myjournal manuscript No.
(will be inserted by the editor)

A Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency

Mirco Kuhlmann, Lars Hamann, Martin Gogolla, Fabian Büttner

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

The date of receipt and acceptance will be inserted by the editor

Abstract Since several years, the Object Constraint Language (OCL) is a central component in modeling and transformation languages like the Unified Modeling Language (UML), the Meta Object Facility (MOF), and Query View Transformation (QVT). Consequently approaches for MDE (Model-Driven Engineering) depend on this language. OCL is present not only in 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 handling of the undefined value, the use of variables and the implementation of OCL standard operations), (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 several OCL tools. The paper clarifies a number of differences in handling particular language features and underspecifications in the OCL standard.

1 Introduction

As a central ingredient in modeling and transformation languages the Object Constraint Language (OCL) [1,2] provides a basis for Model-Driven Engineering (MDE). 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 main fields of application of the Object Constraint language are the determination of model properties and checking the applicability of transformations. In form of imperative OCL it is additionally employed for performing transformations. For example, the QVT standard includes an important part on imperative OCL. Transformation approaches assume the integrated OCL engine to work correctly. Furthermore, the OCL core is employed in language extensions like temporal OCL [5] or real-time OCL [6].

A correct and complete realization of OCL is thus essential for each single tool and indispensable in tool chains. The OCL standard offers two approaches for defining the semantics. Nevertheless, the quality and conformance of concrete OCL implementations have to be guaranteed independently. Our experience shows that already some basic OCL expressions are treated differently in different OCL engines.

1.1 The OCL Benchmark

With this paper we introduce the concepts of a comprehensive OCL benchmark. The benchmark was designed to allow OCL engines to be checked for a correct, complete and efficient implementation of the OCL standard definitions. The focus is on the implementation of the different language constructs, including individual OCL operations, OCL constraints and the handling of basic and complex types in OCL. We divided the benchmark into several parts treating accuracy, determinateness, and efficiency aspects. It covers relevant features of the underlying modeling language and the features of OCL. Currently it includes 1413 OCL expressions handling invariants and operation definitions as well as pre- and postconditions.

We have presented a first version of the OCL benchmark in [7]. Since then the benchmark evolved, so that

we present a new version in this paper. Several missing UML and OCL features like qualifiers and ordered sets were added. Other features were tested more extensively. This resulted in one new part and two extended parts of the benchmark.

In the first part of this paper we describe our benchmark in detail. In the second part we will apply the benchmark to a number of OCL engines: ATL OCL [3], Dresden OCL [8], Eclipse MDT OCL [4], OCLE [9], Octopus [10], RoclET [11], and USE [12]. Further OCL engines like Kermeta OCL [13], KMF [14], OSLO [15], VMTS OCL [16] and other tools would have been possible candidates as well. The evaluation results are presented in this paper in anonymous form (but we provide the detailed results for all tools at [17]), because our aim is to show the applicability and validity of the benchmark concepts as well as to point out specific problem areas. 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 eight parts: The parts B1 to B6 treat accuracy, the part B7 deals with determinateness, and the part B8 handles efficiency. The parts B1, B2, B3, and B6 include at least one 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. The part B6 covers distinctive OCL features which were missing in the previous version of the OCL benchmark. The parts B5 and B7 were considerably extended in the new version.

1.2 Exemplary Benchmark Results

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

(A) `Bag{Seq{1,1}, Seq{2,4}, Seq{3,9}}`
(B) `Seq{Seq{2,4}, Seq{1,1}, Seq{3,9}}`
(C) `Bag{1,1,2,4,3,9}`

Regarding the OCL standard, expression (A) is the only valid result. In addition to this difference, one engine could not handle two or more variables in iterate expressions, another engine did not treat nested variables with identical names correctly, and the last engine calculated `SET=EXPR` as true, but evaluated `SET->one(e|e)` to true and `EXPR->one(e|e)` to false (with respect to a given `SET` and an appropriate expression `EXPR`).

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`)
	B6	Distinctive features (ordered sets, qualifiers, `collect`/`collectNested`) `sortedBy`, `oclAsType`, `oclIsTypeOf`, `oclIsKindOf`)
Determinateness	B7	OCL features with non-deterministic flair (e.g., `any`, `flatten`)
Efficiency	B8	Evaluation for data type, user-defined and collection operations

Fig. 1 Overview on the 8 Parts of the OCL Benchmark

1.3 Outline of the Paper

The structure of the rest of the paper is as follows. In Sect. 2 we discuss related work. In Sect. 3 we reflect the applied methodology for designing the benchmark. Sections 4, 5, and 6 handle accuracy, determinateness, and efficiency, respectively. Section 7 presents the details of the empirical evaluation of the OCL engines. Section 8 discusses uncovered OCL features. In Sect. 9 the paper is finished with a conclusion and future work.

The benchmark sources, i.e., all models, constraints, and queries as well as all details of the evaluation results (partly in german) can be found at [17].

2 Related work

Until today, only few works on benchmarks for OCL were published. The work in [18] focuses on efficiency. It motivates the need for an efficiency benchmark for medium to large scenarios. For instance, the usage of OCL for calculating metrics of Java programs requires an efficient evaluation engine. The authors use a sample model with differently sized system states and compare the evaluation performance of several OCL tools.

While there is a lack of work concerning OCL benchmarks, several conformance test suites for other standards exist. Many test suites can be found on the websites of the U.S. National Institute of Standards and Technology (NIST), for example for XML, COBOL85 and Fortran78 [19]. The general structure of such test suites and the common design criteria described by the

NIST in [20] are a good starting point for developing a benchmark for standard compliance. A fundamental part is the so-called conformance test suite. Such a suite "is a collection of combinations of legal and illegal inputs to the implementation being tested, together with a corresponding collection of 'expected results'" [20]. Finding 'expected results' is a well-known issue in the area of test design where it is called the oracle problem [21].

An automated approach for test suite generation is presented in [22] where valid UML models (demonstrations) and invalid models (counterexamples) are automatically generated by the tool JULE. The generated models can be used as test data for software modeling tools. Although the authors use OCL constraints defined on the UML metamodel they do not generate test data for checking OCL standard conformance.

