JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 7, No. 6, July-August 2008

Efficient Integrity Checking for Essential MOF + OCL in Software Repositories

Miguel Garcia, Institute for Software Systems,
Hamburg University of Science and Technology (TUHH), Germany

The efficient detection of run-time violations of integrity constraints (or their avoidance in the first place) has not been satisfactorily addressed for the combination of object model and constraint definition language most widely accepted in industry, namely OMG's Essential MOF and Object Constraint Language (OCL). We identify the dimensions relevant to this problem, and classify existing proposals by their position in the solution space. After this comparative survey, we propose a solution for the efficient integrity checking of invariants expressed in OCL over the Essential MOF data model, and describe the software architecture of its implementation using object-relational mapping technology.

1 INTRODUCTION

Model-Driven Software Engineering (MDSE) encompasses traditional areas of both Language Design and Software Engineering (language definition and tooling, manipulation of programs and models, refinement of specifications into lower-level abstractions) following a unified conceptual and technical framework (metamodeling and declarative model transformations). By expressing a language definition as a metamodel, the information about abstract syntax and static semantics (including typing rules) becomes machine-processable, enabling language-aware manipulation along a toolchain in a reusable, declarative manner. Metamodels are expressed in Essential MOF (EMOF) [1] (covering structural aspects), and are extended with constraints expressed in OCL [2], to be evaluated over finite populations of instances. An OCL class invariant is a Boolean function over a database snapshot.

As MDSE techniques are applied to development processes of ever increasing complexity, additional demands are placed on the infrastructure supporting those processes. Software repositories [3] play a pivotal role in the management of software artifacts conforming to an EMOF data model, checking the integrity constraints given as OCL invariants. The task of runtime integrity checking has proven non-scalable if performed without regard for optimization techniques, yet many EMOF software repositories in use today do not adequately address this concern. Solving this industrially relevant problem requires identifying a calculus expressive enough to handle OCL yet tractable enough that optimizations of collection operations are feasible. Moreover, an empirical evaluation of the proposed approach should validate the findings before real-world deployment.

Cite this document as follows: Miguel Garcia: *Efficient Integrity Checking for Essential MOF + OCL in Software Repositories*, in Journal of Object Technology, vol. 7, no. 6, July-August 2008, pages 101–119,
http://www.jot.fm/issues/issues_2008_7/article3.pdf

Integrity checking is an instance of the model-checking problem, i.e. determining whether a concrete world satisfies predicates. In turn, query evaluation is an area thoroughly studied in the academic literature. We follow the engineering approach of coherently combining existing scientific knowledge to solve an industrial problem. Our work falls just short of building a concrete product based on the technology choices made (because that's a task for industry). Rather, we disclose the detailed reasoning behind our approach (which industry refrains from doing).

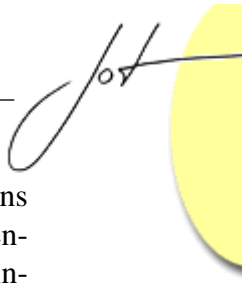
The structure of this article is as follows. Sec. 2 provides context on the artifacts subject to integrity checking in MDSE repositories, followed by a review in Sec. 3 of the strategies for integrity checking available to repository designers. Sec. 4 covers the often overlooked interplay between expressiveness of the constraint language and runtime cost of integrity checking. Sec. 5 presents a technology choice that balances these conflicting requirements. A review of the difficulties associated to checking computationally-complete OCL can be found in Sec. 6, followed by the translation rules into the chosen calculus (Sec. 7) and a sample of the optimization techniques thus enabled (Sec. 8). Related work (Sec. 9) includes pointers to the main-memory case and to recent progress on integrity checking in the SQL/relational setting. Sec. 10 concludes. Familiarity with metamodeling techniques and object-oriented databases is assumed. Knowledge about OCL is helpful but not required.

2 ROLE OF ESSENTIAL MOF AND OCL IN MODEL-DRIVEN SOFTWARE ENGINEERING

The MDSE approach of adopting and extending results from previously separate disciplines can be seen at work in the best practices for defining the syntax, static semantics, and behavior of domain-specific languages (DSLs). Following MDSE principles, the abstract syntax of a DSL is represented as an object-oriented model (expressed in EMOF) thus attaining a number of advantages compared to an EBNF approach. This object-oriented model additionally captures the static semantics of the DSL (e.g., declare-before-use) in the form of invariants expressed in the Object Constraint Language (OCL). As shown in [4], the type checking rules of a DSL are also amenable to an OCL formulation, an area previously treated separately in DSL design. Additional benefits naturally emerge once the language definition is available as a metamodel (and can thus be processed mechanically):

- Abstract Syntax Trees (ASTs) can be exchanged with ease in a toolchain (e.g., between a compiler front-end and an static analyzer), fostering interoperability.

- The declarativeness of the OCL formulation allows applying formal techniques to language processing, in particular Hoare-style program verification of model-transformation algorithms, so as to know at transformation design-time whether well-formed output will always be generated for well-formed input [5].



- Prototypes exist [6, 7, 8] where an AST definition is augmented with annotations to univocally determine a concrete syntax. From this augmented definition, a generator can derive: (a) grammars for different parser generators, making parsers interchangeable; (b) classes whose instances represent Concrete Syntax Tree (CST) nodes, thus allowing for OCL to be used to query and constrain a CST; (c) a visitor to transform a well-formed CST (as checked with OCL) into an AST; (d) an unparser from CST to textual notation (i.e., a pretty-printer); and (e) a text editor supporting usability features such as syntax-directed completion, markers for violations of well-formedness, use-defs navigation, folding, and structural views.

- Following a similar approach, a concrete *visual* syntax can be defined, allowing for the generation of a diagram editor for the DSL in question [9].

3 INTEGRITY CHECKING IN MDSE SOFTWARE REPOSITORIES

Given the ubiquity of EMOF and OCL in MDSE, it comes as no surprise that software repositories are required to manage artifacts conformant to EMOF + OCL metamodels [10]. Infrastructural functionality expected of such repositories includes scalability, concurrent access, integrity checking and enforcement, versioning [11], and view maintenance. These capabilities in turn are needed to support higher-level use cases such as: traceability between requirement specs and implementation artifacts, impact analysis, refactoring, and avoidance of architectural erosion [12].

