Constraint Support in MDA tools: a Survey

Jordi Cabot[1] and Ernest Teniente[2]

[1]Estudis d'Informàtica i Multimèdia, Universitat Oberta de Catalunya
jcabot@uoc.edu
[2] Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya
teniente@lsi.upc.edu

Abstract: The growing interest in the MDA (Model-Driven Architecture) and MDD (Model-Driven Development) approaches has largely increased the number of tools and methods including code-generation capabilities. Given a platform-independent model (PIM) of an application, these tools generate (part of) the application code either by defining first a platform-specific model or by executing a direct PIM to code transformation. However, current tools present several limitations regarding code generation of the integrity constraints defined in the PIMs. This paper compares these tools and shows that they lack expressiveness in the kind of constraints they can handle or efficiency in the code generated to verify them. Based on this evaluation, the features of an ideal code-generation method for integrity constraints are established. We believe such a method is required to extend MDA adoption in the development of industrial projects, where constraints play a key role.

1. Introduction

The goal of automating information systems building was already stated in the late sixties [32]. However, thanks to the definition and standardization of the MDA [28] this goal has revived and seems now more feasible than ever. As a matter of fact, we have recently witnessed an explosion of tools and methods promising a full and automatic generation of the application code from its specification. Even more, nowadays, code-generation capabilities of current CASE tools and their adhesion to the MDA vision is a key issue in their development and marketing strategy.

Nowadays, almost all methods and tools are able to generate the skeleton of Java classes or relational schemas from a platform-independent model (PIM). A few also generate the code of the application operations when its behavior is specified with state diagrams or action semantics [27]. Nevertheless, most methods and tools tend to skip the integrity constraints (ICs) specified in the PIM when generating the application code. We believe this is a major drawback since ICs are a fundamental part in the specification of an application [18], and thus, they must be taken into account when generating its implementation. In fact, the problem of generating an efficient integrity checking code from the ICs defined in a PIM has been classified as one of the open problems to solve before MDA, and in general MDD approaches, can be widely used in the industrial development of information systems [25].

This paper surveys the capabilities of current tools regarding the explicit definition of ICs in a PIM and the code generation to enforce them. As we will see, all of them present important limitations regarding the expressivity of the ICs they can handle and/or the efficiency of the generated code. From such analysis, we will be able to draw the desirable features that should be satisfied by any tool generating code to enforce the ICs.

In our study, we have considered MDA tools in a broad sense since we have chosen the most representative examples from the different kinds of tools (from CASE tools extended with code-generation capabilities to full MDD methods). Moreover, we have included in the study all tools supporting a textual language to define ICs, commonly OCL (Object Constraint Language [26]) or similar. Support for such a language is required in order to specify all kinds of ICs in the PIM.

As far as we know, ours is the first paper addressing this topic. Several tool lists and comparisons exist, like the one we find in [31] which is the closest to our work. We extend the work reported there by evaluating a larger number of tools and by analyzing the support they provide regarding expressivity of the ICs and efficiency of the code generated to enforce them (while [31] only points out whether the tools support OCL or not).

The paper is organized as follows. Next section defines the different evaluation criteria. Section 3 presents the evaluation of the selected tools and methods. Given their limitations, section 4 defines the features that all code-generation method for ICs should have. Finally, section 5 presents some conclusions.

2. Evaluation Criteria

This section presents the criteria used to select and/or to evaluate the tools. We have considered expressivity of the constraint definition language they allow, efficiency of the generated code and target technologies they address since they are the most relevant ones regarding the automatic treatment of ICs defined at the PIM level.

a) *Expressivity of the constraint definition language*

Although some ICs can be expressed by means of the graphical constructs provided by the modeling language (as the cardinality constraints), most ICs require the use of a general-purpose constraint definition language, commonly OCL in our case.

