Technical Report

Technical Report: Incorporating Measurement Uncertainty into OCL/UML Primitive Datatypes

Manuel F. Bertoa, Loli Burgueño,
Nathalie Moreno, Antonio Vallecillo

March 2019

Abstract This technical report contains the formal specification of the OCL and UML primitive datatypes extended with measurement uncertainty, defined in [3], as well as the proofs of their algebraic properties.

Contents

Manuel F. Bertoa, Nathalie Moreno, Antonio Vallecillo are at Universidad de Málaga, Spain. E-mail: {bertoa,moreno,av}@lcc.uma.es
Loli Burgueño is at Open University of Catalonia, IN3, Spain. and Institut LIST, CEA, Université Paris-Saclay, France. E-mail: lburguenoc@uoc.edu

1 Extending type `Boolean`

1.1 Specification of the extended values and operations

Type `UBoolean` is the supertype of type `Boolean` that adds uncertainty to its values. Thus, a `UBoolean` value is a pair (b, c), where b is a Boolean value and c is a Real number that is in the range $[0, 1]$ and represents the confidence that we have in b.

A property of this representation is that $(b, c) = (\neg b, 1 - c)$ for every boolean value b. Then, in the internal representation of uncertain booleans we will always use the *canonical form* of the value by taking $b =$ `true` and c to be the corresponding confidence. Using this canonical form, a true value with 95% confidence is represented as $(\texttt{true}, 0.95)$ and a false value with 95% confidence as $(\texttt{true}, 0.05)$. Then, `true` and `false` are injected into the supertype as $(\texttt{true}, 1.0)$ and $(\texttt{true}, 0.0)$, respectively.

The operations supported by type `UBoolean` extend those of type `Boolean`, as defined by OCL [6]. We have defined the basic operations (`not`, `and` and `or`) and secondary operations (`implies`, `equivalent` and `xor`) of the traditional Boolean algebra by extending them with uncertainty. Assuming that all values are independent, the following list specifies the `UBoolean`-type operations. They also take into consideration the two special values `null` and `invalid`, according to the 4-valued logic defined for OCL [6], relying on how the primitive OCL operations deal with these values.

```
not() : UBoolean
  post: if self=null or self.oclIsInvalid() then result = self
        else (result.b) and
            (result.c = if self.b then 1-self.c else self.c endif)
        endif
and(b : UBoolean) : UBoolean
  post: let C : Real = self.c*b.c in
        if (self.b and b.b) = null then null
        else if (self.b and b.b) = invalid then invalid
            else (result.b) and (result.c=
                if (self.b and b.b) then C else (1-C) endif)
            endif
        endif
or(b : UBoolean) : UBoolean
  post: let C : Real = (self.c + b.c - (self.c * b.c)) in
        if (self.b or b.b) = null then null
        else if (self.b or b.b) = invalid then invalid
            else (result.b) and (result.c=
                if (self.b or b.b) then C else (1-C) endif)
            endif
        endif
implies(b : UBoolean) : UBoolean
  post: let C : Real = ((1-self.c) + b.c - ((1-self.c) * b.c)) in
        if (self.b implies b.b) = null then null
        else if (self.b implies b.b) = invalid then invalid
            else (result.b) and (result.c=
                if (self.b implies b.b) then C else (1-C) endif)
            endif
        endif
```

```
equivalent(b : UBoolean) : UBoolean
  post: result = self.xor(b).not()
xor(b : UBoolean) : UBoolean
  post: if (self.b implies b.b) and (b.b implies self.b) = null
        then null
        else if (self.b implies b.b) and (b.b implies self.b) =
             invalid then invalid
        else let selfc : Real = if self.b then
                                self.c else 1-self.c endif in
             let bc : Real = if b.b then b.c else 1-b.c endif in
        (result.b) and (result.c=(selfc-bc).abs())
        endif endif
equals(b : UBoolean) : Boolean =
  (self.b=b.b)and(self.c=b.c) or (self.b=not b.b)and(self.c=1-b.c)
equalsC(b : UBoolean, C: Real) : Boolean =
  (self.b=b.b) and ((self.c-b.c).abs()<=1-C) or
  (self.b=not b.b) and ((self.c-1+b.c).abs()<=1-C)
distinct(b : UBoolean) : Boolean = not (self.equals(b))
toBoolean() : Boolean =
  if (self.c>=0.5) then (self.b) else (not self.b) endif
toBooleanC(c:Real):Boolean =
  if (self.c>=c) then (self.b) else (not self.b) endif
```

We have preserved the semantics of the `equals()` and `distinct()` operations: two `UBoolean` elements are the same if their confidence values match when they are represented in their canonical form (both operations return a `Boolean` value). We have also extended the `equals()` operation to specify a confidence threshold for the equality of the two `UBoolean` values. Other identity operations, namely `equivalent()` and `xor()`, compare two `UBoolean` values and return another `UBoolean` value.

Finally, some conversion operations allow `UBoolean` values to be converted into `Boolean` values, either approximately, if the confidence is greater than or equal to 0.5, or by specifying a threshold for the confidence.

1.2 Algebraic properties of the operations

To study the algebraic properties of the extended operations that we have defined for the OCL and UML datatypes, we must take into account that relations in this context are no longer evaluated by a Boolean value (i.e., the relation holds or not) but by a real number that is between 0 and 1 and expresses the probability that the relationship holds. For example, $A < B$ is now evaluated as $(\texttt{true}, c)$, with $c \in [0, 1]$. In this context, what does it mean for the $<$ operation to be transitive?

In the following, we will use the extended `UBoolean` version of the "=" operator, which we will denote by $\doteq$ to distinguish it from its Boolean version, whereby two uncertain Booleans $A = (\texttt{true}, c_a)$ and $B = (\texttt{true}, c_b)$ satisfy $A \doteq B$ if their confidences, when expressed in canonical form, match, i.e., $c_a = c_b$. With this, to check the commutativity of an operation $\star$ that returns an `UBoolean` value, we must prove that $A \star B \doteq B \star A$. This extended definition is backward-compatible: if an extended operation is commutative for all

`UBoolean` values, it will be commutative for Booleans, too, since Booleans are particular instances of UBooleans (i.e., Booleans are UBooleans with $c = 1$ or $c = 0$).

Regarding the algebraic properties of the operations for type `UBoolean`, in the following we will assume all `UBoolean` variables are in the canonical form. The following properties hold for every $A = (\texttt{true}, a)$ and $B = (\texttt{true}, b)$ of type `UBoolean`.

`not(not(A))` $\doteq A$
Proof.

- If $A =$ `null` or A`.oclIsInvalid()` (i.e., A`.oclIsUndefined()`) then, by the specification of the `not` operation, we have that `not(`A`)=`A. Therefore, `not(not(`A`))=`A and hence `not(not(`A`)` $\doteq A$.
- If $A = (\texttt{true}, a)$ then `not(`A`)` $= (\texttt{true}, 1 - a)$, by the specification of the `not` operation. Applying this again, `not(not(`A`))` $=$ `not(true,`$1 - a$`)` $= (\texttt{true}, 1 - (1 - a)) = (\texttt{true}, a) = A$. □

`not(A or B)` $\doteq$ `not(A) and not(B)` (AND Morgan's Law)
Proof.

- If $A = (\texttt{true}, a)$ and $B = (\texttt{true}, b)$ (i.e., none of them is undefined), then `not(`A `or` B`)` $=$ `not(true,`$a + b - ab$`)`, by the specification of the `or` operation. Applying the definition of the `not` operation, `not(true,`$a + b - ab$`)` $= (\texttt{true}, 1 - a - b + ab) = (\texttt{true}, (1 - a)(1 - b))$. In turn, `not(`$A$`) and not(`$B$`)` $= (\texttt{true}, 1 - a)$ `and` $(\texttt{true}, 1 - b) = (\texttt{true}, (1 - a)(1 - b))$ applying the definition of `and`.
- If $A = (\texttt{true}, 1)$ and B`.oclIsUndefined()` (i.e., one of them is undefined), then `not(`A `or` B`)` $=$ `not(true,1)`, by the specification of the `or` operation, because according to the OCL standard, "true OR-ed with anything is true" [6, §7.4.13]. Applying the definition of the `not` operation, `not(true,1)` $= (\texttt{true}, 0)$. In turn, `not(`A`)` $= (\texttt{true}, 0)$ and, according to the OCL standard, "false AND-ed with anything is false" [6, §7.4.13]. Therefore, `not(`A`) and not(`B`)`$= (\texttt{true}, 0)$.
- Finally, if $A = (\texttt{true}, a)$, with $a < 1$, and B`.oclIsUndefined()`, then both `not(`A `or` B`)` and `not(`A`) and not(`B`)` result in an undefined value, and hence are equal. □

`not(A and B)` $\doteq$ `not(A) or not(B)` (OR Morgan's Law)
Proof. This proof is similar.