The implementers of some EMOF + OCL software repositories in use today have not paid enough attention to the formal foundation of those languages, with the end result that it cannot be determined anymore whether some tool behaviors are correct or not. Analyses of ambiguities in past revisions of the MOF and OCL specification can be found in [13] and [14]. A formalism that offers rigorous precision is a good start, yet Fegaras and Maier define in [15] additional criteria for a calculus to be suitable for a query language:

- *Coverage*: whether the calculus has enough expressive power to represent all concepts of the query language. In the case of OCL, these concepts include aggregation, duplicate values, sort orders, several collection types (sets, bags, ordered sets, lists), negation, and user-defined (potentially recursive) functions.

- *Ease of manipulation*: expressions in the calculus should lend themselves to uniform matching and rewriting, such as in type-checking or optimization.

- *Evaluation fitness*: whether all valid query plans can be derived from an expression in the calculus. A formalism that expresses queries at too low a level of abstraction acts as a barrier to effective evaluation.

By relying on a formal calculus that is suitable with respect to OCL, precise definitions for the problems of query optimization, integrity checking, and view maintenance

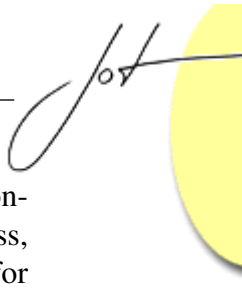
become possible, and correctness of their solutions can be examined. Efficient implementations are the next step. Before discussing a calculus that fulfills the above criteria, we elaborate on the alternative approach of directly anchoring the semantics of EMOF + OCL in terms of the Relational Data Model, turning OCL into a surface syntax. This would acknowledge the fact that results from the object-oriented and deductive database communities have become mainstream in SQL3 and are thus likely to be efficiently supported by conformant DBMSs. We see however some disadvantages with this approach:

- Pre-SQL3 relational formalisms do not fulfill the coverage criteria as defined above. Queries involving aggregation or sort orders need be formulated as a mixture of relational algebra interspersed with control structures. Only those fragments bracketed between control structures are amenable to optimization.
- Post-SQL3 extended-relational formalisms strongly resemble the calculus adopted in our approach. Algorithms for incremental view maintenance based on these formalisms can thus serve as a foundation for our solution architecture.
- It is more efficient to manipulate query plans at the highest level of abstraction possible. Once optimized, object-level queries can be cast in terms of relational algebra thus opening the way for further potential optimizations.
- EMOF concepts cannot be mapped one-to-one to relational "counterparts", thus making a direct relational anchoring non-trivial in itself. For example, a relational view may contain the primary keys of its base relations, while each object in an object view has a globally unique object-ID.

4 CONSTRAINT LANGUAGE EXPRESSIVENESS AND ITS IMPACT ON THE RUNTIME COST OF INTEGRITY CHECKING

There is a mutual dependency between the expressiveness of a constraint language and the computational complexity of evaluating integrity constraints upon updates to database state. Three categories can be distinguished:

1. *Design-time avoidance of integrity violations*: By carefully limiting the expressive power of the data model and constraint language, it is possible to determine at database schema design time whether some ordering of update transactions may violate the integrity constraints. After this proof has been carried out (e.g. based on algorithm model-checking as shown by Lamport in [16]) no run-time checks are needed. An example of this approach for a variant of the F-Logic language is presented in [17]. Actually there is still a run-time overhead in that each transaction is augmented with its generated weakest precondition. Those fragments of the precondition which cannot be proved to be implied by the database invariants have to be checked at runtime.



2. *Run-time integrity checking with efficient evaluation*: For more expressive constraint languages, not all integrity checks can be skipped at runtime. Nevertheless, the evaluation of those remaining checks can be made more efficient than that for arbitrary formulas in first-order predicate logic (PSPACE-complete in the worst-case for finite object graphs [18]). We aim at identifying the subset of OCL whose expressive power fits in this category. An algorithm for incremental view maintenance [19] optimizes integrity checking, as discussed in Sec. 5.

3. *Run-time integrity checking with best-effort evaluation*: For some specific combinations of database schemas and full-OCL invariants, custom checks are derived whose efficiency is comparable to that of category 2 above, sometimes using heuristics. For the remaining cases, large data sets have to be scanned. This approach is followed in [20] and [21] where the non-declarative subset of OCL is also adopted (including control structures and negation).

The chosen complexity of integrity checking (second item above) does not preclude ad-hoc queries from using full-OCL (and require full scans of entity extents in some cases). It seems questionable, however, for the formulation of an integrity constraint to require computational completeness, as the constraint is rendered non-declarative. Those constraints, if really needed, are best enforced by the business logic that manipulates the software repository, e.g. following Design-By-Contract [22], as recommended by best practices evolved over the years for the architecture of multi-tier information systems.

5 INCREMENTAL INTEGRITY CHECKING FOR OCL

Integrity enforcement comprises two runtime phases: (a) *violation detection* and (b) *consistency restoration*. For each OCL invariant, a view to hold the object-IDs of those instances not fulfilling it is defined (a *denial view*). At transaction commit time, all such views should be empty, otherwise a consistency restoration policy is to be applied (roll-back, compensating action, or postponing consistency restoration altogether). Policies for consistency restoration are outside the scope of this article. Given that most transactions leave the majority of invariants unaffected, full recomputation of views after each update is impractical. Instead, *incremental maintenance* is preferred, a process comprising design and runtime activities:

1. At database design time, each view definition is mechanically analyzed to determine which update operations (when performed on certain data elements) affect the resultset .

2. For such events, actions are generated to react to them, taking as input the delta caused by the update and using it to bring the materialized view into an up-to-date state (a *self-maintenance* strategy as opposed to querying the base extents).

3. At runtime, the planned actions are executed upon being triggered by the updates being monitored, performing *change propagation*.

An efficient algorithm for incremental view maintenance in an EMOF + OCL context is not as concise as the above summary might suggest because:

- Update operations on an object model are richer than their relational counterpart, given the additional collection types available (lists, ordered sets, bags).
- Method overriding is an issue in that a subclass may redeclare a side-effects-free operation (an OCL defined one), with that operation being used in an invariant. Instances of the subtype should have the overriding definition evaluated in place of the overridden one.
- Updates may have side effects, which in turn may affect invariants. These side effects result from inverse relationships maintained automatically in EMOF between two entities (its closest counterpart in relational databases is referential integrity). For example, upon deleting an instance which is bidirectionally linked to another, this second instance will have its reference cleared.

