

Formalizing the well-formedness rules of EJB3QL in UML + OCL

Miguel Garcia

Hamburg University of Technology, Hamburg 21073, Germany,
miguel.garcia@tuhh.de,
WWW home page: <http://www.sts.tu-harburg.de/~mi.garcia/>

Abstract. This paper reports the application of language metamodeling techniques to EJB3QL, the object-oriented query language for Java Persistence recently standardized in JSR-220. Five years from now, today's EJB3 applications will be legacy. We see our metamodel as an enabler for increasing the efficiency of reverse engineering activities. It has already proven useful in uncovering spots where the EJB3QL spec is vague. The case study reported in this paper involved (a) expressing the abstract syntax and well-formedness rules of EJB3QL in UML and OCL respectively; (b) deriving from that metamodel software artifacts required for several language-processing tasks, targeting two modeling platforms (Eclipse EMF and Octopus); and (c) comparing the generated artifacts with their counterparts in the reference implementation of EJB3 (which was not developed following a language-metamodeling approach). The metamodel of EJB3QL constitutes the basis for applying model-checkers to aid in assuring conformance of tools claiming to support the specification.

1 Introduction

Language Engineering is an increasingly important area which leverages basic results from Computer Science and a variety of tools developed over the years. Its main goals are (a) simplifying the generation of language processing tools and environments, as well as (b) offering guarantees about their conformance and interoperability. Progress around the first category (productivity) has benefited software practitioners facing the task of developing tooling for domain-specific languages in a cost-effective manner. Our findings confirm that language specifications which increase productivity are also well-suited for the second category (correctness), an issue that has gained less attention overall.

Concrete examples of features under the “productivity” category include syntax-aware editors, support for visual syntax (either for editing or visualization only), and integrated software repositories (cross-artifact detection of inconsistencies, metrics). Support under the “correctness” category includes the model-checking of language processing algorithms to offer guarantees about the transformations they realize. Two basic desirable guarantees are: (a) that all output sentences belong to the target language [1], and (b) that the transformation

function covers the whole input language for which it was designed [2]. Experience with current model-driven tooling shows that these basic requirements are not always met. Beyond these general requirements, guarantees specific to a transformation are also desirable. For example, that an optimized implementation outputs the same result as the non-optimized version.

The structure of this paper is as follows. Sec. 2 presents motivational examples of well-formedness rules and their formulation in the metamodel of EJB3QL. Sec. 3 discusses the impact of language metamodeling techniques on the consistency and completeness of a language specification. Sec. 4 summarizes places where the JSR-220 EJB3QL spec was found to be incomplete or imprecise. Sec. 5 is devoted mostly to tooling, explaining how to integrate in language processing tools the software artifacts generated from metamodels. Sec. 6 discusses related work, with Sec. 7 offering conclusions and possibilities for further work.

This paper assumes knowledge from the reader about object-oriented modeling and database query languages. References to individual techniques and sub-problems are discussed, and context is provided in the form of motivation and related work. The software artifacts developed as part of this case study are available for download from [24]

2 Reverse engineering the EJB3QL spec: how and why

The EJB3QL spec includes an EBNF grammar which, as usual, cannot capture all well-formedness constraints relevant to the language being defined. Implementers of the spec cannot rely on a machine-processable specification of all relevant well-formedness rules (WFRs) thus leaving open the possibility for non-interoperable implementations, given the finite resources that can be devoted to check conformance.

The evaluation of WFRs that are not captured by an EBNF grammar becomes a responsibility of the semantic analysis phase of a language processing tool. As a simple example of one such check, JSR-220 requires “*Entity names are scoped within the persistence unit and must be unique within the persistence unit.*” (Sec. 4.3.1). The OCL formulation is as follows:

<pre>context PersistenceUnit inv WFR_4_3_1 : self.entities ->isUnique(name)</pre>
--

Beyond the productivity gain (once expressed in OCL, Java code to evaluate it can be generated automatically), the fact that this check is specified declaratively instead of implemented procedurally makes the resulting artifacts amenable to formal verification. For this particular WFR the “many-eyeballs principle” is enough for validating an implementation. This strategy does not scale to more subtle, intricate WFRs. Sec. 4 contains the OCL encoding of complex WFRs for which the correctness of a procedural evaluation is non-obvious.

