A Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency

Martin Gogolla, Mirco Kuhlmann, and Fabian Büttner[*]

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany
{gogolla,mk,green}@informatik.uni-bremen.de

Abstract. The Object Constraint Language (OCL) is a central element in modeling and transformation languages like UML, MOF, and QVT. Consequently approaches for MDE (Model-Driven Engineering) depend on OCL. However, OCL is present not only in these areas influenced by the OMG but also in the Eclipse Modeling Framework (EMF). Thus the quality of OCL and its realization in tools seems to be crucial for the success of model-driven development. Surprisingly, up to now a benchmark for OCL to measure quality properties has not been proposed. This paper puts forward in the first part the concepts of a comprehensive OCL benchmark. Our benchmark covers (A) OCL engine accuracy (e.g., for the undefined value and the use of variables), (B) OCL engine determinateness properties (e.g., for the collection operations any and flatten), and (C) OCL engine efficiency (for data type and user-defined operations). In the second part, this paper empirically evaluates the proposed benchmark concepts by examining a number of OCL tools. The paper discusses several differences in handling particular OCL language features and underspecifications in the OCL standard.

1 Introduction

The Object Constraint Language (OCL) [1,2] is a central ingredient in modeling and transformation languages. Thus, approaches for Model-Driven Engineering (MDE) rely on it. OCL is supported in commercial tools like MagicDraw, Together, or XMF Mosaic and in open source tools like ATL [3] or Eclipse MDT OCL [4]. In many approaches, OCL is used as an assembler-like technology underlying model-centric software development. Of course, OCL has a higher degree of abstraction than conventional assemblers, but transformation technology is based on OCL like classical programming languages rely on assemblers.

The Object Constraint Language is employed for determining model properties, for checking the applicability of transformations, and in form of imperative OCL for performing transformations. For example, the QVT standard includes an important part on imperative OCL. Transformation approaches assume that

[*] In cooperation with Benjamin Büttelmann, Lars Hamann, Friederike Jolk, Bin Sun, Hui Wang, Lei Xia.

the integrated OCL engine works correctly. The OCL core is also employed in language extensions like temporal OCL [5] or real-time OCL [6]. A correct and complete realization of OCL is essential for each single tool and indispensable in tool chains. Although the OCL standard offers two approaches for defining the semantics, the quality and conformance of concrete OCL implementations have to be guaranteed. Our experience proves that already basic OCL expressions are treated differently in different OCL engines.

In the first part of this paper we introduce the concepts of a comprehensive OCL benchmark. The benchmark is divided into several parts which treat accuracy, determinateness, and efficiency aspects. Our benchmark covers relevant features of the underlying modeling language and most features of OCL. It currently includes about 950 OCL expressions handling invariants and operation definitions as well as pre- and postconditions.

In the second part of this paper we will apply the benchmark to a number of OCL engines: ATL OCL [3], Dresden OCL [7], Eclipse MDT OCL [4], OCLE [8], Octopus [9], RoclET [10], and USE [11]. Further OCL engines like Kermeta OCL [12], KMF [13], OSLO [14], VMTS OCL [15] and other tools would have been possible candidates as well. The evaluation results are presented in anonymous form, because our aim is to show the applicability and validity of the benchmark concepts. We do not want to recommend or to discourage the use of a particular tool, but would like to emphasize the need for a benchmark which can help to build correct OCL implementations.

As indicated in Fig. 1, our OCL benchmark consists of 7 parts: The parts B1 to B5 treat accuracy, the part B6 deals with determinateness, and the part B7 handles efficiency. The parts B1, B2, and B3 include a UML model in order to check class and object diagram capabilities, invariants, pre- and postconditions, and state-dependent queries. B1 presents core features by checking invariants, B2 adds enumerations and pre- and postconditions, and B3 deals with advanced features like ternary associations and navigation therein. The parts B4 and B5 are based on state-independent queries covering the majority of OCL standard collection operations and their properties. B4 concentrates on the three-valued OCL logic, and B5 features laws between collections operations.

