Adding Random Operations to OCL

Antonio Vallecillo
Universidad de Málaga, Spain
av@lcc.uma.es

Martin Gogolla
University of Bremen, Germany
gogolla@informatik.uni-bremen.de

***Abstract*—This paper presents an extension of OCL to allow modellers to deal with random numbers and probability distributions in their OCL specifications. We show its implementation in the tool USE and discuss some advantages of this new feature for the validation and verification of models.**

I. Introduction

There are many situations in which there is a degree of uncertainty about some aspects, features or properties of the system to be modelled. For example, if we are modelling a community of human beings, what is the percentage of female persons that we want to include in our models? How many of them are expected to be left-handed? This may be important in order to generate the models that will be used during the testing, validation and verification processes so that they are as much accurate and representative as possible. Similarly, assuming that we are modelling a manufacturing system with UML and OCL [10], how to handle the requirements that orders are received following an exponential distribution, or that 0.5 % of the parts are produced with some kind of defect?

In most modelling and simulation environments, the use of random numbers and probability distributions are used to combine definite knowledge (female, male; left-handed, right-handed) with an uncertain view on the result or the population for a test case. Expectations and assumptions that remain uncertain or imprecise at high-level, are made precise and can be realized by stating the corresponding percentages, or the probability distributions that some properties or parameters follow. In this way, confidence on the validation and verification processes can be increased by experimenting with different percentages (that have different likelihoods) and by inspecting the results obtained. Likewise, random numbers are used to generate test models with varying sizes and characteristics, in order to increase the coverage of the test cases.

In this paper, we present an extension to OCL that allows modellers to handle random numbers in their specifications, as well as probability distributions. We show its implementation in the tool USE [4], [5] (UML-based Specification Environment), and discuss some examples for the validation and verification of models.

Non-determinism and random-like behavior is already present in OCL, for example, through the operation `any()`. Most proof-theoretic OCL approaches, such as [1] or [3] do not speak about the evaluation of equations involving `any()`, for example, `Set{7,8}->any(true) = Set{7,8}->any(true)`. In principle, an OCL evaluator could give different results for the two calls of `any()`, although USE [4] (and every other OCL evaluator that we are aware of) gives the same result for both calls. But there are also other OCL operations that introduce non-deterministic, random-like effects, for example, when one converts a collection without order into an order-aware collection, e.g., in `Bag{7,8,7}->asSequence()`. However, randomness also involves returning different (valid) values in separate executions of the operations.

The paper is organized in 7 sections. After this introduction, Sect. 2 provides an overview of what OCL currently offers and what should be required. Then, Sect. 3 and Sect. 4 describe the OCL extensions to deal with random numbers and probability distributions, respectively. Section 5 illustrates the proposal with a couple of examples, and Sect. 6 gives details about how the new OCL operations have been implemented. Finally, Sect. 7 concludes and outlines some lines of future work.

II. Randomness in OCL

Non-determinism and random-like behavior is already present in OCL mainly through collections operations `asSequence()` and `any()` [8]. More precisely, OCL defines `any()` as follows: *Returns any element in the source collection for which body evaluates to true. Returns invalid if any body evaluates to invalid for any element, otherwise if there are one or more elements for which body is true, an indeterminate choice of one of them is returned, otherwise the result is invalid.*

Then, the OCL standard [8, 11.9.1] formally specifies it as follows:

```
source->any(iterator | body) =
  source->select(iterator | body)->asSequence()->first()
```

As we can see, it bases its indeterminism in the behaviour of operation `asSequence()`, which is defined for general type `Collection` and returns a `Sequence` *that contains all the elements from* `self`*, in an order dependent on the particular concrete collection type.* Its specification is as follows:

```
post: result->forAll(elem | self->includes(elem))
post: self ->forAll(elem | result->includes(elem))
```

For example, for a given `Set` it returns a `Sequence` that contains all the set elements, *in undefined order*.

```
post: result->forAll(elem | self->includes(elem))
post: self->forAll(elem | result->count(elem) = 1)
```

As mentioned above, despite in theory this allows any OCL evaluator to return a different value for the same `Set` every

time it is executed, in practice this does not happen and the same element is always returned.