As a further motivating example consider the case where the EBNF grammar underspecifies the WFR about expressions that compare values declared in enumerations. The production `comparison_expression` contains an alternative

(shown in bold on Table 1) for just this case, (in-)equality comparison of values coming from enumerations:

Table 1: EBNF for `comparison_expression`

```

comparison_expression ::=
  string_expression comparison_operator
  {string_expression | all_or_any_expression} |
  boolean_expression { = | <> }
  {boolean_expression | all_or_any_expression} |
  enum_expression { = | <> }
  { enum_expression | all_or_any_expression } |
  datetime_expression comparison_operator
  {datetime_expression | all_or_any_expression} |
  entity_expression { = | <> }
  {entity_expression | all_or_any_expression} |
  arithmetic_expression comparison_operator
  {arithmetic_expression | all_or_any_expression}

```

In fact, comparing values from different enumeration types makes no sense (doing so would defeat the whole purpose of enumeration types) but the grammar does not rule it out. This particular WFR is probably included in the semantic analysis phase of the reference implementation (but we haven't examined its source code to confirm it) while its OCL formulation is quite compact:

```

context EnumCompExp
  inv comparedValuesBelongToTheSameEnumerationType :
    left.type() = right.type()

```

In another category, the grammar in the spec sometimes misses the opportunity to make distinctions that it *could* express, a fact that was brought to our attention by comparing it with its version for the ANTLR parser generator (<http://www.antlr.org>) in the reference implementation. From the spec:

4.6.9 Like Expressions

The syntax for the use of the comparison operator *[NOT] LIKE* in a conditional expression is as follows:

```

string_expression [NOT] LIKE pattern_value [ESCAPE
escape_character]

```

The `string_expression` must have a string value. The `pattern_value` is a string literal or a string-valued input parameter in which ...

According to this, `pattern_value` can be replaced by one of two specific constructs, for which grammar productions are defined (`literalString` and `inputParameter`). The normative document does not make this distinction (in

that `pattern_value` is left undefined.) The reference implementation however reflects the intention of the spec, except that it calls `likeValue` what the spec calls `pattern_value` (Figure 1):

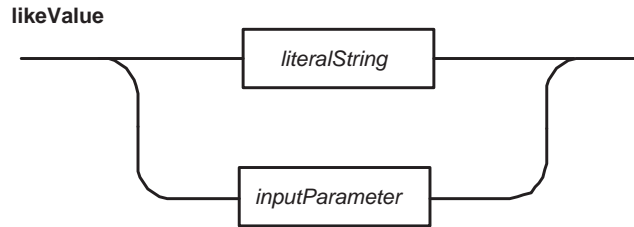


Fig. 1. Reference implementation, what a `pattern_value` can be

The argument can be made that even if abstract syntax is expressed as a UML+OCL metamodel, an EBNF grammar is still required to specify a concrete textual syntax. Proposals exist [3–5] to decorate an object-oriented language description with annotations to specify concrete syntax by choosing among a fixed palette of alternatives (e.g. to indicate whether an operator is prefix or infix). A further use for such information is the generation of an unparser. Additionally, if a representation of the metamodel is available at runtime, the implementation of syntax-sensitive features (e.g. content assist) is made simpler.

A sidenote on the UML tooling involved so far. Our metamodel was first prepared in Octopus [6], an Eclipse-based BSD-licensed tool supporting UML 1.4 class models enriched with OCL 2.0. As explained in Sec. 5, a translation to Eclipse EMF+OCL was achieved automatically. As for the grammarware tooling [22], the details are: “railroad diagrams” for the ANTLR grammar were generated with [7]. Another EJB3QL grammar is available as part of OpenJPA [8]. It is encoded in JJTree [23], a tree builder for the JavaCC parser generator.

We will follow the convention of displaying grammar productions from JSR-220 in EBNF notation. The names of OCL invariants have been chosen to allow for easy cross-referencing with the spec, each such name is prefixed with “WFR_” followed by the section number where the spec introduces the constraint.