The analysis of XQuery benchmarks in [23] gives an idea about the problems with changing standards and faulty tests. Several benchmarks like XMach-1, XMark and X007 are analyzed and compared. Some of the conclusions made in this paper can be used for a general benchmark design. Especially the fact that nearly all examined benchmarks included many queries which do not provide any extra information to the benchmark results influenced our benchmark.

The 'Theory of Benchmarking' presented in [24] describes the benefits of a benchmark for research communities as well as criteria to establish a useful benchmark. The authors also constitute two preconditions a community should fulfill before establishing a benchmark for the area of interest:

- 'a minimum level of maturity' and
- 'an ethos of collaboration within the community'.

In our view the OCL community fulfills both conditions.

3 Benchmark Design and Coverage

Before we discuss the benchmark parts in detail, it is important to dwell on the benchmark design which is generally based on the work presented in the last section. After the design we address the coverage of the benchmark with respect to the OCL specification.

Sim et. al. [24] define three components of a benchmark:

Motivating Comparison: This component should define the technical comparison to be made and the research agenda that will be furthered.

Task Sample: The task sample is a selection of tasks that a tool or technique is expected to solve.

Performance Measures: These measurements can be quantitative or qualitative and can be made by a human or a computer.

The 'Motivating Comparison' of our benchmark was outlined in Sect. 1. Our goal is to provide a common basis for tool developers and users to make statements about the quality of an OCL engine implementation as well as its performance. In addition the benchmark should point out open issues in the OCL specification.

The quality of an implementation can be measured by counting the successful evaluated OCL expressions defined in our benchmark, as we will describe later. Performance measurement is well-known in benchmarking and can also be easily achieved in general.

The 'Task Sample' is the core problem of a benchmark. One has to find a 'representative sample of tasks that the tool or technique is expected to solve in actual practice'. In addition, 'the challenge is to simplify but not to over-simplify' [25].

Due to the fact that OCL is widely used for different purposes, we decided to design several benchmark parts. The basic parts of our benchmark cover the application of OCL in modeling languages and commonly used language features, e.g., basic boolean predicates and connectives, navigation and frequently applied collection operations. They are furthermore divided into three 'feature parts' (B1-B3) which successively add language concepts. This allows an early application of the benchmark during the development of a tool.

The following parts (B4 and B5) focus on OCL language features which can be considered without an underlying model. They perform checks including the handling of the undefined value and frequently used collection operations. Collections and their operations are indispensable with respect to the main purpose of the Object Constraint Language (constraining and querying system states and scenarios).

The handling of the undefined value is likewise important, because it determines the realization of the three valued logic of OCL. Part B4 covers all manageable combinations of boolean operations with respect to the undefined value. However, other methods for provoking the undefined value could be discussed. Currently we use a general approach without using the literal for the undefined value to allow a wider application of the benchmark.

Regarding B5—which is quantitatively the largest part of the benchmark—we chose a compressed approach to reduce the number of the statements by defining and applying several laws which express relationships between collection operations. The defined test cases continuously handle two different collection operations which should result in the same value and therefore can be compared for equality. One benefit of this approach is the eased presentation of the expected results, because the evaluated test cases just yield true or false. The small set of possible results (true, false, undefined) is also helpful in another way, because we noticed that several OCL tools do not directly support OCL queries to retrieve the usually complex evaluation results in a simple way. Another advantage of the equality checks is the possibility to halve the overall needed test cases. However, in case

of a failed test case two operations need to be separately checked in order to identify the erroneous implementation. But the related effort can be neglected.

The parts B6 and B7 check distinctive OCL features which are especially debatable, because the OCL standard does not make a clear statement concerning several tested features, e.g., the determinateness properties. Hence, it is important to offer a guideline and a base for discussion. According to [24] this reveals also a positive sociological effect of benchmarks: They point out open problems within the respective research area.

The efficiency part B8 is divided into two independent sets of test cases. One set bases on queries to differently sized system states, using the collection operations checked in B5. The respective test cases are artificially constructed to represent appropriate efficiency checks, i.e., the queries differ from common OCL applications. The artificial constructs provide meaningful results, because they simulate ordinary OCL expressions in a compressed form. The other set of test cases completes the efficiency tests by checking basic datatypes and their operations, i.e., operations often used in combination with the collection operations.

In appendix A we show all modeling language and OCL concepts covered by our benchmark. The structure also indicates which parts of the benchmark primarily cover the respective features.

On the one hand we covered all essential meta classes of the OCL metamodel. On the other hand this fact cannot be used as a representative indicator for the completeness of our benchmark. As an example, the ten different predefined iterator expressions like `select` or `one` (disregarding the duplicates resulting from the type hierarchy) are represented by a single class in the metamodel, namely IterateExp.

Nevertheless we can make additional statements about the considered OCL types and operations: Our benchmark covers all main types of OCL according to the OCL metamodel (cf. Sect. 8 wrt. type OclMessage). Furthermore the test cases cover more than 90% of the collection operations. A reason for not considering all operations is the presence of changes presented in the beta version of the OCL 2.2 specification [26]. We are going to include the missing operations after the finalization of OCL 2.2, i.e., when the design decisions are definite.

In this section we discussed the overall design of our benchmark. In the next section we start to present the benchmark parts in detail.

4 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. For example, 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. Equally important is the correctness of OCL tools which are used stand-alone.

4.1 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.

In addition to MOF resp. UML 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 provide for the evaluation of state-dependent and state-independent expressions (e.g., `Person.allInstances()->select(age>18)` and `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. Another difference between both versions is the addition of ordered sets in version 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 [27] were our starting point.

4.2 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.