The problem is that when it comes to other collections, the OCL specification of `any()` seems to be wrong, since it only works for `Set` and `Bag` collections because for the other two there is no indeterminism at all. More precisely, for `Bag` it may work, since operation `asSequence()` returns a `Sequence` that contains all the elements from self, in undefined order.

```
post: result->forAll(elem |
            self->count(elem) = result->count(elem))
post: self ->forAll(elem |
            self->count(elem) = result->count(elem))
```

However, the behaviour of `asSequence()` is completely deterministic for collections `OrderedSet` and `Sequence`. For the former, the operation returns a `Sequence` that contains all the elements from self, *in the same order*.

```
post: Sequence{1..self.size()}->
             forAll(i | result->at(i) = self->at(i))
```

Similarly for `Sequence` collections, where `asSequence()` returns *the `Sequence` identical to the object itself. This operation exists for convenience reasons*.

```
post: result = self
```

This means that `any()` applied to a `Sequence` or an `OrderSet` will always return its first element, and not *an indeterminate choice of one of them* as its specification requires.

This is why we propose the following specification for operation `any()`, which does not have this problem:

```
post: self->includes(result)
```

III. Specifying Random Numbers in OCL

Random numbers are generated by extending OCL type Real with an operation called `rand()`. If `x.oclIsOfType(Real)` then `x.rand()` returns a random Real number between `0` and `x`.

```
context Real::rand() : Real
post indeterminism:
    if self > 0.0 then
       (0.0 <= result) and (result < self)
    else if self < 0.0 then
           (result <= 0.0) and (self < result)
        else /* self = 0.0 */ result = self
        endif
    endif
```

For example `1.rand()` returns a random number in the interval `[0..1)`. If you need a number in the interval `[a..b)` you can use the expression "`a + (b-a).rand()`."

Note that every invocation of `rand()` operation may return a different number, and that randomness requires an additional requirement to the postcondition (`indeterminism`) expressed above. This is why operation `any()` is not enough to implement random numbers. Randomness also requires that the sequence of results obtained by consecutive calls to operation `rand()` contains *no recognizable patterns or regularities*—i.e., that the sequence is *statistically random* [7]. However, specifying this property in OCL deserves its own line of research [2] and it is postponed for future work.

In addition, we need *seeds*. Operation `srand()` permits knowing and changing the seed for the random number generator. It is defined over Integers:

```
context Integer::srand() : Integer
```

Then, given an integer `n`, if `n > 0` then `n.srand()` starts a new random sequence with `n` as the new seed (the seed is an integer), and returns the value of the previous seed. To accommodate to the current possibilities of USE, we decided to restrict to integer values below 10^7. Thus, this operation takes the given value modulo 10^7. If `n <= 0`, this operation generates a seed automatically using the current time and other system values.

To illustrate how these operations work, the following listing shows their results when executed in USE:

```
use> ?1.rand()
-> 0.09152860811512553 : Real
use> ?2.rand()
-> 1.8371397364794912 : Real
use> ?2.rand()
-> 0.6646401472302712 : Real
use> ?2.rand()
-> 1.5417649510780334 : Real
use> ?2.rand()
-> 0.990212333639167 : Real
use> ?2.rand()
-> 1.676070756281957 : Real
use> ?2.rand()
-> 1.835645464118648 : Real
use> ?1.srand()
-> 32783 : Integer
use> ?1.srand()
-> 1 : Integer
use> ?0.srand()
-> 1 : Integer
use> ?0.srand()
-> 2297549 : Integer
use> ?0.srand()
-> 3220924 : Integer
use> ?0.srand()
-> 3666729 : Integer
```

IV. Probability Distributions in OCL

Using the random number generator operation, it is easy to build the most commonly used distribution probability functions:

```
context Real::normalDistr( s : Real ) : Real
context Real::pdf01() : Real
context Real::pdf( m : Real, s : Real ) : Real
context Real::cdf01() : Real
context Real::cdf( m : Real, s : Real ) : Real
context Real::expDistr() : Real
```

They are all defined as extensions to type `Real`. With this, if `x` is a real number, then

- `x.normalDistr(s)` returns a value of a Normal (Gaussian) distribution $N(x, s)$ (for example `0.normalDistr(1)` returns the value of a $N(0, 1)$ distribution),
- `x.pdf01()` returns a value of the distribution function $PDF(x)$ of a Gaussian Distribution $N(0, 1)$, i.e., with mean=0 and $\sigma = 1$,
- `x.pdf(m,s)` returns a value of the distribution function $PDF(x)$ of a $N(m, s)$,

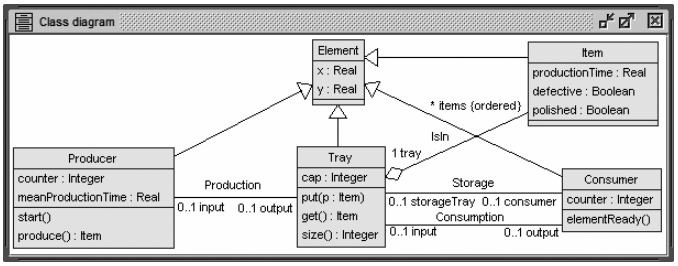

Fig. 1. Class diagram for the production system.