3 Consistency and completeness enforced by language metamodeling

Expressing the structure and WFRs of a language as a UML+OCL metamodel forces the specification authors to consider corner cases that may be easily overlooked otherwise. While encoding in OCL the WFRs around type compatibility for comparison and for assignment expressions, we noticed that the spec is not clear about what combinations of (LHS type, RHS type) are valid in assignments (as part of the UPDATE statement), in case persistent entity types are

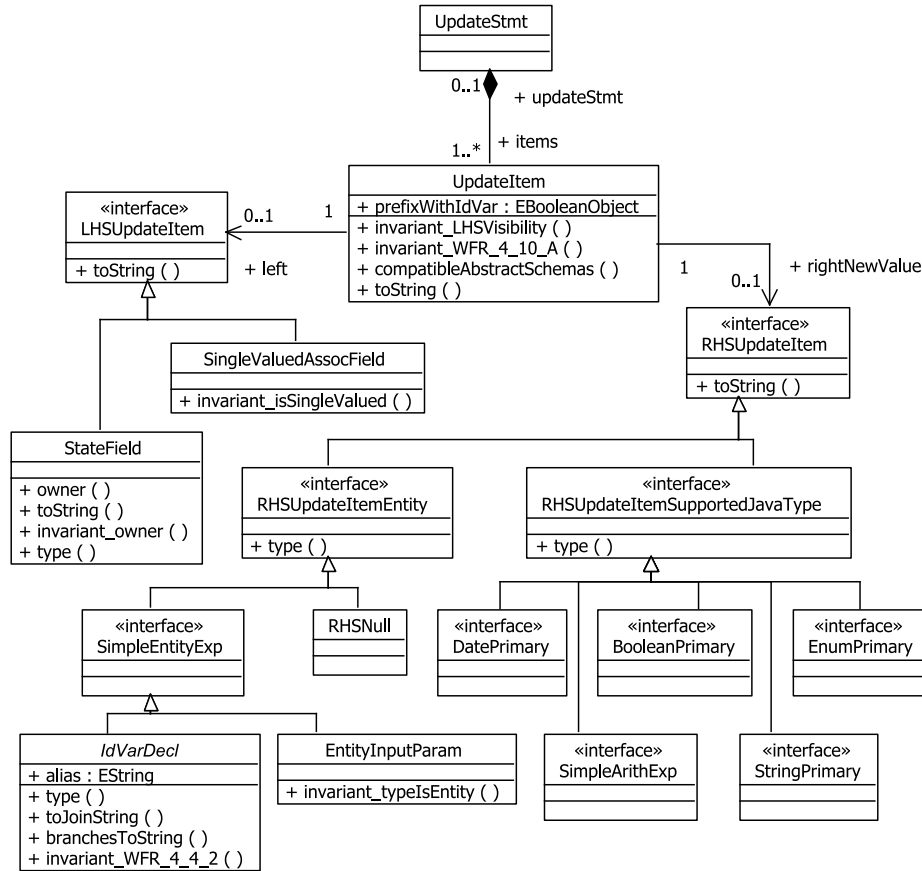


Fig. 2. Metamodel fragment for the UPDATE statement

involved. The spec is silent about whether assigning a B-typed value to a field with declared type A (where B is a subtype of A) is standard across implementations, implementation-dependent, or disallowed. Portability warnings for such cases are encoded in our metamodel as OCL invariants. For example, “*State-fields that are mapped in serialized form or as LOBs may not be portably used in conditional expressions*” (Sec. 4.6 of the spec) can be found by searching for PORTABILITY_4_6. This section discusses in more detail our observations around the UPDATE statement.

Following the grammar in the spec, our metamodel allows an UPDATE statement to own one or more UpdateItem, each representing a LHS := RHS. All constructs that are allowed on the LHS support the UML interface LHSUpdateItem, similarly for those on the right hand side (Figure 2). For comparison, the EBNF counterpart is reproduced in Table 2.

Notice that all shared properties of alternatives in a production rule can be factored out into the interface that covers them. In the UPDATE example, all constructs (and only those) on the RHS that may evaluate to a primitive type conform to the interface `RHSUpdateItemSupportedJavaType`, thus allowing an OCL expression to abstract away from the sub-cases.