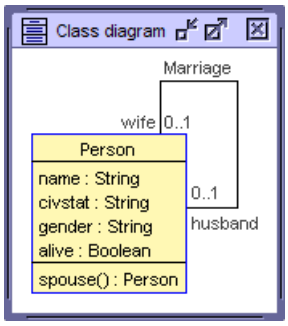

Fig. 2 Class Diagram of the Core Benchmark Model

Frequently used OCL operations and constructs are employed in 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. The collect shortcut denotes an application of a property to a collection of objects which is understood as a shortcut for applying the property inside a `collect` call, e.g., `Person.allInstances.name` stands 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. The expression below represents such an invariant. It checks whether the collect shortcut works correctly.

```
context Person inv abcNameDotShortcutP0_VT:
   let ada=Person.allInstances()->any(
      p:Person|p.name='Ada') in
   let bob=Person.allInstances()->any(
      p:Person|p.name='Bob') in
   let cyd=Person.allInstances()->any(
      p:Person|p.name='Cyd') in
   Set{ada,bob,cyd}.name=Bag{'Ada','Bob','Cyd'}
```

The suffix P0_VT of the invariant name indicates that no parentheses follow the operation `allInstances` and that the variables in collection operations are typed. 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: (1) Iterator variables can be explicitly named (indicated by VN), (2) they can be typed (VT), and (3) several operations also accept implicit variables (VI). For example:

```
(1) Person.allInstances()->
       reject(p|p.gender='male')
(2) Person.allInstances()->
       reject(p:Person|p.gender='male')
(3) 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 (P1 instead of P0).

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.

4.3 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 and in the example query below which has to return all pairs of persons who possibly can get married.

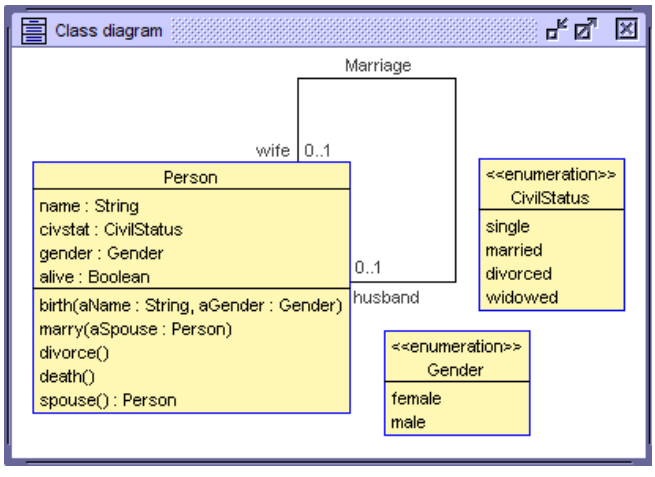

Fig. 3 Class Diagram of the Extended Core Benchmark Model

```
Person.allInstances()->iterate(w,h:Person;
   res:Set(Tuple(bride:Person,
                 bridegroom:Person)) =
   Set{} |
   if w.gender=Gender::female and
      h.gender=Gender::male and
      w.alive and h.alive and
      w.civstat<>CivilStatus::married and
      h.civstat<>CivilStatus::married
   then res->including(Tuple{bride:w,
                             bridegroom:h})
   else res endif)
```

In the extended core 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 have to 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.

One of the operations to be simulated is the operation `divorce`. It is constrainted by pre- and postconditions. The latter particularly test the `@pre` operator which allows for accessing the former system state, e.g., the expression `husband@pre` results in the husband who was linked to the current person before the operation was invoked.

```
operation Person::divorce()
pre isMarried: civstat=CivilStatus::married
pre isAlive: alive
pre husbandAlive:
   gender=Gender::female implies husband.alive
pre wifeAlive:
   gender=Gender::male implies wife.alive
post isDivorced: civstat=CivilStatus::divorced
post husbandDivorced:
   gender=Gender::female implies
      husband.isUndefined and
      husband@pre.civstat =
      CivilStatus::divorced
post wifeDivorced:
   gender=Gender::male implies
      wife.isUndefined and
      wife@pre.civstat=CivilStatus::divorced
```

4.4 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.

For this reason, the accuracy benchmark B3 is based on a model specifying a ternary reflexive association

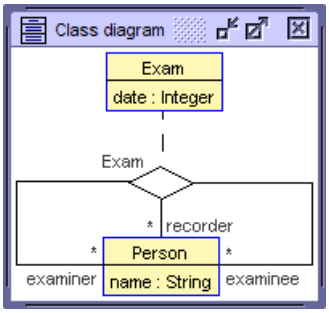

Fig. 4 Class Diagram of the Advanced Model

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, the expression `ada.examiner[examinee]` results in all persons being an examiner of the examinee Ada.

4.5 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. The following expression for example checks the implementation of the operation `implies`.

```
let B=Sequence{Undefined,false,true} in
B->iterate(b1,b2:Boolean;
   r:Sequence(Boolean) = Sequence{} |
   r->including(b1 implies b2))
```

The variables `b1` and `b2` consecutively take the undefined value, `false` and `true` as value. By this, we build up the whole truth table. So the expected result is:

```
Sequence{Undefined, Undefined, true,
         true, true, true,
         Undefined, false, true}
```

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.

4.6 OCL Laws Benchmark (B5)

Benchmark B5 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 [27] provides a basis for this benchmark. All test cases check for the equivalence of two different OCL expressions, i.e., they test whether the laws between two operations as exposed in [27] hold. 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, false and the undefined value 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, ordered 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))
```

Since the first version of the benchmark this part was extended by test cases based on ordered sets. These test cases were merged into the existing test structure.

4.7 OCL Distinctive Features Benchmark (B6)

The OCL distinctive features part was newly added to the first version of the benchmark. It treats (A) qualified associations and the corresponding OCL navigation expressions, (B) ordered sets and the OCL standard operations defined on this collection type, (C) the operations `oclAsType`, `oclIsTypeOf` and `oclIsKindOf`, (D) the operation `sortedBy`, and (E) the operations `collect` and `collectNested`.

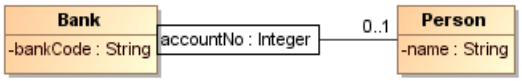

Fig. 5 Class Diagram of the Bank and Customer Example

Qualifiers are attributes on binary associations as shown in Fig. 5. The value of the qualifier accountNo selects a unique person owning a bank account with this specific account number. OCL provides for navigation with qualifier values by appending the values to role names. The examples below base on a system state in which the persons Ada and Bob are customers of a bank. The account number 123456 belongs to Ada and the number 654321 belongs to Bob.

```
Bank.allInstances()->any(bankCode=1).
   person.name=Bag{'Ada','Bob'}
Bank.allInstances()->any(bankCode=1).
   person[123456].name='Ada'
Bank.allInstances()->any(bankCode=1).
   person[654321].name='Bob'
```

The benchmark covers additional qualifier test cases, e.g., models with more than one qualifier attribute and different multiplicities.

