Expressing Measurement Uncertainty in OCL/UML Datatypes

Manuel F. Bertoa[1], Nathalie Moreno[1], Gala Barquero[1], Loli Burgueño[1], Javier Troya[2], and Antonio Vallecillo[1](✉)

[1] Universidad de Málaga, Málaga, Spain
{bertoa,moreno,gala,loli,av}@lcc.uma.es
[2] Universidad de Sevilla, Sevilla, Spain
jtroya@us.es

Abstract. Uncertainty is an inherent property of any measure or estimation performed in any physical setting, and therefore it needs to be considered when modeling systems that manage real data. Although several modeling languages permit the representation of measurement uncertainty for describing certain system attributes, these aspects are not normally incorporated into their type systems. Thus, operating with uncertain values and propagating uncertainty are normally cumbersome processes, difficult to achieve at the model level. This paper proposes an extension of OCL and UML datatypes to incorporate data uncertainty coming from physical measurements or user estimations into the models, along with the set of operations defined for the values of these types.

1 Introduction

It has been claimed that the expressiveness of a model is at least as important as the formality of its expression [19]. This expressiveness is determined by the suitability of the language for describing the concepts of the problem domain or for implementing the design. While in software engineering there exists a variety of modeling languages tailored at addressing different problems, they may not be well suited for capturing some key aspects of the real world [3,17,27], and in particular for managing data uncertainty in a natural manner. In this respect, the emergence of Cyber-Physical Systems (CPS) [3] and the Internet of Things (IoT), as examples of systems that have to interact with the physical world, has made evident the need to faithfully represent some extra-functional properties of the modeled systems and their elements, as well as to overcome current limitations of existing modeling languages and tools.

One aspect of particular relevance is related to the *uncertainty* of the attribute values of the modeled elements, specially when dealing with certain *quality characteristics* such as precision, performance or accuracy. Data uncertainty can come from different reasons, including variability of input variables, numerical errors or approximations of some parameters, observation errors, measurement errors, or simply lack of knowledge of the true behavior of the system or its underlying physics [12]. On other occasions estimations are needed

A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 46–62, 2018.
https://doi.org/10.1007/978-3-319-92997-2_4

because the exact values cannot be obtained since the associated properties are not directly measurable or accessible, values are too costly to measure, or simply because they are unknown.

In a previous paper [28] we presented an extension of the OCL/UML datatype `Real` to deal with measurement uncertainty of numerical values, by incorporating their associated uncertainty [12,13]. However, we soon realized that this was not enough: data uncertainty rapidly extends to all OCL/UML datatypes since it is not just a matter of propagating the uncertainty through the arithmetical operations, but also of dealing with the uncertainty when we compare two uncertain numbers, or need to make a decision about a collection of elements. This requires the definition of uncertain Booleans—values that are true or false with a given probability (level of confidence). Similarly, integers should also be endowed with uncertainty, e.g. when they are used to represent timestamps in milliseconds, and we need to deal with imprecise clocks. This extends to collections too (e.g., a `forAll` statement in a set of uncertain values), and to datatypes operations.

This paper shows how measurement uncertainty can be incorporated into OCL [21] primitive data types and their collections (and hence into UML [23], since both languages share the same primitive types), by defining super-types for them, as well as the set of operations defined on the values of these types. Both analytical and approximate algorithms have been developed to implement these operations. We provide a Java library and a native implementation in USE [9,10].

This paper is structured as follows. First, Sect. 2 briefly introduces the concepts related to measurement uncertainty that will be used throughout the paper. Then, Sect. 3 describes our proposal and the algebra of operations on uncertain values and the implementations we have developed for these operations. Section 4 illustrates some usage scenarios and applications of the proposal. Section 5 compares our work to similar proposals. Finally, we conclude the paper in Sect. 6 with an outlook on future work.

2 Background

Uncertainty is the quality or state that involves imperfect and/or unknown information. It applies to predictions of future events, estimations, physical measurements, or unknown properties of a system [12].

Measurement uncertainty is the special kind of uncertainty that normally affects model elements that represent properties of physical elements. It is defined by the ISO VIM [14] as "a parameter, associated with the result of a measurement, that characterizes the dispersion of the values that could reasonably be attributed to the measurand."

The Guide to the Expression of Uncertainty in Measurement (GUM) [12] defines measurement uncertainty for `Real` numbers representing values of attributes of physical entities, and states that they cannot be complete without an expression of their uncertainty. Such an uncertainty is given by a *confidence interval*, which can be expressed in terms of the *standard uncertainty*—i.e., the

standard deviation of the measurements for such value. Therefore, a real number x becomes a pair (x, u), also noted $x \pm u$, that represents a random variable X whose average is x and its standard deviation is u. For example, if X follows a normal distribution $N(x, u)$, we know that 68.3% of the values of X will be in the interval $[x - u, x + u]$.

