OCL Compiler for EMF

Miguel Garcia A. Jibran Shidqie

Institute for Software Systems (STS)
Hamburg University of Technology (TUHH), 21073 Hamburg, Germany
http://www.sts.tu-harburg.de/~mi.garcia

Abstract: The Eclipse infrastructure for modeling is based on EMF, with support for OCL 2.0 provided by the Model Development Tools (MDT) project. The combined expressive power of Ecore + OCL allows capturing a sizable amount of development requirements in a declarative manner: (a) data modeling aspects can be expressed as an Ecore-based schema further constrained by OCL invariants; (b) a number of functional requirements can be specified as operation pre- and postconditions, together with side-effects-free queries. The OCL compiler reported in this paper extends the code generation process of EMF by performing a translation from OCL types and expressions into Java types and statements, increasing productivity and quality measures of a Model-Driven Software Development (MDSD) process. The generated code can be directly used as the Model component in an MVC architecture (for example, as part of a graphical editor generated with Eclipse GMF).

1 Introduction

The operation of a compiler [2] can be broken down into phases: (1) lexical analysis; (2) parsing a stream of tokens into a language-independent Concrete Syntax Tree (CST); (3) transforming such CST into a language-specific Abstract Syntax Tree (AST). During this phase, usages are resolved to their declarations, and symbol tables are built for later use in succeeding phases, as for example during (4) semantic analysis, where the static semantics (also called Well-Formedness Rules, WFRs) are checked, i.e. the conditions that a program should satisfy beyond those captured by the grammar alone. For example, a program may be syntactically valid yet not pass type checking, with type checking being a case in point of well-formedness. For input that has progressed this far, the remaining phases can generate executable code: (5) translation to intermediate code, (6) detection of unreachable blocks, (7) optimization (e.g. constant propagation) based on control and dataflow analyses, (8) detailed decisions of instruction selection and register allocation.

In our setting, things are a bit different. The output language is a high-level language (Java 5), moreover constrained to a number of code idioms. The particular patterns to generate partly depend on user preferences. For example, either (a) POJO-style (Plain Old Java Objects) or (b) EMF-enabled style can be chosen. The latter is suited whenever EMF services will be accessed at runtime (reflection, dynamic object model, interaction with Eclipse editors). As an example of (b), method signatures generated for OCL invariants follow the contract expected by the EMF Validator Framework. Similarly, there are code idioms resulting from OCL itself (most notably, the implicit *source* argument resulting from OCL's compact syntax).

Regarding the input languages, the basic activities of parsing, AST building, and well-formedness checking for Ecore + OCL are carried out by reusing building blocks provided by the Eclipse Modeling infrastructure [7]. EMF support for parametric polymorphism ("Generics" [8]) is in line with the strongly-typed nature of OCL and makes for a seamless transition from OCL collection types to those from the `java.util` package.

The structure of this paper is as follows. Sec. 2 provides details about the core of our contribution, i.e. the compilation algorithm as a whole, which comprises a conversion from OCL types to Java counterparts (Sec. 2.2) and the translation of OCL constructs proper (Sec. 2.3). Implementation-specific aspects have been gathered for the interested reader in Sec. 2.4. The techniques for OCL translation presented here can be applied to a variety of target languages (e.g., database query languages). Sec. 3 discusses some of these possibilities. The integration of compiled code with other artifacts produced in an MDSD toolchain is the focus of Sec. 4, as well as performance evaluation. On average,

generated code exhibits a 2x speedup over its interpreted counterpart. Sec. 5 discusses related work and offers our conclusions. Familiarity with OCL syntax is necessary to follow the discussion in Sec. 2. The source code of the compiler is available under the Eclipse Public License (EPL) v1.0 and can be downloaded from `CVS-repository-TODO`.