In contrast, test cases for checking the implementation of ordered sets do not need an underlying UML model. They are context-free. Besides the completion of the OCL laws part in terms of ordered sets, this kind of collection is checked in-depth in this part of the benchmark. That is, all OCL standard operations defined on ordered sets are included.

Unfortunately the OCL standard is incomplete in respect of ordered sets. On the one hand, expressions like the following ones must definitely yield true.

```
OrderedSet{1,2,3}<>OrderedSet{1,3,2}
OrderedSet{1,1,2,3,1}->sum()=6
```

On the other hand, the handling of duplicates is not specified or discussed in the standard, i.e., it is not clear

whether the expressions listed below result in the value `OrderedSet{1,2,3}` or `OrderedSet{1,3,2}`. A consistent approach would be to choose the former set for all four expressions. Thereby, the values to be added which are already present in ordered sets should be ignored. This topic has to be further discussed within the OCL community.

```
OrderedSet{1,2,3,2}
OrderedSet{1,2,3}->including(2)
OrderedSet{1,2,3}->append(2)
OrderedSet{1,2,3}->prepend(2)
```

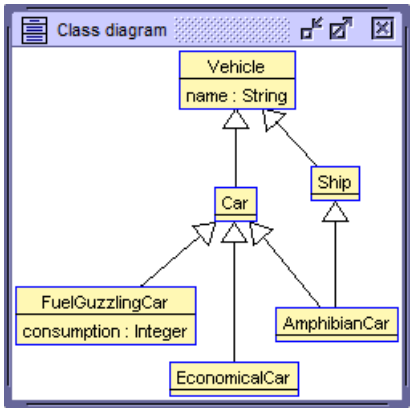

Fig. 6 Class Diagram of the Class Hierarchy Example

The class hierarchy in Fig. 6 makes up the basis for expressions checking type related operations. The `let` expressions of the following examples reflect the underlying system state. First we test the implementation of the operation `oclIsTypeOf` in context of each specified class.

```
let vehicle:Vehicle =
   Vehicle.allInstances()->any(name='A380') in
let car:Car =
   Car.allInstances()->any(name='Golf') in
let ecocar:EconomicalCar =
   EconomicalCar.allInstances()->
      any(name='Smart') in
let guzcar:FuelGuzzlingCar =
   FuelGuzzlingCar.allInstances()->
      any(name='Veyron') in
let amphcar:AmphibianCar =
   AmphibianCar.allInstances()->
      any(name='AmphiCar') in
Sequence{vehicle,car,ecocar,guzcar,amphcar}->
   collect(v|v.oclIsTypeOf(EconomicalCar)) =
Sequence{false,false,true,false,false}
```

Analogous test cases are defined for the operation `oclIsKindOf`, but the operation `oclAsType` needs a special treatment. The source collection of the test case below is of type `Sequence(Vehicle)`. Vehicles cannot access the attribute `consumption` which is only defined for fuel-guzzling cars. The operation `oclAsType` is used to convert the fuel-guzzling car from type `Vehicle` to `FuelGuzzlingCar`.

```
letExpressions
Sequence{vehicle,car,ecocar,guzcar,amphcar}->
   any(oclIsTypeOf(FuelGuzzlingCar)).
      oclAsType(FuelGuzzlingCar).consumption =
   80
```

This, for example, would not be possible for instances of type `Vehicle`. Consequently a corresponding expression must result in the undefined value.

```
let vehicle:Vehicle =
   Vehicle.allInstances()->any(name='A380') in
vehicle.oclAsType(Car)=Undefined
```

While the upper expression has to be evaluated during runtime, the next expression has to result in a type error, because an amphibian car can never be converted to an economical car.

```
let amphcar:AmphibianCar =
   AmphibianCar.allInstances()->
      any(name='AmphiCar') in
amphcar.oclAsType(EconomicalCar)
```

Some operations used as auxiliary operations in other parts of the benchmark need to be handled more explicitly. Examples are the operations `collectNested`, `collect` and `sortedBy`. They are systematically checked, e.g., by contrasting `collect` with `collectNested`, choosing differently nested source collections, or by sorting duplicate values.

```
let s:Set(Integer) = Set{1,2,3,4} in
s->collect(i|i) =
s->collectNested(i|i)->flatten()
```

```
let s:Set(Set(Set(Integer))) =
   Set{Set{Set{1,2},Set{3,4}},
       Set{Set{5,6},Set{7,8}}} in
s->collect(i|i) =
s->collectNested(i|i)->flatten()
```

```
Bag{1.5,1.0,0.0,-1.0,-1.0,-1.5}->
   sortedBy(r:Real|r*-1) =
Sequence{1.5,1.0,0.0,-1.0,-1.0,-1.5}
```

5 OCL Engine Determinateness Properties (B7)

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 the operation `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 B7 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.