- If $A = (\texttt{true}, a)$ and $B = (\texttt{true}, b)$ (i.e., none of them is undefined), then `not(`A `and` B`)` $=$ `not(true,`ab`)` $= (\texttt{true}, 1 - ab)$, by the specification of the `and` operation. Applying the definition of the `not` operation, `not(true,`ab`)` $= (\texttt{true}, 1 - ab)$. In turn, `not(`$A$`) or not(`$B$`)` $= (\texttt{true}, 1 - a)$ `or` $(\texttt{true}, 1 - b) = (\texttt{true}, (1 - a) + (1 - b) - (1 - a)(1 - b)) = (\texttt{true}, 1 - ab)$ applying the definition of operation `or`.

- If $A = (\texttt{true}, 0)$ and B`.oclIsUndefined()` (i.e., one of them is undefined), then `not(`A `and` B`)` = `not(true,0)`, by the specification of the `and` operation, because according to the OCL standard, "false AND-ed with anything is false" [6, §7.4.13]. Applying the definition of the `not` operation, `not(true,0)` $= (\texttt{true}, 1)$. In turn, `not(`A`)` $= (\texttt{true}, 1)$ and, according to the OCL standard, "true OR-ed with anything is true" [6, §7.4.13]. Therefore, `not(`A`) or not(`B`)` $= (\texttt{true}, 1)$.
- Finally, if $A = (\texttt{true}, a)$, with $a < 1$, and B`.oclIsUndefined()`, then both `not(`A `or` B`)` and `not(`A`) and not(`B`)` result in an undefined value, and hence are equal. □

Operation ***and*** *is commutative.*

- If $A = (\texttt{true}, a)$ and $B = (\texttt{true}, b)$ (i.e., none of them is undefined), then $(A$ `and` $B) = (\texttt{true}, ab) = (\texttt{true}, ba) = (B$ `and` $A)$ because the multiplication operation on real values is commutative.
- If $A = (\texttt{true}, 0)$ and B`.oclIsUndefined()` then `(`A `and` B`)` = `(true,0)`, by the specification of the `and` operation, because according to the OCL standard, "false AND-ed with anything is false" [6, §7.4.13]. Given that the Boolean operation `and` is commutative, $(B$ `and` $A) = (\texttt{true}, 0)$ for the same reason.
- Finally, if $A = (\texttt{true}, a)$, with $a < 1$, and B`.oclIsUndefined()`, or both A and B are undefined, then both `(`A `and` B`)` and `(`B `and` A`)` equate to an undefined value, and hence they are equal. □

Operation ***and*** *is associative.*

- If $A = (\texttt{true}, a)$, $B = (\texttt{true}, b)$ and $C = (\texttt{true}, c)$ (i.e., none of them is undefined), then $((A$ `and` $B)$ `and` $C) = ((\texttt{true}, ab)$ `and` $(\texttt{true}, c) = (\texttt{true}, abc) = (A$ `and` $(B$ `and` $C))$ because the multiplication operation on real values is associative.
- If one or more of the three operands are undefined, the proof is similar to the one we used to prove that this operation is commutative in this case. □

Operation ***and*** *has* $(\texttt{true}, 1)$ *as identity element.*

- If $A = (\texttt{true}, a)$ then: $(A$ `and` $(\texttt{true}, 1)) = ((\texttt{true}, 1)$ `and` $A) = (\texttt{true}, a)$.
- If A is undefined, $(A$ `and` $(\texttt{true}, 1))$ is undefined, too (cf. [6, §7.4.13]). □

Operation ***or*** *is commutative.*

- If $A = (\texttt{true}, a)$ and $B = (\texttt{true}, b)$ (i.e., none of them is undefined), then $(A$ `or` $B) = (\texttt{true}, a+b-ab) = (\texttt{true}, b+a-ba) = (B$ `or` $A)$ because both the addition and multiplication operations on real values are commutative.

- If $A = (\texttt{true}, 1)$ and B`.oclIsUndefined()` then (*A* `or` *B*) = `(true,1)`, by the specification of the `or` operation, because according to the OCL standard, "true OR-ed with anything is true" [6, §7.4.13]. Given that the Boolean operation `or` is commutative, $(B \texttt{ or } A) = (\texttt{true}, 1)$ for the same reason.
- Finally, if $A = (\texttt{true}, a)$, with $a < 1$, and B`.oclIsUndefined()`, or both A and B are undefined, then both `(`A `or` B`)` and `(`B `or` A`)` equate to an undefined value, and hence they are equal. □

Operation `or` *is associative.*

- If $A = (\texttt{true}, a)$, $B = (\texttt{true}, b)$ and $C = (\texttt{true}, c)$ (i.e., none of them is undefined), then $((A \texttt{ or } B) \texttt{ or } C) = ((\texttt{true}, a + b - ab) \texttt{ and } (\texttt{true}, c) = (\texttt{true}, a + b + c - ab - ac - bc + abc) = (A \texttt{ or } (B \texttt{ or } C))$ because the addition and multiplication operations on real values are associative.
- If one or more of the three operands are undefined, the proof is similar to the one we used to prove that this operation is commutative. □

Operation `or` *has* $(\texttt{true}, 0)$ *as identity element.*

- If $A = (\texttt{true}, a)$ then: $(A \texttt{ or } (\texttt{true}, 0)) = ((\texttt{true}, 0) \texttt{ and } A) = (\texttt{true}, a)$.
- If A is undefined, $(A \texttt{ or } (\texttt{true}, 0))$ is undefined, too (cf. [6, §7.4.13]). □

"AND complement" property (`A and not(A)`$\doteq$`(false,`1`)`*).*
This property does not hold in all cases. However, we can always affirm that `A and not(A)`$\doteq$`(false,`c`)`, with $c \geq 0.75$. To prove this, we assume that all values are in normal form. Then, we have that `A and not(A)` $\doteq$ `(true,`c`) and (true,1`$-c$`)` $\doteq$ `(true,` c`*(1`$-c$`))` $\doteq$ `(false,1`$-c$`*(1`$-c$`))`. However, $1 - c * (1 - c)$ is a function whose minimum is 0.75 (which is attained for $c = 0.5$).

"OR complement" property (`A or not(A)`$\doteq$`(true,`1`)`*).*
A similar result can be proved for the "OR complement" property (`A or not(A)` $\doteq$ `true`), for which we can only affirm, in case of uncertain Booleans, that `A or not(A)` $\doteq$ `(true,`c`)`, with $c \geq 0.75$.

Secondary operations on `UBoolean`*.*
The secondary operations (`xor`, `implies` and `equivalent`) of type `UBoolean` work with `Boolean` values as before and respect their properties, even when lifted to `UBoolean` values. In particular:

- Operation `implies` is non-commutative and associative, since `(A implies B)` $\doteq$ `(not A or B)`.
- Operation `equivalent` is commutative, associative, and its identity element is $(\texttt{true}, 1)$. To check whether `equivalent` is an equivalence relation or not, we must prove that it is reflexive, symmetric and transitive.
  - It is reflexive because `A equivalent A`$\doteq (\texttt{true}, 1)$.

  - Symmetry holds because the operation is commutative on `Boolean` values.
  - Transitivity cannot be ensured: assuming $a \geqslant b \geqslant c$, `(`$A$ `equivalent` B`) and (`B `equivalent` C`)`$\doteq (\texttt{true}, (1-b+c)+(a-b)(b-c))$, while `(`$A$ `equivalent` C`)`$\doteq (\texttt{true}, (1-b+c)$. They coincide in case of `Boolean` values, or if any of the two values are equal. .
- Similarly, `xor` is commutative, associative, and its identity element is $(\texttt{false}, 1)$. However, `(A xor A)` $\doteq (\texttt{false}, c(2-c))$.