2 Compilation Phases

2.1 Information initially available to the compiler

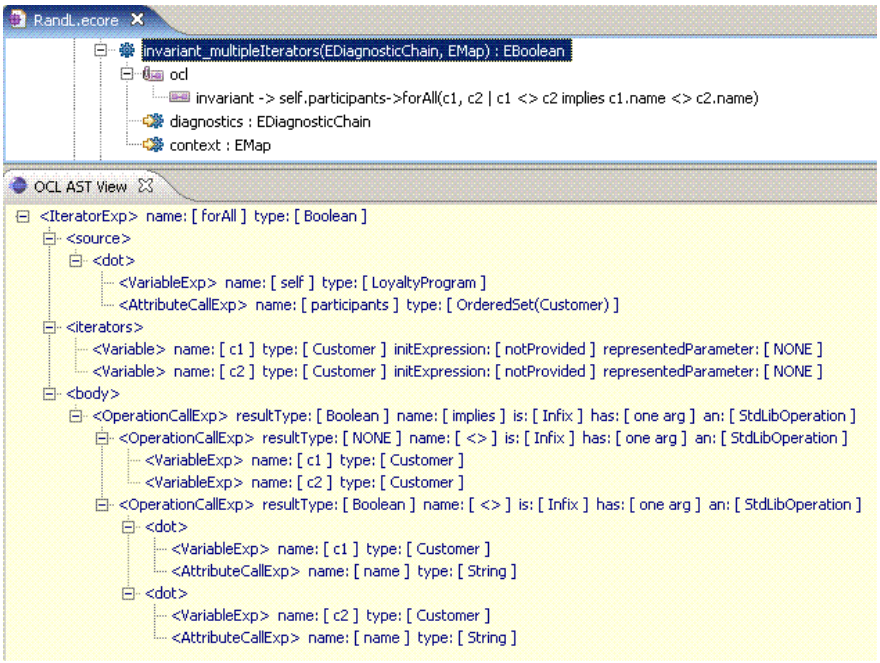


Figure 1: Visual depiction using OCLASTView [7] of the AST for the invariant `self.participants->forAll(c1,c2 | c1<>c2 implies c1.name<>c2.name)`

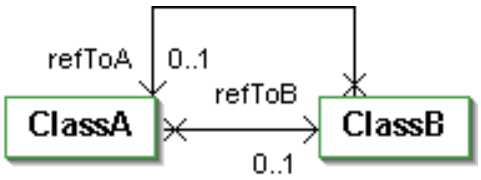
The ASTs prepared by MDT OCL encode not only structural aspects of OCL expressions (operators, operands, precedence) but also reveal the statically computed types for each sub-expression, down to the leaf nodes (literals, read access to variables). Not all of this type information has been explicitly stated by the developer, most of it is inferred from the typing rules of OCL, the types of arguments, and the involved operation. Less frequently, a type declaration itself may be implicit and the resulting type has no user-visible name (this is the case for tuple literals and for implicit iterator variables in loop expressions).

The translation algorithm comprises (a) types conversion and (b) expressions translation. The latter involves a structural mapping that considers one node at a time of the input OCL AST, and produces a Java counterpart. This translation has to abide by the type conversion, i.e. the Java type of an output expression has to conform to that specified by the rules elaborated in Sec. 2.2. Translation (b) requires information available locally at each input node. Such nodes stand for a function application (internal node) or for a read access (leaf node). As explained in detail in Sec. 2.3, the translation of a function application is constructive: provided that the arguments have already been translated, the output for the function application as a whole will be well-formed.

Regarding the possible OCL constructs, Figure 4 depicts the relevant fragment of the OCL metamodel [14], i.e. the classes whose instances are nodes in an AST (for more details, see [7]). For illustration, one such AST is shown in Figure 1, depicting the *static dataflow* of an OCL expression. For simple expressions, such AST depiction has the same

shape as the *dynamic dataflow* (i.e., the tree of call stack activations). For OCL expressions involving recursion or loops, the dynamic dataflow is data-dependent and a visual representation would involve unwinding the call hierarchy for a particular execution trace.