Table 2: Metamodel fragment for the UPDATE statement

```

update_statement ::= update_clause [where_clause]
update_clause ::=
  UPDATE abstract_schema_name [[AS] identification_variable]
  SET update_item {, update_item}*
update_item ::=
  [identification_variable.]
  {state_field | single_valued_association_field } = new_value
new_value ::=
  simple_arithmetic_expression |
  string_primary |
  datetime_primary |
  boolean_primary |
  enum_primary
  simple_entity_expression |
  NULL

```

Sec. 4.10 of the spec deals with assignments involving primitive types only: “*The new_value specified for an update operation must be compatible in type with the state-field to which it is assigned.*” For completeness, the WFR for type compatibility for comparison (not assignment) between entities is also mentioned here, although it does not shed light on this issue: “*Two entities of the same abstract schema type are equal if and only if they have the same primary key value.*” (Sec. 4.12)

Making explicit the underspecified assignment case is forced upon us by OCL type checking. It all starts when we consider the two sub-cases for a LHS: interface `LHSUpdateItem` is realized by classes `StateField` and by `SingleValuedAssocField` only (our metamodel faithfully enforces the partition semantics: the sub-cases cover completely and are disjoint with each other).

Listing 1.1 reproduces the OCL if-statement that specifies the compatibility condition for the primitive-types case (the then-branch) as well as the entity-types case (the else-branch). The else-branch in turn has to consider again the two partitioning sub-cases of the RHS: primitive or entity. The first case prompts returning false (the types are not assignment-compatible). The second case embodies a conservative approach: only assignments of entities of exactly the same declared type are allowed, for lack of additional assurances from the specification. This can be revised as the spec is updated.

Listing 1.1. OCL encoding of type compatibility for assignments in an UpdateItem

```

— "The new_value specified for an update operation must be
— compatible in type with the state-field to which
— it is assigned"
context UpdateItem
inv WFR_4_10_A :
if left.ocIsKindOf(ejb3qlmm :: pathExp :: StateField)
  then
  — LHS is typed with SupportedJava Type
  if not rightNewValue.ocIsKindOf(
   .ejb3qlmm :: stmts :: RHSUpdateItemSupportedJava Type)
  then false
  else let
    t1 :.ejb3qlmm :: schema :: SupportedJava Type
      = left.ocAsType(ejb3qlmm :: pathExp :: StateField)
        .type(),
    — RHS is either SimpleArithExp, StringPrimary,
    — BooleanPrimary, DatePrimary, or EnumPrimary
    t2 :.ejb3qlmm :: schema :: SupportedJava Type
      = rightNewValue.ocAsType(
        .ejb3qlmm :: stmts :: RHSUpdateItemSupportedJava Type)
        .type()
    in .ejb3qlmm :: schema :: SupportedJava Type ::
      areTypeCompatible(t1, t2)
  endif
  else
  — LHS is typed with AbstractSchema
  if not rightNewValue.ocIsKindOf(
    .ejb3qlmm :: stmts :: RHSUpdateItemEntity)
  then false
  else let
    t1 : AbstractSchema = left.ocAsType(
      .ejb3qlmm :: schema :: SingleValuedAssocField).type(),
    — RHS is either RHSNull, IdVarDecl, or EntityInputParam
    t2 : AbstractSchema = rightNewValue.ocAsType(
      .ejb3qlmm :: stmts :: RHSUpdateItemEntity).type()
    in t1 = t2 — TODO spec incomplete.
      — What about inheritance?
  endif
endif

```

Notice that the WFR discussion so far lies still within the realm of language structure, not operation. We don't claim that behavioral semantics should be specified in OCL. However, sooner or later, such additional information is needed for answering some decision problems. For example, without knowledge of the prescribed evaluation order of the LHSs in an UPDATE statement, what can be said about the following statement? Does it exchange the references being held in fields `workAddress` and `homeAddress` or not?

```
UPDATE Employee
SET workAddress = homeAddress,
    homeAddress = workAddress
```

The metamodeling approach allows expressing “details” which are taken for granted as unstated assumptions in most language specs. Continuing with the example of UpdateItem, it can be made explicit that the fields being assigned are actually visible (declared or inherited) at the type of the entity being updated:

```
context UpdateItem
  inv LHSVisibility :
    self.updateStmt.fromClause.type().isVisible(self.left)
```

Making explicit these assumptions is a precondition for applying formal approaches to reasoning about software artifacts.