A concrete realization of the above ideas, satisfying the complexity requirements introduced in Sec. 4, is provided by the MOVIE algorithm for incremental view maintenance [19], explained in detail by its author in his PhD thesis [23]. A thorough performance evaluation [24] confirms its practical usefulness. The MOVIE algorithm is based on the translation of queries into the monoid calculus and their subsequent optimization, as discussed in [25] and [26]. The monoid calculus embodies the relational calculus, and has proven versatile enough to support both traditional as well as innovative optimizations. The software architecture of the proposed solution comprises:

1. The design time mapping of a model expressed in EMOF into a relational database schema (performed by a ready made component [27]). Data manipulation occurs at runtime only as EMOF-level update operations that are intercepted and matched against event patterns derived by MOVIE from view definitions for invariants.
2. The design time translation of OCL invariants into monoid calculus expressions. The resulting event patterns (derived by MOVIE for runtime interception) correspond to EMOF-level update operations. The accompanying actions generated by MOVIE to effect view maintenance are also EMOF-level updates.

The data definition, manipulation, and query languages (DDL, DML, DQL) of our solution are: EMOF, EMOF-level update operations, and full-OCL. The (incrementally maintainable) constraint language is the subset of OCL translatable into monoid calculus, and moreover valid as input for MOVIE (as defined in Sec. 4.1.2.2 of [23]). Although full-OCL is our standard DQL, nothing prevents the user from expressing read-only queries directly in SQL or in the ORM-level query language, JPQL [28] (Java Persistence Query Language, sometimes referred to as EJB3QL). Writing these "pass-through queries" in SQL requires knowledge of the mapping decisions encapsulated in item 1 above.

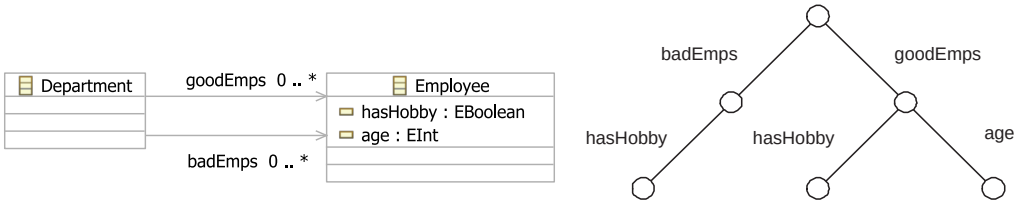
The barriers to efficient evaluation introduced by full-OCL are covered next, followed by an in-depth discussion of the translation of OCL into monoid calculus as a prerequisite to applying the MOVIE algorithm.

6 COMPUTATIONALLY COMPLETE CONSTRAINT LANGUAGE

Proposals using full-OCL for integrity constraints [20, 21] involve re-evaluating candidate broken invariants on a set of instances collected at runtime. The applied strategy consists in minimizing the amount of relevant instances, instead of avoiding re-computing subexpressions whose value has not changed (e.g., by caching their values). This is a major difference with incremental view maintenance. The essential aspects of the full-OCL approaches are illustrated with two examples, including the difficulties introduced by recursion. For a more detailed presentation see Sec. 4 in [21].

A core aspect of [20] and [21] is the observation that for each data element on which an OCL invariant depends, it is possible to derive a navigation-based query in the direction from the data element back to the instance where the invariant is evaluated. On the wake of an update on some data element, these navigation paths lead to a set of instances relevant for re-evaluating the invariant in question. For example, in an scenario where `Departments` may have good and bad `Employees` (Figure 1(a)), an integrity constraint may require the union of two sets (all bad employees and those good employees over forty) not to contain a hobbyist:

```
context Department
inv     noHobbyst  : badEmps->union(goodEmps->select(age > 40))
                     ->select(hasHobby)->isEmpty()
```



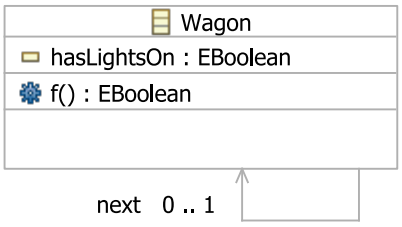
(a) Departments have good and bad employees (b) Reachability for `noHobbyst`

Figure 1: The `noHobbyst` example

Given a `Department` d, adding or removing employees (good or bad), as well as changing their hobby status may affect the invariant `noHobbyst` when evaluated for d. However, for this particular invariant, age updates are relevant only for good employees. This intuition is reflected in the reachability path shown as a tree in Figure 1(b). Thanks to bidirectional associations, upon an update to a node in that tree, the fixed-length path to the root can be followed to find the `Department` instance (i.e., d) on which invariant `noHobbyst` should be re-evaluated at transaction-commit time.

Special care is required for recursive functions ranging over dynamic data structures, as illustrated by the forward-only list of Figure 2. In that example, the invariant `lastWagonHasLightsOn` is fulfilled for a `Wagon` w in a train as long its last wagon has the lights on. In this case, a statically fixed back-navigation path will not achieve the desired result, as the required number of links to traverse changes at runtime. A conservative approach consists in re-evaluating recursive invariants for all instances of their contexts,

thus achieving completeness at the expense of efficient evaluation. It is not clear from [20, 21] how recursion over dynamic data structures is dealt with.




```
context Wagon
inv     lastWagonHasLightsOn : f()

context Wagon::f()
def   : if next.oclIsUndefined()
        then hasLightsOn
        else next.f()
        endif
```

Figure 2: The `lastWagonHasLightsOn` example

7 TRANSLATION OF OCL INTO MONOID CALCULUS

Queries translated into the monoid calculus refer to the same object-oriented schema as their OCL counterparts. No schema mapping is needed because most EMOF constructs have a direct counterpart in the monoid calculus, with the following exceptions: (a) EMOF-level ordered sets (no duplicates, insertion order preserved) are represented as monoid lists; (b) EMOF *dictionaries* (`Maps` in Java) are represented as sets of *(key, value)* pairs, where pairs are monoid lists. Under these conventions, for the purposes of side-effect-free queries, the result of evaluating the monoid translation agrees with its original OCL formulation. For update purposes instead, these conventions would not be consistent, as for example monoid lists do not capture the semantics of EMOF ordered sets (which require membership testing before insertion). We do not claim to optimize updates, whose semantics are enforced by the ORM engine. The fact that the same data schema is shared by both OCL and monoid expressions makes possible to optimize the monoid formulation without the additional complication of data mapping. No schema changes are introduced during rewriting for optimization. Finally, the optimized version is semantics preserving with respect to its original formulation.