The OCL Standard does not specify the order in which AST building should take place, but it cannot be arbitrary: `def` statements can be used to add: (a) attributes, (b) references and (c) operations to a class model specified in Ecore. In order for other OCL expressions to parse correctly (as they may contain usages of these newly added model elements), the declaration part of all `def` statements is processed first, affecting an in-memory copy of the original input model. Thereafter, ASTs are built for the initialization part of the `def` statements and for the remaining OCL statements. An example of mutual forward references occurring in the initializers of `def` statements is depicted in Figure 2.




```
context A
def dA: sa : String = refToB.sb

context B
def dB: sb : String = refToA.sa
```

Figure 2: Mutual forward references in the initializers of `def` statements

2.2 Types Conversion

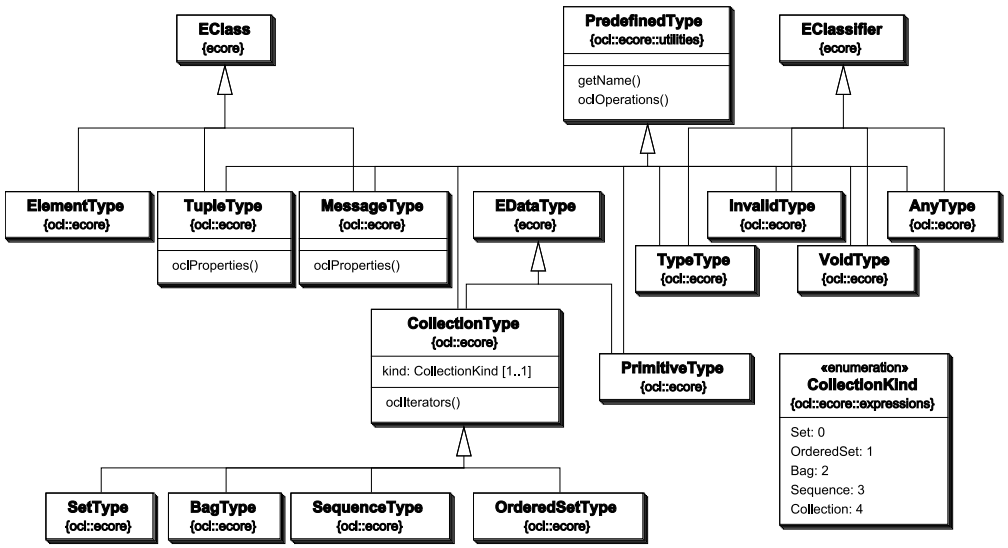


Figure 3: Type information in MDT OCL

The declared types appearing in a particular Ecore + OCL specification are a subset of the *universe of types* resulting from applying OCL type construction operators to the union of the OCL built-in types and those in the user-specified class model. The subtype relationship over the types universe is a partial order. The types conversion implemented by our compiler must map this graph G into an isomorphic graph H (whose nodes represent Java 5 types). This isomorphic mapping is a bijection f between the vertices of G and H such that any two vertices u and v from G are adjacent if and only if $f(u)$ and $f(v)$ are adjacent in H. This ensures that, if two nodes in the target graph are connected, such statement about subtyping is valid under the subtyping relationship of Java 5 (§4.10 in [10]). Type formation in OCL is summarized in Figure 3 and covered in Sec. 8.2 of the OCL 2.0 Standard [14]. In fact, the OCL

→ Java types conversion can be made more concise by translating into Ecore types [8] (which are shorthands for the Java 5 types that will appear in the code generated by the EMF CodeGen component). The algorithm to achieve this conversion appears on Tables 1 and 2, as pairs of $LHS \rightarrow RHS$ transformations from OCL types into Ecore types. This algorithm is applied to instances of Ecore's `ETypedElement` (attributes and references in classes, formal parameters and return type in operations) after ASTs have been built as discussed at the end of Sec. 2.1.

OCL type	Ecore counterpart

Collection(T)
Sequence(T)
Set(T)
OrderedSet(T)
Bag(T)
	(wrapped inside an `EDataType<T>` with `instanceClassName` as below)
	`java.util.Collection<? extends T>`
	`java.util.ArrayList<? extends T>`
	`java.util.HashSet<? extends T>`
	`java.util.LinkedHashSet<? extends T>`
	`org.eclipse.ocl.util.Bag<? extends T>`

Boolean, Integer String, Real	`EBoolean`, `EInt` `EString`, `EDouble`
TupleType	A dedicated `EClass` is added to the in-memory copy of the input model, with structural features standing for the tuple's fields
VoidType	All OCL-defined expressions return some value, including `body` statements defining `EOperation`s. The Java counterpart of VoidType is a `void` return type for a method, but again, such methods cannot be defined with OCL.
MessageType	Not handled by our compiler, MessageTypes denote method invocations, which are not reified in Java.
ElementType	This metaclass appears in the OCL standard just to introduce vocabulary for later use in English sentences. There is no "ElementType" as such, the item type of a collection must be one of the types defined above.