The expressivity of tools supporting such a language differs depending on the complexity of the operators permitted in the constraint definitions. We distinguish three basic complexity levels (adapted from [33]):

- Intra-object ICs: constraints restricting the value of the attributes of a single object.
- Inter-object ICs: constraints restricting the relationships between an object and other objects, instances of different classes. Within this category, it is worth to distinguish the subcategory of ICs containing aggregator operators (like *sum*, *count*, *size*…).
- Class-level ICs: constraints restricting a set of objects of the same class (in OCL, these ICs require the *allInstances* operator).

b) *Efficiency of the code generated to enforce the ICs*
An IC states a condition that each state of the Information Base (IB), i.e. the set of objects and links of the class diagram at a certain time point, must satisfy. Hence, after each change of the contents of the IB the generated code must check efficiently that the new state of the IB satisfies also the ICs. We define two different levels of efficiency:

1. An IC must only be enforced after changes that may induce its violation [5]. For instance, if one of the ICs states that the value of an attribute *at* of a class *cl* must be lower than *X*, we do not need to verify the IC after changes over other attributes of *cl* or when deleting *cl* instances.
2. The enforcement of an IC must be done incrementally by considering the lowest possible number of objects [6]. In the previous example, once a new instance of *cl* is created we should only evaluate the constraint over that new instance instead of taking all instances of *cl* into account.

c) Target technologies of the code generation process
The IB must be implemented in a particular technology. Typically the IB is implemented by means of a (relational) database or by means of a set of classes in some object-oriented language.

When using a database, the ICs are checked over the tuples of the tables created to represent the classes of the class diagram. When using a set of classes (for instance Java classes) representing the class diagram, the ICs are verified over the set of objects instance of these classes. Usually, after the objects have been verified they are also permanently stored in a database or a file system.

Therefore, to study the constraint code-generation capabilities we focus on these two technologies: 1 – Relational databases and 2 – Object-oriented languages, in particular Java. Even though some tools also deal with other technologies (like .NET or C++), this decision does not restrict the set of tools to study since these two are the most widely covered.

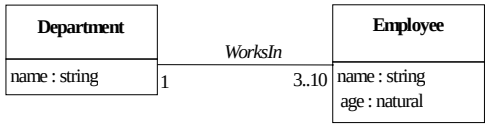
3. Tool Evaluation

To facilitate the evaluation, we have classified the different tools in the following categories: CASE tools, MDA specific tools, MDD methods and OCL tools. For each group we have selected the tools we believe are the most representative or the ones that offer a better IC support. Some of the tools could be classified in more than category since most of them are usually considered as MDA-tools although in our classification we reserve this category to the tools closest to the MDA standard.

Following the criteria stated in the previous section, for each tool we have evaluated its constraint generation capabilities for Java and for relational databases. For each technology we have studied the allowed expressivity and the efficiency of the generated implementation. See the appendix for a summary table.

As a running example we will use the simple PIM of Figure 1. Apart from the cardinality constraints (each employee works in a department and each department has from three to ten employees) the PIM includes three textual ICs defined with OCL.

The ICs state that all employees are over 16 years old (*ValidAge*), that all departments contain at least three employees over 45 (*SeniorEmployees*) and that no two employees have the same name (*UniqueName*).



context Employee **inv** ValidAge: self.age>16

context Department **inv** SeniorEmployees:
self.employee->select(e| e.age>45)->size()>=3

context Employee **inv** UniqueName:
Employee.allInstances()->isUnique(name)

Figure 1. PIM used as a running example

As we will see in the next subsections, even such a simple example cannot be fully generated using the current tools since none of them is able to provide an efficient implementation of the schema and its ICs.

3.1 CASE Tools

Even though the initial aim of CASE tools was to facilitate the modeling of software systems, almost all of them have extended their functionality to offer, to some extent, code-generation capabilities. From all CASE tools (see [23] for an exhaustive list) we have selected the following ones: *Poseidon*, *Rational Rose*, *MagicDraw*, *Objecteering/UML* and *Together*. In what follows we comment them in detail:

a) *Poseidon* [15] is a commercial extension of *ArgoUML*. The Java generation capabilities are quite simple. It does not allow the definition of OCL ICs and it does not take the cardinality constraints into account either. It only distinguishes two different multiplicity values: '1' and 'greater than one'. In fact, when the multiplicity is greater than one the values of the multivalued attributed created in the corresponding Java class are not restricted to be of the correct type (see the *employee* attribute of the *Department* class in Figure 2, the attribute may hold any kind of object and not only employee instances).