An internal node in the AST of an OCL class invariant stands for a function application, with each subnode providing actual arguments. Some OCL constructs (e.g. `let v = ... in ...`) add identifiers to the scope visible in subtrees. Such syntax can be removed by expanding definitions, thus achieving the shape of "function application only" mentioned before. This rewriting does not alter meaning as OCL has call-by-value evaluation semantics. Terminal nodes are not tagged with function applications but with any of: (a) a literal constant; (b) the predefined OCL variable `self`; (c) entity extents of the form *`ClassName`*`.allInstances()`. The variable `self` ranges over an entity extent, namely that for the class where the invariant was defined. Unlike UML, there are no class-scoped attributes or associations in EMOF. We assume furthermore that invocations of user-defined, non-recursive functions have been replaced with their definitions (this may involve substituting usages of formal arguments by their corresponding actual arguments). To account

for late binding (choosing a function definition based on the actual type of a usage instead of its declared type) a potentially verbose case distinction is needed. This is no principle obstacle with *whole-model analysis*: all possible actual types are known at translation time and the actual type of an object can be queried with `oclIsTypeOf()`. After this preprocessing step, each internal node stands for the invocation of either an OCL predefined function or a user-defined (directly or indirectly) recursive function.

The Monoid Calculus

The monoid calculus provides a uniform notation for collections such as lists, bags and sets, based on the observation that the operations of set and bag union and list concatenation are monoid operations (that is, they are associative and have an identity element). Monoids for collection types are known as *collection monoids*. Operations like conjunctions and disjunctions on booleans and integer addition are instead *primitive monoids*. Borrowing notation from [25], a monoid of type T is a pair $(\oplus, \mathcal{Z}_\oplus)$ where $\oplus$ is an associative function of type $T \times T \to T$ and $\mathcal{Z}_\oplus$ is the left and right identity of $\oplus$. A monoid may be commutative (i.e., when $\oplus$ is commutative), idempotent (i.e., when $\forall x : x \oplus x = x$), or both. For example, $(+, 0)$ is a commutative and anti-idempotent monoid, while $(\cup, \{\})$ is both commutative and idempotent.

An expression of the form $\oplus[\![e \mid e_1 \ldots e_n]\!]$ is a *comprehension over monoid* $\oplus$. Unlike the prominent role granted in functional programming languages to *list* comprehensions, the notation above uniformly captures collection operations, whose kind is revealed by the outermost braces ([] for lists, {{ }} for bags, {} for sets). Each e_i is a qualifier, which can either be a generator of the form $v \leftarrow E$, where v is a variable and E is a collection-valued expression, or a filter p (a boolean valued predicate). Informally, each generator $v \leftarrow E$ sequentially binds variable v to the elements of expression E's value, making it visible in successive qualifiers. A filter evaluating to *true* results in successive qualifiers (if any) being evaluated under the current bindings, otherwise 'backtracking' takes place. The *head* expression e is evaluated for those bindings that satisfy all the filters, and taken togethers these values constitute the resulting collection. For example, the following SQL-like nested query:

```
select distinct e(x)
from ( select d(y) from E as y where q(y) ) as x
where p(x)
```

is translated as $\{\ e(x)\ \mid\ x\ \leftarrow\ \{\!\{\ d(y)\ \mid\ y\ \leftarrow\ E\ ,\ q(y)\ \}\!\}\ ,\ p(x)\ \}$

Applying a function f to each element in a collection (`map f xs` in Haskell) is thus expressed as $[\![\ f(x)\ \mid\ x \leftarrow xs\ ]\!]$, while `filter p xs` becomes $[\![\ f(x)\ \mid\ x \leftarrow xs,\ p(x)\ ]\!]$. Comprehensions in turn are syntactic sugar for *monoid homomorphisms*, which express structural recursion on the collection constructor ($+\!\!+$ for lists, $\cup$ for sets, $\uplus$ for bags), as shown pictorially in Figure 3 [29]. For example, taking $\otimes$ to be `max(x,y) = case x<y of true -> y | false -> x` makes $(\!|-\infty; max|\!)\ C$ find the maximum of collection C.

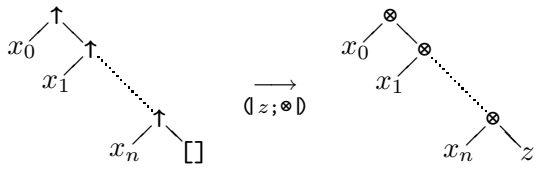


Figure 3: Graphical representation of the homomorphism from monoid $(\uparrow, [])$ to $(z, \otimes)$ (the latter not necessarily a collection monoid)

Translation Rules

Transformations for languages with a number of syntactic constructs (such as OCL) take the form of $LHS \rightarrow RHS$ pattern-based substitutions, where each OCL construct is matched by only one LHS. The transformation algorithm can be shown to correctly preserve meaning if each rewrite transformation is proved meaning-preserving. This follows case by case from definitions (in the respective semantic domains of OCL and monoid calculus). The rewrite rules are terminating because they decrease the number of occurrences of OCL constructs available for matching, and are confluent given that the LHS s partition the set of shapes that OCL constructs may take (each OCL construct being matched by one rewrite rule). Translation operates bottom-up from the leaves of the AST. For each node all required information is available locally due to pre-processing: no lookup of the correct binding for an OCL variable is needed as no such usages are left except for `self`.

Regarding the possible OCL constructs, Figure 4 depicts the relevant fragment of the OCL metamodel [30], i.e. the classes whose instances are nodes in an AST. As part of preprocessing, some constructs have been desugared (`LetExp`, `VariableExp`), while others do not appear in invariants (`MessageExp`). Occurrences of `UnspecifiedValueExp`, `InvalidLiteralExp`, and `NullLiteralExp` stand for the result of applying a partial function outside its domain. `StateExp` and `TypeExp` are functions that access instance-level data (the current state, given an associated statechart) and the actual type (which remains constant througout the lifetime of the instance, as EMOF lacks dynamic reclassification). Related to this, the boolean operation `oclIsKindOf()` reports whether a pair of types belongs to the transitive closure of the direct subtype relationship $\leq_1$ of EMOF + OCL [4].

OCL	Monoid calculus
`c->select( e | boolExpr(e) )`	$[\![\, e \mid e \leftarrow c\,,\ \text{boolExpr}(e) \,]\!]$
`c->reject( e | boolExpr(e) )`	$[\![\, e \mid e \leftarrow c\,,\ \text{boolExpr}(e) = \text{false} \,]\!]$
`c->exists( e | boolExpr(e) )`	$\vee\{\, \text{boolExpr}(e) \mid e \leftarrow c \,\}$
`c->forAll( e | boolExpr(e) )`	$\wedge\{\, \text{boolExpr}(e) \mid e \leftarrow c \,\}$
`c->collect( e | expr(e) )`	$[\![\, \text{expr}(e) \mid e \leftarrow c \,]\!]$
`c->one( e | boolExpr(e) )`	$1 = \text{length}(\, [\, e \mid e \leftarrow c\,,\ \text{boolExpr}(e)] \,)$ where $\text{length}(x) \equiv +[1 \mid e \leftarrow x]$

Table 1: Non-recursive subcases of `LoopExp`

Figure 4: Fragment of the OCL 2.0 metamodel (only inheritance relationships shown)

OCL constructs of the form `LiteralExp` are translated as follows: (a) a literal of the primitive types (integer, real, string, or boolean) has a corresponding monoid constant, the same goes for literals of a user-defined enumerations; (b) a collection literal of type ordered set or list is translated as a monoid list, while set and bag collections have direct counterparts; (c) a tuple literal is translated as a set of pairs $(tag, value)$.