6 OCL Engine Efficiency (B8)

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
```

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, (B) the connected components of the graph, i.e., the maximal node sets in which all nodes are connected directly or indirectly, and (C) the number of directed paths beginning at a given node. The example model has a single class and a single association as displayed in Fig. 7.

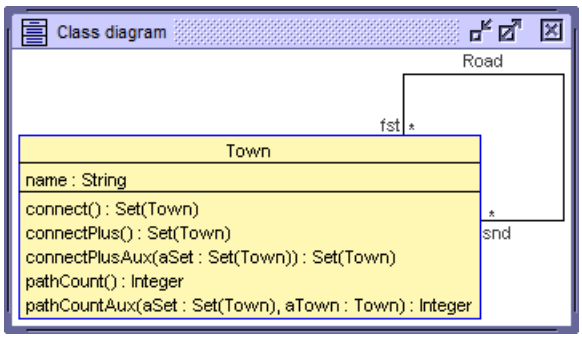

Fig. 7 Class Diagram for Towns and Roads

An example state with 42 towns and 42 roads is built up. It is illustrated in Fig. 8. For maintaining a clear

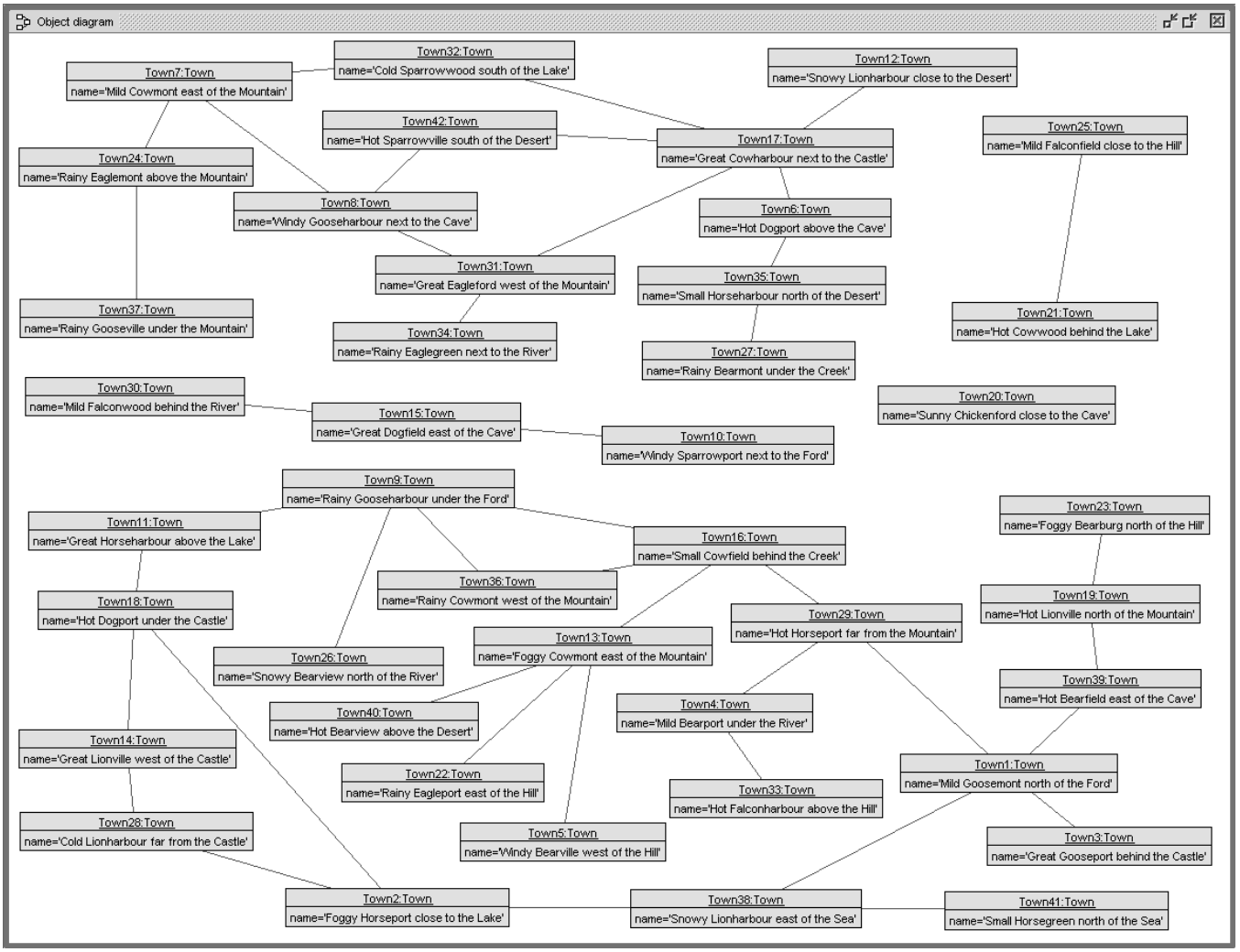

Fig. 8 Object Diagram for an example State with 42 Towns and 42 Roads

arrangement the figure hides the role names. 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)
```

The operations `connectPlus()` computes all towns directly or indirectly reachable from the current node with the roles fst and snd. It makes use of the operation `connect` which returns all directly connected nodes disregarding the direction of the edges and the recursive operation `connectPlusAux`.

```
connect():Set(Town) =
   fst->union(snd)
connectPlus():Set(Town) =
   connectPlusAux(connect())
connectPlusAux(aSet:Set(Town)):Set(Town) =
   let oneStep:Set(Town)=aSet->collect(t |
      t.connect())->flatten()->asSet() in
   if oneStep->exists(t|aSet->excludes(t))
   then connectPlusAux(aSet->union(oneStep))
   else aSet endif
```

Another OCL query examines a different property of the graph by calculating the number of all directed paths which do not include cycles. A direction is given by the role names fst and snd. An edge starts at fst and ends at snd. The expression consecutively takes each town instance as source node, determines the number of directed paths starting at the source node, and adds up the results.

```
Town.allInstances()->
   collect(t|t.pathCount())->sum
```

The operation `pathCount` only navigates via the the role name snd. It relies on a recursive auxiliary operation which returns 1 if a leaf is reached and 0 if it detects a cycle.

```
pathCount():Integer =
   if snd->isEmpty then 0
   else pathCountAux(Set{},self)
   endif
pathCountAux(aSet:Set(Town),
             aTown:Town):Integer =
   if aTown.snd->isEmpty then 1
   else aTown.snd.collect(t |
      if aSet->includes(t) then 0
      else pathCountAux(aSet->including(t),t)
      endif)->sum()
   endif
```