Table 1: Types present in OCL since version 1.0

OCL type	Ecore counterpart
AnyType	`EObject`
InvalidType	As in Java, there is no name in Ecore for the type whose only allowed value is `null`. Whenever an OCL expression would evaluate to InvalidType, the Java counterpart will compute `null`.
TypeType	"TypeType" appears in the OCL spec only in diagrams (in particular, no definition for it is given). Its apparent intent, type reification, is already handled by the Ecore metamodel and the above definitions, which suffice for ASTs involving `oclIsTypeOf()`, `oclIsKindOf()`, and `oclAsType()`.

Table 2: Types added to OCL 2.0

2.3 Expressions Translation

The internal nodes in an OCL AST stand for the application of a function to its arguments. The OCL constructs subclassing `CallExp` receive, besides the argument list, an additional *source expression* as implicit argument. In the example shown in Figure 1, `self.participants->forAll(c1,c2 | c1<>c2 implies c1.name<>c2.name)`, the source expression of the `forAll` is `self.participants`. In general, the Java code generated to compute the function application could assume that the values of arguments are available in local variables. This recursive pattern fits

perfectly the visit order that can be followed by subclassing `org.eclipse.ocl.utilities.AbstractVisitor`. To enforce the pattern, each method in the compilation visitor (one for each OCL construct) should abide by the following contract:

(a) visit the nodes of arguments so that Java statements to compute them are added to a visitor-local running list; and

(b) return the name of the local variable where the result of the expression rooted at the visited node will be available (the upstream node will need this name to complete its own code generation)

While performing (a) for each argument to an OCL function invocation, the name of the local variable holding the argument's value can be obtained: this name was returned as per (b).

Listing 1: Template of the code generated for an `IfExp`

```
/* 'NCS' below stands for the nearest common supertype
for the types of the Then and the Else branches */
NCS if123 = null;
// statements generated by getCondition().accept(this)
// returning the local variable name 'cond456'
if (cond456) {
    // statements generated by getThenExpression().accept(this)
    // returning the local variable name 'then789'
    if123 = then789;
} else {
    // statements generated by getElseExpression().accept(this)
    // returning the local variable name 'else789'
    if123 = else789;
}
```

For example, the code generated for an `if` C `then` E_1 `else` E_2 `endif` appears in Listing 1, making clear that generated local variables will be in scope (and assigned) by the time they are used: the local variable containing the result of C is in-scope and assigned by the time it is referred in the generated `if` statement. As a further example, the visitor method in charge of compiling an OCL `let` statement is shown in Listing 2. A more comprehensive input-output pair (involving iterators and implicit variables) can found in Listing 4.

The compilation algorithm is encapsulated in class `CompilationVisitor`. The implementation of OCL visitors in general is discussed in [7], 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.

Listing 2: Visit order for a `let` expression: initializer, in-part

```
@Override
public String visitLetExp(LetExp<EClassifier, EParameter> letExp) {
    OCLExpression<EClassifier> initExpr = letExp.getVariable().getInitExpression();
    // add the Java stmts for the initializer part of the letExp
    String srcInitVal = initExpr.accept(this);
    String srcJavaType = getSrcType(letExp.getVariable().getType());
    String srcVarName = letExp.getVariable().getName();
    addAssignment(srcJavaType, srcVarName, srcInitVal);
    // add the Java stmts for the in part of the letExp
    String res = letExp.getIn().accept(this);
    return res;
}
```