To give an impression of the different evaluations in the examined OCL engines, let us take some examples from the details discussed further down and consider the term `Set{1,2,3}->collect(i|Seq{i,i*i})` (`Sequence` abbreviated by `Seq`). We obtained three different answers from three OCL engines:

Accuracy	B1	Core (data types, invariants, properties, binary associations)
	B2	Extended core (enumerations, pre- and postconditions, queries)
	B3	Advanced modeling (ternary associations, association classes)
	B4	Three-valued logic (e.g., `1/0=1 or true`)
	B5	OCL laws (e.g., `select` versus `reject`)
Determinateness	B6	OCL features with non-deterministic flair (e.g., `any, flatten`)
Efficiency	B7	Evaluation for data type, user-defined and collection operations

Fig. 1. Overview on the 7 Parts of the OCL Benchmark

(A) `Bag{Seq{1,1}, Seq{2,4}, Seq{3,9}}`, (B) `Seq{Seq{2,4}, Seq{1,1}, Seq{3,9}}` and (C) `Bag{1,1,2,4,3,9}`. The answer (A) is the only correct representation of the expected result. Among the remaining three OCL engines, one engine could not handle two or more variables in iterate expressions, another engine did not treat nested variables with identical names correctly, and, for a given `SET` and an appropriate expression `EXPR`, the last engine calculated `SET=EXPR` correctly as `true`, but evaluated `SET->one(e|e)` to `true` and `EXPR->one(e|e)` to `false`.

The structure of the rest of the paper is as follows. Sections 2, 3, and 4 handle accuracy, determinateness, and efficiency, respectively. Section 5 presents the details of the empirical evaluation of the OCL engines. In Sect. 6 the paper is finished with a conclusion and future work. The technical details of the benchmark, i.e., all models, constraints, and queries, can be found in [16], and all details of the evaluation results are presented (partly in German) in [17].

2 OCL Engine Implementation Accuracy

Implementation accuracy is a measurement for the completeness and the correctness of the realization of OCL (and the needed modeling language features) in an OCL engine. Accuracy relates to syntactic and semantic features and is essential, because in tool chains each single tool must rely on the correct and complete OCL handling in the preceding tools. High accuracy is the premise for compatibility of OCL tools. Therefore situations like the following ones should be prevented: (1) The parser of the first tool does not accept the OCL expressions written with the second tool, or (2) the third tool accepts the syntax of the first tool, but shows different evaluation results.

2.1 Accuracy in the Modeling Language and in OCL

OCL constraints and queries refer to a context like a class or an operation. Therefore an OCL engine must provide support for a subset of the underlying modeling language. The most common features are class diagrams and object diagrams for state-dependent evaluation. Our benchmark assumes that central MOF resp. UML class diagram features are supported, e.g., classes, attributes, associations (binary, ternary, reflexive), roles, multiplicities, association classes, and enumerations.

All central OCL features must be available in an OCL engine and are therefore used in our benchmark. Central in this respect are, for example, object properties (attributes and roles), collection operations, and navigation with the collect shortcut. An OCL engine must be able to evaluate state-dependent expressions (e.g., `Person.allInstances()->select(age>18)`) and state-independent expressions (e.g., `Set{1..9}->collect(i|i*i)`). As indicated in the OCL standard, query evaluation by returning a result value and a result type is an important task of an OCL engine.

State-dependent expressions refer to objects, their attributes and roles. Typically these kinds of expressions are used in OCL pre- and postconditions specifying side-effected operations and OCL invariants. Our benchmark covers the mentioned OCL elements. The OCL standards 1.3 and 2.0 show minor differences for certain syntactic constructs. For example, according to OCL 1.3 all instances of a class are retrieved by `allInstances`, but in OCL 2.0 `allInstances()` is used. Our benchmark therefore formulates one constraint in particular syntactic variations in order to check for support of OCL 1.3 and OCL 2.0.

Beside checking for completeness of OCL features, a correct and consistent evaluation of OCL constraints and queries is required. The basis for an accurate evaluation of a complex expression is the correct implementation of every individual OCL operation. Such tests are put into practice by applying OCL collection operations, OCL data type operations and enumeration literals in complex terms. For OCL collection operations, the laws and relationships from [18] are our starting point.

2.2 Core Benchmark (B1)

The core benchmark checks rudimentary OCL and modeling language features. With regard to the modeling language, the applied model includes a class with simple attributes, a side-effect free user-defined operation and a reflexive binary association as shown in Fig. 2. The model is instantiated with an object diagram in order to check the capabilities of object creation, value assignment, handling of String, Integer and Boolean literals as well as link insertion and deletion. The core benchmark avoids special and advanced features like enumerations, empty collections and the undefined value and provides several different syntactic variations for the same expression.