1.3 Type-A specification of `UBoolean`

We have also specified an alternative implementation of these operations, in case no assumption can be made about the independence of the variables in a Boolean expression. It is based on the Monte-Carlo simulation method that was proposed in [4] for Type-A measurement uncertainty in real numbers and has been adapted to Boolean values. Basically, every `UBoolean` value contains a sequence of `Boolean` values that represent the sample that is obtained when measuring that value. Operations are performed on the samples, and b and c become derived values. Boolean value `true` is lifted to a sequence where all the samples are `true`, and similarly for `false`. An additional invariant, which is at the end of the list, requests that all samples be of the same size.

```
class UBoolean_A
-- canonical form: triplet (sample[],c), with:
--   sample: the set of measured values obtained for self
--   c: the confidence that self is true
attributes
   sample : Sequence(Boolean)
   b : Boolean derive: true
   c : Real derive: self.sample->count(true)/self.sample->size()
operations
not() : UBoolean_A
  post: (result.c = 1-self.c) and
        (Sequence{1..self.sample->size}->forAll(i|
            result.sample->at(i)=not self.sample->at(i)))
and(b : UBoolean_A) : UBoolean_A
  post: (Sequence{1..self.sample->size}->forAll(i|
           result.sample->at(i) =
              (self.sample->at(i) and b.sample->at(i))))
or(b : UBoolean_A) : UBoolean_A
  post: (Sequence{1..self.sample->size}->forAll(i|
           result.sample->at(i) =
              (self.sample->at(i) or b.sample->at(i))))
implies(b : UBoolean_A) : UBoolean_A
  post: (Sequence{1..self.sample->size}->forAll(i|
           result.sample->at(i) =
              (self.sample->at(i) implies b.sample->at(i))))
xor(b : UBoolean_A) : UBoolean_A
  post: (Sequence{1..self.sample->size}->forAll(i|
           result.sample->at(i) =
              (self.sample->at(i) xor b.sample->at(i))))
equivalent(b : UBoolean_A) : UBoolean_A
```

```
  post: (Sequence{1..self.sample->size}->forAll(i|
           result.sample->at(i) = not
              (self.sample->at(i) xor b.sample->at(i))))
equals(b : UBoolean_A) : Boolean = (self.c=b.c)
equalsC(b : UBoolean_A, c: Real) : Boolean =
   ((self.c-b.c).abs()<=(1-c))
distinct(b : UBoolean_A) : Boolean = not (self.equals(b))
context UBoolean_A inv SameSampleSize:
UBoolean_A.allInstances->forAll(u,v|u.sample->size=v.sample->size)
```

This specification respects the algebraic properties of the operations, since what the newly specified operations only transfer the operations to the corresponding `Boolean` ones in the samples.

2 Extending type `Real`

2.1 Specification of the extended values and operations

The values of `UReal` are pairs of `Real` numbers $X = (x, u)$. They determine the expected value (x) and associated standard uncertainty (u) of quantity X. Real numbers x are naturally injected into type `UReal` by pairs $(x, 0)$.

The following list presents the specifications of the `UReal` operations:

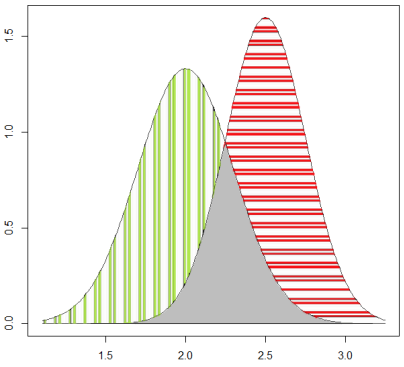
```
-- ARITHMETIC OPERATIONS
context UReal::add(r : UReal) : UReal
post: result.x=self.x + r.x and
      result.u=(self.u*self.u + r.u*r.u).sqrt()
context UReal::minus(r : UReal) : UReal
post: result.x=self.x - r.x and
      result.u=(self.u*self.u + r.u*r.u).sqrt()
context UReal::mult(r : UReal) : UReal
post: result.x=(self.x*r.x) and
      result.u=(r.u*r.u*self.x*self.x +
                self.u*self.u*r.x*r.x).sqrt()
context UReal::divideBy(r : UReal) : UReal
pre:  (r.x - r.u).max(0) > (r.x + r.u).min(0) --not r.equals(0,0)
post: result.x=self.x/r.x and
      result.u=if (r=self) then 0.0
               else if (r.u=0.0) then  -- r scalar, self may be
                     self.u/r.x
               else if (self.u=0.0) then -- self scalar, r is not
                     r.u/(r.x*r.x)
               else ((self.u*self.u/r.x)+((r.u*r.u*self.x*self.x)/
                     (r.x*r.x*r.x*r.x))).sqrt()
             endif endif endif
-- FURTHER OPERATIONS
context UReal::abs() : UReal
post: result.x=self.x.abs() and
      result.u=self.u
context UReal::neg() : UReal
post: result.x=-self.x and
      result.u=self.u
context UReal::floor() : UReal
post: result.x=self.x.floor() and
      result.u=self.u
```

```
context UReal::round() : UReal
post: result.x=self.x.round() and
      result.u=self.u
context UReal::sqrt() : UReal
post: result.x=self.x.sqrt() and
      result.u=self.u/(2*self.x.sqrt())
context UReal::power (s : Real) : UReal
post: result.x=self.x.power(s) and
      result.u=s*self.u*self.x.power(s-1)
-- COMPARISON OPERATIONS
context UReal::equals(r : UReal) : Boolean
   = (self.x - self.u).max(r.x - r.u) <=
        (self.x + self.u).min(r.x + r.u)
context UReal::distinct(r : UReal) : Boolean
    = not self.equals(r)
context UReal::compareTo(r : UReal) : Integer
    = if self.equals(r) then 0
      else if self.lt(r) then -1 else 1 endif endif
context UReal::lt(r : UReal) : Boolean
   = (self.x<r.x) and ((self.x + self.u)<(r.x - r.u))
context UReal::le(r : UReal) : Boolean
   = self.lt(r) or self.equals(r)
context UReal::gt(r : UReal) : Boolean = not self.le(r)
context UReal::ge(r : UReal) : Boolean = not self.lt(r)
context UReal::max(r : UReal) : UReal
    = if r.lt(self) then self else r endif
context UReal::min(r : UReal) : UReal
    = if r.lt(self) then r else self endif
-- TRIGONOMETRIC OPERATIONS
context UReal::sin() :UReal
post: result.x=self.x.sin() and
      result.u=(self.x.cos()*self.u).abs()
context UReal::cos() :UReal
post: result.x=self.x.cos() and
      result.u=(self.x.sin()*self.u).abs()
context UReal::tan():UReal
post: result.x=self.x.tan() and
      result.u= self.u / (self.x.cos()*self.x.cos())
context UReal::atan() :UReal
post: result.x=self.x.atan() and
      result.u=self.u / (1 + self.x*self.x)
context UReal::asin() :UReal
post: result.x=self.x.asin() and
      result.u=self.u / ((1 - self.x*self.x).sqrt())
context UReal::acos() :UReal
post: result.x=self.x.acos() and
      result.u=self.u / ((1 - self.x*self.x).sqrt())
```

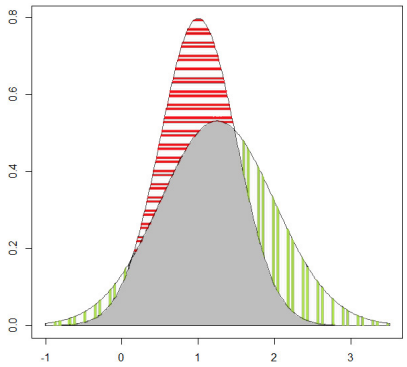
In addition to the traditional comparison operations between uncertain reals ($=, \neq, <, \leq, >, \geq$), which return `Boolean` values, comparisons between real numbers with uncertainty could also return uncertain Boolean values.

Then, given two `UReal` values x and y, we define three real numbers, namely, (l, e, g), that represent, respectively, the probability of x being less than, equal to or greater than y. Expression $l + e + g = 1$ always holds.

For example, the triplet that we obtain for values a and b (Fig. 1(a)) is $(0.893, 0.106, 1.11 \cdot 10^{-16})$, which specifies that $a < b$ with probability 0.893,



((a)) Representation of $a = 2.0 \pm 0.3$ and $b = 2.5 \pm 0.25$.



((b)) Representation of $c = 1.0 \pm 0.5$ and $d = 1.25 \pm 0.75$.

Fig. 1: Graphical representations of `UReal` values.

$a = b$ with probability 0.106, and $a > b$ with a probability $1.11 \cdot 10^{-16}$. Similarly, the triplet for c and d (Figure 1(b)) is $(0.152, 0.754, 0.094)$. These 3 numbers correspond to the three areas into which the curve that represents the first of the values can be divided (this is clearer in Figure 1(b), where the three areas of the curve that represents `UReal` number d can be easily distinguished: the left and right areas with vertical stripes represent the areas where $c < d$ and $c > d$, respectively, and the central area (in gray) represents the area where both numbers coincide; it corresponds to the intersection of both curves).

