_i$ or $L \wedge bil_i$, where L is a non-empty list of literals, l_i is a positive or negated literal, and bil_i a built-in literal.

In the first case, if l_i is positive, the translation consists in an inner join between the translations of l_i and L joining the columns corresponding to their bound variables. If no variable is bound, then, a cross join (i.e., a Cartesian product) is performed instead. Thus, the translation is as follows:

$$\begin{aligned} &\mathit{fromTransl}(L \wedge l_i) = \mathit{fromTransl}(L) \ \texttt{JOIN} \ \mathit{sqlTable}(l_i) \ \texttt{ON} \ (\mathit{binding}(L, l_i)) \\ &whereTransl(L \wedge l_i) = whereTransl(L) \end{aligned}$$

Where $\mathit{binding}(L, l_i)$ is a function that returns the column joins according to the variables of l_i that are bound to L.

If l_i is negative, the translation consists in performing an anti join to get those tuples of L that do not join l_i. For performing the anti join, we use a left join and check the joined columns to be *null*. Thus, the translation results in:

$$\begin{aligned} &\mathit{fromTransl}(L \wedge l_i) = \mathit{fromTransl}(L) \ \texttt{LEFT JOIN} \ \mathit{sqlTable}(l_i) \ \texttt{ON} \ (\mathit{binding}(L, l_i)) \\ &whereTransl(L \wedge l_i) = whereTransl(L) \ \texttt{AND} \ columnName(l_i, 1) \ \texttt{IS NULL} \end{aligned}$$

Where *columnName*(l_i, 1) returns the name of the first SQL column of the SQL table corresponding to l_i. Such column is used to check whether the joined columns resulted into *null*.

Also, notice that $binding(L, l_i)$ will not be empty because all the variables of l_i are necessarily bound to L since any EDC satisfies the *safeness* property.

Finally, the translation when we deal with a built-in literal bil_i is as follows:

$$fromTransl(L \wedge bil_i) = fromTransl(L)$$
$$whereTransl(L \wedge bil_i) = whereTransl(L) \texttt{ AND } sqlComparison(L, bil_i)$$

Where *sqlComparison* is a function that translates the bil_i into the SQL syntax. I.e. it changes the variables of bil_i for the corresponding SQL column names of the L variables they are bound to. Note that all variables of bil_i are bound to L because the *safeness* property.

Translation example To illustrate our translation, we show how the EDC in rule 2 is specified as an SQL query. First of all, we sort the EDC literals to move the negated ones to the end of the formula, thus obtaining the following EDC':

$$\iota owner(G, U), member(G, U), \delta member(G, U) \neg owner(G, U) \rightarrow \bot$$

Next, by applying the translation we have just defined, we get:

```
SELECT T0.user
FROM ins_owner AS T0
   JOIN Member AS T1 ON (T1.group = T0.group AND T1.user = T0.user)
   JOIN del_Member AS T2 ON (T2.group = T0.group AND T2.user = T0.user)
   LEFT JOIN Group AS T3 ON (T3.group = T0.group AND T3.group = T0.group)
WHERE T3.group IS NULL
```

Lastly, and since violations of an OCL constraint are specified by means of several EDCs, the final SQL query we obtain to check the OCL constraint is given by the SQL union of all the queries we have obtained from each EDC.

4 Experiments

We have conducted an experiment to illustrate the scalability improvement provided by our SQL queries as compared to OCL2SQL[8] and MySQL4OCL[9]. In this experiment, we show that we can scale up to scenarios with $5*10^6$ instances with $5{\cdot}10^4$ updates to check a set of OCL constraints whereas OCL2SQL and MySQL4OCL can not deal with some of them after 1 hour of execution. Since the time required to check a constraint is independent from the others, we stayed at a reduced number of constraints without altering the relevance of our results.

Given the schema of a message service application in our example, the conducted experiment consisted into adding some new message instances to some conversation groups in several randomly generated database states of increasing size. Then, we checked all the constraints of the example by means of the queries generated by MySQL4OCL, OCL2SQL and our incremental approach.

Table 1. Time results comparison with OCL2SQL and MySQL4OCL

	5*10^3	5*10^4	5*10^5	5*10^6
1st Const. - MySQL4OCL	0.71 s	7.45 s	58.0 s	$>$ 1 h
1st Const. - OCL2SQL	0.17 s	0.40 s	3.37 s	$>$ 1 h
1st Const. - inc.	0.09 s	0.13 s	0.46 s	2.63 s
2nd Const. - MySQL4OCL	0.70 s	7.04 s	58.9 s	$>$ 1 h
2nd Const. - OCL2SQL	0.15 s	0.54 s	3.71 s	$>$ 1 h
2nd Const. - inc.	0.10 s	0.12 s	0.34 s	2.38 s
3rd Const. - MySQL4OCL	0.60 s	2.06 s	17.0 s	223 s
3rd Const. - OCL2SQL	0.11 s	0.17 s	0.51 s	42.15 s
3rd Const. - inc.	0.09 s	0.09 s	0.10 s	0.10 s
4th Const. - MySQL4OCL	1.94 s	15.0 s	126 s	$>$ 1 h
4th Const. - OCL2SQL	0.11 s	0.25 s	1.72 s	$>$ 1 h
4th Const. - inc.	0.09 s	0.09 s	0.24 s	1.63 s

The randomly generated data consisted in N users who randomly had already sent $N * 10$ messages distributed in $N/10$ groups and $N/10$ pairs. The update consisted in $N/10$ new messages. The experiments were carried out on MySQL 5.6 running on Windows 7 in a Intel T4500 2.30GHz machine with 4GB of RAM. The database had the MySQL default indexes for primary/foreign keys.