The iterator expressions (`LoopExp`) comprise non-recursive subcases (Table 1). The remaining subcases are first desugared to their `iterate()` form as defined in the OCL standard ([30], Sec. 11.9 and A.3.1.3). `iterate()` in turn can be expressed as a left-fold. To capture this primitive recursive function, the *function composition monoid* $(\circ, \lambda x.x)$ is needed [25] where the function composition, $\circ$, defined as $(f \circ g)\, x = f(g(x))$, is associative but neither commutative nor idempotent. Even though the type of this monoid, $T_\circ(\alpha) = \alpha \to \alpha$, is parametric, it is still a primitive monoid. For a list $L = [a_1, a_2, \ldots, a_n]$, applying $\circ[\lambda x.f(x, a) \mid a \leftarrow L]$ to z expands to $(\lambda x.f(x, a_1)) \circ \ldots \circ (\lambda x.f(x, a_n))(z)$ which computes the left-fold $f(\ldots(f(f(z, a_n), a_{n-1}), \ldots a_1)$. The formulation of OCL's `c->iterate(a ; acc=init | expr(acc,a))` is thus the comprehension $\circ[\lambda\, acc.expr \mid a \leftarrow c](init)$. The expressive power of comprehensions involving $\circ$ lies in their ability to compose functions that propagate a state during list iteration. For example, the reverse of list L is $\circ[\lambda x.x{+\!\!+}[a] \mid a \leftarrow L]([])$. Actually, the OCL standard defines the semantics of all `LoopExp` in terms of `iterate()`, but as can be seen from Table 1 the additional expressive power is not necessary, and may complicate optimization by hiding properties that $\otimes$ may exhibit (commutativity, idempotence).

OCL	Monoid calculus
`c->count(m)`	$+[1 \mid e \leftarrow c, e = m]$
`c->excludes(m)`	$\wedge\{e \neq m \mid e \leftarrow c\}$
`c1->excludesAll(c2)`	$\wedge\{\wedge\{e \neq m \mid e \leftarrow c_1\} \mid m \leftarrow c_2\}$
`c->includes(m)`	$\vee\{e = m \mid e \leftarrow c\}$
`c1->includesAll(c2)`	$\wedge\{\vee\{e = m \mid e \leftarrow c_1\} \mid m \leftarrow c_2\}$
`c->isEmpty()`	$c = [\![\,]\!]$
`c->sum()`	$+[e \mid e \leftarrow c]$
`c->size()`	$+[1 \mid e \leftarrow c]$
`c1->product(c2)`	$\{(x, y) \mid x \leftarrow c_1, y \leftarrow c_2\}$

Table 2: Standard OCL operations on all collection types

In EMOF terminology, a *structural feature* is either (a) an instance field or association end; or (b) a method. Accessing the value of (a) is represented with `PropertyCallExp`. Invoking an (OCL-defined, side-effect free) method is represented with `OperationCallExp`. Therefore, occurrences of these constructs are translated as function application in monoid expressions. The sibling of `PropertyCallExp` (`AssociationClassCallExp`) is not relevant for EMOF, as class-scoped structural features are not allowed. The pending cases of `OperationCallExp` not translated so far comprise: (a) operations on the primitive types boolean, integer, real, and string; and (b) collection operations (not to be confused with iterator operations). The first group can be translated as-is given that all storage engines implement them natively. From the point of view of optimization, they are handled as black-boxes. Translation rules for collection operations appear in Tables 2 to 4, classified by computational complexity, which is not apparent from the uniform OCL syntax.

The implementation of OCL AST transformations is discussed in [31], including techniques such as the encapsulation of walker code, instantiation of type-parametric visitors with type substitutions, and tracking the input-output relationship between AST nodes along a chain of visitors.

8 OPTIMIZATIONS WITH MONOID CALCULUS

The invariant `noHobbyst` (Figure 1 in Sec. 6) is amenable to a basic optimization, pushing selections below joins (the predicate `hasHobby = true` appears only after building partial results, performing it earlier increases selectivity). The vast body of query optimization algorithms is not applicable to the surface syntax of OCL: the same concept can be expressed in so many different ways that *ease of manipulation* (Sec. 3) is impracticable.

We claim that query optimization is required for two purposes in an EMOF + OCL setting: (a) for ad-hoc queries, and (b) to optimize expressions obtained from OCL invariants before their *maintenance plans* are derived by MOVIE. The case for (a) should be evident. As for (b), the authors of [21] observe that invariant rewriting may disconcert users, who would be faced with integrity violation errors based on expressions they have

OCL	Monoid calculus
`c1->excluding(c2)`	$[\![e \mid e \leftarrow c_1, \wedge\{d \neq e \mid d \leftarrow c_2\}]\!]$
`c->append(m)`	$c{+}{+}[m]$
`c->asBag()`	$\{\!\{e \mid e \leftarrow c\}\!\}$
`c->asOrderedSet()` `c->asSequence()`	$[e \mid e \leftarrow c]$
`c->asSet()`	$\{e \mid e \leftarrow c\}$
`c->flatten()`	$\oplus[\![e \mid s \leftarrow c, e \leftarrow s]\!]$
`c->including(m)`	$c \oplus [\![m]\!]$
`c1->intersection(c2)`	$\oplus[\![e \mid e \leftarrow c_1, \vee\{e = d \mid d \leftarrow c_2\}]\!]$
`c->prepend(m)`	$[\![m]\!] \oplus c$
`c1->union(c2)`	$c_1 \cup c_2$
`c1->symmetricDifference(c2)`	same translation as for `(c1->union(c2))->` `excluding((c1->intersection(c2)))`

Table 3: Overloaded collection operators ($\oplus$ stands for the merge operator of the resulting collection monoid)