The GUM framework also identifies two ways of evaluating the uncertainty of a measurement, depending on whether the knowledge about the quantity X is inferred from repeated measured values ("Type A evaluation of uncertainty"), or scientific judgment or other information concerning the possible values of the quantity ("Type B evaluation of uncertainty").

In Type A evaluation of uncertainty, if $X = \{x_1, \ldots, x_n\}$ is the set of measured values, then the estimated value x is taken as the mean of these values, and the associated uncertainty u as their *experimental standard deviation*, i.e., $u^2 = \frac{1}{(n-1)} \sum_{i=1}^{n} (x_i - x)^2$ [12]. In Type B evaluation, uncertainty can also be characterized by standard deviations, evaluated from assumed probability distributions based on experience or other information. For example, if we know or assume that the values of X follow a Normal distribution, $N(x, \sigma)$, then we take $u = \sigma$. And if we can only assume a uniform or rectangular distribution of the possible values of X, then x is taken as the midpoint of the interval, $x = (a+b)/2$, and its associated variance as $u^2 = (b-a)^2/12$, and hence $u = (b-a)/(2\sqrt{3})$ [12].

In addition to the measure or estimation of individual attributes, in general we need to combine them to produce an aggregated measure, or to calculate a derived attribute. For example, to compute the area of a rectangle we need to consider its height and its width, combining them by multiplication. The individual uncertainties of the input quantities need to be combined too, to produce the uncertainty of the result. This is known as the *propagation of uncertainty*, or *uncertainty analysis*.

Uncertainty can also apply to Boolean values. For example, in order to implement equality and comparison of numerical values with uncertainty, the traditional values of `true` and `false` returned by boolean operators are no longer enough. They now need to return numbers between 0 and 1 instead, representing the probabilities that one uncertain value is equal, less or greater than other [20]. This leads to the definition of *Uncertain Booleans*, which are Boolean values accompanied by the level of confidence that we assign to them. This is a proper supertype of Boolean and its associated operations. Note that this approach should not be confused with *fuzzy logic*: although both probability and fuzzy logic represent degrees of subjective belief, fuzzy set theory uses the concept of fuzzy set membership, i.e., how much an observation belongs to a vaguely defined set, whilst probability theory uses the concept of subjective probability, i.e., the likelihood of an event or condition [16].

3 Extension of OCL and UML DataTypes

Our goal is to extend the OCL and UML languages by declaring new types able to express uncertainty. The benefits are twofold. First, uncertainty can be expressed

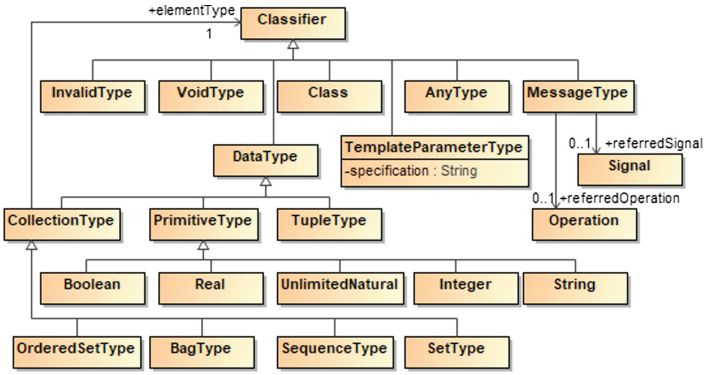


Fig. 1. OCL types, from [21].

in models, i.e., our approach allows the user to define and manipulate uncertainty in a high-level and platform-independent way. Second, information at the model level can be transferred to standard algorithms and tools, so that these can also manage uncertainty by dealing with complex types in their computations.

We propose to extend the OCL types, which are shown in Fig. 1, with uncertainty information. Of course, not all of them need such information, such as types `oclInvalid`, `oclAny`, or `oclVoid`. Other types, such as `Class` and `Tuple`, are user-defined and composed of other heterogeneous types that will convey such information, so there is no need to extend them at this level. Similarly for `TemplateParameter` types, which refer to generic types. Therefore, we need to cover the primitive types (`Real`, `Integer`, `Boolean`, `String`, and `UnlimitedNatural`), collections (`Set`, `Bag`, `OrderedSet`, and `Sequence`) and messages. In this paper we focus on the primitive types, excluding `String`, and on collections. Uncertainty in Strings, Messages and Enumerations—which are datatypes both in OCL and UML—is of different nature, and therefore their extension is left for future work.

3.1 Extension Strategy

In order to extend the OCL/UML primitive types, we apply *subtyping* [18]. We say that type A is a *subtype* of type B (noted $A <: B$), if all elements of A belong to B, and the behavior of operations of B, when applied to elements of A, is the same as those of A [1], i.e., they respect behavioral subtyping [18]. If $A <: B$, then we say that B is a supertype of A.

For instance, `Integer` is a subtype of `Real` because every `Integer` number can be seen as a `Real` number whose decimal part is zero. Besides, `Real` operations, when applied to `Integer` numbers, behave as those of type `Integer`.