Frequently used OCL operations and constructs are added to the model through the invariants, e.g., basic boolean predicates, the operations `collect` and `flatten`, `let` expressions, nested collections and navigation with the collect shortcut. Using the collect shortcut means to apply a property to a collection of objects which is understood as a shortcut for applying the

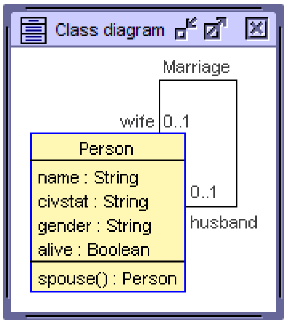

Fig. 2. Class Diagram of the Core Benchmark Model

property inside a `collect` call, e.g., `Person.allInstances().name` is a shortcut for `Person.allInstances->collect(name)`. Because not all considered OCL engines provide support for OCL queries, we restricted the core benchmark to invariants. Therefore, the core benchmark involves so-called query invariants which compare the query with the expected result in order to obtain a boolean expression.

Up to six different syntactic notations are provided for each invariant. Ideally the parser of an OCL engine accepts all variants, but at least one of them has to be accepted. Three choices arise from the naming and typing of variables in collection operations: Iterator variables can be explicitly defined (e.g., `Person.allInstances()->reject(p|p.gender='male')`); additionally they can be typed (e.g., `Person.allInstances()->reject(p:Person|p.gender='male')`), and several operations also accept implicit variables (e.g., `Person.allInstances()->reject(gender='male')`). The number of choices is doubled when we incorporate the notation of `allInstances()` without parentheses as it is permitted in OCL 1.3.

After the syntactic check the evaluation accuracy is identified with the aid of an example object diagram representing a snapshot of a valid system state. All core invariants are designed to be fulfilled in context of this system state.

2.3 Extended Core Benchmark (B2)

While the core benchmark only checks basic model elements, the extended core benchmark adds enumerations, side-effected operations with pre- and postconditions and state-dependent queries. Focus of the queries is object access (including cases treating the undefined value) and navigation as well as handling of enumeration literals and enumeration type attributes as shown in Fig. 3.

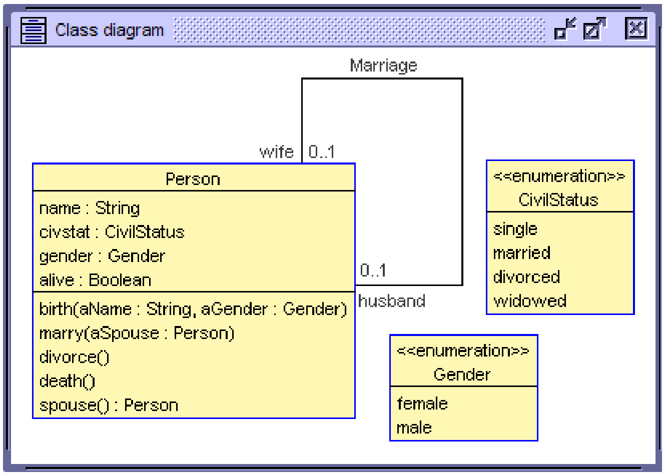

Fig. 3. Class Diagram of the Extended Core Benchmark Model

In this scenario several successive object diagrams are constructed to represent an evolving system. Each pair of successive states represents the execution of an operation specified in the extended model. We do not dictate whether user-defined operations should be directly executable, for example as Java methods, or whether they can be simulated on the modeling level. But in each case we demand the possibility to evaluate pre- and postconditions in context of one pair of system states.

2.4 Advanced Modeling Benchmark (B3)

Navigating ternary and higher-order associations and association classes is an advanced chapter in the OCL standard [1]. Higher-order associations are sometimes needed for concise modeling and are common in database modeling.