4 Selected examples of additional corner cases

4.1 Visibility of declarations

Just like in SQL, queries and subqueries may declare one or more identification variables in a FROM clause. The SELECT, WHERE, GROUP BY, and HAVING clauses may then refer to these variables. In case subqueries are present, the spec is not clear about how to interpret a nested variable declaration with the same name as a declaration in the outer scope. Is it disallowed or does it hide the outer declaration? For example:

```
SELECT c
FROM Customer c
WHERE c.balanceOwed
      < ( SELECT avg(c.balanceOwed)
          FROM Customer c )
```

Scopes for identification variables are not defined as such in Ch. 4 of the spec.: “An identification variable always designates a reference to a single value. It is declared in one of three ways: in a range variable declaration, in a join clause, or in a collection member declaration. The identification variable declarations are evaluated from left to right in the FROM clause, and an identification variable declaration can use the result of a preceding identification variable declaration of the query string.” (Sec. 4.4.2). However, Sec. 4.6.2 implicitly introduces the notion of a visibility scope for identification variables: “All identification variables used in the WHERE or HAVING clause of a SELECT or DELETE statement must be declared in the FROM clause, as described in Section 4.4.2. The identification variables used in the WHERE clause of an UPDATE statement must be declared in the UPDATE clause.”

Our interpretation of the scope rules can be summarized as: A FROM clause (and other constructs) introduces a new scope for identification variables. Scopes may be nested forming a tree hierarchy, with (new) variables declared in an

inner scope hiding those with the same name in surrounding scopes. To confirm whether ORM (Object-Relational Mapping) engines conforming to the JSR-220 spec follow this interpretation, EJB3QL queries involving variable hiding were translated to SQL with two different engines. The resulting SQL exhibits variable hiding by explicitly renaming the declaration and usages of the inner variables.

In terms of our metamodel, we check in each query (including subqueries) whether all usages of variables refer to variables which are visible:

Listing 1.2. Declarations-before-usages for a SelectStmt

```

context SelectStmt
  inv WFR_4_6_2_A :
    ( not self.whereClause->isEmpty()
      implies
        self.whereClause.areAllReferredVarsVisible (
          self.locallyDeclaredIdVars() )
    ) and (
      not self.havingClause->isEmpty()
      implies
        self.havingClause.areAllReferredVarsVisible (
          self.locallyDeclaredIdVars() )
    )

```

The argument received by function `areAllReferredVarsVisible()` is a set containing the declarations of visible variables. The recursive nature of the check performed by `areAllReferredVarsVisible()` can be seen at work for a subquery. The overriding OCL definition is shown in Listing 1.3. Before checking whether its WHERE and HAVING clauses (if any) fulfill the declares-before-usages constraint, the scope is augmented with the locally declared variables by using the OCL `union()` operator:

Listing 1.3. Declarations-before-usages for a subquery

```

context Subquery :: areAllReferredVarsVisible ( varsInScope :
  Set(ejb3qlmm :: idVarDecl :: IdVarDecl) ) : Boolean
  body :
    ( not self.whereClause->isEmpty()
      implies
        self.whereClause.areAllReferredVarsVisible (
          varsInScope->union( self.locallyDeclaredIdVars() )
        )
    ) and (
      not self.havingClause->isEmpty()
      implies
        self.havingClause.areAllReferredVarsVisible (
          varsInScope->union( self.locallyDeclaredIdVars() )
        )
    )

```

As in other situations, a visitor could have been written to procedurally validate scope visibility. Again, arguments related to the “productivity” and “correctness” categories introduced in Sec. 1 can be made in favor of the declarative approach.

4.2 GROUP BY

GROUP BY is a rich source of WFRs that amount to exceptions to otherwise valid queries. For example, groups can be formed based on the values of an entity-typed column, with an exception: “*Grouping by an entity is permitted. In this case, the entity must contain no serialized state fields or lob-valued state fields.*” (Sec. 4.7 of the spec):

Listing 1.4. Constraint on entities used as grouping criteria

```

context SelectStmt
inv WFR_4_7_B:
  let entitiesUsedAsGroupBy :
    Set(ejb3qlmm :: schema :: AbstractSchema)
  = groupbyClause->iterate(
    gbi : ejb3qlmm :: selectStmt :: GroupByItem ;
    r : Set(ejb3qlmm :: schema :: AbstractSchema) = Set{ } |
    if gbi.oclIsKindOf(ejb3qlmm :: idVarDecl :: IdVarDecl)
      then r->including( gbi.oclAsType(
        ejb3qlmm :: idVarDecl :: IdVarDecl ). type () )
      else r
    endif )
  in entitiesUsedAsGroupBy->forAll(
    as : ejb3qlmm :: schema :: AbstractSchema |
    as.lobFields->isEmpty() )