Then, for extending a primitive OCL datatype `T` we will define a supertype that incorporates information about the uncertainty of the values of `T`, and

Table 1. New OCL primitive types and their operations.

Type	Operations
`UReal`	+, −, ∗, /, `abs()`, `neg()`, `power()`, `sqrt()`, `inv()`, `floor()`, `round()`, <, ≤, >, ≥, =, <>, `uEquals()`, `uDistinct()`, `min()`, `max()`, `toString()`, `toInteger()`, `toReal()`, `toUInteger()`
`UInteger`	+, −, ∗, `div`, /, `abs()`, `neg()`, `power()`, `sqrt()`, `inv()`, `mod()`, <, ≤, >, ≥, =, <>, `uEquals()`, `uDistinct()`, `min()`, `max()` `toString()`, `toInteger()`, `toUReal()`, `toUInteger()`
`UUnlimitedNatural`	+, ∗, `div`, /, `mod`, <, ≤, >, ≥, =, <>, `uEquals()`, `uDistinct()`, `min()`, `max()`, `toString()`, `toInteger()`, `toUReal()`, `toUInteger()`
`UBoolean`	`not`, `and`, `or`, `xor`, `implies`, `equivalent`, =, <>, `equalsC()`, `uEquals()`, `uDistinct()`, `toString()`, `toBoolean()`, `toBooleanC()`

defines the operations for the extended type, which are also applicable to the base type—i.e., the subtype. This uncertainty information will vary depending on whether the values of the base type are numbers (types `Real`, `Integer` and `UnlimitedNatural`) or boolean values. In the first case, the uncertainty information will record *measurement uncertainty*, and will be expressed as specified in the GUM [12]. Thus, numbers of the extended types will be pairs (x, u), with u the associated uncertainty (cf. Sects. 3.2–3.4). Operations will respect the subtyping relationship, ensuring safe-substitutability. In the case of booleans, the uncertainty will be given by means of a real number between 0 and 1 that represents the assigned confidence (cf. Sect. 3.5).

Table 1 shows the newly defined types and their operations. Besides, the subtyping relationships (<:) among the numeric datatypes—both standard and extended—are shown below:

$$\begin{array}{ccccc} \texttt{UnlimitedNatural}\backslash\{*\} & <: & \texttt{Integer} & <: & \texttt{Real} \\ \wedge & & \wedge & & \wedge \\ \texttt{UUnlimitedNatural}\backslash\{*\} & <: & \texttt{UInteger} & <: & \texttt{UReal} \end{array}$$

In addition, `Boolean` <: `UBoolean`, completing the relationships. To extend collections we will specify them using the corresponding extended operations of their element types. The following sections describe these extensions in detail.

3.2 Extending Type `Real`

To represent real values with measurement uncertainty, we make use of type `UReal` and the algebra of operations defined on the values of that type, which we presented in our previous work [28]. Basically, the values of `UReal` are pairs of `Real` numbers $X = (x, u)$. They determine the expected value (x) and associated standard uncertainty (u) of a quantity X, as defined in Sect. 2. Real numbers x are naturally injected into type `UReal`, corresponding to pairs $(x, 0)$.

We have specified in OCL, and also implemented in Java, all the operations on the values of type `UReal`, to allow modelers to use them for defining derived attributes and for specifying operations and invariants in OCL and UML models. Furthermore, to validate our proposal we have also extended the tool USE by implementing the new types as native ones—see Sect. 3.7.

As an example, the following listing shows the specification of two of the `UReal` operations[1]:

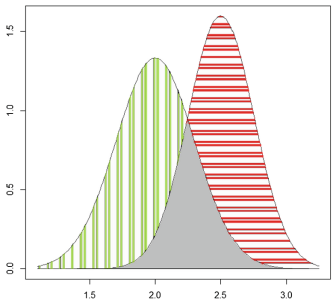
```
context UReal::add(r : UReal) : UReal
post: result.x = self.x + r.x and
      result.u = (self.u*self.u + r.u*r.u).sqrt()
context UReal::mult(r : UReal) : UReal
post: result.x = (self.x*r.x) and
      result.u = (r.u*r.u*self.x*self.x + self.u*self.u*r.x*r.x).sqrt()
```

In addition to the traditional comparison operations between uncertain reals ($<$, $\leq$, $>$, etc.), which return a Boolean value, comparisons between real numbers with uncertainty should return uncertain booleans. To illustrate this need, consider the graphical representation of two pairs of uncertain reals shown in Fig. 2. We can see that there is indeed an overlap (represented by the gray area): it constitutes the probability that the two values are equal.