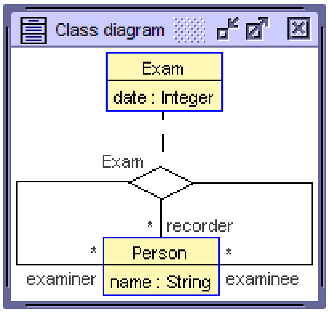

Fig. 4. Class Diagram of the Advanced Model

For this reason, the accuracy benchmark B3 is based on a model specifying a ternary reflexive association class. A link, i.e., an instance of the association class `Exam`, is identified by a triple of persons. Each person is allowed to attend exams in different roles. The following expression navigates within the ternary association.

```
let ada = Person.allInstances()->any(name='Ada') in
  ada.examiner[recorder]
```

The brackets indicate the direction from which an association is navigated. Therefore the above expression results in the set of examiners being present in an exam in which Ada is the recorder. In contrast, `ada.examiner[examinee]` results in all persons being an examiner of the examinee Ada.

2.5 Three-Valued Logic Benchmark (B4)

OCL offers a sophisticated handling of undefined values. This induces a three-valued logic which is tested in the fourth part B4 of the accuracy benchmark. Following the semantics defined in [1], B4 systematically checks the correct implementation of boolean OCL operations with context-free queries.

We emphasize that the OCL standard explicitly requires that, for example, 'True OR-ed with anything is True' and 'False AND-ed with anything is False'. This means that in these cases the undefined value is not allowed to be propagated.

2.6 OCL Laws Benchmark (B5)

Benchmark B4 was set up to check systematically the correct implementation of individual operations, with focus on collection operations. The analysis of semantic properties between OCL operations presented in [18] provides a basis for this benchmark. Each test case checks for the equivalence of two different OCL expressions, i.e., it tests whether the laws between two operations as exposed in [18] holds. If an OCL evaluation engine negates an equivalence, an erroneous implementation of at least one participating operation is indicated. The following example shows a law considered in the benchmark. The variable `e` represents a boolean OCL expression.

```
let c = sourceCollection in
c->exists(i|e) = c->select(i|e)->notEmpty()
```

Another important aspect is the use of the general collection operation `iterate` for substituting other operations. An example is shown below.

```
let c = sourceCollection in
c->exists(i|e) = c->iterate(i;r:Boolean=false|r or e)
```

For checking a law in the benchmark we have to substitute corresponding expressions by concrete source collections (`c`) and OCL subexpressions (`e`). In the case of boolean expressions a very simple form (`i<4`) is sufficient for testing, because we only need an expression which can result to true and to false depending on the value of the iterator variable. The complexity of the expression provoking the boolean value is irrelevant. A correct evaluation of the subexpressions has to be assured by other parts of the benchmark.

In contrast, the source collections have to be systematically chosen, because several inconsistencies do not occur until a particular element constellation is present. On this account each law is instantiated with (1) sets, bags and sequences, (2) empty collections, singleton collections and collections with many elements, (3) collections including the undefined value and excluding the undefined value as well as (4) collections including elements which fulfill the boolean expression and collections excluding these elements. In case of bags and sequences we additionally differentiate between (5) collections excluding equal elements and collections including equal elements which (6) fulfill or not fulfill the boolean expression. The combination of these six situations results in 29 cases for each equivalence. In some test cases like the one checking the law between the operations `collect` and `iterate` this number varies, because of the absence of a boolean expression, i.e., the cases 4 and 6 are not relevant. An example case is shown below.

```
let c = Set{-1,0,1,2} in
c->collect(i|i*i) =
c->iterate(i;r:Bag(Integer)=Bag{}|r->including(i*i))
```

3 OCL Engine Determinateness Properties (B6)

This part of the benchmark deals with OCL engine implementation properties for non-deterministic OCL features and operations for which the OCL standard allows a choice in the implementation like `any` or `flatten`. The aim of this benchmark is to reduce the freedom for implementation choice as far as possible.