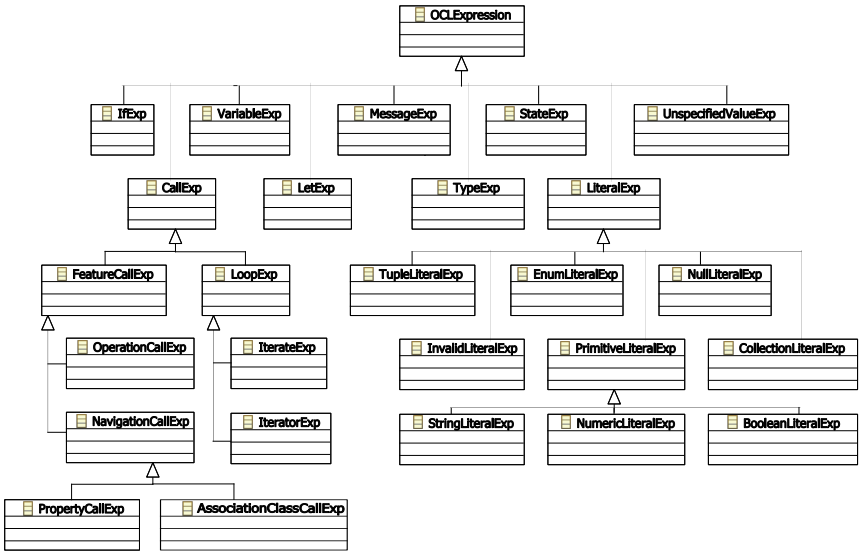


Figure 4: Fragment of the OCL 2.0 metamodel (only inheritance relationships shown)

2.4 Implementation Notes

There are some differences in the metamodel-level representation of OCL types between the OCL Standard and MDT OCL, as is usually the case for paper specs vs. conforming implementations. These differences are mentioned here to save time to compiler extenders (Sec. 3). We have found no inconsistency in the MDT formulation. Besides the obvious names differences (`EClassifier` instead of Classifier), MDT OCL defines `ElementType`, `MessageType`, and `TupleType` as direct subtypes of `EClass` (and not of Classifier resp. DataType as the OCL standard suggests). Figure 3 depicts the situation in MDT OCL (namespace URI `http://www.eclipse.org/ocl/1.1.0/Ecore`).

EMF-enabled code generation (i.e., `suppressEMFTypes == false`) results in `EList` being the supertype of all `ETypedElement`s that are marked `isMany()` (`EList` is responsible for handshaking references in bidirectional associations, among other services). Given that `EList` subtypes `java.util.Collection` (and not the other way around), we need to wrap and unwrap as `EList` at method boundaries, so that the method bodies compiled from OCL conform with the method signatures generated by EMF CodeGen. By "method boundaries" is meant receiving `EList` arguments and returning an OCL-computed collection as an `EList`.

Our current implementation of `allInstances()` relies on AspectJ-based interception of instantiation, as reported in [17]. Given that we control the code generation process, it is not really necessary to resort to AspectJ in order to instrument methods we have generated. A next version will coordinate with EMF CodeGen the generation of such instance-tracking code as part of protected argless constructors, removing thus the AspectJ dependency. The options supported by our compiler are:

(a) suppress EMF types (this controls POJO vs. EMF-enabled style)
(b) suppress generation of interfaces
(c) make helper methods (for pre, post, invariant, init) not part of the business interface
(d) generate code to track `allInstances()`

The `@pre` construct (used in postconditions to obtain the value a `FeatureCallExp` had before the operation was run) is not supported, as it would require keeping `WeakReference` backup copies of potentially large data structures. Preconditions and postconditions are translated as `assert` statements, thus letting the user decide whether to run them in production (`http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html`).

Expressions involving type literals (`oclIsTypeOf()`, `oclIsKindOf()`, `oclAsType()`) might be initially thought to pose a challenge, given that not all type information is kept at runtime by Java (in particular, type arguments are subject to *type erasure*). Improved reflection support will be available in Java 6 (`javax.lang.model.util.Types`), but anyway the only requirement on runtime type reification imposed by OCL is subtype checking and casting, tasks for which the Java 5 facilities are enough (there is no requirement, for example, to obtain at runtime the textual representation of a type in full, which would require querying Ecore-provided type information).