All this has been specified in OCL using an auxiliary operation, namely, `calculate(r:UReal)`, which returns a tuple with the triplet. With it, the comparison operations between `UReal` numbers are specified as follows:

```
context UReal::lt(r :UReal) :UBoolean -- less than
    post: (result.b) and
          (result.c = self.calculate(r).l)
context UReal::le(r :UReal) :UBoolean -- equal or less than
    post: (result.b) and
          (result.c = let x:Tuple(l:Real,e:Real,g:Real)=
                            self.calculate(r) in
                      x.l + x.e)
context UReal::gt(r :UReal) :UBoolean -- greater than
    post: (result.b) and
          (result.c=self.calculate(r).g)
context UReal::ge(r :UReal) :UBoolean -- greater or equal
    post: (result.b) and
          (result.c=let x:Tuple(l:Real,e:Real,g:Real)=
                        self.calculate(r) in x.g + x.e)
context UReal::uEquals  (r :UReal) :UBoolean
    post: (result.b) and (result.c = self.calculate(r).e)
context UReal::uDistinct(r :UReal) :UBoolean
    post: (result.b) and (result.c = 1.0 - self.uEquals(r))
```

Operation `calculate()` is specified in OCL/SOIL as follows:

```
context UReal::calculate(r:UReal) :Tuple(lt:Real,eq:Real,gt:Real)
begin
  declare m1 : Real, m2 : Real, s1 : Real, s2 : Real,
          crossing1 : Real, crossing2 : Real,
          r1 : Real, r2 : Real, c1 : Real, c2 : Real,
          aux : Tuple(lt:Real,eq:Real,gt:Real), swap : Boolean;
  if (self.x <= r.x) then
    m1:=self.x; m2:=r.x; s1:=self.u; s2:=r.u; swap:=false;
  else
    m2:=self.x; m1:=r.x; s2:=self.u; s1:=r.u; swap:=true;
  end;
  if (s1=0.0) and (s2=0.0) then  -- Real numbers
    if (m1=m2) then aux:=Tuple{lt:0.0,eq:1.0,gt:0.0};
    else if (m1<m2) then aux:=Tuple{lt:1.0,eq:0.0,gt:0.0};
         else aux:=Tuple{lt:0.0,eq:0.0,gt:1.0};
         end
    end
  else
    if ((s1=0.0)) then --- s1 is degenerated, s2 is not
      aux:=Tuple{lt:1.0-m1.cdf(m2,s2),eq:0.0,gt:m1.cdf(m2,s2)};
    else
      if ((s2=0.0))  then  --- s2 is degenerated, s1 is not
        aux:=Tuple{lt:m2.cdf(m1,s1),eq:0.0,gt:1.0-m2.cdf(m1,s1)};
      else --- none of the two numbers are degenerated.
        --- This is where the fun starts...
        if (s1=s2) then
          crossing1 := (m1+m2)/2;
          aux:=Tuple{
            lt:crossing1.cdf(m1,s1)-crossing1.cdf(m2,s2),
            eq:crossing1.cdf(m2,s2)+1.0-crossing1.cdf(m1,s1),
            gt:1.0-crossing1.cdf(m2,s2)-(1.0-crossing1.cdf(m1,s1))
            };
          else
            r1:=(s2/s1).log();
            r2:=((m1-m2)*(m1-m2)-2.0*(s1*s1-s2*s2)*r1).sqrt();
            crossing1:=-(-m2*s1*s1+m1*s2*s2+s1*s2*r2)/(s1*s1-s2*s2);
            crossing2:=(m2*s1*s1-m1*s2*s2+s1*s2*r2)/(s1*s1-s2*s2);
            c1:=crossing1.min(crossing2);
            c2:=crossing1.max(crossing2);
            if (s1<s2) then
              aux:=Tuple{
                lt:1.0-c2.cdf(m2,s2)-(1.0-c2.cdf(m1,s1)),
                eq:c1.cdf(m1,s1) + (1.0-c2.cdf(m1,s1)) +
                   c2.cdf(m2,s2) - c1.cdf(m2,s2),
                gt:c1.cdf(m2,s2)-c1.cdf(m1,s1)
                };
              else
                aux:=Tuple{
                  lt:c1.cdf(m1,s1)-c1.cdf(m2,s2),
                  eq:c1.cdf(m2,s2) + (1.0-c2.cdf(m2,s2)) +
                     c2.cdf(m1,s1) - c1.cdf(m1,s1),
                  gt:1.0-c2.cdf(m1,s1)-(1.0-c2.cdf(m2,s2))
                  };
              end
          end --- end of fun
      end
    end
  end;
```

```
  if (swap) then result:=Tuple{lt:aux.gt,eq:aux.eq,gt:aux.lt};
  else result:=Tuple{lt:aux.lt,eq:aux.eq,gt:aux.gt};
  end
end -- calculate()
```

Basically, operation `calculate` computes the areas of the curves that represent the Normal distributions of the `UReal` numbers.

For simplicity, in these expressions we assume that the variables are independent and use a closed-form solution to compute the aggregated measurement uncertainty. As in the case of type `UBoolean`, an alternative specification uses samples to implement a Type A evaluation of uncertainty (see Section 2.3, where it is specified).

2.2 Algebraic properties of the operations

The algebraic properties of the operations of type `UReal` follow.

Operation `+` is commutative.
$X_1 + X_2 = (x_1, u_1) + (x_2, u_2) = (x_1 + x_2, \sqrt{u_1^2 + u_2^2}) = (x_2 + x_1, \sqrt{u_2^2 + u_1^2})$
$= (x_2, u_2) + (x_1, u_1) = X_2 + X_1$

Operation `+` is associative.
$X_1 + (X_2 + X_3) = (x_1, u_1) + (x_2 + x_3, \sqrt{u_2^2 + u_3^2})$
$= (x_1 + (x_2 + x_3), \sqrt{u_1^2 + (\sqrt{u_2^2 + u_3^2})^2})$
$= (x_1 + x_2 + x_3, \sqrt{u_1^2 + u_2^2 + u_3^2})$
$= ((x_1 + x_2) + x_3, \sqrt{(\sqrt{u_1^2 + u_2^2})^2 + u_3^2}) = (x_1 + x_2, \sqrt{x_1^2 + x_2^2}) + (x_3, u_3)$
$= (X_1 + X_2) + X_3$

The identity of operation `+` is $(0.0, 0.0)$.
$X_1 + (0, 0) = (x_1, u_1) + (0, 0) = (x_1, \sqrt{u_1^2 + 0^2}) = (x_1, u_1) = X_1$

Operation `*` is commutative.
$X_1 * X_2 = (x_1, u_1) * (x_2, u_2) = (x_1 * x_2, \sqrt{x_1^2 * u_2^2 + x_2^2 * u_1^2})$
$= (x_2 * x_1, \sqrt{x_2^2 * u_1^2 + x_1^2 * u_2^2}) = (x_2, u_2) * (x_1, u_1) = X_2 * X_1$

Operation `*` is associative.
$X_1 * (X_2 * X_3) = (x_1, u_1) * (x_2 * x_3, \sqrt{x_2^2 * u_3^2 + x_3^2 * u_2^2})$
$= (x_1 * (x_2 * x_3), \sqrt{x_1^2 * (x_2^2 * u_3^2 + x_3^2 * u_2^2) + u_1^2 * x_2^2 * x_3^2})$
$= (x_1 * x_2 * x_3, \sqrt{x_1^2 * x_2^2 + u_3^3 + x_1^2 * u_2^2 + x_3^3 + u_1^2 * x_2^2 * x_3^2})$
$= ((x_1 * x_2) * x_3, \sqrt{x_3^2 * (x_1^2 * u_2^2 + x_2^2 * u_1^2) + x_1^2 * x_2^2 * u_3^2})$
$= (x_1 * x * 2, \sqrt{x_1^2 * u_2^2 + x_2^2 * u_1^2}) * (x_3, u_3) = (X_1 * X_2) * X_3$

The identity of operation `*` is $(1.0, 0.0)$.
$X_1 * (1, 0) = (x_1, u_1) * (1, 0) = (x_1, \sqrt{u_1^2 * 1^2 + x_1^2 * 0^2}) = (x_1, u_1) = X_1$

The distributive property of `+` *and* `*` *only holds if* $u_1 = 0$*, i.e., the multiplicator is a Real number*

$X_1 * (X_2 + X_3) = (x_1, u_1) * ((x_2, u_2) + (x_3, u_3))$
$= (x_1, u_1) * (x_2 + x_3, \sqrt{u_2^2 + u_3^2})$
$= (x_1x_2 + x_1x_3, \sqrt{u_1^2(x_2 + x_3)^2 + x_1^2(u_2^2 + u_3^2)}\,)$
$= (x_1x_2 + x_1x_3, \sqrt{u_1^2x_2^2 + u_1^2x_3^2 + \mathbf{2u_1^2x_2x_3} + x_1^2u_2^2 + x_1^2u_3^2}\,)$
$\neq (x_1x_2 + x_1x_3, \sqrt{(x_1^2u_2^2 + x_2^2u_1^2) + (x_1^2u_3^2 + x_3^2u_1^2)}\,)$
$= (x_1x_2, \sqrt{x_1^2u_2^2 + x_2^2u_1^2}) + (x_1x_3, \sqrt{x_1^2u_3^2 + x_3^2u_1^2}) = X_1 * X_2 + X_1 * X_3.$

In general, the uncertainty of $X_1 * (X_2 + X_3)$ is not the same as that of $X_1 * X_2 + X_1 * X_3$.