In OCL there are at least five possibilities for converting sets and bags to sequences. Here, we will only discuss the ones for sets because the conversions for bags are analogous to the set conversions. Roughly speaking, sets can be made into sequences by using (1) `asSequence`, (2) `iterate`, (3) `any`, (4) `flatten` or (5) `sortedBy`. In the expressions below, `intSet` is an arbitrary OCL expression with type `Set(Integer)`, e.g., `Set{1..12}`.

```
(1) intSet->asSequence()
(2) intSet->iterate(e:Integer;
      r:Sequence(Integer)=Sequence{}|
      r->including(e))
(3) intSet->iterate(u:Integer;
      r:Tuple(theSet:Set(Integer),theSeq:Sequence(Integer))=
        Tuple{theSet:intSet,theSeq:Sequence{}}|
      let e=r.theSet->any(true) in
        Tuple{theSet:r.theSet->excluding(e),
              theSeq:r.theSeq->including(e)}).theSeq
(4) Sequence{intSet}->flatten()
(5) intSet->collect(e:Integer|Sequence{0,e})->
      sortedBy(s:Sequence(Integer)|s->first())->
        collect(s|s->last())
```

The first possibility is the direct conversion with `asSequence`. The second term uses an `iterate` over the integer set with an element variable and successively builds the sequence by appending the current element. The basic idea behind the third term is to choose an arbitrary element with `any` and to append this element to the result sequence. The fourth term calls `flatten` on a sequence possessing the integer set as its only element. The fifth possibility uses `sortedBy` to give an order to a bag of integer sequences. Each of the five terms represents a particular way to produce a sequence from a set. We are using the notion determinateness in this context because the OCL engine has to determine the order in the sequence. Our benchmark tests whether the orders produced by terms 2 to 5 coincide with the direct order given by `asSequence`. The benchmark part B6 checks also minor other points, for example, whether the following two properties hold which consider operation applications to a given set and its corresponding bag.

```
aSet->any(true) = aSet->asBag()->any(true)
aSet->asSequence() = aSet->asBag()->asSequence()
```

We understand such determinateness properties as points of underspecification in the OCL standard. Our benchmark gives the possibility to reduce this underspecification and with this the amount of freedom for the OCL engine implementor.

4 OCL Engine Efficiency (B7)

In this section we propose OCL expressions checking the evaluation efficiency in an OCL engine. The expressions are assumed to be evaluated in the different engines and the evaluation time has to be recorded. In order to have easily measurable and reliable evaluation times the expressions are usually evaluated in an iterate loop not only once but many times. The expressions in this section are divided on the one hand into expressions concerning the OCL standard data types Boolean, String, Integer and Real and on the other hand into expressions of a small model of towns and roads in between.

The expressions for the data types compute (A) the truth tables of the Boolean connectives available in OCL, (B) the inverse of a longer String value, (C) the prime numbers as Integer values up to a given upper bound, and (D) the square root of a Real number. As an example consider the following OCL expression for the prime numbers up to 2048.

```
Sequence{1..2048}->iterate(i:Integer;
  res:Sequence(Integer)=Sequence{}|
    if m.isPrime(i) then res->including(i) else res endif)
```

The operation `isPrime(i)` is defined in a singleton class MathLib as specified below. The operation is called on the singleton object m of class MathLib.

```
isPrime(arg:Integer):Boolean=
  if arg<=1 then false else
    if arg=2 then true else isPrimeAux(arg,2,arg div 2) endif endif
isPrimeAux(arg:Integer,cur:Integer,top:Integer):Boolean=
  if arg.mod(cur)=0 then false else
    if cur+1<=top then isPrimeAux(arg,cur+1,top) else true endif
  endif -- algorithm inefficiency irrelevant for benchmark
```

The expressions for the example model with towns and roads consider the underlying data structure as a graph with objects (nodes) and links (edges). They compute (A) the transitive closure, i.e., the directly and indirectly reachable nodes of a given node, and (B) the connected components of the graph, i.e., the maximal node sets in which all nodes are connected directly or indirectly. The example model has a single class and a single association as displayed in Fig. 5.

An example state with 42 towns and 42 roads is built up. The underlying graph has 5 connected components with 1, 2, 3, 13 and 23 nodes. In the example state the following OCL expression for the transitive closure is evaluated.

```
Set{1..1024*1024}->iterate(i:Integer;
  res:Bag(Set(Town))=Town.allInstances->collect(t|t.connectPlus())|
  res)
```