3 Extending the compiler

Early on the decision was made to base our compiler on Ecore instead of UML2. The modeling abstractions supported by Ecore are a subset of those available in UML2. In detail, Ecore does not allow: class-scoped features or operations, association classes, association-end qualifiers (which office as primary keys to identify an item in a collection), and the marking of operations as `isQuery()`. With the expressive power of OCL however, every datamodel that can be expressed in UML2 can be reformulated as a corresponding Ecore + OCL model (for example, a Singleton pattern can be stated with an invariant of the form *`Type`*`->allInstances()->size() = 1`). The OCL infrastructure itself provides uniform support for both UML2 and Ecore, by relying on bounded type parameters, which allow writing algorithms minimally dependent on the types of the input while preserving static type-safety. We thus see no principle obstacle to refactor the compiler to take as input UML2-based models instead of an Ecore-based ones.

We plan to explore query-optimization techniques [17] originating in RDBMSs to avoid nested looping, which holds the prospect of significantly reducing execution time. Regarding database query languages, the translations of OCL into SQL'92 required proprietary extensions (control structures , stored procedures) [4]. This situation must have changed with SQL3 and JPQL (Java Persistence Query Language, [6]) but we are not aware of any fully working OCL compilers targeting those languages, which would be a useful complement to EMF persistence solutions such as Teneo, `http://www.elver.org/`.

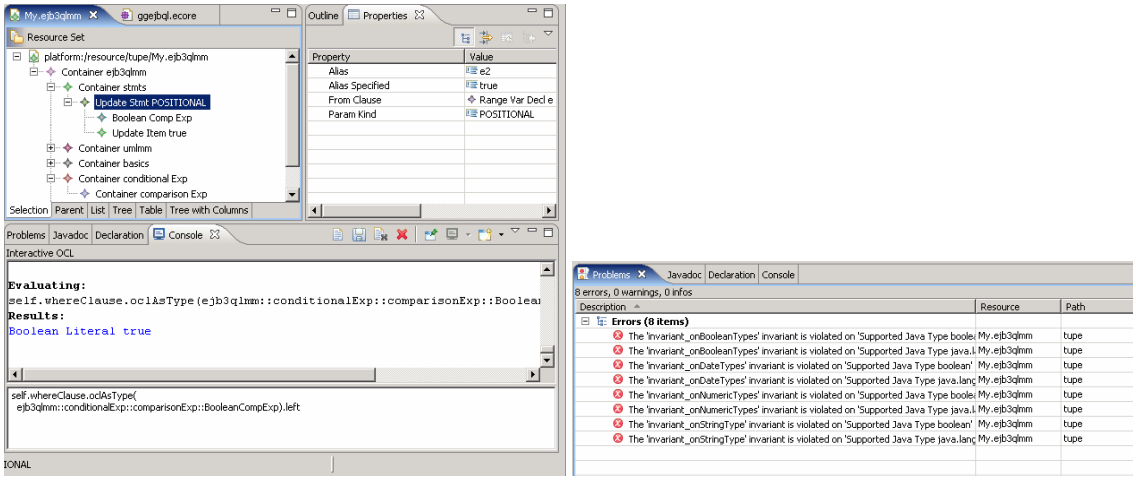
The detection of the minimal subset of invariants that need rechecking (due to updates to the object population, intercepted at runtime) has been addressed a number of times. One technique to achieve detection relies on AspectJ and is described by Dzidek [3]. Altenhofen et. al. also address this problem [1]. Existing approaches re-evaluate from scratch each invariant in the "potentially affected" set, upon detecting an update to any location referred from the AST of the invariant. However, it would be enough to evaluate those nodes upstream of the updated one, propagating values as long as the new value differs from the cached previous value (using memoization, applied to Java in [16]). Recursion and looping result in the dynamic dataflow not matching the shape of the compile-time AST (Sec. 2.1), an issue to consider to avoid false-positives (invariants that actually need no rechecking) as well as, more importantly, mispredictions (overlooking re-evaluating an invariant whose value has actually changed). Checking invariants at transaction boundaries (moreover, in the concurrent case) is addressed for Concurrent Haskell in [11]. To our knowledge, the closest work in Java to achieve *software transaction memory* is [12].

Another area for future work consists in re-architecting our (batch) compiler to support incremental compilation. This would require reacting to the deltas that the workspace notification mechanism provides.