Operations minus (−) and division (/).
Operations − and `/` exhibit the same properties as their `Real`-type counterparts. However, when combined with operations `+` and `*`, their behaviors change because `UReal` values accumulate uncertainty in every operation application. For instance, if X and Y are of type `UReal`, then $(X - Y) + (Y - X) \neq (0.0, 0.0)$ unless the uncertainties in X and Y are both 0. In this context, formulas that involve `UReal` values should be algebraically simplified, if needed, before the final results are computed.

Reciprocal and opposite of a `UReal` *number.*
The propagation of uncertainty through operations also influences the behavior of inverse and reciprocal operations:

The reciprocal (or multiplicative inverse) of a `UReal` number (x, u)—the `UReal` number (y, z) such that $(x, u) * (y, z) = (1.0, 0.0)$—exists if and only if $x \neq 0.0$ and $u = 0.0$, i.e., for non-null `Real` numbers, and coincides with $(1/x, 0.0)$. This is because the uncertainty is always non-negative and can only grow when propagated through operations.

Similarly, the opposite (or additive inverse) of a `UReal` number (x, u)—the `UReal` number (y, z) such that $(x, u) + (y, z) = (0, 0)$—exists if and only if $x \neq 0$ and $u = 0$, i.e., for non-null `Real` numbers, and coincides with $(-x, 0.0)$.

Equality of `UReal` *values.*
The properties of the equality operations `equals()`, `distinct()`, "=" (also called `uEquals()`), and "<>" (or `uDistinct()`) that are defined on `UReal` values are as follows:

- Operations `equals()` (that corresponds to $\doteq$) and `distinct()` return `Boolean` values with the result of the comparison of the two `UReal` values as pairs of numbers. That is is, $(x, u) \doteq (y, z)$ iff $x = y \wedge u = z$. Therefore, operation `equals` is reflexive, symmetric and transitive, while operation `distinct` is anti-reflexive, symmetric and not transitive—like their `Real` counterparts.

– The behaviors of operations = (or `uEquals`) and `<>` (`uDistinct`), which return `UBoolean` values, coincide by construction with that of operations = and <> when applied to `Real` values, but their properties differ when used with `UReal` numbers. In particular, operation "=" (`uEquals`) is reflexive and symmetric, but not necessarily transitive.

First, `uEquals()` is reflexive because `x.uEquals(x)=(true,1.0)` always holds, given that the two normal curves coincide. See also the specification of operation `calculate`.

Second, `uEquals()` is symmetric because the operation `calculate`, which computes the area of the intersection of both curves, is symmetric.

To show that `uEquals()` is not necessarily transitive, consider `UReal` numbers $X = (3.0, 1.0)$, $Y = (4.0, 1.0)$ and $Z = (5.0, 1.0)$. In this case we have that `uEquals(`X, Y`)=(true,`0.617`)`, `uEquals(`Y, Z`)=(true,`0.617`)` and that `uEquals(`X, Z`) = (true,`0.317`)`. Transitivity, in the context of `UBoolean` values, can be stated as `(`X `uEquals` Y`) and (`Y `uEquals` Z`) implies (`X `uEquals` Z`)`, which, in this case, evaluates to `(true,`0.577`)`, which does not coincide with `(true,1)`.

Comparison operations.
Finally, the behaviors of operations $<$, $\leq$, $>$ and $\geq$, which return `UBoolean` values, coincide with the behavior of the `Real` operations of the same name when they are applied to `Real` values (cf. the specification of operation `calculate`); however, their behaviors may differ when used with `UReal` values, mainly because, in that case, we are assigning probabilities (numbers in the range $[0, 1]$) to these relationships.

Operations $\leq$ and $\geq$ are reflexive because $(X \leq X) \doteq (X \geq X) \doteq (true, 1)$; however, we cannot guarantee that they are antisymmetric or transitive unless they are applied to `Real` numbers, because these operators return probabilities—which are 0 or 1 only in case the numbers are of equal, of type `Real` or very distant apart. Similar results are obtained for operations $<$ and $>$, since we are dealing here with probabilities. This can be proved via the same arguments that were used to prove the properties of the equality operations on `UReal` values.

2.3 Type A specifications of `UReal` numbers

The ISO GUM [4] also defines al alternative implementation of the uncertainty of real values, when it cannot be assumed that the variation of the values of the variables do not follow a Normal distribution and the variables are independent.

The implementation is based on the Monte-Carlo simulation method that is proposed in [4] for Type-A measurement uncertainty in real numbers. Basically, every `UReal` value contains a sequence of `Real` values that represent the sample that is obtained when measuring that value. Operations are performed

on the samples, and x and u become derived values (the average and standard deviation of the samples, respectively). An additional invariant requests that all samples be of the same size.

```
context UReal_A inv SameSampleSize:
UReal_A.allInstances->forAll(x,y|x.sample->size=y.sample->size)
class UReal_A
attributes
   sample : Sequence(Real)
   x : Real derive: self.sample->avg()
   u : Real derive: self.sample->stdDev()
-- ARITHMETIC OPERATIONS
context UReal_A::add(r : UReal_A) : UReal_A
post: (Sequence{1..self.sample->size}->
        forAll(i|result.sample->at(i)=
          (self.sample->at(i) + r.sample->at(i))))
context UReal_A::minus(r : UReal_A) : UReal_A
post: (Sequence{1..self.sample->size}->
        forAll(i|result.sample->at(i)=
          (self.sample->at(i) - r.sample->at(i))))
context UReal_A::mult(r : UReal_A) : UReal_A
post: (Sequence{1..self.sample->size}->
        forAll(i|result.sample->at(i)=
          (self.sample->at(i) * r.sample->at(i))))
context UReal_A::divideBy(r : UReal_A) : UReal_A
pre:  (Sequence{1..self.sample->size}->
        forAll(i|r.sample->at(i)<>0.0))
post: (Sequence{1..self.sample->size}->
        forAll(i|result.sample->at(i)=
          (self.sample->at(i) / r.sample->at(i))))
-- FURTHER OPERATIONS
context UReal_A::abs() : UReal_A
post: (Sequence{1..self.sample->size}->
      forAll(i|result.sample->at(i)=(self.sample->at(i)).abs()))
context UReal_A::neg() : UReal_A
post: (Sequence{1..self.sample->size}->
      forAll(i|result.sample->at(i)=(-self.sample->at(i))))
context UReal_A::floor() : UReal_A
post: (Sequence{1..self.sample->size}->
      forAll(i|result.sample->at(i)=(self.sample->at(i)).floor()))
context UReal_A::round() : UReal_A
post: (Sequence{1..self.sample->size}->
      forAll(i|result.sample->at(i)=(self.sample->at(i)).round()))
context UReal_A::sqrt() : UReal_A
post: (Sequence{1..self.sample->size}->
      forAll(i|result.sample->at(i)=(self.sample->at(i)).sqrt()))
context UReal_A::power (s : Real) : UReal_A
post: (Sequence{1..self.sample->size}->
      forAll(i|result.sample->at(i)=
          (self.sample->at(i)).power(s)))
-- COMPARISON OPERATIONS
context UReal_A::equals(r : UReal_A) : Boolean
   = (self.x - self.u).max(r.x - r.u) <=
        (self.x + self.u).min(r.x + r.u)
context UReal_A::distinct(r : UReal_A) : Boolean
    = not self.equals(r)
context UReal_A::compareTo(r : UReal_A) : Integer
    = if self.equals(r) then 0
```

```
      else if self.lt(r) then -1
           else 1 endif endif
context UReal_A::lt(r : UReal_A) : Boolean
   = (self.x<r.x) and ((self.x + self.u)<(r.x - r.u))
context UReal_A::le(r : UReal_A) : Boolean
    = self.lt(r) or self.equals(r)
context UReal_A::gt(r : UReal_A) : Boolean = not self.le(r)
context UReal_A::ge(r : UReal_A) : Boolean = not self.lt(r)
context UReal_A::max(r : UReal_A) : UReal_A
    = if r.lt(self) then self else r endif
context UReal_A::min(r : UReal_A) : UReal_A
    = if r.lt(self) then r else self endif
-- TRIGONOMETRIC OPERATIONS
context UReal_A::sin() :UReal_A
post: (Sequence{1..self.sample->size}->
       forAll(i|result.sample->at(i)=(self.sample->at(i)).sin()))
context UReal_A::cos() :UReal_A
post: (Sequence{1..self.sample->size}->
       forAll(i|result.sample->at(i)=(self.sample->at(i)).cos()))
context UReal_A::tan():UReal_A
post: (Sequence{1..self.sample->size}->
       forAll(i|result.sample->at(i)=(self.sample->at(i)).tan()))
context UReal_A::atan() :UReal_A
post: (Sequence{1..self.sample->size}->
       forAll(i|result.sample->at(i)=(self.sample->at(i)).atan()))
context UReal_A::asin() :UReal_A
post: (Sequence{1..self.sample->size}->
       forAll(i|result.sample->at(i)=(self.sample->at(i)).asin()))
context UReal_A::acos() :UReal_A
post: (Sequence{1..self.sample->size}->
       forAll(i|result.sample->at(i)=(self.sample->at(i)).acos()))
```

This specification respects the algebraic properties of the operations, since the newly specified operations only transfer the operations to the corresponding `UReal` ones in the samples.