- `x.cdf01()` returns a value of the *cumulative* distribution function $CDF(x)$ of a Gaussian Distribution $N(0,1)$,
- `x.cdf(m,s)` returns a value of the *cumulative* distribution function $CDF(x)$ of a $N(m,s)$, and
- `x.expDistr()` returns a value of an exponential distribution with mean `x`, i.e., $Exp(1/x)$, or 0.0 if `x=0.0`.

V. Two Simple Examples

A. A Production System

To illustrate our proposal let us suppose first a simple production system whose metamodel is depicted in Fig. 1. Producers produce items that are placed in trays (bounded buffers), from where consumers collect them when informed that elements are ready (by operation `elementReady()`), polish them, and finally place them in the storage trays. We want to simulate the system with some uncertainty about the time producers take generating items and the probability of producers and consumers to introduce defects when handling the items.

For example, suppose that we want producers to produce items according an exponential distribution with mean 5.0, and that the probability of machines to introduce defects is 0.05. Using our OCL extension and its implementation in USE the description of operations `Producer::produce()` and `Consumer::elementReady()` is as follows:

```
produce():Item
begin
  result:=new Item;
  self.counter:=self.counter+1;
  result.productionTime :=
          self.meanProductionTime.expDistr();
  result.polished := false;
  result.defective :=
        if 1.rand() < 0.05 then true
        else false
        endif;
end
elementReady()
begin
  declare it: Item;
  it:=self.input.get();
  it.polished := if 1.rand() < 0.05 then false
                 else true
                 endif;
  it.defective := it.defective or
                 (if 1.rand() < 0.05 then true
                  else false
                  endif);
  self.storageTray.put(it);
  self.counter:=self.counter+1;
end
```

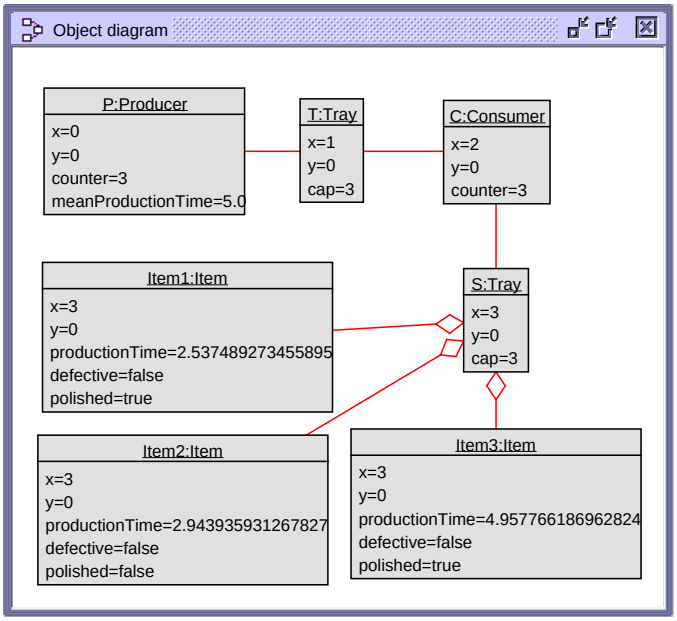

Fig. 2. Object diagram after producing three items.

One possible result of executing the system after the production of 3 items is shown in Fig. 2.

B. A Social Network

Random numbers can also be used to determine other parameters of the system, or even the number of objects that we would like to have in our test models.

In Fig. 3 a simple model of a social network is shown. For validation purposes, two object diagrams (shown in Fig. 4) have been generated by operation `generate()` using the proposed random features. The generated object diagrams differ with respect to attribute values and the structure that is defined by the `Friendship` links.