4 Integration in an MDSD Toolchain, Performance Evaluation

Ecore + OCL specs, while declarative, still lack any form of behavioral specification, as is possible with statecharts, or Event-Condition-Action rules. If such behavioral specs were available, fully working components could be generated by a model compiler (as done by Executable UML [15] tools, which usually target the C programming language). Even without behavioral specs the productivity and quality gains are significant: Figure 5(a) depicts a screenshot of an EMF-generated tree editor that allows editing sentences of a custom DSL (Domain Specific Language). Ad-hoc queries and method invocations can be performed through a (generated) OCL Interpreter. Figure 5(b) displays a close-up of the Problems View, whose entries list the OCL invariants currently broken for the object population being edited.

No single Java statement was manually written to realize this editor. In the specific case of DSL editors, projects are underway to generate a text editor supporting usability features such as syntax-directed completion, markers for violations of well-formedness, use-defs navigation, folding, and structural views [5].



(a) Querying an object population (sentences of a custom DSL) (b) Broken well-formedness rules for the DSL sentences

Figure 5: Using the generated code in an EMF-generated editor

Measuring wall-clock time, compiled code runs up to six times faster than its interpreted counterpart (twice as fast on average). In all cases, elapsed times for the interpreter do not include runtime parsing and AST building, as these operations can be amortized among several evaluations. The largest speedups correspond to nested loops, as is the case for the invariant shown in Listing 3 evaluated over 10000 instances (84 sec vs. 580 sec).

Listing 3: Nested loop in an invariant, resulting in a cartesian product

```
context LoyaltyProgram
inv: self.participants->forAll( c1 | self.participants->forAll( c2 |
         c1 <> c2 implies c1.name <> c2.name ))
```

Better speedups could be achieved if our compiler were an *optimizing* compiler [17]. Some compile-time optimizations (e.g. constant propagation) are performed by the JIT (Just-in-Time) compiler of Java anyway. Additional algorithms for OCL rewriting appear in [7] and [9].

5 Related Work and Conclusions

Another OCL → Java compiler has been available for Eclipse since 2005 (`http://octopus.sourceforge.net/`), providing syntax-aware text editors for integrated UML + OCL specs. The main differences with our work are: (a) Octopus adopts a code generation strategy where separate helper methods compute subexpressions, we inline them instead; (b) the notification, serialization, and reflection mechanisms that most EMF-based editors rely on are not present in the POJO-style code generated by Octopus. In particular, (c) ad-hoc OCL queries (i.e., known only at runtime) cannot be evaluated, a task that MDT OCL supports with OCL Interpreter (and associated GUI).

Once OCL specs are edited alongside Ecore-based class models, it is possible for individually correct refactorings at the class model level to invalidate OCL expressions referring to affected model elements. Besides detecting these situations, a tool could attempt to re-establish consistency by refactoring the damaged OCL expressions. RoclET [13] is an Eclipse-based tool for such integrated refactorings, `http://www.roclet.org/`.

The availability of compilers constitutes an acid-test for the specifications of their input languages. The metamodel approach to language specification [6] has proved to be a step forward, provided that the same level of precision attained by previous language definition techniques is followed (i.e., formulation of static semantics as OCL invariants, including typing rules). Several synergy effects are yet to be realized from the unified mechanism for constraining object models that OCL offers (synergies in the fields of program verification, software repositories, and automatic generation of tools, to name a few). Their impact will be amplified by the availability of such capabilities on the Eclipse platform.

References

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

[2] A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2003. http://www.cs.princeton.edu/~appel/modern/java/.

[3] L. C. Briand, W. J. Dzidek, and Y. Labiche. Instrumenting contracts with aspect-oriented programming to increase observability and support debugging. In I. C. Society, editor, *21st IEEE International Conference on Software Maintenance (ICSM), Budapest, Hungary, September 25-30*, pages 687–690. IEEE, 2005. http://www.simula.no/research/engineering/publications/Briand.2005.1/downloadPdfFile.

[4] B. Demuth, H. Hußmann, and S. Loecher. OCL as a Specification Language for Business Rules in Database Applications. In M. Gogolla and C. Kobryn, editors, *UML*, volume 2185 of *LNCS*, pages 104–117. Springer, 2001.

[5] M. Garcia. Generation of DSL Tools based on Language Definitions. Presentation at MDSD Today 2007, http://www.sts.tu-harburg.de/mi.garcia/SoC2007/GenDSLToolsFromLangDef.pdf.