```

Some constraints stated in the spec are vague. For example, if a GROUP BY clause is used to reduce a dataset into groups, a boolean expression may be given in the HAVING clause to leave out some of the reduced groups. Such expression may refer only to the groups already reduced, not to their base data. The phrasing in the spec does not make it clear: “*The HAVING clause must specify search conditions over the grouping items or aggregate functions that apply to grouping items.*” (from Sec. 4.7 of the spec). Our reading is that a HAVING clause may contain usages of the following:

1. items which are grouped in the GROUP BY clause, as well as
2. aggregate functions on non-grouped items. Given that applying an aggregate function to a grouped item consisting of just one value always returns that single value, we rule out this possibility

The set of item 1 consists of instances of any class implementing the interface GroupByItem. All usages in the HAVING clause not belonging to that set must be arguments to an aggregate function (i.e. must be an instance of a subclass of AggregateExp). An OCL function returning all data-access expressions being used in a CondExp returns the set for checking the condition in item 2. Such function is reused in the formulation of other WFRs.

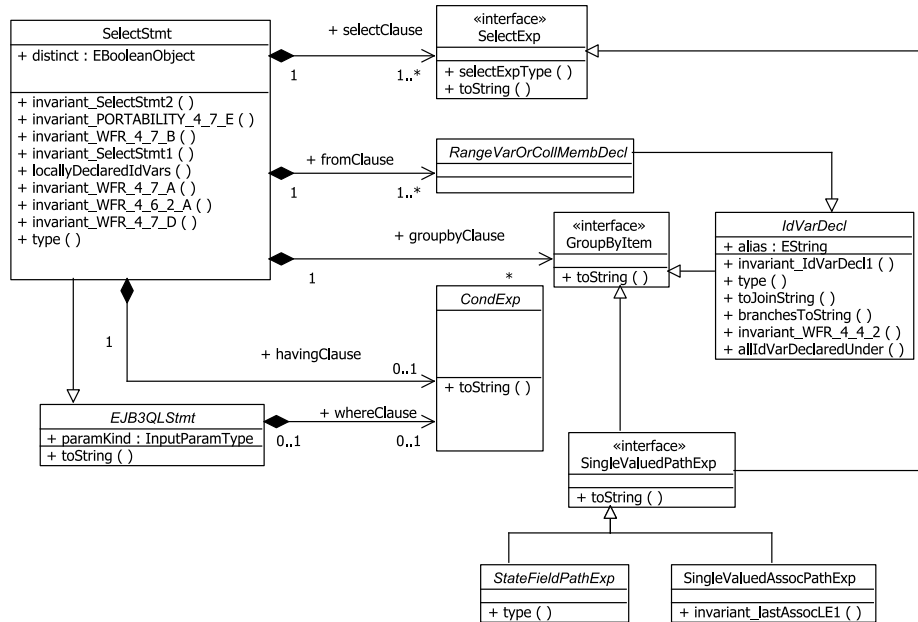


Fig. 3. SELECT statement, including GROUP BY and HAVING clauses

5 Integrating the artifacts generated from the language metamodel in a software project

After the EJB3QL metamodel passed the validation checks enforced by Octopus [6], artifacts were generated for use as building blocks in a larger toolset. These artifacts are available for download from [24]:

1. An in-house developed extension to Octopus was used to generate an AST library in Java. Our extension generates Java 5 code (using generics, enums) by building upon Octopus' approach to compiling OCL expressions into Java statements instead of leaving them as strings for interpretation at runtime as done by EMF OCL.
2. In another project, an Eclipse plugin was developed to translate a valid UML+OCL specification from Octopus into its EMF counterpart, by generating a human-readable Emfatic document [9] with annotations containing OCL expressions for interpretation at runtime [10]. In anticipation of this step, our metamodel was prepared using only those constructs of UML 1.4 amenable for translation into EMF.
3. The evaluation of OCL invariants takes place for individual ASTs which are instantiated either programmatically or by means of a GUI. The GUI was generated by EMF and consists of a tree editor with property sheets. A console allows typing ad-hoc OCL queries for direct evaluation on selected

An interesting issue around translating OCL queries into their EJB3QL realization is the adjustment to a similar but not identical type system (that for EJB3QL is depicted in Figure 4). This transformation surfaces in the context of the refinement of a Platform Independent Model (PIM) to a Platform Specific Model (PSM), as advocated by model-driven development. In our case, the PIM is expressed in UML+OCL and the PSM in terms of Java Enterprise Edition, including EJB3QL.