The generation of the relational schema is not much powerful either. The only constraints appearing in the relational schema are the *primary keys*. The designer must explicitly indicate which attributes act as primary keys by means of modifying the corresponding property in the attribute definition.

```
public class Department {
  private string name;
  public java.util.Collection employee = new java.util.TreeSet();
}
```

Figure 2. *Department* class as generated by Poseidon

b) *Rational Rose* [30]. The Java generation process is similar to that of *Poseidon*. The database generation is better because the class diagram can be complemented with the definition of additional properties. For instance, the *ValidAge* constraint can be specified as a property of the *age* attribute (Figure 3). Given this property, the tool adds to the *Employee* table the constraint *check(age>16)* to control the employees' age.

Recently, a Rational Rose plug-in [13] is available to permit the definition of OCL ICs on rose models. However, these ICs are not considered when generating the application code.

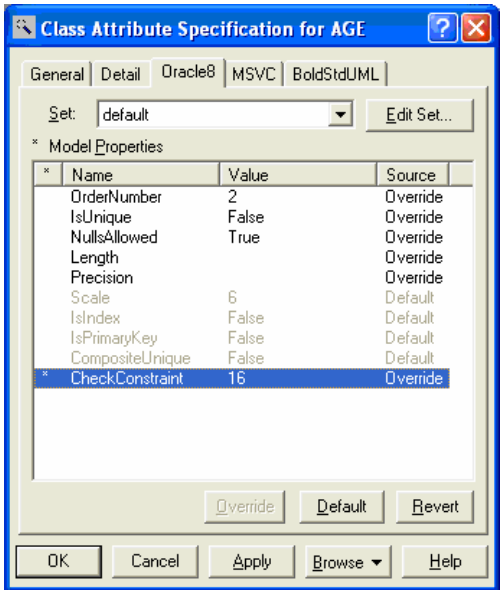


Figure 3. Properties of the *Age* attribute in Rational Rose

c) *MagicDraw* [22] offers a specific UML profile to define relational schemas which allows to improve the code generation for that kind of databases. In this way, the user may annotate the class diagram with all the necessary information (*primary* and *foreign keys*, *unique* constraints and checks over attributes).

Figure 4 shows the relational schema definition of the PIM in Figure 1 once annotated with the profile. This diagram could be considered as the PSM (Platform-Specific Model) of the initial PIM. The tool partially generates this PSM from the PIM. The schema includes the primary keys of each table, the foreign key from employee to department and the *ValidAge* IC. The other ICs cannot be specified since the database does not provide any predefined mechanism to verify them (and MagicDraw does not generate for itself any code excerpt to verify them either).

Though *MagicDraw* allows the definition of OCL ICs, they are completely omitted during the PSM or code generation. For instance, when transforming the initial PIM (Figure 1) to the PSM (Figure 4), *MagicDraw* is unable to transform *ValidAge* in the corresponding *check* in the PSM, we are force to manually redefine the constraint again in the PSM.

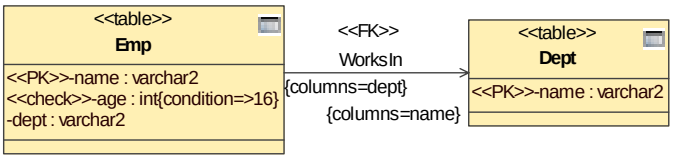


Figure 4. PSM for the relational schema in *MagicDraw*

d) *Objecteering/UML* [29] presents as a special feature with respect to the previous tools that supports (and generates) any multiplicity value in the associations. When generating the Java code it uses a predefined class library to enforce the cardinality constraints. Moreover, it creates a set of triggers during the generation of the relational schema. For instance, the trigger in Figure 5 checks that a department contains less than ten employees before allowing the assignment of a new employee. Otherwise, it raises an exception. The starting point for the database generation is, as in the previous tool, a PSM that can be obtained from the initial PIM. It does not allow the definition of ICs in OCL.

```
CREATE TRIGGER TI_dept_emp_INSERT
 ON employee
 FOR INSERT
 AS
 IF NOT((SELECT COUNT(*)
     FROM employee, inserted
     WHERE employee.department = inserted.department) <= 10)
 BEGIN
   ROLLBACK TRANSACTION
   RAISERROR 20501, "dept_emp : May not insert element,
         dept_emp_FK maximum cardinality constraint violation"
 END
```