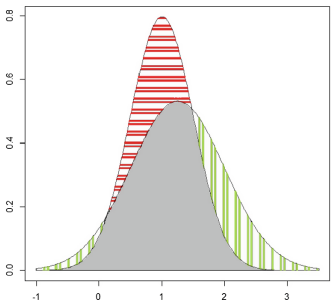
Then, given two `UReal` values x and y we define three real numbers (l, e, g) that represent, respectively, the probability of x being less, equal or greater than y. Of course, it is always the case that $l+e+g = 1$. For example, the triplet that we obtain for values a and b (Fig. 2a) is the following: $(0.893, 0.106, 1.11 \cdot 10^{-16})$. This means that $a < b$ with probability 0.893; $a = b$ with probability 0.106, and $a > b$ with a probability $1.11 \cdot 10^{-16}$. Similarly, the triplet for c and d (Fig. 2b) is: $(0.152, 0.754, 0.094)$. Note that these 3 numbers correspond to the three areas in which the curve that represents the first of the values can be divided (this is clearer in Fig. 2b).

All this has been specified in OCL using an auxiliary operation on type `UReal` called `calculate(r:UReal)` that returns a tuple with the triplet. With it, the specification of comparison operations between `UReal` numbers is as follows (`lt` and `gt` mean *lower/greater than*, `le` and `ge` mean *lower/greater or equal than*, b and c conform the `UBoolean` type, as explained in Sect. 3.5).

[1] Operations on basic datatypes normally use infix notation (e.g., $x + y$, $a < b$, P `and` Q). This is the notation that we already support in our USE implementation for the newly defined types (`UReal`, `UBoolean`, etc.), see Sect. 3.7. However, other languages that we have used to implement these new types (e.g., Java) do not support infix notation. Therefore, in the following we will use either an infix or prefix notation (`x.add(y)`, `a.lt(b)`, `P.and(Q)`) for the operations of these types, depending on the context and on the particular language used.



(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. 2. Graphical representation of `UReal` values.

```
context UReal::lt(r :UReal) :UBoolean
  post: (result.b) and (result.c = self.calculate(r).l)
context UReal::le(r :UReal) :UBoolean
  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
  post: (result.b) and (result.c=self.calculate(r).g)
context UReal::ge(r :UReal) :UBoolean
  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))
```

The complete OCL specifications of these types and operations, and their implementation in SOIL [4]—an OCL extension that permits the execution of OCL specifications for simulation purposes—is available from [2], together with the two implementations in Java that we provide, depending on whether we assume values are independent and normally distributed—and therefore a closed form expression can be used for the calculations—or using Monte-Carlo simulations in case variables follow arbitrary distributions.

Table 1 shows the set of operations defined for type `UReal`, including conversion operations to other OCL datatypes (both standard and extended).

3.3 Extending Type `Integer`

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

This extension is straightforward. Every `UInteger` element is of the form (n, u) with n an `Integer` value and u a `Real` value that represents the uncertainty. The injection of any `Integer` value n into type `UInteger` is naturally

defined by $(n, 0)$. In turn, the behavior of `UInteger` operations is defined by lifting the operation to type `UReal`, and then projecting the corresponding result, if needed. This, together with the subtyping relationship `Integer` <: `Real` existing in OCL, ensures the proper subtyping relationship between `Integer` and `UInteger`.

3.4 Extending Type `UnlimitedNatural`

An OCL `UnlimitedNatural` is either a non-negative `Integer` or a special *unlimited* value (`*`) that represents the upper value of a multiplicity specification [21].

First, we have that `UnlimitedNatural`\{`*`} <: `Integer`, that is, excluding value `*`, unlimited naturals are just non-negative integers. This special value `*` cannot be used in any arithmetic operation with unlimited naturals, but only with comparison (including max and min) operations. 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 hence to `Real`).

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

Operations on `UUnlimitedNatural` values not involving special value `*` are defined by lifting them to type `UInteger`. Comparison operations need to consider the particular case of special value `*` (internally represented by "−1"), lifting the operation to the supertype if this value is not involved. For illustration purposes, the following listing shows 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
```

3.5 Extending Type `Boolean`

Type `UBoolean` is the supertype for type `Boolean` that adds uncertainty to its values. In this case, the uncertainty does not refer to measurement uncertainty, but to confidence. Thus, a `UBoolean` value is a pair (b, c) where b is a boolean value $(true, false)$ and c is a real number in the range $[0 \ldots 1]$, representing the confidence that b is certain. Boolean values `true` and `false` are injected into the supertype as $(\texttt{true}, 1)$ and $(\texttt{false}, 1)$, respectively.

A property of this representation is that $(b, c) = (\neg b, 1 - c)$ for every boolean value b. Then, in its internal representation we will use a canonical form, always taking b the value of $true$ and c 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 $(\texttt{true}, 0.05)$.

The operations supported by type `UBoolean` extend those of type `Boolean`, as defined by OCL [21]. We have defined the basic (`not`, `and` and `or`) and secondary operations (`implies`, `equivalent` and `xor`) of the traditional Boolean algebra, extending them with uncertainty. Assuming all values are independent, the following listing shows the specification of all the `UBoolean` type operations.