`c->at(i)`	$(\circ[\![\lambda(x,k).(\text{if } k = i \text{ then } a \text{ else } x, k-1) \mid a \leftarrow c]\!]$ $(\text{NULL}, \text{length}(x))).\text{fst}$
`c->first()`	$\circ[\![\lambda x.a \mid a \leftarrow c]\!]$
`c->last()`	same translation as for `c->at(c->size())`
`c->indexOf(m)`	$(\circ[\![\lambda(x,k).(\text{if } a = m \text{ then } k \text{ else } -1, k-1) \mid a \leftarrow c]\!]$ $(-1, \text{length}(x))).\text{fst}$
`c->subOrderedSet(j,k)` `c->subSequence(j,k)`	$(\circ[\![\lambda(x,i).\text{ if } j \leq i \leq k \text{ then } ([a]{+}{+}x, i-1)$ $\text{else } (x, i-1)\,]\!]([], \text{length}(c))).\text{fst}$
`c->insertAt(k,m)`	$(\circ[\![\lambda(x,k).\text{if } i = k \text{ then } ([m]{+}{+}[a]{+}{+}x, i-1)$ $\text{else } ([a]{+}{+}x, i-1) \mid a \leftarrow c]\!]([], \text{length}(c))).\text{fst}$

Table 4: Collection operations involving comprehensions of function composition

never seen before. As a consequence, rewriting in general (and optimization in particular) is explicitly avoided. The usability concern in question can be addressed in that error messages can be produced by evaluating the original OCL invariant once it is known (by optimized evaluation) that it has been broken. Actually, re-evaluation is inherent to the approach in [21], thus incurring no additional overhead.

The primitive operations supported by storage managers or query engines correspond to query algebra operators (*semi-joins*, selection supported by indexes, etc.) The monoid calculus takes advantage of this fact by offering a uniform framework for query translation, rewriting for optimization, and execution plan generation: query optimization can be made aware of the physical schema (table partitioning applied as part of ORM), saving I/O costs. To illustrate this kind of optimization, we show an end-to-end example of translation, optimization, and plan generation aware of physical schema, adapted to the EMOF + OCL setting from [26].

Consider a database of `Films` and the `actors` appearing in them (recording in how many scenes, `scenes`), together with the films' `directors`, as shown in Figure 5.

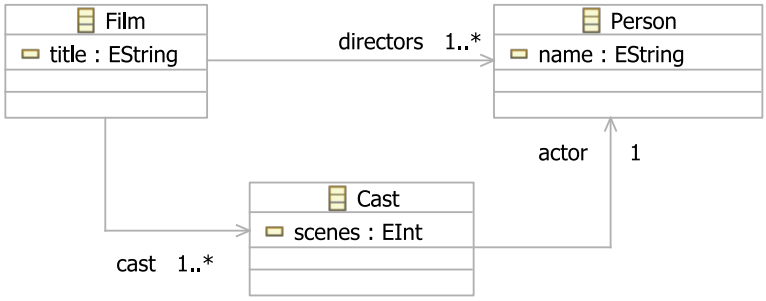


Figure 5: EMOF-level logical schema for the films, actors, and directors database

Assuming that most queries access either actors or directors, it makes sense to *vertically decompose* the logical schema into four tables (see Figure 6). Clustering table columns that are frequently accessed together avoids unnecessary I/O, as its elements are stored physically contiguous.

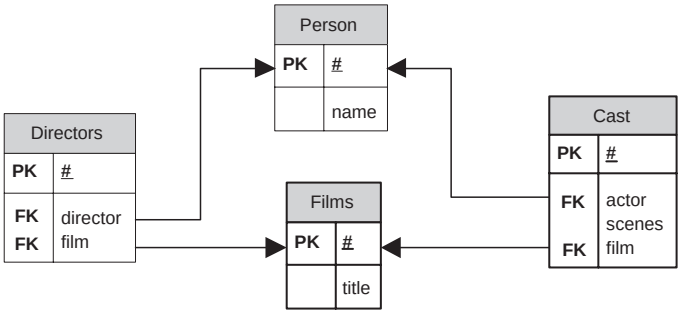


Figure 6: Physical schema for the films, actors, and directors database

The OCL query below (in terms of the logical schema in Figure 5) returns the titles of Hitchcock-style films: the director appears as an actor in exactly one scene.

```
Film.allInstances()->select( f |
      f.directors->exists( d |
           f.cast->exists( c | c.scenes = 1 and c.actor = d ) ) )
->collect( f | f.title )
```

Its translation as a monoid comprehension is as readable as the OCL version:

$$\begin{array}{l}\{f.\text{title} \mid \quad f \leftarrow \text{film}, \\ \qquad\quad some\{some\{d = c.\text{actor} \mid c \leftarrow f.\text{cast}, c.\text{scenes} = 1\} \mid \\ \qquad\qquad\quad d \leftarrow f.\text{directors}\} \\ \}\end{array}$$

Before optimization can start, the connection to the physical schema is established by replacing *film* by its definition in terms of the vertically decomposed tables, using the *nest* operator to reconstruct the owned collections of actors and directors for each film. With that, the monoid comprehension can be normalized: all variables ranging over collections (i.e., f, d, c) appear first followed by a predicate in conjunctive normal form. This not yet optimized formulation has a direct counterpart in query algebra (Figure 7), a tree of cartesian products. In principle, relational optimizations could start from there, thus guaranteeing that monoid-based optimizations do not end up in execution plans worse than relational optimization (e.g., exchanging two generators results in join reordering).

$$\pi_{f_p.\mathrm{title}}((\mathrm{Films}_{f_p} \underset{f_p.\#=d_p.\mathrm{film}}{\bowtie} \mathrm{Directors}_{d_p}) \underset{f_p.\#=c_p.\mathrm{film}\wedge d_p.\mathrm{director}=c_p.\mathrm{actor}}{\bowtie} (\sigma_{c_p.\mathrm{scenes}=1}(\mathrm{Cast}_{c_p})))$$

Figure 7: Query algebra formulation, non-optimized stage

The semijoin $E_1 \ltimes_p E_2$ is a join variant that delivers only those left operand objects having at least one join partner with respect to the join predicate p. Its implementation is efficient because as soon as a join partner is found for an E_1 object, then it is known to belong to the result and no further E_2 objects need be accessed. The monoid comprehension formulation allows detecting those access patterns that correspond to semijoins. In the example, after partial flattening of subqueries (not shown), the shaded subexpression in Figure 8 is one such case.

$$\{f.\mathrm{title} \mid f \leftarrow \mathrm{films},\ d \leftarrow f.\mathrm{directors},\ \mathit{some}\{d = c.\mathrm{actor} \mid c \leftarrow f.\mathrm{cast}, c.\mathrm{scenes} = 1\}\}$$

Figure 8: Semi-join access pattern

The resulting optimized formulation appears in Figure 9.

$$\mathrm{Films}_{f_p} \underset{f_p.\#=d_p.\mathrm{film}}{\ltimes} (\mathrm{Directors}_{d_p} \underset{c_p.\mathrm{actor}=d_p.\mathrm{director}\wedge d_p.\mathrm{film}=c_p.\mathrm{film}}{\ltimes} (\sigma_{c_p.\mathrm{scenes}=1}(\mathrm{Cast}_{c_p})))$$

Figure 9: Query algebra formulation, optimized stage

The resulting query plan expressed in relational algebra has a straightforward translation into JPQL [28], the ORM-level query language. Further potential relational optimizations may be performed by the RDBMS. In keeping with MDSE principles, this translation is not implemented as string manipulations but as an AST-to-AST transformation [32]. Well-formedness is thus ensured before delivering output for further processing.