2.4 Correlated `UReal` variables

When the variables are correlated, and their covariance is know, it is also possible to represent the behavior of operations using closed-form expressions. In the case of type `UReal`, the specification of operations when the variables are correlated and their covariance is known, is as follows:

```
-- ARITHMETIC OPERATIONS
context UReal::add(r : UReal, cov : Real) : UReal
post: result.x=self.x + r.x and
      result.u=(self.u*self.u+r.u*r.u+2*cov).sqrt()
context UReal::minus(r : UReal, cov : Real) : UReal
post: result.x=self.x - r.x and
      result.u=(self.u*self.u+r.u*r.u-2*cov).sqrt()
context UReal::mult(r : UReal, cov : Real) : UReal
post: result.x=(self.x*r.x) and
      result.u=(r.u*r.u*self.x*self.x +
                self.u*self.u*r.x*r.x +
                2*self.x*r.x*cov).sqrt()
```

```
context UReal::divideBy(r : UReal, cov : Real) : UReal
pre:  (r.x - r.u).max(0) > (r.x + r.u).min(0) --not r.equals(0,0)
post: result.x=if (r=self) then 1.0
              else if (r.u=0.0) then  -- r scalar, self may be
                     self.x/r.x
              else if (self.u=0.0) then --self scalar, r is not
                     self.x/r.x
              else (self.x/r.x + (self.x*r.u*r.u)/(r.x*r.x*r.x))
               endif endif endif
    and
      result.u=if (r=self) then 0.0
              else if (r.u=0.0) then  -- r scalar, self may be
                     self.u/r.x
              else if (self.u=0.0) then -- self scalar, r is not
                     r.u/(r.x*r.x)
              else ((self.u*self.u/r.x.abs()) +
                     ((r.u*r.u*self.x*self.x)/(r.x*r.x*r.x*r.x)) -
                     self.x*cov/((r.x*r.x*r.x).abs())
                     ).sqrt()
               endif endif endif
```

3 Extending type `Integer`

3.1 Specification of the extended values and operations

Type `UInteger` is the supertype of OCL type `Integer` that defines measurement uncertainty. It is needed, for instance, when representing timestamps of events, which are normally expressed in milliseconds and may have uncertain values due to a lack of clock accuracy.

This extension is straightforward. Every `UInteger` element is of the form (n, u), where n is an Integer value and u a Real value that represents its measurement uncertainty. The injection of any `Integer` value n into type `UInteger` is naturally defined by $(n, 0.0)$. The behaviors of `UInteger` operations are defined by lifting the operations to type `UReal` and projecting the corresponding results, if necessary. This, together with the subtyping relationship `Integer` <: `Real` in OCL, ensures the proper subtyping relationship between `Integer` and `UInteger`.

3.2 Algebraic properties of the operations

Given that the behavior of the operations on `UInteger` values is defined by lifting them to the type `UReal`, their algebraic properties coincide with those of the corresponding `UReal` operations.

4 Extending type `UnlimitedNatural`

4.1 Specification of the extended values and operations

An OCL `UnlimitedNatural` is either a non-negative `Integer` or a special *unlimited* value (`*`) that represents the upper value of a multiplicity specification [6]. This special value `*` cannot be used in any arithmetic operation with unlimited naturals; it can only be used with comparison (including max and min) operations.

Excluding value `*`, unlimited naturals are non-negative integers, more precisely: `UnlimitedNatural`$\backslash\{*\}$ <: `Integer`. Although subtraction is not defined in OCL for unlimited naturals, it can be naturally defined as a partial operation, and hence lifted to type `Integer` (and, therefore, to `Real`).

The extension of `UnlimitedNatural` to `UUnlimitedNatural` consists of adding a new component to every unlimited natural value, with the expression of its uncertainty. The uncertainty of special value `*` will be 0.

Operations on `UUnlimitedNatural` values that do not involve special value `*` are defined by lifting them to type `UInteger`. Comparison operations must consider the particular case of special value `*` (internally represented by "−1") and lift the operation to the supertype if this value is not involved. For illustration purposes, the following list provides the OCL specifications of the comparison operations between `UUnlimitedNatural` values.

```
uEquals(r : UUnlimitedNatural) : UBoolean
  post: result=if (self.x<>-1) and (r.x<>-1) then
                 self.toUInteger().uEquals(r.toUInteger())
               else (self.x=-1) and (r.x=-1) endif
lt(r : UUnlimitedNatural) : UBoolean
  post: if (self.x<>-1) and (r.x<>-1) then
             result=self.toUInteger().lt(r.toUInteger())
        else (result.b = ((self.x<>-1) or (r.x=-1)))
             and (result.c=1.0) endif
le(r : UUnlimitedNatural) : UBoolean
  post: result=self.lt(r).or(self.equals(r))
gt(r : UUnlimitedNatural) : UBoolean
  post: result = not self.le(r)
ge(r : UUnlimitedNatural) : UBoolean
  post: result = not self.lt(r)
max(r : UUnlimitedNatural) : UUnlimitedNatural
  post: result = if (self.x=-1) then self
          else if (r.x=-1) then r
             else if r.lt(self).toBoolean() then self
                  else r endif
             endif
          endif
min(r : UUnlimitedNatural) : UUnlimitedNatural
  post: result = if (self.x=-1) then r
          else if (r.x=-1) then self
             else if r.lt(self).toBoolean() then self
                  else r endif
             endif
          endif
```

4.2 Algebraic properties of the operations

Again, given that the behavior of the operations on `UUnlimitedNatural` values is defined by lifting them to the type `UReal`, their algebraic properties coincide with those of the corresponding `UReal` operations.

5 Extending type `String`

5.1 Specification of the extended values and operations

Type `UString` extends type `String` by associating uncertainty to its values, by means of a real number that is in the range $[0, 1]$ and represents the confidence that we have in the contents of the string. Therefore, values of type `UString` are pairs (S, c), where S is the string and c the associated confidence. Values of type `String` are injected into `UString` as $(S, 1.0)$.

To calculate the confidence of a string S we will use the Levenshtein distance [5], which is defined as "the minimum number of changes (insertion, deletion, or substitution of a single character) needed to transform one string into another." Then, if `dist` is a number that represents the number of changes that we estimate that a string S may have, and `l` is the size of the string, the following operations enable the calculation of the corresponding confidence of the string and vice versa: given a confidence, the estimated distance of that string is calculated with respect to its real value.

```
context UString::distToConf(dist:Real, l:Integer) : Real =
                  (1.0-dist/l).max(0.0)
context UString::confToDist() : Real =
                  self.S.size()*(1.0 - self.c)
```

For example, if $S =$ `'Hello world!'`, a confidence of 1 would mean that the string is completely accurate; a confidence of 0.92 (=`1-(1/S.size)`=11/12) would mean that we could allow one change (i.e., a deletion, addition or modification of one character of the String).

In addition to the operations that are defined for OCL type `String`, which can also be applied to `UString` values, Table 1 lists the newly defined operations for Type `UString`. The former operations simply act on the string value, without changing its confidence. In contrast, the operations that are listed in Table 1 operate on both the string value and the confidence.

The following list presents the specifications of these operations.

```
equals(us:UString):Boolean =
      (self.S=us.S) and (self.c=us.c)
uSize() : UInteger
  post: result.x = self.S.size() and
        result.u = self.confToDist()
uConcat(us:UString):UString
  post: result.S = self.S.concat(us.S) and
        result.c = self.distToConf(self.confToDist() +
          us.confToDist(), self.S.size()+us.S.size())
uSubstring(lower:Integer, upper:Integer) : UString
```

Table 1: Extended operations on type `UString`.