We show our results in table 1. The title of the columns indicates the size of the database giving the number of current preexisting messages. As it can be seen, we are able able to check the integrity of the constraints in at most 3 seconds per constraint with a database state with $5{\cdot}10^6$ messages and $5{\cdot}10^4$ new insertions. In contrast, OCL2SQL/MySQL4OCL cannot afford some of the constraints after 1 hour of execution.

The result of the 3rd constraint requires to take special attention. Since adding new messages cannot cause the violation of the 3rd OCL constraint (*UserIsMemberOfOwnedGroups*), our incremental approach performs in almost constant time because the query begins from an empty auxiliary table. For the other approaches, we argue that its better performance might be because it contains one level less of subqueries in comparison to the other constraints.

5 Conclusions

We have proposed a method for efficiently checking OCL constraints by means of SQL queries that perform incrementally since only the relevant constraints and the relevant values are taken into account during the computation. In addition, they are written in such a way not to nest any other query nor procedure inside.

To achieve it, we first specify the OCL constraint violations through an EDC formalism. Such formalism offers the advantage of stating which events may cause the violation of an integrity constraint. Then, each EDC is translated into an SQL query. When doing such translation, we create some new SQL tables for temporary storing the new instances that are going to be inserted/deleted. Such auxiliary tables are used by our SQL queries to perform incrementally.

We made some experiments showing that our approach is able to check in seconds four OCL constraints over an scenario containing $5{\cdot}10^6$ instances with $5{\cdot}10^4$ updates while other approaches like OCL2SQL and MySQL4OCL could not afford some of these constraints after 1 hour of execution. As future work, we would like to extend the fragment of OCL we are dealing with and to implement in SQL the rules proposed in [16] to compare the efficiency achieved in this case.

Acknowledgements This work has been partly supported by the Ministerio de Ciencia e Innovación under project TIN2011-24747 and by the FI grant from the Secreteria d'Universitats i Recerca of the Generalitat de Catalunya. We would like also to thank the anonymous reviewers for their comments.

References

1. Olivé, A.: Conceptual Modeling of Information Systems. Springer, Berlin (2007)
2. Object Management Group (OMG): Unified Modeling Language (UML) Superstructure Specification, version 2.4.1. (2011) `http://www.omg.org/spec/UML/`.
3. Object Management Group (OMG): Object Constraint Language (UML), version 2.4. (2014) `http://www.omg.org/spec/OCL/`.
4. Hamann, L., Hofrichter, O., Gogolla, M.: On integrating structure and behavior modeling with OCL. In: International Conference on Model Driven Engineering Languages & Systems (MODELS 2012), Springer (2012) 235–251
5. Bergmann, G.: Translating OCL to graph patterns. In: Model-Driven Engineering Languages and Systems. Volume 8767 of LNCS. Springer (2014) 670–686
6. Clavel, M., Marina, E., Miguel Angel, G.d.D.: Building an efficient component for OCL evaluation. Electronic Communications of the EASST **15** (2008)
7. Demuth, B., Hussmann, H.: Using UML/OCL constraints for relational database design. In: UML99 - The Unified Modeling Language. Springer (1999) 598–613
8. Demuth, B., Wilke, C.: Model and object verification by using Dresden OCL. In: Proceedings of the Russian-German Workshop "Innovation Information Technologies: theory and practice", Russia. (2009) 81
9. Egea, M., Dania, C., Clavel, M.: MySQL4OCL: A stored procedure-based MySQL code generator for OCL. Electronic Communications of the EASST **36** (2010)
10. Oriol, X., Tort, A., Teniente, E.: Fixing up non-executable operations in UML/OCL conceptual schemas. In: Conceptual Modeling–ER 2014. (2014) 232–245
11. Uhl, A., Goldschmidt, T., Holzleitner, M.: Using an OCL impact analysis algorithm for view-based textual modelling. ECEASST **44** (2011)
12. Groher, I., Reder, A., Egyed, A.: Incremental consistency checking of dynamic constraints. In: Fundamental Approaches to Software Engineering. Springer (2010) 203–217
13. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. Journal of Systems and Software **82**(9) (2009) 1459–1478
14. Szárnyas, G., Izsó, B., Ráth, I., Harmath, D., Bergmann, G., Varró, D.: IncQuery-D: A distributed incremental model query framework in the cloud. In: Model-Driven Engineering Languages and Systems. Volume 8767 of LNCS. Springer (2014) 653–669
15. Decker, H.: Translating advanced integrity checking technology to SQL. In: Database Integrity. (2002) 203–249
16. Miranker, D.P.: TREAT: A better match algorithm for AI production systems. In: Proc. of the 6th National Conf. on Artificial Intelligence. Volume 1 of AAAI., AAAI Press (1987) 42–47
17. Costal, D., Gómez, C., Queralt, A., Raventós, R., Teniente, E.: Improving the definition of general constraints in UML. Software & Systems Modeling **7**(4) (2008) 469–486