Without the conceptual framework of the monoid calculus, applying similar rewritings directly on OCL ASTs would not have been feasible. This is further evidence to the claim that integrity checking for EMOF + OCL should follow the approach proposed here.

9 RELATED WORK

The influence of the Object Query Language (OQL) defined in the 1990s by the Object Data Management Group (ODMG) cannot be understated, reaching to JPQL today. Trigoni [33] formalizes type inference for OQL queries. Additionally, algorithms are provided for applying two semantic optimization heuristics: constraint introduction and constraint elimination. These refined heuristics take into consideration *association rules* discovered with data mining, which are not as strong as integrity constraints (they may have exceptions in fact). Given that these "rules" statistically hold most of the time, it pays off to monitor their validity status at runtime. Unless they become invalid, they can be used during optimization to increase selectivity and to skip evaluations, thus improving performance. As with other heuristic techniques, safety measures are built in to prevent the cost of analysis to exceed optimization speed-up.

Ritter et. al. [10] also aim at integrity checking by translating OCL into a view definition language, this time SQL'92. However, no systematic performance analysis is made. The Dresden OCL Toolkit [34] compiles full-OCL into RDBMS stored procedures including control structures, thus compromising query optimization in the general case.

The optimization of object-based queries is not only relevant for the persistent case: naïve evaluation over instances in main-memory also results in unacceptable performance. A succint account of this problem and a heuristic solution for θ-joins in Java 5 appears in [35]. A recent book on the subject of database integrity is [36]. Most contributions focus on the relational case. The book [37] is devoted to view materialization.

10 CONCLUSIONS AND FURTHER WORK

We have addressed an industrially relevant problem by going back to first principles, leveraging research results from object databases to improve the efficiency of software repositories for EMOF + OCL. Our choice of integrity checking mechanism does not require for the database to be in a consistent state before an update can take place, yet reporting of integrity violations is sound and complete (no false positives, no missed violations). This is deemed vital to account for the realities of collaborative design environments.

Incremental view maintenance adds a measure of reactivity to the monitoring of invariants. Unlike the more powerful Event Condition Action rules (ECA) of an active DBMS, view definitions based on OCL invariants cannot make statements about events external to the database state, nor range over several snapshots as in versioned data models [11] (values in pre- and poststates can only be referred from OCL postconditions, not from class invariants). OCL-based views are however sufficient to support a variety of use cases in software repositories, such as monitoring the conformance of artifacts to coding and modeling conventions [12]. Moreover, not all views need be maintained incrementally (as required for integrity constraints): in some cases results are only periodically needed (e.g., after an integration build, or on a daily or weekly basis). Examples include: (a) detecting opportunities for applying refactorings; (b) checking mutual consistency be-

tween artifacts and documentation; and (c) deriving software metrics.

As usual, irrespective of whether an OCL-based view is tagged for incremental or batch evaluation, it makes for concise composite queries. Materialized views naturally support OCL's `derive` statement, which is used to specify values for attributes or association ends. Looking into the future, the proposed infrastructure can serve as a basis for supporting ECA and versioning functionality through extensions to the OCL language.

REFERENCES

[1] Object Management Group: Meta Object Facility (MOF) Core Specification, formal/06-01-01, http://www.omg.org/docs/formal/06-01-01.pdf (Jan 2006)

[2] Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley, Boston, MA, USA (2003) ISBN 0321179366.

[3] Dittrich, K.R., Tombros, D., Geppert, A.: Databases in Software Engineering: a Roadmap. In: ICSE - Future of SE Track. (2000) 293–302

[4] Garcia, M.: Rules for Type-checking of Parametric Polymorphism in EMF Generics. In Bleek, W.G., Schwentner, H., Züllighoven, H., eds.: Software Engineering 2007 – Beiträge zu den Workshops. Volume 106 of GI-Edition Lecture Notes in Informatics. (2007) 261–270 http://www.sts.tu-harburg.de/~mi.garcia/pubs/2007/mdsdHeute/garcia-emfgen-2.pdf.

[5] Garcia, M., Möller, R.: Certification of Transformations Algorithms in Model-Driven Software Development. In Bleek, W.G., Räsch, J., Züllighoven, H., eds.: Software Engineering 2007. Volume 105 of GI-Edition Lecture Notes in Informatics. (2007) 107–118 http://www.sts.tu-harburg.de/~mi.garcia/pubs/2007/se2007/GarciaMoeller.pdf.

[6] Muller, P.A., Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Gérard, S., Jézéquel, J.M.: Model-Driven Analysis and Synthesis of Concrete Syntax. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: MoDELS. Volume 4199 of LNCS., Springer (2006) 98–110 http://lglpc35.epfl.ch/lgl/docs/papers/MDASOCS-5-pam.pdf.

[7] Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L., eds.: GPCE, ACM (2006) 249–254

[8] Daly, C.J.: AST framework generation with Gymnast. In: Tech Exchange Panel: Language Toolkits, EclipseCON 2005 (2005)

[9] Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Generation of Visual Editors as Eclipse Plugins. In: ASE '05: Proceedings of the 20th IEEE/ACM Intnl Conf on Automated Software Engineering, New York, NY, USA, ACM Press (2005) 134–143

[10] Ritter, N., Steiert, H.P.: Enforcing Modeling Guidelines in an ORDBMS-based UML Repository. In: Intnl Resource Mgmt. Assoc. Conf. 2000 (Information Modeling Methods and Methodologies Track of IRMA 2000), Anchorage, Alaska (May 2000) 269–273

[11] Kovse, J.: Model-Driven Development of Versioning Systems. PhD thesis, TU Kaiserslautern, Germany (August 2005)

[12] Ruokonen, A., Hammouda, I., Mikkonen, T.: Enforcing Consistency of Model-Driven Architecture Using Meta-Designs. In: European Conf. on MDA: Workshop on Consistency in Model Driven Engineering (C@MoDE 2005). (Nov. 2005) 127–141 http://practise.cs.tut.fi/files/publications/EEWES/metadesign.pdf.