Figure 5. Trigger to control the maximum number of employees per department

e) *Together* [4] offers similar capabilities to *Rational Rose* regarding the database generation. Moreover, it includes full OCL support to define constraints and pre/postconditions in the PIMs. However, when generating the Java code, only intra-object constraints are correctly generated. Moreover the generation is not efficient since constraints are verified after every single method and not only after methods possibly violating the constraints.

3.2 MDA Tools

We classify in this category tools having as main goal to support the definition and execution of model transformations from PIMs to PSMs and from the PSMs to the final code. We evaluate in this section some of the well-known MDA tools: *ArcStyler*, *OptimalJ* and *AndroMDA*.

ArcStyler [17] concentrates in the generation of Java, J2EE and .NET applications (with its *cartridge* architecture the designer can define additional transformations).

When generating Java programs, the constraint support is like the one in Poseidon, with the only difference that automatically creates a set of methods to modify the attributes representing the associations of the original class diagram. *ArcStyler* includes the *Dresden OCL* tool (see section 3.4) to define and generate ICs.

OptimalJ [9] is devoted to generate J2EE applications where all the business logic concentrates in the Java classes (Enterprise Java Beans in this case). It only supports constraints over the attribute values using constant values or regular expressions. For more complex ICs, the designer must write the corresponding Java code directly.

AndroMDA is an open source code generation framework that follows the MDA paradigm. According to the tool information, it takes model(s) from CASE-tool(s) and generates fully deployable applications. *AndroMDA* supports the definition of OCL query expressions and transforms them to the Hibernate-QL or EJB-QL query languages. However, no explicit support for OCL ICs is provided.

3.3 MDD Methods

In this section we grouped several MDD methods although some of them may not follow exactly the MDA approach nor use OMG standard languages.

OO-Method [14] is based on the formal language OASIS, though admits the definition of UML class diagrams with constraints defined in an OCL-like language. ICs may include aggregator operators but class-level constraints are not allowed. ICs are verified over the objects instance of the Java classes implementing the class diagram. Every time a method of a Java class is executed, all ICs defined on that class are verified (and not only the ICs that may be affected by that method execution). To verify the ICs, they add a special method in the Java classes (Figure 6). The method contains a set of conditions (one for each constraint defined on the class). When a condition is not satisfied, the method throws an exception.

```
Protected void checkIntegrityConstraints() throws Error
{
  if (! (age>16))
  throw new error (“Constraint Violation. Invalid age”);
}
```

Figure 6. Java method verifying the *ValidAge* constraint

WebML [8] is specialized in the generation of web applications. It presents little support for defining ICs. It only admits the definition of *validity predicates* on the web page forms. A validity predicate is a boolean expression that checks the correctness of the value entered by the user in a form included in a web page. The boolean expression may consist of boolean operators, arithmetic operators, comparisons (=,>,<,…) and constant values.

Executable UML [21] proposes to specify the behavior of an application in sufficient detail so that it can be directly executed. Specifications in executable UML consist merely of class diagrams, state diagrams and action semantics to describe the operation behavior. Using a model compiler, then, the specification is internally

transformed into Java or C++. It supports a predefined set of constraints like cardinality constraints, unique constraints or checks over the attribute' values. These ICs are afterwards expressed using the *Action Language* they provide. For more general ones, the designer must define them using this Action Language directly. That is, the designer is forced to define them in an imperative way and not declaratively (although action languages may contain query expressions they are basically an imperative language). Figure 7 shows the *UniqueName* expressed in the action language. Tools following this approach (like *BridgePoint* or *iUML*) are mainly used in the real time and embedded domains.

```
select many employees from instances of Employee
Where selected.name==self.name
Return (cardinality employees)==0)
```

Figure 7. *UniqueName* defined with an Action Language

3.4 OCL Tools

This section evaluates all tools generating code from OCL constraints. Tools supporting OCL with other purposes (as model validation [16] or verification [1]) are not considered.