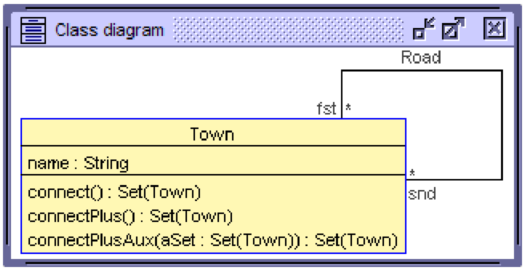
Class diagram
Road
fst
*
*
snd
Town
name : String
connect() : Set(Town)
connectPlus() : Set(Town)
connectPlusAux(aSet : Set(Town)) : Set(Town)

Fig. 5. Class Diagram for Towns and Roads

The operations `connectPlus()` computes all towns directly or indirectly reachable from the current node with the roles fst and snd. All details, i.e., models, invariants, and queries, of this part and the other parts of the OCL benchmark can be found in [16].

5 Empirical Evaluation of the Benchmark

The aim of this empirical evaluation of OCL engines is (A) to show the applicability of the benchmarks concepts developed before and (B) to make a contribution for improvements of current OCL engines. We want to emphasize the applicability of the benchmark and will therefore not go into details concerning the single tools. OCL engine developers can find all details in the cited material and are encouraged to perform our benchmark by themselves.

The OCL benchmark presented in this paper comprises 949 test cases, composed of 71 invariants and 878 query expressions. 121 expressions are state-dependent and 828 context-free. We checked 7 OCL evaluation engines including two code generators. One of the tools was only partly checked because of resource limitations. 401 accuracy and determinateness queries ($46,8\%$) were correctly evaluated by all engines except the aforementioned tool. All evaluation results of performed checks can be found in [17] (partly in German).

5.1 Core Benchmark (B1)

Even though benchmark B1 only includes very basic modeling language and OCL constructs and expressions, it reveals several problems. No evaluation engine accepts all syntactic variations. In general all tools except for one either demand parentheses for the operation `allInstances` or forbid them. Additionally 5 of 6 engines require typing of `let` variables.

Before checking the first OCL invariants one of the tools showed grave restrictions in context of the modeling language features, because no well-formedness rules of the UML metamodel are checked. Thus the tool, for example, does not require unique attribute names within a class definition.

If we disregard the syntactic variations the benchmark B1 checks 18 invariants. Assuming that an invariant is regarded as correctly evaluated when at least one notation is syntactically accepted and the corresponding expression results in true, only one tool evaluates all invariants correctly (18/18). The other tools evaluate from 6 to 16 expressions correctly (6/18, 6/18, 14/18, 14/18, 16/18). Responsible for these results are small discrepancies in the implementations. One parser does not accept range expressions in constructors of collections (e.g., `Set{1..9}`), another parser incorrectly treats string literals, because it handles quotation marks as part of the string. The same tool implements the operation `substring` with character numbers running from `0` to `self.size()-1`, while the OCL standard requires numbers from `1` to `self.size()`. Another noticeable problem is the general handling of iterator variables. Some tools do not permit equal variable names in nested collection operations (e.g., `c->select(p|...any(p|...)...)`). One of them additionally forbids implicit variables in case of nested operations (e.g., `c->select(...any(...)...)`). Even more demonstrative, more than half of the tested OCL engines do not have the ability to handle more than one iterator variable inside the operation `iterate` (e.g., `c->iterate(x,y|...)`) or other operations like `forAll`. On the other hand a tool which allows for more than one iterator variable evaluates the corresponding query incorrectly, because it implicitly flattens nested collections (e.g., `Bag{Sequence{'Ada',18}}` results in `Bag{'Ada',18}`). The latter example shows a sequence with elements having a different basic type. This constellation is however allowed, because both elements have the same super type `OclAny`. Three engines do not define a common super type and throw a type mismatch exception.

5.2 Extended Core (B2) and Advanced Modeling Benchmark (B3)

The extension of benchmark B1 uncovers further restrictions. Some of them are not profound, while others clearly decrease the accuracy of the respective tools. One tool does not provide for query expressions, so they have to be embedded as boolean expressions into invariants (e.g., `Set{1,2,3}->collect(i|i*i)` is transformed to `Set{1,2,3}->collect(i|i*i)=Bag{1,4,9}`). Another tool completely ignores postcondition definitions.

Many test cases directly access the identifiers of objects. Since no tool supports this feature except for one, the expressions can be adapted. The following example shows the adaption of a test case using the object identifier `ada` which represents the Person Ada.