One such adjustment has to do with the data modeling capabilities of PIM and PSM. For example, a UML+OCL (platform-independent) model abstracts the realization mechanism for `{ordered}` associations ends. A particular PIM to PSM refinement will choose one of several mapping patterns to realize the `{ordered}` feature (involving at least an additional column, possibly additional tables). As a result, the automatic translation to EJB3QL of OCL queries relying on ordered collections must take the chosen pattern into account, making explicit use of it in the platform-specific representation. Patterns for mapping OCL constructs to SQL'92 (with stored procedures) have been reported in [21].

A recurring problem in ORM (Object-Relational-Mapping) is the propagation of changes from a logical-level UML-based model to physical database schemas, with the associated impact analysis of those changes (e.g. determining which queries become invalid). This scenario makes unavoidable the processing of EJB3QL ASTs, as it implies cross-artifact consistency checks between the UML class model and its mapping to an instantiation of 4.

7 Conclusions and Future Work

Improving the quality of enterprise-class software systems requires at some point advanced decision procedures, which in turn build upon precise language definitions. Reverse engineering and other activities in the software development process can cope with the increasing complexity of software architectures only by adopting precise language definitions as a foundation, while simultaneously addressing “productivity” and “correctness” as defined in Sec. 1. Our case study shows by construction how to achieve better language definitions by applying metamodeling techniques to a language used in building enterprise-class systems.

It is difficult to conceive algorithms to process EJB3QL in the current state of affairs, where several important notions of the language are not specified declaratively. For example, the type of the resultset of a SELECT statement can be determined statically (JSR-220 explains informally how). An OCL function encapsulating that algorithm can be used in deciding whether a dynamically built query will be rejected by the ORM engine. Such checks can offload the ORM engine, by applying them before queries are shipped for processing, or by performing them at compile time.

Language-processing algorithms can rely on tree walkers and visitor skeletons generated from language metamodels. For EJB3QL, an obvious example is a visitor for translating to SQL'92. More sophisticated visitors can also be implemented once the infrastructure reported in this paper is in place: predict-

ing execution time, or displaying the access paths of a query (to visualize the depends-on relationships between materialized views). A metamodel for the syntax of SQL 99/2003 (without well-formedness as of now) is available for EMF [15], thus allowing AST-to-AST translation from EJB3QL to SQL.

As we have seen, different levels of formality are sufficient for different purposes. We plan to leverage the OCL formulation of WFRs by translating them into a logical formalism for which a model checker (TLA⁺ [16], ⁺CAL [17]) or theorem prover (HOL [18]) is available.

⁺CAL [17] is an imperative language meant to replace pseudo-code for writing high-level descriptions of algorithms, for which a translator to TLA⁺ [16] is available. Once the data structures and typing conditions specified in a language metamodel are expressed in ⁺CAL, assertions made for the algorithms can be model-checked. As mentioned in the ⁺CAL and TLA⁺ literature, the expressive power of their underlying formalism precludes decidability (i.e. not all valid theorems can be proved by a tool), but experience has shown that the tools can cope with most specifications that engineers write. Another high-level frontend to a model checker is JML [20], for algorithms expressed in Java.

As for HOL (Higher-Order Logic), formalizations of a Hoare-logic for simple imperative languages are available [18, 19]. Once a language-processing algorithm has been formulated in a procedural language amenable for Hoare-analysis, those verification conditions that cannot be automatically derived by the tool have to be specified manually. Proof tactics are then to be applied (automatically or assisted by the user) to discharge the verification conditions and therefore the pre- and postconditions for the algorithm as a whole.

An area we plan to explore is the suitability of an UML+OCL metamodel as a “formalism-independent” way to jumpstart a formal language specification. Once translated into the formalism of choice, we expect to give additional detail (e.g. behavioral semantics, security, quality-of-service) to be used in the formal verification of properties of interest.

Acknowledgement. Liu Yao proficiently contributed to the implementation of this case study as part of his master thesis on Eclipse-based support for Domain-Specific Languages.