Dresden OCL [11] generates the Java classes corresponding to the classes in a class diagram, including all ICs except for the class-level ICs, which are not supported. ICs are verified only after modifications over the attributes and associations (represented also as attributes in the Java classes) referenced in the constraint definition. This represents an efficiency improvement regarding previous methods, but, as shown in [5], it is still inefficient since not all kinds of changes over the attributes may violate the IC. For instance, the *SeniorEmployees* IC can be violated when removing an employee from a department but not when assigning a new one. This is exactly the same limitation of [34].

OCLtoSQL is another tool comprised in the previous toolkit, based on the method proposed in [10]. It generates the relational schema from the class diagram. Additionally, for each IC, it creates an SQL view. The view selects those tuples of the database not satisfying the constraint, and thus, a non-empty view indicates that the IC has been violated. As an example, Figure 8 shows the view corresponding to the *ValidAge* IC. Note that the view selects those employees *not* verifying the age condition. The views are not efficient since they examine the whole table population instead of considering only those tuples modified during the transaction (in the example, the view accesses all employees and not just the inserted or updated ones).

```
CREATE OR REPLACE VIEW ValidAge as
(select * from EMPLOYEE SELF where not (SELF.AGE > 16));
```

Figure 8. View for the *ValidAge* constraint

The code-generation capabilities of Octopus [20] are more restricted. For each IC, it creates a new method in the Java class corresponding to the context type of the IC. To know whether the IC holds we call this method. If it does not hold the method

throws an exception. However, the decision about *when* the IC needs to be verified (i.e. when we should call this method) is left to the designer. *OCLE* [2] and *KMF* {KentModellingFramework, #154} provide a similar functionality.

OCL2J [12] (and similarly *OCL4Java* [24]) generate a Java implementation of a PIM including all intra-object and inter-object ICs. The constraint verification is inefficient since ICs are verified before and after any method of the class executes.

Finally, *BoldSoft* [3] permits to execute an OCL expression over a set of objects stored in main memory or in the database (in this latter case, the expressivity is restricted, for instance, operators as count, collect, difference, asSet, asBag and so forth are not allowed). However, the tool is focused in the definition of derived elements and not in the verification of ICs.

4. Desirable Features of an IC Generation Method

From the previous evaluation it is clear that tools must incorporate and/or develop new methods to cope with IC enforcement. The aim of this section is to propose a set of features that should be considered when developing such methods.

Apart from the two basic features (expressivity and efficiency) we define also the *technology-aware generation*, *technological independence* and *checking time* characteristics. The description of each characteristic is the following:

1. *Expressivity*: The whole expressivity of the OCL language should be allowed.
2. *Efficiency*: The generated code should verify the ICs only when it is strictly necessary and using an efficient approach.
3. *Technology-aware generation*: To improve the efficiency of the generated code, the method should take into account the special characteristics of each target technology platform. For instance, when Java is the target technology *disjoint* ICs (stating that the intersection between objects of two given classes must be empty) can be discarded since they are enforced by the Java language itself (Java does not admit multiple classification, and thus, all classes are necessarily disjoint).
   The same idea applies when the target technology is a relational database. Relational databases offer some predefined constraint constructs as primary keys, checks over attribute values or unique constraints. OCL ICs should be mapped into one of this predefined constructs when possible, instead of creating our own checking code. It is reasonable to assume that the database management system will be always more efficient in managing them.
4. *Technological independence*: Following the MDA vision, at least the first stages of the code generation process (as the processing of the ICs to determine the kind of changes that can violate them) should be independent of the target technology platform. In this way we could reuse the same method to generate the checking code in several technologies. Just the last step (the code generation itself) should be technologically dependent and take into account the previous *technology-aware generation* characteristic.
5. *Checking time*: In general there are two different possibilities regarding the moment when ICs are verified. They can be verified immediately after each single

modification or their verification can be deferred until the end of the operation/s or the transaction. The method should be flexible enough to allow the designer define the preferred checking time for each IC.