[13] Amelunxen, C., Schürr, A.: On OCL as part of the Metamodeling Framework MOFLON. In: 6th OCL Workshop at the UML/MoDELS Conference. (2006) http://st.inf.tu-dresden.de/OCLApps2006/topic/acceptedPapers/13_Amelunxen_MOFLON.pdf.

[14] Brucker, A.D., Wolff, B.: The HOL-OCL Book. Technical Report 525, ETH Zürich (2006) http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-ocl-book-2006.

[15] Fegaras, L., Maier, D.: Towards an Effective Calculus for Object Query Languages. In: SIGMOD '95: Proceedings of the 1995 ACM SIGMOD Intl Conf. on Management of Data, New York, NY, USA, ACM Press (1995) 47–58 http://lambda.uta.edu/sigmod95.ps.gz.

[16] Lamport, L.: The +CAL Algorithm Language. In: NCA '06: Proc of the Fifth IEEE Intnl Symposium on Network Computing and Applications, Washington, DC, USA, IEEE Computer Society (2006) 5–10 See also http://research.microsoft.com/users/lamport/pubs/pluscal.pdf.

[17] Lawley, M.: Transaction Safety in Deductive Object-Oriented Databases. In Ling, T.W., Mendelzon, A., Vieille, L., eds.: DOOD. Volume 1013 of LNCS., Springer (1995) 395–410

[18] Stockmeyer, L.J.: The Complexity of Decision Problems in Automata Theory and Logic. Technical Report MAC TR-133, MIT, Cambridge MA, Project MAC (1974)

[19] Ali, M.A., Fernandes, A.A.A., Paton, N.W.: MOVIE: an incremental maintenance system for materialized object views. Data Knowl. Eng. **47**(2) (2003) 131–166

[20] Cabot, J., Teniente, E.: Incremental Evaluation of OCL Constraints. In Dubois, E., Pohl, K., eds.: CAiSE. Volume 4001 of LNCS., Springer (2006) 81–95 Project homepage http://www.lsi.upc.edu/~jcabot/research/IncrementalOCL/index.html.

[21] Altenhofen, M., Hettel, T., Kusterer, S.: OCL Support in an Industrial Environment. In Demuth, B., Chiorean, D., Gogolla, M., Warmer, J., eds.: OCL for (Meta-)Models in Multiple Application Domains, Dresden, University Dresden (2006) 126–139 http://st.inf.tu-dresden.de/OCLApps2006/topic/acceptedPapers/03_Altenhofen_OCLSupport.pdf.

[22] Meyer, B., Mingins, C., Schmidt, H.: Providing trusted components to the industry. Computer **31**(5) (1998) 104–105

[23] Ali, M.A.: Incremental Maintenance of Materialized Views in Object-Oriented Databases. PhD thesis, University of Manchester, UK (September 2002) http://computing.unn.ac.uk/staff/CGMA2/projectlinks.html.

[24] Ali, M.A., Paton, N.W., Fernandes, A.A.A.: An Experimental Performance Evaluation of Incremental Materialized View Maintenance in Object Databases. In Kambayashi, Y., Winiwarter, W., Arikawa, M., eds.: DaWaK. Volume 2114 of LNCS., Springer (2001) 240–253

[25] Fegaras, L., Maier, D.: Optimizing Object Queries using an Effective Calculus. ACM Trans. Database Syst. **25**(4) (2000) 457–516

[26] Grust, T., Scholl, M.H.: Translating OQL into Monoid Comprehensions—Stuck with Nested Loops? Technical Report 3a/1996, Dept. of Computer and Information Science, Database Research Group, U Konstanz, (September 1996) http://www.inf.uni-konstanz.de/dbis/publications/download/GS:TR96a.ps.gz.

[27] Elver Project: Teneo EMF Persistency, http://www.eclipse.org/emft/projects/teneo/ (2007)

[28] EJB 3.0 Expert Group: JSR 220: Enterprise JavaBeans, Version 3.0: EJB 3.0 Simplified API. Available at http://java.sun.com/products/ejb/docs.html (2005)

[29] Grust, T.: Monad Comprehensions. A Versatile Representation for Queries. In: The Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data. Springer Verlag (Sept 2003) 288–311 ISBN: 978-3-540-00375-5.

[30] Object Management Group: OMG OCL Specification v2.0, formal/2006-05-01 (May 2006) http://www.omg.org/technology/documents/formal/ocl.htm.

[31] Garcia, M.: How to process OCL Abstract Syntax Trees (2007) http://www.eclipse.org/articles/article.php?file=Article-HowToProcessOCLAbstractSyntaxTrees/index.html, Eclipse Technical Article.

[32] Garcia, M.: Formalizing the Well-formedness Rules of EJB3QL in UML + OCL. In Kühne, T., ed.: Reports and Revised Selected Papers, Workshops and Symposia at MoDELS 2006, Genoa, Italy. LNCS 4364, Springer-Verlag (2006) 66–75

[33] Trigoni, A.: Semantic Optimization of OQL Queries. PhD thesis, University of Cambridge, UK (October 2002) http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-547.html.

[34] Demuth, B., Hußmann, H., Loecher, S.: OCL as a Specification Language for Business Rules in Database Applications. In Gogolla, M., Kobryn, C., eds.: UML. Volume 2185 of LNCS., Springer (2001) 104–117

[35] Willis, D., Pearce, D.J., Noble, J.: Efficient Object Querying for Java. In Thomas, D., ed.: ECOOP. Volume 4067 of LNCS., Springer (2006) 28–49 http://www.mcs.vuw.ac.nz/~djp/files/WPN_ECOOP06.ps.

[36] Doorn, J.H., Rivero, L.C., eds.: Database Integrity: Challenges and Solutions. Idea Group Publishing (2002)

[37] Gupta, A., Mumick, I.S., eds.: Materialized views: techniques, implementations, and applications. MIT Press, Cambridge, MA, USA (1999)

ABOUT THE AUTHORS

Miguel Garcia is a PhD candidate and research assistant at the Institute for Software Systems at the Hamburg University of Science and Technology (TUHH), Germany. He can be reached at miguel.garcia@tuhh.de. See also http://www.sts.tu-harburg.de/~mi.garcia.