```
not() : UBoolean
  post: (result.b) and
        (result.c = if self.b then 1-self.c else self.c endif)

and(b : UBoolean) : UBoolean
  post: let C : Real = (self.c * b.c) in (result.b) and
        (result.c=if (self.b and b.b) then C else (1-C) endif)

or(b : UBoolean) : UBoolean
  post: let C : Real = (self.c + b.c - (self.c * b.c)) in
        (result.b) and
        (result.c = if (self.b or b.b) then C else (1-C) endif)

implies(b : UBoolean) : UBoolean
  post: let C : Real = (self.c + b.c - (self.c * b.c)) in
        (result.b) and
        (result.c = if (self.b implies b.b) then C else (1-C) endif)

equivalent(b : UBoolean) : UBoolean
  post: let C : Real = (self.c + b.c - (self.c * b.c)) in
        (result.b) and
        (result.c = if (self.b implies b.b) and (b.b implies self.b)
                    then C else (1-C) endif)

xor(b : UBoolean) : UBoolean
  post: result = self.uEquivalent(b).not()

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)

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 kept '=' (`equals()`) and '<>' (`distinct()`) operations with their usual semantics, that is, two `UBoolean` elements are the same if their boolean and confidence values match. We have also extended the `equals()` operation with the possibility of indicating a confidence threshold that both `UBoolean` values are equal. Other identity operations (`uEquals()`, `uDistinct()`) compare two `UBoolean` values, returning another `UBoolean`. 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 indicating a threshold for the confidence.

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 proposed in [13] for Type-A measurement uncertainty in real numbers, adapted to boolean values. Basically, every `UBoolean` value contains a sequence of `Boolean` values that represent the sample obtained when measuring that value. Operations are performed on the samples, and then b and c become just derived values. An excerpt of such specification, showing only the first two operations, is shown in the listing below. Note that an additional invariant, at the end of the listing, requests that all samples should be of the same size.

```
class UBoolean_A
   -- canonical form: triplets (sample[],true,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()
not() : UBoolean_A
  post: (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))))
...
context UBoolean_A inv SameSampleSize:
  UBoolean_A.allInstances->forAll(u1,u2|u1.sample->size=u2.sample->size)
```

Similar specifications (and their corresponding implementations in Java) are also available for the rest of the extended types.

3.6 Extending OCL Collections

OCL collections can be easily extended based on the extended operators for primitive datatypes. The following listing shows the specification of all collection operations. Those that return a `UBoolean` value incorporate a 'u' at the start of their name, to distinguish them from their boolean versions:

```
source->uForAll(e | P(e)) : UBoolean
::= source->iterate(e, acc:UBoolean=UBoolean(true,1) | acc.and(P(e)))

source->uExists(e | P(e)) : UBoolean
::= source->iterate(e, acc:UBoolean=UBoolean(true,0) | acc.or(P(e)))
```

```
source->uIncludes(e) : UBoolean
::= source->iterate(v, acc:UBoolean=UBoolean(true,0) |
        if v.uEquals(e).c > acc.c then v.uEquals(e) else acc endif)

source->uIncludesAll(collection) : UBoolean
::= collection->uForAll(e | source->uIncludes(e))

source->uExcludes(e) : UBoolean
::= source->uForAll(v | v.uEquals(e).not())

source->uExcludesAll(collection) : UBoolean
::= collection->uForAll(e | source->uExcludes(e))

source->uSelect(P():UBoolean) : collection
::= source->iterate(v, acc:collection=collection{} |
        if P(v).toBoolean() then acc->including(v) else acc endif)

source->uSelect(P():UBoolean, c :Real) : collection
::= source->iterate(v, acc:collection=collection{} |
        if P(v).toBooleanC(c) then acc->including(v) else acc endif)

source->uReject(P():UBoolean):collection
::= source->iterate(v, acc:collection=collection{} |
        if P(v).toBoolean() then acc->excluding(v) else acc endif)

source->uReject(P():UBoolean, c:Real) : collection
::= source->iterate(v, acc:collection=collection{} |
    if P(v).toBooleanC(c) then acc->excluding(v) else acc endif)

source->uCount(e) : Integer
::= source->iterate(v, acc:Integer=0 |
        if v.uEquals(e).toBoolean() then acc + 1 else acc endif)

source->uCountC(e,c) : Integer
::= source->iterate(v, acc:Integer=0 |
        if v.uEquals(e).toBooleanC(c) then acc + 1 else acc endif)

source->uOne(P():UBoolean) : Boolean
::= source->uSelect(e | P(e))->size()=1

source->uOneC(P():UBoolean, c:Real) : Boolean
::= source->uSelect(e | P(e).toBooleanC(c) )->size()=1

source->uIsUnique(P():UBoolean) : UBoolean
::= source->uForAll(e|source->uForAll(v|e<>v implies
                                        P(e).uEquals(P(v).not())))
source->sum() : UReal
::= source->iterate(v, acc:UReal=UReal(0,0) | acc.add(v))
```

3.7 Implementation in USE

USE [9] is a modeling tool that allows the validation of OCL and UML models by means of executing the UML models and checking its OCL constraints. The tool is open-source and distributed under a GNU General Public License. To validate our proposal we have extended the OCL/UML language in USE by adding the previously described uncertain types as basic primitive data types, as well as their native operations, so they become available to any OCL/UML modeler. An example of how the new types can be effectively used is illustrated in Sect. 4. The extended tool can be downloaded from our website[2].