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

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

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

[9] M. Giese and D. Larsson. Simplifying transformations of OCL constraints. In L. C. Briand and C. Williams, editors, *MoDELS*, volume 3713 of *LNCS*, pages 309–323. Springer, 2005.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.

[11] T. Harris and S. P. Jones. Transactional memory with data invariants. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006. (To appear). http://research.microsoft.com/users/simonpj/papers/stm/stm-invariants.pdf.

[12] B. Hindman and D. Grossman. Strong Atomicity for Java Without Virtual-Machine Support. Technical Report 2006-05-01, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, USA, May 2006. http://www.cs.washington.edu/homes/djg/papers/atomjava_tr_may06.pdf.

[13] C. Jeanneret, L. Eyer, S. Marković, and T. Baar. RoclET: Refactoring OCL Expressions by Transformations. In *Software & Systems Engineering and their Applications,19th International Conference, ICSSEA 2006*, 2006. http://infoscience.epfl.ch/getfile.py?recid=90714&mode=best.

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

[15] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie. *Model Driven Architecture with Executable UML*. Cambridge University Press, Cambridge, UK, 2004.

[16] A. Shankar and R. Bodík. DITTO: Automatic incrementalization of data structure invariant checks (in Java). In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 310–319, New York, NY, USA, 2007. ACM Press. http://www.cs.berkeley.edu/~aj/cs/ditto/.

[17] D. Willis, D. J. Pearce, and J. Noble. Efficient Object Querying for Java. In D. Thomas, editor, *ECOOP*, volume 4067 of *LNCS*, pages 28–49. Springer, 2006. http://www.mcs.vuw.ac.nz/~djp/files/WPN_ECOOP06.ps.

Listing 4: The noAccounts invariant translated into Java

```
public boolean invariant_noAccounts(DiagnosticChain diagnostics,
    Map<Object, Object> context) {
/*
context LoyaltyProgram
inv invariant_noAccounts :
  --  when the LoyaltyProgram does not offer the possibility to earn
  -- or burn points, the program members do not have LoyaltyAccounts
  partners.deliveredServices->forAll(pointsEarned = 0 and pointsBurned = 0)
  implies memberships.account->isEmpty()
*/
  org.eclipse.ocl.util.Bag<RandL.Service> collect1 =
    org.eclipse.ocl.util.CollectionUtil.createNewBag();
  for (RandL.ProgramPartner i_ProgramPartner :
    org.eclipse.ocl.util.CollectionUtil.asSet(this.getPartners())) {
    collect1.addAll(org.eclipse.ocl.util.CollectionUtil.asSet(
      i_ProgramPartner.getDeliveredServices()));
  }
  Boolean forAll2 = true;
  for (RandL.Service i_Service : collect1) {
    if (forAll2) {
      Boolean equal3 = Boolean.valueOf(i_Service.getPointsEarned() == 0);
      Boolean and4 = equal3;
      if (and4) {
        Boolean equal5 = Boolean.valueOf(i_Service.getPointsBurned() == 0);
        and4 = equal5;
      }
      forAll2 = and4;
    }
  }
  Boolean implies6 = forAll2;
  if (!(implies6)) {
    implies6 = Boolean.TRUE;
  } else {
    java.util.List<RandL.LoyaltyAccount> collect7 =
      org.eclipse.ocl.util.CollectionUtil.createNewSequence();
    for (RandL.Membership i_Membership :
      org.eclipse.ocl.util.CollectionUtil.asOrderedSet(
        this.getMemberships())) {
      collect7.add(i_Membership.getAccount());
    }
    implies6 = (new Boolean(collect7.isEmpty()));
  }
  if (!(implies6)) {
    if (diagnostics != null) {
      diagnostics.add(new BasicDiagnostic(Diagnostic.ERROR,
      RLValidator.DIAGNOSTIC_SOURCE,
      RLValidator.US_ADDRESS__HAS_US_STATE,
      EcorePlugin.INSTANCE.getString("_UI_GenericInvariant_diagnostic",
        new Object[] { "LoyaltyProgram_noAccounts",
      EObjectValidator.getObjectLabel(this, context) }),
      new Object[] { this }));
    }
    return false;
  }
  return true;
}
```