The definition of operation `generate()` is given below. The example demonstrates that, with the newly introduced OCL features, the generation of test cases showing different characteristics is supported.

```
generate(numObj:Int,numLink:Int)
begin
declare i:Int,
        p,q:Profile,
        ps:Seq(Profile);
ps:=Sequence{};
for i in Sequence{1..numObj} do
  p:=new Profile;
  ps:=ps->including(p);
  p.firstN:=
    names->at(1+names->size().rand().floor());
  end;
for i in Sequence{1..numLink} do
  p:=ps->at(1+ps->size().rand().floor());
  q:=ps->at(1+ps->size().rand().floor());
  if p.inviter->excludes(q) and
     p.invitee->excludes(q) then
        insert (p,q) into Friendship
     end;
  end;
end
```

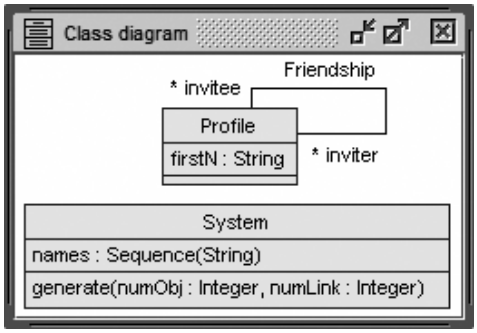

Fig. 3. Class diagram for the Social Network.

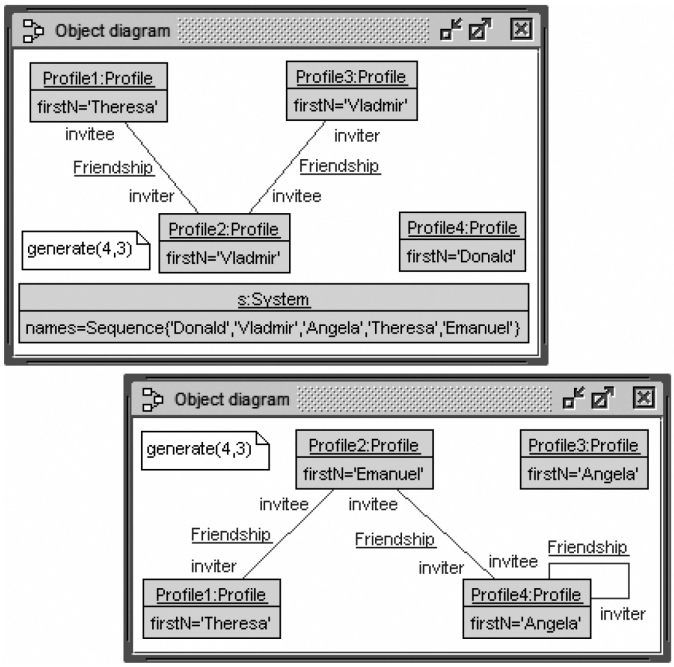

Fig. 4. Object diagrams with random links.

VI. Implementation in USE

Let us describe now how we have implemented this extension in USE. First, USE provides an extension mechanism that permits adding operations to basic types. Folder `oclextensions` in the USE directory permits adding new files with the signature of the new operations, and their implementation in Ruby [9].

For example, to add operation `sqrt` to OCL type `Real` we use the following piece of code in one of the files (e.g. `Real.xml`) in the `oclextensions` folder:

```
<operation source="Real" name="sqrt" returnType="Real">
  <body><![CDATA[
      Math.sqrt($self)
    ]]>
  </body>
 </operation>
```

Making use of this mechanism, and the `Random` library available in Ruby, the implementation of `rand()` and `srand()` operations is simple:

```
<operation source="Real" name="rand" returnType="Real">
  <body><![CDATA[
      $self * Random.rand
    ]]>
  </body>
 </operation>
<operation source="Integer" name="srand"
           returnType="Integer">
  <body><![CDATA[
    if $self > 0
      return Random.srand($self) % 1000000
    else
      return Random.srand() % 10000000
    end
  ]]>
  </body>
</operation>
```

If `self` is positive then `srand()` starts a new random sequence with `self` as new seed (the seed is integer), and it returns the current seed. Given that Ruby's initial seed is a huge integer number that cannot be handled by USE, this operation takes the modulo with 10^7. If `self` is equal or less than 0 then the operation uses the default Ruby `srand()` operation that generates a seed automatically using the time and other system values.