5. Conclusions

In this paper we have surveyed the support of current tools regarding the automatic generation of the code required to enforce the ICs specified in a PIM. These tools have been evaluated regarding the expressivity of the IC definition language they provide, the efficiency of the generated code and the target technology they allow.

From our study, we may conclude that current tools have not yet seriously addressed such an issue since the support they provide is still rather limited. The main shortcomings encountered are the lack of expressivity of the ICs that may be defined; the need to use proprietary profiles and/or properties of each tool since in many tools ICs may not be expressed directly in OCL; and the lack of efficiency of the code generated to enforce the ICs.

We believe that the main reasons why the tools are making so little progress in this matter are the difficulty of performing IC checking with OCL (because of the high expressivity of this language) and the focus of the tools on the automatic code generation for more basic capabilities (such as translation of the class diagram, automatic interface generation, etc.). Additionally, we think that the increasing number of OCL auxiliary tools (parsers, compilers, APIs…) provides the tool vendors with a feasible opportunity to enhance tools' functionality with full OCL support.

So, there is still a huge amount of research to be pursued to achieve the goal of generating automatically the code required to enforce the OCL ICs defined in the PIM. Methods dealing with this problem are likely to be an extension of previous work on incremental integrity checking in relational and deductive databases. A first proposal has been recently presented in [7].

Acknowledgments

This work has been partly supported by the Ministerio de Educacion y Ciencia under project TIN 2005-06053.

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P. H.: The KeY tool, Integrating object oriented design and formal verification. Software and Systems Modeling 4 (2005) 32-54
2. Babes-Bolyai. Object Constraint Language Environment 2.0. http://lci.cs.ubbcluj.ro/ocle/
3. Borland. Bold for Delphi. http://info.borland.com/techpubs/delphi/boldfordelphi/
4. Borland. Together Architect 2006. http://www.borland.com/us/products/together/
5. Cabot, J., Teniente, E.: Determining the Structural Events that May Violate an Integrity Constraint. In: Proc. 7th Int. Conf. on the Unified Modeling Language, LNCS, 3273 (2004) 173-187