The example state was created automatically by the system state generator of the USE tool [28]. The generator allows us to build up large states with more than thousand towns and roads. In this way we were able to produce a basis for more meaningful efficiency tests, e.g., for checking the scalability of OCL engines. Beside the demonstrative example with 42 towns and roads there are also test states with 128, 256, 512 and 1024 nodes and edges. The underlying ASSL [28] (A Snapshot Sequence Language) procedure is shown below. It takes the number of towns and roads to be created as parameters. Each town gets a randomly chosen name which consists of five parts. The roads randomly connect the towns.

```
procedure genWorld(numTown:Integer,
                   numRoad:Integer)
var theTowns: Sequence(Town),
   one:Town, two:Town,
   part1:String, part2:String, part3:String,
   part4:String, part5:String;
begin
theTowns:=CreateN(Town,[numTown]);
for i:Integer in [Sequence{1..numTown}] begin
   part1:=Any([Sequence{'Snowy ','Foggy ',
      'Great ','Small ','Cold ','Mild ',
      'Hot ','Windy ','Sunny ','Rainy '}]);
   part2:=Any([Sequence{'Eagle','Falcon',
      'Bear','Cow','Horse','Lion',
      'Dog','Chicken','Goose','Sparrow'}]);
   part3:=Any([Sequence{'wood','port',
      'harbour','ford','green','ville',
      'burg','field','mont','view'}]);
   part4:=Any([Sequence{' close to the ',
      ' under the ',' north of the ',
      ' south of the ',' east of the ',
      ' west of the ',' behind the ',
      ' above the ', ' next to the ',
      ' far from the '}]);
   part5:=Any([Sequence{'Sea','Lake',
      'Mountain','River','Desert','Creek',
      'Castle','Ford','Hill','Cave'}]);
   [theTowns->at(i)].name :=
      [part1.concat(part2).concat(part3).
      concat(part4).concat(part5)];
   end;
```

	Invs	Queries	State dependent	Correctly evaluated (⌀)
B1	71	0	71	68,5% (6 tools)
B2	0	17	17	82,4% (5 tools)
B3	0	14	14	75,0% (2 tools)
B4	0	12	0	58,3% (6 tools)
B5	0	875	0	82,0% (6 tools, 710 test cases checked)
B6	0	297	33	N/A
B7	0	92	0	66,9% (6 tools)
B8	0	34	28	N/A
Sum	72	1341	163	79,3%

Table 1 Overview of the evaluation results.

```
for i:Integer in [Sequence{1..numRoad}] begin
   one:=Any([theTowns]);
   two:=Any([theTowns->excluding(one)->
      reject(t|one.fst->includes(t))->
      reject(t|one.snd->includes(t))]);
   Insert(Road,[one],[two]);
   end;
end;
```

7 Empirical Evaluation of the Benchmark

The aim of this empirical evaluation section and the subsequent discussion section is

(A) to show the applicability of the benchmarks concepts developed before, by revealing significant problem fields with the application of each individual benchmark part,
(B) to make a contribution for improvements of current OCL engines by showing particular problem classes which have to be regarded during the development of an OCL engine, and
(C) to encourage discussions about the mentioned language constructs with respect to the (sometimes ambiguous or incomplete) OCL standard definitions.

Table 1 quantitatively overviews the benchmark and the results. It lists the number of defined OCL invariants and queries for each part of the benchmark, determines the number of state dependent expressions (a subset of the invariants and queries) and presents the average benchmark results as well as the number of regarded tools.

Our OCL benchmark comprises 1413 test cases, composed of 71 invariants and 1341 query expressions. 1250 expressions are context-free and 163 state-dependent. We evaluated 949 test cases—the number of test cases in the first version of the benchmark—in 7 OCL evaluation engines including two code generators. One of the tools was only partly checked because of resource limitations and part B2 of the benchmark was only applied for two engines, because the others did not support some of the

advanced UML concepts. Altogether about 80% of the test cases were correctly evaluated. This is a good basis for further improvements and tool harmonization.

We would like to emphasize the applicability of the benchmark as well as the general problem fields and will therefore not go into details concerning the single tools. OCL engine developers can find all details at [17] (partly in German) and are encouraged to perform our benchmark by themselves.

7.1 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 17 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 (18/17). The other tools evaluate from 6 to 16 expressions correctly (6/17, 6/17, 14/17, 14/17, 16/17). 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 related 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.

7.2 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. The expression `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.

7.3 Three-Valued Logic Benchmark (B4)

The 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 conforming 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 B7. 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 fulfilling the boolean subexpression (e.g., `Sequence{undefinedValue,1,4}->exists(i|i<4)`). Other operations generate the undefined value in either case.

7.4 Laws (B5) and Determinateness Benchmark (B7)

Benchmark B5 and B7 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 always 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.

8 Discussion of uncovered OCL Features

Some aspects of OCL are not covered by our benchmark, because further discussion in the OCL community is required. For instance, the type system described in the OCL standard is not clear with regard to the types OclVoid and OclAny. The standard states in the section 'Well-formedness Rules of the Expressions package' that empty collection literals have OclVoid as their element type. On the other hand the definition of the operation `including` for the type Set(T) is defined as `including(object:T):Set(T)`. Considering terms like `Set{}->including(1)`, this raises the question whether these terms are valid with respect to OCL 2.0.

Actually, they are valid, because the OCL standard library defines several operations `including(T')` for a Set(T). By definition of subtyping, the type Set(T) comprises all operations `Set(T')->including(T')`, for each supertype T' of T. Therefore, as `Set{}` which is of type Set(OclVoid), is also of type Set(Integer), it provides the operation `including(Integer)`. Consequently, `Set{}->including(1)` is of type Set(Integer) if we refer to the operation `Set(Integer)->including(Integer)`.

We could select the operation `Set(OclAny)->including(OclAny)` just as well, because Integer is a subtype of OclAny. If we do so, the term `Set{}->including(1)` would be of type Set(OclAny).

To gain the intended typing, i.e., Set(Integer), we need a choosing rule for overloaded collection operations that selects the most specific common supertype for the element and operand types. But at the time being, the OCL standard does not state clearly how the choosing from several (overloaded) collection operations should work exactly. The situation becomes unclear as soon as multiple inheritance comes into play. Under this situation, a unique most specific supertype cannot be found. A proposal addressing this issue was made for OCL 1.3 in [29]. But it has not been included in the type system of OCL. Therefore, we do not include tests in our benchmark that require collection operations from supertypes.

Another part of OCL not included in the benchmark is the message sending operator ^, because the standard is not definitive about its semantics. The semantical model does not handle state transitions which are induced by sent messages. Another pragmatic reason

for not considering message sending is the fact that this OCL feature is not supported by known OCL tools.