Finally, we have also implemented the probability distributions mentioned in Sect. 3 and show some of them in the following listing.

```
<operation source="Real" name="expDistr" returnType="Real">
  <body><![CDATA[
    if $self != 0
      return $self * (6.9077553 -
        Math.log(Random.rand(1000) + 1))
    else
      return 0.0
    end
  ]]>
  </body>
</operation>

<operation source="Real" name="normalDistr" returnType="
    ↪Real">
  <parameter>
  <par name="s" type="Real" />
  </parameter>
  <body><![CDATA[
    return $self + ( $s * Math.sqrt(-2.0 *
           Math.log(Random.rand)) *
           Math.cos(6.283185307 * Random.rand))
   ]]>
  </body>
</operation>

<operation source="Real" name="pdf" returnType="Real">
  <parameter>
    <par name="m" type="Real" />
    <par name="s" type="Real" />
  </parameter>
  <body><![CDATA[
    (1.0 / (Math.sqrt(2 * Math::PI))) *
     Math::exp(-((($self-$m)/$s)**2)/2.0))/$s
    ]]>
  </body>
</operation>

<operation source="Real" name="cdf" returnType="Real">
  <parameter>
    <par name="m" type="Real" />
    <par name="s" type="Real" />
  </parameter>
  <body><![CDATA[
  # Distribution.Normal.cdf(($self-$m)/$s)
  def cdf01(z)
    0.0 if z < -12
    1.0 if z > 12
    0.5 if z == 0.0
```

```
    if z > 0.0
      e = true
    else
      e = false
      z = -z
    end
    z = z.to_f
    z2 = z * z
    t = q = z * Math.exp(-0.5 * z2) / (Math.sqrt(2 * Math::
        ↪PI))
    3.step(199, 2) do |i|
      prev = q
      t *= z2 / i
      q += t
      if q <= prev
        return(e ? 0.5 + q : 0.5 - q)
      end
    end
    e ? 1.0 : 0.0
  end
  cdf01(($self-$m)/$s)
  ]]>
  </body>
</operation>
```

VII. Conclusions

In this paper we have introduced a simple extension of OCL to deal with random numbers and probability distributions in OCL specifications. It uses the USE extension mechanisms to implement the new operations, employing the underlying Ruby implementation and some of its supported functions. All files and operations described here can be downloaded from https://www.dropbox.com/s/2j9tgejbj507id0/oclextensions.zip?dl=0. To our knowledge, the only similar proposal is [6], a modelling framework for the predictive analysis of architectural properties.

Counting on these new operations offers interesting benefits to model developers and testers. For example, they are now able to capture some assumptions of the real world that correspond to stochastic events, or for which there is little information. We are also able to generate random sets of models, and models with random values in their elements' attributes, thus permitting richer input test suites for achieving model-based testing.

Our current plans for extensions of this work include the experimentation with larger case studies, in order to analyze the applicability and expressiveness of our approach, and the addition of further probability distributions that could be required in other situations.

Acknowledgments. This work was supported by Research Project TIN2014-52034-R.

References

[1] T. Baar. Non-deterministic Constructs in OCL: What Does any() Mean. In *Proc. 12th Int. SDL Forum*, LNCS 3530, pages 32–46, 2005.

[2] Robert Bill, Achim D. Brucker, Jordi Cabot, Martin Gogolla, Antonio Vallecillo, , and Edward D. Willink. Workshop in ocl and textual modelling. report on recent trends and panel discussions. In *Proc. of STAF 2017 Satellite Events*, LNCS. Springer, 2017.

[3] A.D. Brucker and B. Wolff. HOL-OCL: A Formal Proof Environment for UML/OCL. In *Proc. FASE'08*, LNCS 4961, pages 97–100, 2008.

[4] M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69:27–34, 2007.

[5] M. Gogolla and F. Hilken. Model Validation and Verification Options in a Contemporary UML and OCL Analysis Tool. In *Proc. Modellierung'2016*, pages 203–218. GI, LNI 254, 2016.

[6] Pontus Johnson, Johan Ullberg, Markus Buschle, Ulrik Franke, and Khurram Shahzad. *P2AMF: Predictive, Probabilistic Architecture Modeling Framework*, pages 104–117. Springer, 2013.

[7] M.G. Kendall and B. Babington Smith. Randomness and random sampling numbers. *Journal of the Royal Statistical Society*, 101(1):147–166, 1938.

[8] OMG. *Object Constraint Language (OCL), version 2.4*. Object Management Group, December 2014. OMG formal/2014-02-03.

[9] D. Thomas. *Programming Ruby 1.9 & 2.0*. Pragmatic Bookshelf, 4 edition, 2013.

[10] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2nd Edition, 2004.