Operation	Parameters	Return Value
toString()	-	String
uSize()	-	UInteger
uConcat(), +	s:UString	UString
uSubstring()	lower:Integer, upper:Integer	UString
toInteger()	-	Integer
toReal()	-	Real
toBoolean()	-	Boolean
uToUpperCase()	-	UString
uToLowerCase()	-	UString
indexOf()	s:UString	Integer
equals()	s:UString	Boolean
uEquals()	s:UString	UBoolean
uEqualsIgnoreCase()	s:UString	UBoolean
at()	i:Integer	String
uAt()	i:Integer	UString
uCharacters()	-	USequence(UString)
uToBoolean()	-	UBoolean
lt,gt,le,ge	s:UString	UBoolean

```
  pre validLimits: (1<=lower) and (lower<=upper)
  post: result.S = self.S.substring(lower,upper) and
        result.c = self.c
uEquals (us:UString) : UBoolean
  post: result.b and
        result.c = if (self.S = us.S) then self.c*us.c
                   else 1.0 - self.c * us.c endif
uAt(i:Integer) : UString
  pre validArg: i > 0 and i <= self.size()
  post: result.S = self.substring(i,i) and result.c = self.c
lt(us:UString) : UBoolean
  post: result.b and
        result.c = if (self.S < us.S) then self.c*us.c
                   else 1.0 - self.c * us.c endif
```

5.2 Algebraic properties of the operations

The algebraic properties of the extended operations respect the properties of the `String` operations, except when the combination of the operations specifies concatenations that are followed by substring extractions of `UString` values with different uncertainties. This is due to the way in which the uncertainty of the composite string is calculated and propagated to the substrings (see the OCL specifications of these two operations in the list above).

For example, consider the following three uncertain strings: `S1=('ABC',0.8)`, `S2=('FGHIJ',0.8)` and `S3=('FGHIJ',0.6)`. The operations that concatenate these strings and extract the corresponding substrings produce different results if the uncertainties differ, as shown below:

```
-- when the uncertainty of the substrings is the same
C1 = S1.uConcat(S2) = ('ABCFGHIJ',0.8)
R1 = C1.uSubstring(1,3) = ('ABC',0.8) = S1
```

```
R2 = C1.uSubstring(4,8) = ('FGHIJ',0.8) = S2
-- when the uncertainties of the substrings are different
C2 = S1.uConcat(S3) = ('ABCFGHIJ',0.675)
T1 = C2.uSubstring(1,3) = ('ABC',0.675) <> S1
T2 = C2.uSubstring(4,8) = ('FGHIJ',0.675) <> S2
```

The uncertainty associated to a String refers to the confidence we have on it. Should we need to assign a confidence to individual characters of a String, we can consider the String as a sequence of characters (in OCL, Strings of size 1), and associate a confidence to each one.

6 Extending Enumeration types

Enumerations are user-defined types. An enumeration type is defined by a set of literals $\{l_1, ..., l_n\}$. An enumeration value is one of the literals defined for the type.

Type `UEnum` is the embedding supertype for type `Enum` that adds uncertainty to each of its values. Then, a value of an uncertain enumeration type will no longer be a single literal, but a set of pairs $\{(l_1, c_1), ..., (l_n, c_n)\}$ where $\{c_1, ..., c_n\}$ are numbers that are in the range $[0, 1]$ and that represent the probabilities that the variable takes each literal as its value, and where $\sum_{i=1}^{n} c_i = 1$. For convenience, pairs with a zero confidence can be omitted from the set. Regular (non-uncertain) enumeration values are injected into the extended type by a set with only one pair, which is composed of the literal with a confidence of 1.0.

The only operations that are defined for enumeration types are `equals()` and `literals()`. We extend them to `UEnum` types and add operations `equals()` and `uEquals()`.

Operation `equals()` returns a `Boolean` value that checks if the two sets of pairs are the same. Operation `uEquals()` returns a `UBoolean` value whose confidence is defined by $\sqrt{(\sum_{i=1}^{n}(a_i - b_i)^2)}/2$, where a_i and b_i are the confidences of the literals of the two `UEnum` values that are being compared. The last operation, namely `literals()`, returns the set of literals of the type; therefore, it coincides with the original operation that was defined in OCL for enumeration types.

Logically, to specify these kinds of types, we build one class for every `UEnum` that we want to specify. For example, for enumeration type `Color`, the following listing specifies the corresponding `UColor` class and its operations. The two class invariants at the end check that the values of uncertain enumerations have unique literals, and their confidences are correct, namely, that they all are in the range $[0, 1]$ and their sum is 1.

```
enum Color{White, Red, Blue, Green, Yellow, Black}

class UColor
attributes values:Sequence(Tuple(literal:Color,conf:Real))
operations
equals(ue:UColor):Boolean = self.values->asSet = ue.values->asSet
```

```
conf(lit:Color):Real =
  -- confidence of a literal
  let L:Sequence(Color) = self.values->collect(literal) in
  if L->includes(lit) then self.values->collect(conf)->at(L->
      ↪indexOf(lit))
  else 0.0 endif
literals():Sequence(Color) = self.values->collect(literal)
uEquals(ue:UColor):UBoolean
  post: result.b and
        result.c = if self.equals(ue) then 1.0
          else let L:Sequence(Color) =
            self.literals()->union(ue.literals()) in
            1.0-(L->iterate(l ; s : Real = 0.0 |
                  let x1 : Real = self.conf(l) in
                  let x2 : Real = ue.conf(l) in
                  s + (x2-x1)*(x2-x1))/4 ).sqrt()
-- type invariants
context UColor inv UColorUniqueLiterals:
  self.values->size() = self.values->collect(literal)->asSet->size
context UColor inv UColorProbabilities:
  self.values->collect(conf)->sum()=1.0 and
  self.values->collect(conf)->select(c|c<0.0 or c>1.0)->isEmpty()
```

The extended class is automatically generated from the `UEnum` type. In our implementation of this type in the USE environment, this is why a `UEnum` type is defined in a similar way to an `enum` type: `UEnum` $name\{l_1, ..., l_n\}$; e.g., `UEnum Color{White, Red, Blue, Green, Black}`. Constants are expressed in a compact form, e.g., `UColor{(#Yellow,0.8),(#White,0.2)}`.

7 Extending OCL collections

Two kinds of uncertainty can be considered when extending OCL collections.

a) Collections with uncertain values This first case corresponds to collections whose elements are uncertain values. OCL defines an abstract datatype `Collection`, with a set of operations common to all kinds of collections, plus a set of operations which are specific to each subtype: `Set`, `OrderedSet`, `Bag`, `Sequence`. Table 2 shows the operations supported by OCL collections. The extension consists in extending these operations to deal with uncertain values. As before, they are evaluated in the higher type of the type hierarchy of the elements of the collection—note that this hierarchy includes now the extended datatypes. For instance, if a `Sequence` is composed of values of types `Real` and `UReal`, the type of the collection would be `Set(UReal)` and the corresponding operations will be evaluated within this type; e.g., operation `sum()` will return a `UReal` value. Similarly, logic predicates in collection operations that return `Boolean` values (such as `forAll`, `exists` or `isUnique`) might now be of type `UBoolean`, and therefore the operations may also return a `UBoolean` value—for example, when deciding whether all the elements of a set of `UReal` values are greater than a given number. However, we do not allow logic predicates of type `UBoolean` to act as filters to select elements of the collections, since

Table 2: OCL collections and their operations.

Type	Operations
Collection	select(), reject(), collect(), collectNested(), forAll(), exists(), isUnique(), one(), any(), closure(), iterate(), sortedBy(), =, <>, size(), includes(), excludes(), includesAll(), isEmpty(), excludesAll(), notEmpty(), max(), min(), sum(), product(), selectByKind(), selectByType(), asSet(), asBag(), asOrderedSet(), asSequence(), flatten(), count()
Set	union(), intersection(), -(), including(), excluding(), symmetricDifference(),
OrderedSet	append(), prepend(), insertAt(), subOrderedSet(), at(), indexOf(), first(), last(), reverse()
Bag	union(), intersection(), including(), excluding(),
Sequence	union(), append(), prepend(), insertAt(), subSequence(), at(), indexOf(), first(), last(), reverse()

we need to clearly decide whether an element belongs or not to the collection; this is the case, for instance, of operations `select`, `any`, or `collect`. Operation `confidence()`, which allows to know the confidence of a `UBoolean` value, can be used in these cases. Listing **??** shows some examples of executions of these operations in USE. A question mark ('?') command in the USE shell is used to evaluate an OCL expression. The result is shown in the following line, preceded by the '->' symbol.