References

1. Huang, S. S., Zook, D., Smaragdakis, Y.: Statically Safe Program Generation with SafeGen, In R. Glück and M. Lowry, editors, *4th Intl. Conf. on Generative Programming and Component Engineering (GPCE'05)*, Tallin, Estonia. September 2005. LNCS, vol. 3676, pp. 309-326. Springer-Verlag, 2005.
2. Wang, J., Kim, S-K., Carrington, D.: Verifying Metamodel Coverage of Model Transformations. *aswec*, pp. 270-282, Australian Software Engineering Conference (ASWEC'06), Sydney, Australia. April 2006.
3. Jouault, F., Bézivin, J., and Kurtev, I.: TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. To appear in *Proc. of the 5th Intl. Conf. on Generative programming and Component Engineering*, Portland, Oregon. October 2006.

4. Muller, P-A., et. al.. Model-Driven Analysis and Synthesis of Concrete Syntax. To appear in *Proc. of the MoDELS/UML 2006*, Genova, Italy. October 2006.
5. Grammatech Synthesizer Generator.
<http://www.grammatech.com/research/RandD-bibliography.html>
6. Klasse Objecten, Octopus: OCL Tool for Precise Uml Specifications.
<http://octopus.sourceforge.net/>
7. Malakanov, M.: Syntax Diagram Generator for ANTLR 2.
<http://www.antlr.org/share/list>.
8. OpenJPA: An ASL-licensed implementation of the Java Persistence API (JPA).
<http://incubator.apache.org/openjpa/>
9. Emfatic Language for EMF, <http://www.alphaworks.ibm.com/tech/emfatic>
10. Damus, C. W.: Implementing Model Integrity in EMF with EMFT OCL. IBM developerWorks, August 2006. <http://www.eclipse.org/articles/Article-EMF-Codegen-with-OCL/article.html>
11. Alanen, M., Porres, I.: A Relation Between Context-Free Grammars and Meta Object Facility Metamodels. Tech. Rep. No. 606, Turku Centre for Computer Science, March 2003.
12. Jin, D., Cordy, J.R., Dean, T. R.: Where's the Schema? A Taxonomy of Patterns for Software Exchange, *10th International Workshop on Program Comprehension (IWPC'02)*, pp. 65-74, 2002.
13. Antoniol, G. Di Penta, M. Merlo, E.: YAAB (Yet another AST browser): using OCL to navigate ASTs. *11th IEEE International Workshop on Program Comprehension (IWPC '03)*, pp. 13- 22. Washington, DC, USA. May 2003.
14. De Volder, K.: JQuery: A Generic Code Browser with a Declarative Configuration Language, *8th Intl. Symposium Symposium on Practical Aspects of Declarative Languages (PADL 2006)* LNCS Vol. 3819/2005, pp. 88-102.
15. SQL 99/2003 Metamodel. http://www.eclipse.org/datatools/project_modelbase/
16. Specifying Systems: The TLA⁺ Language and Tools for Hardware and Software Engineers. Leslie Lamport, Addison-Wesley (2002). ISBN 032114306X.
17. Lamport, L.: The ⁺CAL Algorithm Language. 2006. Submitted for publication, <http://research.microsoft.com/users/lamport/pubs/pluscal.pdf>
18. Gordon, M. J. C.: Mechanizing Programming Logics in Higher Order Logic. Tech. Rep. 145, September 1988. University of Cambridge Computer Laboratory.
19. Brucker, A., Wolff, B.: A Package for Extensible Object-Oriented Data Models with an Application to IMP++. *Intl. Workshop on Software Verification and Validation (SVV 2006)*. Computing Research Repository (CoRR), pp. 73-87. Seattle, USA. August 2006.
20. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G. T., Rustan, K., Leino, M., Poll, E.: An overview of JML tools and applications. *Intl. J. on Software Tools for Technology Transfer*, Vol. 7, No. 3, pp. 212-232, June 2005.
21. Demuth, B., Hussmann, H.: Using OCL Constraints for Relational Database Design. *Proc. 2nd International Conference UML'99*, Springer LNCS 1723, pp. 598-613, 1999.
22. Klint, P., Lämmel, R., and Verhoef. C., Towards an Engineering Discipline for Grammarware, *ACM Transactions on Software Engineering and Methodology*, Vol. 14, No. 3, July 2005, pp. 331-380.
23. JJTree, Tree builder generator. <https://javacc.dev.java.net/doc/JJTree.html>
24. EJB3QL Metamodel and accompanying software artifacts.
<http://www.sts.tu-harburg.de/~mi.garcia/pubs/atem06>