`let o:OclAny=ada in o` is transformed to

```
let ada=Person.allInstances()->any(name='Ada') in
   let o:OclAny=ada in o
```

Although enumerations are lightweight extensions of a UML model, several tools have problems applying enumeration values. Whereas enumeration literals are generally correctly handled, enumeration type attributes eventually prevent a correct evaluation. In one case it is not possible to compare enumeration type attribute values among each other. In another case the comparison of enumeration type attribute values and enumeration literals fails.

A special bug emerges in one tool. Here the essential substitution property for equality is violated. An expression in the form of `SET->one(e|e)` results in true as well as `EXPR=SET`, but the expression `EXPR->one(e|e)` results in false.

Benchmark B3 discovers more obvious limitations and checks advanced modeling and OCL features. 6 of 7 tools do not support ternary associations and 3 tools do not provide for association classes at all.

5.3 Three-Valued Logic Benchmark (B4)

Benchmark B4 emphasizes a fact that already appeared in benchmark B2. Only one of the tested OCL engines sophisticatedly treats the undefined value. All other tools show in different ways a behavior which is not conformant to the OCL standard. One engine sometimes throws an exception if an operation is invoked on an undefined value, but the boolean operations are correctly implemented. Another engine does in some cases explicitly not allow for operation calls which result in an undefined value (e.g., the invocation of the operation `last` on an empty sequence). If, nevertheless, the undefined value occurs in a subexpression the whole expression will be undefined. A third engine does not allow undefined values in collections, i.e., it filters them out. So an expression like `Set{undefinedValue}->includes(undefinedValue)` results in false.

Especially the implementation of the boolean operations is analyzed in benchmark B4. In case of 4 tools we have to differentiate between the left hand side and the right hand side of a boolean operator. If the left hand side already determines the resulting value, the whole expression is correctly evaluated (e.g., `false and undefinedValue` results in false, `true or undefinedValue` results in true and `false implies undefinedValue` results again in true). Otherwise the expression is either not evaluable or results in undefined.

The inconsistent treatment of the undefined value continues in benchmark B5 and B6. Only 3 of 6 OCL engines evaluate all queries correctly in presence of the undefined value, but the other half produces at least partly wrong results. One tool completely refuses the evaluation if one or more undefined elements are included in a source collection. Another tool primarily fails in case of sequences including undefined values. A third tool only implements some operations like `iterate` and `collect` correctly. In contrast, operations like `exists` and `one` need at least one value which fulfills the boolean subexpression (e.g., `Sequence{undefinedValue,1,4}->exists(i|i<4)`). Other operations generate the undefined value in either case.

5.4 Laws (B5) and Determinateness Benchmark (B6)

Benchmark B5 and B6 discover additional problems. One tool, a code generator, does not support tuple types, and implements the `including` (as well as `iterate` and `forAll`) and `implies` erroneously. The former operation is transformed into a Java method which on the one hand declares primitive type parameters, on the other hand requires object type arguments in the methods body. An example extract of a corresponding method is shown below.

```
private Set including(..., boolean b1, boolean b2) {
  ...
  if ( !result.contains(new Boolean((b1.booleanValue() ||
       b2.booleanValue()))) ) { ... } ... }
```

The latter operation and its right hand side is simply unconsidered during the transformation process if the left hand side is not explicitly parenthesized (e.g., `expr1 and expr2 implies expr3` results in `expr1 and expr2`).

One tool does not regard the binding of boolean operations and predicates. They are evaluated from left to right (e.g., in case of (`expr1 implies expr2 and expr3`) the subexpression (`expr1 implies expr2`) is evaluated first). Another tool exhibits a bug which is easily overlooked. Collections used as components of tuples strangely include the `null` value falsifying several evaluation results. Yet another tool generally does not evaluate the operation size invoked on sequences, and additionally shows many unexplainable errors.

6 Conclusion

This paper proposes a comprehensive benchmark for OCL engines. OCL is employed as a basic technology in model-centric development approaches. The quality of an OCL engine is therefore crucial for the success of transformation-driven techniques. We have empirically evaluated our benchmark by considering a number of different OCL engines.