[2] http://atenea.lcc.uma.es/downloads/uncertainOCLTypes/use-5.0.0_extended.zip.

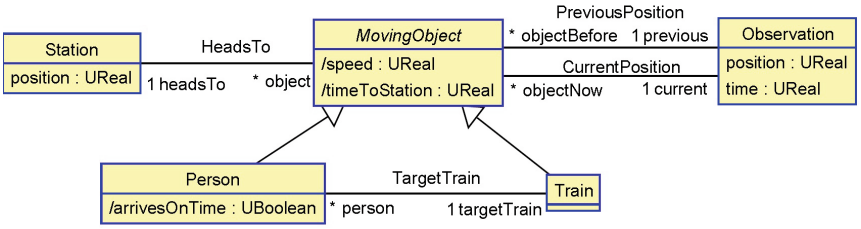


Fig. 3. UML class diagram for train example.

4 Applications

To illustrate our proposal, let us consider the system described by the metamodel shown in Fig. 3. It is composed of people, trains and stations. Both persons and trains move towards stations. For simplicity, we assume they all move in one single direction. Monitors observe their movements, and record their last two positions and the time in which they were observed. The speed is automatically calculated from this information, as well as the expected time to arrive at the station. For a person it is also important to know if she will be able to catch her target train, i.e., reach the station at least 3 s before the train does. All these calculations can be specified by means of OCL expressions:

```
context MovingObject::speed:Real
  derive: (self.current.position−self.previous.position) /
          (self.current.time−self.previous.time)

context MovingObject::timeToStation: Real
  derive: (self.headsTo.position−self.current.position) / self.speed

context Person::arrivesOnTime:Boolean
  derive: (self.timeToStation + 3) <= self.targetTrain.timeToStation
```

Figure 4 shows a UML object diagram with an example of such a system, using conventional UML datatypes `Real` and `Boolean`.

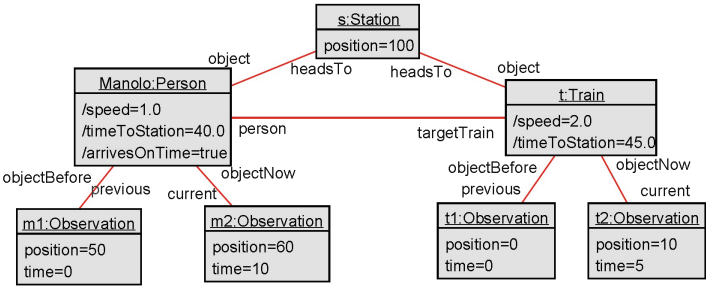


Fig. 4. UML object diagram with `Real` and `Boolean` types.

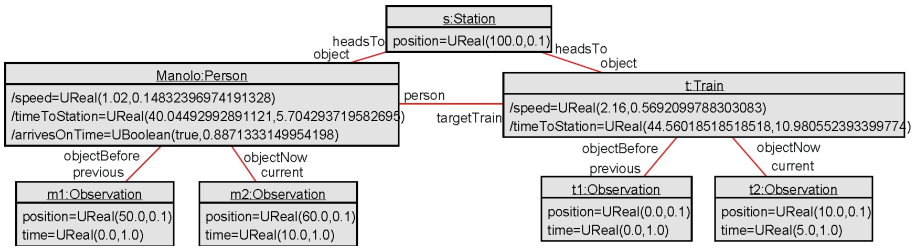


Fig. 5. UML object diagram with uncertain types.

Note, however, that in practice all these attributes and operations are subject to uncertainty: positions and times are never 100% precise, and this imprecision is propagated to derived values and estimated times. For example, suppose our positioning system is correct up to one centimeter, and our clock has a precision of 1 s. This can be captured in our model by simply updating the types of `Real` and `Boolean` variables to `UReal` and `UBoolean`, respectively (this was already showed in Fig. 3).

Figure 5 shows an object diagram with uncertain variables. Those variables take into account the measurement uncertainty in the observations, and propagate it through the computations. We can see how the expected train and user arrival times at the station are $T = 44.560 \pm 10.581$ and $M = 40.045 \pm 5.704$, respectively, and therefore their difference is $T - M = -4.515 \pm 12.374$. Using a `UBoolean` comparison operation, $M \leq T = (true, 0.887)$, which means that the user will be able to arrive on time to catch the train with a probability of 0.887. This is much more realistic than the first model, which probably was too naïve to be of real use. Something worth noticing is that we only had to change the types of the variables; all the OCL expressions that were used to compute the values of derived attributes remained exactly the same.