9 Conclusion

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. With this paper we propose a comprehensive benchmark for OCL engines. We have empirically evaluated our benchmark by considering a number of different OCL engines. The evaluation accuracy is generally high. On an average the engines evaluate about 80 percent of the applied test cases correctly.

The set of incorrectly evaluated expressions varies depending on the particular tool. 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) a feature for handling 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 1413 expressions) can be carried out in an automatic way.

Some OCL engines considered in this paper do not offer the above functionality: RoclET does only allow for checking 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.

The benchmark sources (available in generic formats like XML and XMI) and a technical description on how to apply the current version of the benchmark in OCL tools, i.e., information on requirements and setup, are presented at [17].

Last but not least we would like to thank the OCL engine users and developers for their valuable feedback. It has given us the possibility to improve and supplement our benchmark. We expect that our benchmark continues to be a basis for discussions on OCL.

References

1. OMG, 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. Addison-Wesley (2003) 2nd Edition.
3. ATL-Team: ATL Development Tools. http://www.sciences.univ-nantes.fr/lina/atl/atldemo/adt (2008)
4. MDT-OCL-Team: MDT OCL. http://www.eclipse.org/modeling/mdt/?project=ocl (2008)
5. Ziemann, P., Gogolla, M.: OCL Extended with Temporal Logic. In: Proc. Ershov Memorial Conference, LNCS 2890 (2003) 351–357
6. Flake, S., Müller, W.: An OCL Extension for Real-Time Constraints. In: Object Modeling with OCL, LNCS 2263 (2002) 150–171
7. Gogolla, M., Kuhlmann, M., Büttner, F.: A Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency. In Czarnecki, K., ed.: Proc. 11th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'2008), LNCS 5301, Springer, Berlin (2008) 446–459
8. Dresden-OCL-Team: Dresden OCL Toolkit. http://dresden-ocl.sourceforge.net (2008)
9. Chiorean, D., OCLE-Team: Object Constraint Language Environment 2.0. http://lci.cs.ubbcluj.ro/ocle (2008)
10. Kleppe, A., Warmer, J.: Octopus: OCL Tool for Precise UML Specifications. http://octopus.sourceforge.net (2008)
11. RoclET-Team: Welcome to RoclET. http://www.roclet.org (2008)
12. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming **69** (2007) 27–34
13. Kermeta-Team: Kermeta: Breathe Life into your Metamodels. http://www.kermeta.org (2008)
14. Akehurst, D., Patrascoiu, O.: KMF (Kent Modeling Framework) OCL Library. http://www.cs.kent.ac.uk/projects/ocl/tools.html (2008)
15. Hein, C., Ritter, T., Wagner, M.: Open Source Library for OCL (OSLO). http://oslo-project.berlios.de (2008)
16. VMTS-Team: Visual Model and Transformation System (VMTS). http://vmts. aut.bme.hu (2008)
17. Kuhlmann, M., Hamann, L., Gogolla, M., Büttner, F.: OCL Benchmark. http://www.db.informatik.uni-bremen.de/publications/OCLbench/ (2010)
18. Clavel, M., Egea, M., de Dios, M.A.G.: Building an Efficient Component for OCL Evaluation. In: 8th OCL Workshop at the UML/MoDELS Conferences: OCL Concepts and Tools. (2008)
19. National Institute of Standards and Technology: Conformance Test Suite Software. http://www.itl.nist.gov/div897/ctg/software.htm
20. Gray, M., Goldfine, A., Rosenthal, L., Carnahan, L.: Conformance Testing. http://xml.coverpages.org/conform20000112.html

21. Gaudel, M.C.: Testing can be formal, too. In: TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development, Springer-Verlag (1995) 82–96
22. Bunyakiati, P., Finkelstein, A., Rosenblum, D.: The certification of software tools with respect to software standards. (Aug. 2007) 724–729
23. Afanasiev, L., Marx, M.: An Analysis of the Current XQuery Benchmarks. In: In International Workshop on Performance and Evaluation of Data Management Systems (EXPDB). (2006)
24. Sim, S.E., Easterbrook, S., Holt, R.C.: Using Benchmarking to Advance Research: A Challenge to Software Engineering. In: ICSE '03: Proceedings of the 25th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2003) 74–83
25. Pfaller, C., Wagner, S., Gericke, J., Wiemann, M.: Multidimensional measures for test case quality. In: Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on. (April 2008) 364–368
26. OMG, ed.: Object Constraint Language 2.2 - Beta 2 (ptc/2009-05-02). OMG (2009) `http://www.omg.org`.
27. Kuhlmann, M., Gogolla, M.: Analyzing Semantic Properties of OCL Operations by Uncovering Interoperational Relationships. Electronic Communications of the EASST, http://eceasst.cs.tu-berlin.de/index.php/eceasst **9** (2008) UML/MoDELS Workshop on OCL (OCL4ALL'2007), 17 Pages.
28. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Journal on Software and System Modeling **4**(4) (2005) 386–398
29. Schürr, A.: A new type checking approach for OCL version 2.0 ? In Clark, T., Warmer, J., eds.: Object Modeling with the OCL: The Rationale behind the Object Constraint Language. Springer (2002) 21–41

A Covered UML and OCL Elements

The following two lists show all UML and OCL concepts covered by the OCL benchmark. The prefixes **B1** to **B8** indicate the benchmark part which primarily checks the respective concepts.