On the one hand, the results have shown incompatibilities following from different interpretations of the OCL standard. On the other hand, the benchmark has discovered faulty implementations of OCL features. The benchmark can help to harmonize the implementation of OCL features in different tools in order to allow for consistent modeling and transformation support. It can be applied as quality measure in OCL engine development.

After having carried out this benchmark, we can state a number of helpful preliminaries for performing OCL benchmarks in the future. An OCL engine should support (A) an import feature for class diagrams including operation definitions, invariants and pre- and postconditions as well as for system states in which evaluations are performed, (B) checking of boolean OCL expressions in the context of a system state, (C) evaluation of OCL expressions in the context of a system state and presentation of results, and (D) composition of the above steps in a single command line script so that comprehensive checks (our benchmark covers about 950 expressions) can be carried out in an automatic way.

Not all OCL engines considered in this paper offer the above functionality: RoclET does only allow to check invariants; ATL OCL concentrates on transformations with OCL conditions to be checked; both engines do not offer the direct evaluation of OCL expressions in a system state. Our goal is that OCL tool builders provide a suitable infrastructure with their tools and self-commit to perform a benchmark like ours on their own. Last, but not least, our benchmark has to be completed because not all relevant modeling language features (e.g., qualifiers) or all OCL features (e.g., ordered sets) are covered yet. Feedback from OCL engine users and developers will give us the possibility to improve and supplement our benchmark.

References

1. O.M.G. (ed.): Object Constraint Language 2.0 (formal/06-05-01). OMG (2006), `http://www.omg.org`
2. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML, 2nd edn. Addison-Wesley, Reading (2003)
3. ATL-Team: ATL Development Tools (2008), `http://www.sciences.univ-nantes.fr/lina/atl/atldemo/adt`
4. MDT-OCL-Team: MDT OCL (2008), `http://www.eclipse.org/modeling/mdt/?project=ocl`
5. Ziemann, P., Gogolla, M.: Extended with Temporal Logic. In: Broy, M., Zamulin, A.V. (eds.) PSI 2003. LNCS, vol. 2890, pp. 351–357. Springer, Heidelberg (2004)
6. Flake, S., Müller, W.: An OCL Extension for Real-Time Constraints. In: Clark, A., Warmer, J. (eds.) Object Modeling with the OCL. LNCS, vol. 2263, pp. 150–171. Springer, Heidelberg (2002)
7. Dresden-OCL-Team: Dresden OCL Toolkit (2008), `http://dresden-ocl.sourceforge.net/`
8. Chiorean, D.: OCLE-Team: Object Constraint Language Environment 2.0 (2008), `http://lci.cs.ubbcluj.ro/ocle/`
9. Kleppe, A., Warmer, J.: Octopus: OCL Tool for Precise UML Specifications (2008), `http://octopus.sourceforge.net/`
10. RoclET-Team: Welcome to RoclET (2008), `http://www.roclet.org/`
11. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming 69, 27–34 (2007)
12. Kermeta-Team: Kermeta: Breathe Life into your Metamodels (2008), `http://www.kermeta.org/`
13. Akehurst, D., Patrascoiu, O.: KMF (Kent Modeling Framework) OCL Library (2008), `http://www.cs.kent.ac.uk/projects/ocl/tools.html`
14. Hein, C., Ritter, T., Wagner, M.: Open Source Library for OCL (OSLO) (2008), `http://oslo-project.berlios.de/`
15. VMTS-Team: Visual Model and Transformation System (VMTS) (2008), `http://vmts.aut.bme.hu/`
16. Gogolla, M., Kuhlmann, M., Büttner, F.: Sources for a Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency, pages 242 (2008), `http://www.db.informatik.uni-bremen.de/publications/Gogolla_2008_BMSOURCES.pdf`
17. Büttelmann, B., Hamann, L., Jolk, F., Sun, B., Wang, H., Xia, L.: Evaluating a Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency, pages 69 (2008), `http://www.db.informatik.uni-bremen.de/publications/Buettelmann_2008_BMEVAL.pdf`
18. Kuhlmann, M., Gogolla, M.: Analyzing Semantic Properties of OCL Operations by Uncovering Interoperational Relationships. In: Electronic Communications of the EASST, UML/MoDELS Workshop on OCL (OCL4ALL 2007) , vol. 9, pages 17 (2008), `http://eceasst.cs.tu-berlin.de/index.php/eceasst`