6. Cabot, J., Teniente, E.: Computing the Relevant Instances that May Violate an OCL constraint. In: Proc. 17th Int. Conf. on Advanced Information Systems Engineering, LNCS, 3520 (2005) 48-62
7. Cabot, J., Teniente, E.: Incremental Evaluation of OCL Constraints. In: Proc. 18th Int. Conf. on Advanced Information Systems Engineering, LNCS, 4001 (2006) 81-95
8. Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., Matera, M.: Designing Data-Intensive Web Applications. Morgan Kaufmann (2002)
9. Compuware. OptimalJ. http://www.compuware.com/products/optimalj/
10. Demuth, B., Hussmann, H., Loecher, S.: OCL as a Specification Language for Business Rules in Database Applications. In: Proc. 4th Int. Conf. on the Unified Modeling Language, LNCS, 2185 (2001) 104-117
11. Dresden. Dresden OCL Toolkit. http://dresden-ocl.sourceforge.net/index.html
12. Dzidek, W. J., Briand, L. C., Labiche, Y.: Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In: Proc. MODELS 2005 Workshops, LNCS, 3844 (2005) 10-19
13. EmPowerTec. OCL-AddIn for Rational Rose. http://www.empowertec.de/products/rational-rose-ocl.htm
14. Fons, J., Pelechano, V., Albert, M., Pastor, Ó. Development of Web Applications from Web Enhanced Conceptual Schemas. In: Proc. 22nd Int. Conf. on Conceptual Modeling, LNCS, 2813 (2003) 232-245
15. Gentleware. Poseidon for UML v. 4. http://www.gentleware.com
16. Gogolla, M., Bohling, J., Richters, M.: Validation of UML and OCL Models by Automatic Snapshot Generation. In: Proc. 6th Int. Conf. Unified Modeling Language. LNCS, 2863 (2003)
17. Interactive Objects. ArcStyler v.5. http://www.interactive-objects.com/
18. ISO/TC97/SC5/WG3: Concepts and Terminology for the Conceptual Schema and Information Base. ISO, (1982)
19. Kent Modelling Framework. Kent OCL Library. http://www.cs.kent.ac.uk/projects/kmf/
20. Klasse Objecten. Octopus: OCL Tool for Precise Uml Specifications. http://www.klasse.nl/octopus/index.html
21. Mellor, S. J., Balcer, M. J.: Executable UML. Object Technology Series. Addison-Wesley (2002)
22. No Magic Inc. MagicDraw UML v. 10.5. http://www.magicdraw.com/
23. Objects by Design. List of UML tools. Available: http://www.objectsbydesign.com/
24. OCL4Java. http://www.ocl4java.org
25. Olivé, A.: Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research. In: Proc. 17th Int. Conf. on Advanced Information Systems Engineering, LNCs, 3520 (2005) 1-15
26. OMG: UML 2.0 OCL Specification. Adopted Specification (ptc/03-10-14) (2003)
27. OMG: UML 2.0 Superstructure Specification. Adopted Specification (ptc/03-08-02) (2003)
28. OMG: MDA Guide Version 1.0.1. (2003)
29. Softeam. Objecteering/UML v. 5.3. http://www.objecteering.com/products.php
30. Software, R. Rational Rose. http://www-306.ibm.com/software/rational/
31. Tariq, N. A., Akhter, N.: Comparison of Model Driven Architecture (MDA) based tools. In: Proc. 13th Nordic Baltic Conference (NBC), IFMBE Proceedings, 9 (2005)
32. Teichroew, D.: Methodology for the Design of Information Processing Systems. In: Proc. 4th Australian Computer Conference, (1969) 629-634
33. Türker, C., Gertz, M.: Semantic integrity support in SQL:1999 and commercial (object-)relational database management systems. The VLDB Journal 10 (2001) 241-269
34. Verheecke, B., Straeten, R. V. D.: Specifying and implementing the operational use of constraints in object-oriented applications. In: Proc. Tools Pacific 2002, (2002) 23-32

Appendix A

The following table summarizes the comparison of the different tools. For each tool we indicate its expressivity and efficiency regarding the Java and relational database generation of the ICs. In the *expressivity* columns, the symbol *X* means that the tool does not support any kind of constraint definition while the symbol √ means a full constraint support and *n/a* indicates that the tool does not generate code for that technology. Otherwise, we explicitly indicate the type of ICs admitted, according to the classification of section 2. Likewise for *efficiency* columns. In the *DB efficiency* column, cells are defined as *DBMS* when the tool relies on the constraint constructs offered by the database-management system (*primary keys*, *checks*…) to verify the ICs.

Table A.1. Tool comparison

Tools	Java		DB	
	Expressivity	**Efficiency**	**Expressivity**	**Efficiency**
Poseidon	X	n/a	PK	DBMS
Rational Rose	X	n/a	PK, intra	DBMS
Magic Draw	X	n/a	PK, intra	DBMS
Objecteering	cardinality	√	PK,cardinality	√
Together	√	ICs are verified after every method	PK, intra	DBMS
ArcStyler	Uses DresdenOCL	n/a	PK, intra	DBMS
OptimalJ	intra	√	PK, intra	DBMS
AndroMDA	X	n/a	PK, intra	DBMS
OO-Method	intra, inter	ICs are verified after every method	PK	DBMS
WebML	intra	√	PK	DBMS
ExecutableUML	intra, predefined types	√	n/a	n/a
DresdenOCL	intra, inter	ICs are verified after methods modifying the constrained elements	n/a	n/a
OCLtoSQL	n/a	n/a	√	Views evaluate all table population
Octopus	intra, inter	n/a	n/a	n/a
OCLE	intra, inter	n/a	n/a	n/a
KMF	intra, inter	n/a	n/a	n/a
OCL2J	intra, inter	ICs are verified before and after every method	n/a	n/a
OCL4Java	intra, inter	ICs are verified before and after every method	n/a	n/a