5 Related Work

The need to represent and manipulate physical values in software models is emerging, in particular units or real-time properties of cyber-physical systems [27]. For example, given that timing values are by nature uncertain (they are very often estimates and/or measured by means of monitoring), the real-time community is used to represent probability distributions and intervals for timing properties, and their influence is evident in the MARTE Profile [22] and SysML [24]. However, neither MARTE nor SysML support operations for performing calculations with these values, they remain at the descriptive level.

Similarly, in [31], the authors propose a conceptual model, called *Uncertum*, which is supported by a UML profile (UUP, the UML Uncertainty Profile)

that enables including uncertainty in test models. Uncertum is based on the U-Model [32], extending it for testing purposes. UUP is a very complete profile that covers all different kinds of uncertainties, in particular measurement uncertainty. Again, their focus, testing, is slightly different from ours, and they only need to represent uncertainty but not to perform operations with it, and therefore they also remain at a descriptive level.

Other works on Business Process Models (e.g., [15]) also consider uncertainty when modeling the arrival time of clients, the availability of some resources or the duration of some tasks. These works use probabilistic mass functions for modeling the values of the corresponding attributes. We have preferred to use the way defined by the GUM [12,13]. Apart from being simpler and widely adopted by other engineering disciplines, it has the main benefit of permitting operations on variables that do not follow any particular probabilistic distribution.

The work in [30] defines an XML-based modeling language for measurement uncertainty evaluation based on the GUM, and a simulation framework for it. This work can be in principle considered closely related to our proposal, but the fact that it is not integrated with the type system of a mainstream modeling language (such as OCL or UML), and its low-level syntax (based on plain XML) hindered its usability. Similarly, the work in [11] defines a datatype that incorporates measurement uncertainty and provides some libraries to perform computations with its values. The integration of these works with OCL/UML models is not straightforward, and therefore their adoption and usage by UML modelers might be limited. To the best of our knowledge these works are more closely related to the mathematical libraries and tools already existing [29] for propagating measurement uncertainty and operating with uncertain values, than to our work.

Other works deal with model uncertainty, but focusing on aspects different from the ones we have described here. For instance, on the uncertainty on the models themselves and on the best models to use depending on the system properties that we want to capture [19]. Other works deal with the uncertainty of the design decisions, of the modeling process, or of the domain being modeled [5–8,26]. We depart from them since we are concerned with the uncertainty of the values of the quantities being measured, which is a different problem.

Finally, the OMG defined the Structured Metrics Meta-model (SMM) [25], which is part of the Architecture Driven Modernization (ADM) effort, and aims at representing measurement information related to software, its operation and design. The SMM is a specification for the definition of measures and the representation of their measurement results, including uncertainty, independently of the representation of the measured entities. In this sense, our proposal can be considered as a refinement of the SMM metamodel, particularizing it to the domain of OCL and UML datatypes.

6 Conclusion and Future Work

In this paper we have focused on representing and managing measurement uncertainty in OCL and UML software models, something required in order

to precisely capture and manipulate some of the essential quality properties of any physical system. We have extended the OCL datatypes and their related operations with uncertainty information. OCL and Java libraries have also been developed to implement the type and its operations in MDE settings. Our implementation is available on [2].

This work opens several interesting lines of research that we would like to explore next. First, we would like to analyze how uncertainty could be added to those OCL datatypes not covered here, namely Strings and Enumerations. As mentioned earlier, the nature of their uncertainty seems to be rather different from the rest. Second, we would like to provide mappings from our high-level OCL/UML specifications to other specification and simulation languages and tools, in particular Modelica and Simulink. The objective is to achieve a stepwise refinement heterogeneous specification and simulation process, whereby high-level specifications (and hence more lightweight) can be progressively refined into more concrete, complete (and more complex) specifications. Finally, we would like to further validate our proposal with different kinds of examples, checking the expressiveness and applicability of this type of specifications.

Acknowledgements. This work has been partially supported by the Spanish Government under Grant TIN2014-52034-R. We would like to thank Martin Gogolla for his help and support during the development of the USE tool extension, and to the reviewers for their constructive comments and very valuable suggestions.

References

1. America, P.: Inheritance and subtyping in a parallel object-oriented language. In: Bézivin, J., Hullot, J.-M., Cointe, P., Lieberman, H. (eds.) ECOOP 1987. LNCS, vol. 276, pp. 234–242. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-47891-4_22
2. Bertoa, M.F., Moreno, N., Barquero, G., Burgueño, L., Troya, J., Vallecillo, A.: Uncertain OCL Datatypes, April 2018. http://atenea.lcc.uma.es/projects/UncertainOCLTypes.html
3. Broy, M.: Challenges in modeling cyber-physical systems. In: Proceedings of the ISPN 2013, pp. 5–6. IEEE (2013)
4. Büttner, F., Gogolla, M.: On OCL-based imperative languages. Sci. Comput. Program. **92**, 162–178 (2014)
5. Eramo, R., Pierantonio, A., Rosa, G.: Managing uncertainty in bidirectional model transformations. In: Proceedings of SLE 2015, pp. 49–58. ACM (2015)
6. Esfahani, N., Malek, S.: Uncertainty in self-adaptive software systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 214–238. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_9
7. Famelis, M., Salay, R., Chechik, M.: Partial models: towards modeling and reasoning with uncertainty. In: Proceedings of ICSE 2012, pp. 573–583. IEEE Press (2012)
8. Garlan, D.: Software engineering in an uncertain world. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER 2010), pp. 125–128. ACM (2010)