For illustration purposes, the following list specifies some of the newly defined collection operations.

```
source->forAll(e | P(e)) : UBoolean
  body: source->iterate(e, acc:UBoolean=UBoolean(true,1) |
        acc.uAnd(P(e)))
source->exists(e | P(e)) : UBoolean
  body: source->iterate(e, acc:UBoolean=UBoolean(true,0) |
        acc.uOr(P(e)))
source->includes(e) : UBoolean
  body: source->iterate(v, acc:UBoolean=UBoolean(true,0) |
        if v.uEquals(e).c > acc.c then v.uEquals(e)
        else acc endif)
source->includesAll(collection) : UBoolean
  body: collection->uForAll(e | source->uIncludes(e))
source->excludes(e) : UBoolean
  body: source->uForAll(v | v.uEquals(e).uNot())
source->excludesAll(collection) : UBoolean
  body: collection->uForAll(e | source->uExcludes(e))
source->isUnique(P():UBoolean) : UBoolean
  body: source->uForAll(e|source->uForAll(v|e<>v implies
        P(e).uEquals(P(v).not())))
source->sum() : UReal
  body: source->iterate(v,acc:UReal=UReal(0,0)|acc.add(v))
```

Note that, due to the way in which `UBoolean` operations have been defined (see Section 1), these operations respect the behavior defined in the OCL standard when they involve OCL null and invalid values.

b) Uncertain collections This second case provides extensions to represent the lack of confidence about the contents of the collections as we defined in Section **??**. To represent the degree of belief that we have on the real presence/absence of the elements to a collection, we have defined new collection types, namely `USet`, `UBag`, `UOrderedSet` and `USequence`, each one extending the corresponding OCL collection type. These extensions are implemented in a similar way to how we extended type `String`, associating a Real value within the range $[0, 1]$ that represents the confidence. Thus, values of type `UCollection` are pairs (S, c), where S is a `Collection` and c the associated confidence. Basic collection types (i.e., collection without uncertainty) are injected into the extended uncertain collection types by associating them a confidence of 1.0.

To calculate the confidence of a collection S we follow the same procedure that we applied in the case of strings, using the Levenshtein distance [5]. In this context, it provides the minimum number of changes (insertion, deletion, or substitution of a single element) needed to transform one collection into another. This provides a measure of the possible changes in a collection, which is an aggregated measure of its uncertainty. Of course, these changes need to be understood in the context of each kind of collection; for example, adding a repeated element to a `USet` does not represent a change (nor changing the order of two of its elements), while, for example, it makes a difference when the type of collection is a `USequence`.

With this, if d is an Integer value that represents the number of changes that we estimate that a collection S may have, and l is the size of the collection, its confidence c will be computed as $c = \max\{1.0 - (d/l), 0\}$. Therefore, if $S =$ `Set{0,1,2,3,4,5,6,7,8,9}`, a confidence of $c = 1.0$ would mean that S is completely accurate; a confidence of $c = 0.9$ (=`1-(1/S.size)`$=9/10$) would mean that the set S could allow one change in its content, i.e., one of its actual elements could have been mistakenly included, missed, or modified.

When the OCL collection operations are applied to these new collection types, their behavior changes accordingly. Thus, the results of OCL operations that return `UBoolean` values (such as `uSelect` or `uForAll`), when applied to an uncertain collection (S, c), are obtained by multiplying the result of the corresponding operation on S, by the confidence c of the uncertain collection. Operations that return base OCL types (e.g., Real or Integer) are extended so that they return uncertain types. For example, operation `size` returns a `UInteger` value when applied to a uncertain collection. The uncertainty of the result is computed as we did for uncertain strings. Finally, operations that return uncertain values are extended taking into account the confidence of the uncertain collection. Note that there are some operations that cannot be extended, such as `sum()`. In these cases, we need to restrict the calculations only to the known (certain) elements of the collection. For illustration purposes,

the following list presents the specifications of some of these operations (the complete specifications of all operations are available from [1,2])

```
UCollection::size() : UInteger
post: result.value() = self.S->size() and
      result.uncertainty() = self.confToDist()
USet::union(us:USet(T)) : USet(T)
post: result.S = self.S.union(us.S) and
      result.c = self.distToConf(
        self.confToDist() + us.confToDist(),
        self.S->size() + us.S->size())
UCollection::includes(e) : UBoolean
post: let r:UBoolean=self.S->exists(v|v.equals(e)))
       in result.x = r.value() and
          result.c = r.confidence()*self.c
```

This second case provides extensions to represent the lack of confidence about the contents of the collections. To represent the degree of belief that we have on the real presence/absence of the elements to a collection, we have defined new collection types, namely `USet`, `UBag`, `UOrderedSet` and `USequence`, each one extending the corresponding OCL collection type. These extensions are implemented in a similar way to how we extended type `String`, associating a Real value within the range $[0, 1]$ that represents the confidence. Thus, values of type `UCollection` are pairs (S, c), where S is a Collection and c the associated confidence. Basic collection types (i.e., collection without uncertaint) are injected into the extended uncertain collection types by associating them a confidence of 1.0.

To calculate the confidence of a collection S we follow the same procedure that we applied in the case of strings, using the Levenshtein distance [5]. In this context, it provides the minimum number of changes (insertion, deletion, or substitution of a single element) needed to transform one collection into another. This provides a measure of the possible changes in a collection, which is an aggregated measure of its uncertainty. Of course, these changes need to be understood in the context of each kind of collection; for example, adding a repeated element to a `USet` does not represent a change (nor changing the order of two of its elements), while, for example, it makes a difference when the type of collection is a `USequence`.

With this, if d is an integer that represents the number of changes that we estimate that a collection S may have, and l is the size of the collection, its confidence c will be computed as $c = \max\{1.0 - (d/l), 0\}$. Therefore, if $S =$ `Set{0,1,2,3,4,5,6,7,8,9}`, a confidence of $c = 1.0$ would mean that S is completely accurate; a confidence of $c = 0.9$ (=`1-(1/S.size)`$=9/10$) would mean that the set S could allow one change in its content, i.e., one of its actual elements could have been mistakenly included, missed, or modified.

When the OCL collection operations are applied to these new collection types, their behavior changes accordingly. Thus, the results of OCL operations that return `UBoolean` values (such as `uSelect()` or `uForAll()`), when applied to an uncertain collection (S, c), are obtained by multiplying the result of the corresponding operation on S, by the confidence c of the uncertain

collection. Operations that return base OCL types (e.g., Real or Integer) are complemented with new operations that return uncertain types. For example, operation `uSize` complements `size`, returning a `UInteger` value when applied to a uncertain collection. The uncertainty of the result is computed as we did for uncertain strings. Finally, operations that return uncertain values are extended taking into account the confidence of the uncertain collection. Note that there are some operations that cannot be extended, such as `sum()`. Thus, in these cases, we need to restrict the calculations only to the known (certain) elements of the collection.

```
source->uSize() : UInteger
post: result.x = self.S->size() and
      result.u = self.confToDist()
source->uUnion(us:USet(T)) : USet(T)
post: result.S = self.S.union(us.S) and
      result.c = self.distToConf(self.confToDist() +
          us.confToDist(), self.S->size()+us.S->size())
source->uIncludes(e) : UBoolean
post: let r : UBoolean = source.S->exists(v | v.uEquals(e))
      in result.x = r.x and result.c = r.c*source.c
```

Acknowledgements This work was partially funded by the Spanish Research Projects TIN2014-52034-R and TIN2016-75944-R.

References

1. Atenea research group Git repository (2018). `https://github.com/atenearesearchgroup/uncertainty`. Accessed: March 2019
2. Bertoa, M.F., Moreno, N., Barquero, G., Burgueño, L., Troya, J., Vallecillo, A.: Uncertain OCL Datatypes (2018). URL `http://atenea.lcc.uma.es/projects/UncertainOCLTypes.html`
3. Bertoa, M.F., Moreno, N., Burgueño, L., Vallecillo, A.: Incorporating Measurement Uncertainty into OCL/UML Primitive Datatypes (2019). Submitted
4. JCGM 101:2008: Evaluation of measurement data – Supplement 1 to the "Guide to the expression of uncertainty in measurement" – Propagation of distributions using a Monte Carlo method. Joint Committee for Guides in Metrology (2008). URL `http://www.bipm.org/utils/common/documents/jcgm/JCGM_101_2008_E.pdf`
5. Levenshtein, V.: Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Soviet Physics Doklady **10**, 707 (1966)
6. Object Management Group: Object Constraint Language (OCL) Specification. Version 2.2 (2010). OMG Document formal/2010-02-01