A.1 UML

- **B1** models (class diagram)
  - **B2** enumerations
    - **B2** names, literals
  - **B1** classes
    - **B1** names
    - **B1** attributes
      - **B1** primitive types
        - **B1** `String`
        - **B1** `Boolean`
        - **B3** `Integer`
      - **B2** enumeration types
    - **B1** operations
      - **B1** names
      - **B1** parameters
        - **B1** implicit `self` parameter
        - **B2** explicit parameters
      - **B1** return types
      - **B2** no return types
  - **B1** associations
    - **B1** names, multiplicities, explicit role names
    - **B1** reflexiveness
    - **B1** arity
      - **B1** binary
      - **B3** ternary
    - **B3** properties (association class)
    - **B6** qualification
- **B1** states (object diagram)
  - **B1** objects
    - **B1** creation
    - **B1** attribute values
      - **B1** setting
      - **B2** changing
  - **B1** links
    - **B1** insertion
    - **B2** deletion

A.2 OCL

- **B1** expressions
  - **B1** constraints
    - **B1** invariants
      - **B1** names
      - **B1** evaluation
    - **B2** pre- and postconditions
      - **B2** `@pre`
        - **B2** applied to objects
      - **B2** names
      - **B2** evaluation
    - **B2** queries
      - **B2** state dependent
      - **B5** state independent
      - **B2** evaluation
    - **B1** types
      - **B1** basic
      - **B2** enumeration
      - **B1** object
      - **B2** `OclAny`
      - **B2** `Tuple`
      - **B1** collection
        - **B1** `Set`, `Bag`, `Sequence`
        - **B6** `OrderedSet`
        - **B1** element types
          - **B1** homogeneous, heterogeneous
          - **B1** complex
    - **B1** constructors
      - **B1** `Set{}`, `Bag{}`, `Sequence{}`
        - **B1** literals
        - **B1** variables
        - **B2** object identifiers
        - **B2** tuples
        - **B1** nested constructors
        - **B1** range expressions
          - **B1** `Integer`
          - **B1** complex expressions
        - **B1** complex expressions
      - **B6** `OrderedSet{}`
      - **B2** `Tuple{}`
        - **B2** component access
    - **B1** literals
      - **B1** `Integer`, `String`, `Boolean`
      - **B6** `Real`
      - **B2** enumeration
  - **B1** operations
    - **B1** parameters
      - **B1** literals
      - **B1** variables
      - **B2** object identifiers
      - **B2** tuples
      - **B1** complex expressions
      - **B5** collections
        - **B5** empty
        - **B5** singleton
          - **B5** including undefined value
          - **B5** excluding undefined value
        - **B5** many elements
          - **B5** including undefined value
          - **B5** excluding undefined value
    - **B1** types
      - **B1** class
        - **B1** `allInstances`
          - **B1** parentheses
          - **B1** no parentheses
      - **B1** object
        - **B1** `<>`

- **B2** `oclIsUndefined`
- **B1** user-defined query operations

B2 enumeration
- **B2** `=`, `<>`, `oclIsUndefined`

B5 `Integer`
- **B5** `=`, `<`, `>`, `<=`, `>=`, `-`, `+`, `*`

B1 `String`
- **B1** `=`, `<>`, `substring`
- **B1** `size`
  - **B1** parentheses
  - **B2** no parentheses
- **B2** `isUndefined`

B1 `Boolean`
- **B1** predicates
  - **B2** `=`, `oclIsUndefined`
- **B4** connectives
  - **B4** `and`, `or`, `implies`, `not`, `xor`
- **B1** binding

B1 `Collection`
- **B1** `Set`, `Bag`, `Sequence`
- **B6** `OrderedSet`
- **B1** collection of objects
  - **B1** user-defined operations (dot shortcut)
  - **B1** `=`
- **B1** `excluding`, `flatten`, `includes`, `including`
- **B5** `asBag`, `asSequence`, `asSet`, `at`, `count`
- **B5** `excludes`, `exists`, `first`, `last`, `notEmpty`
- **B5** `reject`
- **B6** `asOrderedSet`, `sortedBy`, `oclAsType`
- **B6** `oclIsTypeOf`, `oclIsKindOf`
- **B1** `any`
  - **B1** (no) variable declaration
  - **B1** (no) type declaration
  - **B1** literal `false`
  - **B5** literal `true`
- **B1** `forAll`
  - **B1** one iterator variable
  - **B1** two iterator variables
  - **B1** type declarations
  - **B1** no type declarations
- **B1** `iterate`
  - **B5** one iterator variable
  - **B1** two iterator variables
  - **B1** type declarations
  - **B5** no type declarations
- **B1** `collect`, `isUnique`, `one`, `select`
  - **B1** (no) variable declaration
  - **B1** (no) type declaration
- **B1** `isEmpty`
  - **B1** parentheses
  - **B2** no parentheses
- **B5** `size`
- **B5** checked relationships
  - **B5** `reject`, `select`
  - **B5** `exists`, `reject`
  - **B5** `one`, `reject`
  - **B5** `exists`, `collect`, `one`
  - **B5** `collect`, `iterate`
  - **B5** `one`, `iterate`
  - **B5** `exists`, `forAll`
  - **B5** `forAll`, `reject`
  - **B5** `one`, `select`
  - **B5** `forAll`, `collect`, `excludes`
  - **B5** `exists`, `iterate`
  - **B5** `reject`, `iterate`
  - **B5** `one`, `exists`, `forAll`
  - **B5** `forAll`, `select`
  - **B5** `forAll`, `collect`, `one`
  - **B5** `exists`, `collect`, `includes`
  - **B5** `forAll`, `iterate`
  - **B5** `select`, `iterate`

B1 other statements and concepts
- **B1** navigation
  - **B1** dot shortcut
    - **B1** sources
      - **B1** objects
      - **B1** collection constructors
      - **B1** complex expressions
    - **B1** attribute access
    - **B1** role name access
      - **B1** directly
      - **B3** in role of `[ ]` (n-ary reflexive association)
    - **B1** operation calls
      - **B1** standard operations
      - **B1** user-defined query operations
    - **B1** implicit collect and flatten
  - **B1** `if-then-else-endif`
  - **B1** `let`
    - **B1** variable names
    - **B1** variable types
      - **B1** collections
    - **B1** nested
  - **B1** self
    - **B1** explicit
    - **B2** implicit

B1 undefined value
- **B1** provocation
  - **B1** navigating from undefined object
  - **B5** invoking `first`/`last` on empty sequence
  - **B1** `any(false)`
- **B1** handling (three-valued logic with equality)
  - **B1** `=`
  - **B4** `and`, `or`, `implies`
    - **B4** left argument undefined
    - **B4** right argument undefined
  - **B4** `if-then-else-endif`
    - **B4** first argument undefined
    - **B4** second argument undefined

B7 determinateness properties
- **B7** `asSequence`, `asBag->asSequence`, `iterate`
- **B7** `any`, `flatten`

B8 evaluation efficiency