9. Gogolla, M., Büttner, F., Richters, M.: USE: a UML-based specification environment for validating UML and OCL. Sci. Comp. Prog. **69**, 27–34 (2007)
10. Gogolla, M., Hilken, F.: Model validation and verification options in a contemporary UML and OCL analysis tool. In: Oberweis, A., Reussner, R. (eds.) Proceedings of the Modellierung (MODELLIERUNG 2016). LNI, vol. 254, pp. 203–218. GI (Gesellschaft für Informatik), Karlsruhe (2016)
11. Hall, B.D.: Component interfaces that support measurement uncertainty. Comput. Stand. Interfaces **28**(3), 306–310 (2006)
12. JCGM 100:2008: Evaluation of measurement data - Guide to the expression of uncertainty in measurement (GUM). Joint Committee for Guides in Metrology (2008). http://www.bipm.org/utils/common/documents/jcgm/JCGM_100_2008_E.pdf
13. 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). http://www.bipm.org/utils/common/documents/jcgm/JCGM_101_2008_E.pdf
14. JCGM 200:2012: International Vocabulary of Metrology - Basic and general concepts and associated terms (VIM), 3rd edn. Joint Committee for Guides in Metrology (2012). http://www.bipm.org/utils/common/documents/jcgm/JCGM_200_2012.pdf
15. Jiménez-Ramírez, A., Weber, B., Barba, I., del Valle, C.: Generating optimized configurable business process models in scenarios subject to uncertainty. Inf. Softw. Technol. **57**, 571–594 (2015)
16. Kosko, B.: Fuzziness vs. probability. Int. J. Gen. Syst. **17**(2–3), 211–240 (1990)
17. Lee, E.A.: Cyber physical systems: design challenges. In: Proceedings of ISORC 2008, pp. 363–369. IEEE (2008)
18. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (1994)
19. Littlewood, B., Neil, M., Ostrolenk, G.: The role of models in managing the uncertainty of software-intensive systems. Reliab. Eng. Syst. Saf. **50**(1), 87–95 (1995)
20. Mayerhofer, T., Wimmer, M., Burgueño, L., Vallecillo, A.: Specifying quantities in software models (2018, submitted). Technical report: http://atenea.lcc.uma.es/index.php/Main_Page/Resources/DataUncertainty
21. Object Management Group: Object Constraint Language (OCL) Specification. Version 2.2, February 2010. OMG Document formal/2010-02-01
22. Object Management Group: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Version 1.1, June 2011. OMG Document formal/2011-06-02
23. Object Management Group: Unified Modeling Language (UML) Specification. Version 2.5, March 2015. OMG Document formal/2015-03-01
24. Object Management Group: OMG Systems Modeling Language (SysML), Version 1.4, January 2016. OMG Document formal/2016-01-05
25. Object Management Group: Structured Metrics Metamodel (SMM) Specification. Version 1.1.1, April 2016. OMG Document formal/16-04-04
26. Salay, R., Chechik, M., Horkoff, J., Sandro, A.: Managing requirements uncertainty with partial models. Requir. Eng. **18**(2), 107–128 (2013)
27. Selic, B.: Beyond mere logic - a vision of modeling languages for the 21st century. In: Proceeding of MODELSWARD 2015 and PECCS 2015, p. IS–5. SciTePress (2015)

28. Vallecillo, A., Morcillo, C., Orue, P.: Expressing measurement uncertainty in software models. In: Proceedings of the 10th International Conference on the Quality of Information and Communications Technology (QUATIC), pp. 1–10 (2016)
29. Wikipedia: List of uncertainty propagation software. https://en.wikipedia.org/wiki/List_of_uncertainty_propagation_software. Accessed 13 Apr 2018
30. Wolf, M.: A modeling language for measurement uncertainty evaluation. Ph.D. thesis, ETH Zurich (2009)
31. Zhang, M., Ali, S., Yue, T., Norgren, R., Okariz, O.: Uncertainty-wise cyber-physical system test modeling. Softw. Syst. Model. (2017). https://doi.org/10.1007/s10270-017-0609-6
32. Zhang, M., Selic, B., Ali, S., Yue, T., Okariz, O., Norgren, R.: Understanding uncertainty in cyber-physical systems: a conceptual model. In: Wąsowski, A., Lönn, H. (eds.) ECMFA 2016. LNCS, vol. 9764, pp. 247–264. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42061-5_16