*This is a pre-print of an article published in **Software and Systems Modeling (SoSyM).** The final authenticated version is available online at: https://doi.org/10.1007/s10270-019-00741-0 and the specific citation should be:*
Bertoa, M.F., Burgueño, L., Moreno, N., Vallecillo, A. "Incorporating measurement uncertainty into OCL/UML primitive datatypes." Softw Syst Model (2019). https://doi.org/10.1007/s10270-019-00741-0

Incorporating Measurement Uncertainty into OCL/UML Primitive Datatypes

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

Received: date / Accepted: date

Abstract The correct representation of the relevant properties of a system is an essential requirement for the effective use and wide adoption of model-based practices in industry. Uncertainty is one of the inherent properties of any measurement or estimation that is obtained in any physical setting; as such, it must be considered when modeling software systems that deal with real data. Although a few modeling languages enable the representation of measurement uncertainty, these aspects are not normally incorporated into their type systems. Therefore, operating with uncertain values and propagating their uncertainty become cumbersome processes, which hinder their realization in real environments. This paper proposes an extension of OCL/UML primitive datatypes that enables the representation of the uncertainty that comes from physical measurements or user estimates into the models, together with an algebra of operations that are defined for the values of these types.

Keywords Measurement Uncertainty · OCL · UML · Primitive Datatypes

Manuel F. Bertoa
Universidad de Málaga, Spain
E-mail: bertoa@lcc.uma.es

Loli Burgueño
Open University of Catalonia, IN3, Spain
Institut LIST, CEA, Université Paris-Saclay, France
E-mail: lburguenoc@uoc.edu

Nathalie Moreno
Universidad de Málaga, Spain
E-mail: moreno@lcc.uma.es

Antonio Vallecillo
Universidad de Málaga, Spain
E-mail: av@lcc.uma.es

1 Introduction

The emergence of cyber-physical systems (CPSs) [9] and the internet of things (IoT) [29], which are examples of systems that must interact with the physical world, and the current industrial practices, such as the Industry 4.0 [50], have made evident the need to faithfully represent extra-functional properties in models of systems and their elements. This is an essential requirement for leveraging some of the potential benefits of model-based software engineering (MBSE) [59,8,16] in industrial settings—particularly if MBSE is indeed going to become widely adopted in practice.

It has been claimed that the expressiveness of a model is as important as the formality of its expression [47]. This expressiveness is determined by the suitability of the language for describing the concepts of the problem domain or for implementing the design. Although in software engineering a variety of modeling languages are tailored to various problems, they may not be well suited for capturing key aspects of the real world [9,43,60] and, in particular, for managing data *uncertainty* in a natural manner. One relevant issue is related to the uncertainty of the attribute values of the modeled elements, especially when dealing with physical quantities such as lengths, times, weights, or other measurable elements.

Data uncertainty can originate from various sources, including variability of input variables, numerical errors or approximations of parameters, observation errors, measurement errors and lack of knowledge of the true behavior of the system or its underlying physics [34]. On other occasions, estimates are needed because the exact values cannot be obtained since the associated properties are not directly measurable, or accessible or values are too costly to measure or are simply unknown.

Until recently, most software modeling notations have permitted only exact values to be used for representing system properties, which has limited their precision for modeling and analyzing any realistic system. UML [54], and its constraint language, namely, OCL [52] are the main examples of this. They support primitive datatypes Real, Integer, Boolean and String, in addition to enumerations, and UnlimitedNatural, which is used to represent the unbounded set of non-negative integers (with "`*`" representing "∞") to express the cardinalities of model elements.

Currently, several modeling languages, such as MARTE [53] and SysML [55], which are both extensions of UML, support the representation of measurement uncertainty for describing various system properties. Typically, stereotypes that are added to class attributes of type Real are used to represent the tolerance or precision of their values. Various business process modelling notations (e.g., [37]) also consider measurement uncertainty, for example, when modeling the arrival times of clients, the availability of resources or the duration of tasks. These works use probabilistic mass functions to model the values of the corresponding attributes instead of fixed values. However, these aspects are not incorporated into their type systems; therefore, operating with uncertain values or propagating uncertainty are typically cumbersome processes and very difficult to implement at the model level.

In a previous paper [63], we presented an extension of the OCL/UML primitive datatype `Real` for dealing with measurement uncertainty of numerical values, by incorporating their associated uncertainty [34,35]. However, we soon realized that this was insufficient: uncertainty may also affect the other OCL/UML primitive datatypes since it is not just a matter of propagating the uncertainty through the arithmetical operations but also of dealing with the uncertainty when, for example, we compare two uncertain numbers, which would require the definition of uncertain Booleans—values that are true or false with a specified probability (level of confidence). Similarly, integers should be endowed with uncertainty, e.g., when they are used to represent timestamps expressed in milliseconds in systems with imprecise clocks. This also extends to collections (e.g., a `forAll` statement on a set of uncertain values) and to the remaining primitive datatypes.

In [5], we proposed the extension of some OCL/UML types (Real, Integer, Boolean and UnlimitedNatural) with measurement uncertainty, and partially to OCL collections which involved them. However, it did not cover all OCL primitive datatypes (Strings and enumerations were not included), the extension did not consider the 4-valued OCL logic, and neither the correctness nor the algebraic properties of these extensions were studied. No tool support was provided for these extensions either.

In this paper, we incorporate measurement uncertainty into all primitive OCL datatypes, complete the extension of the OCL collections and the Boolean values, check the algebraic properties of all the extended types and their operations to ensure that they satisfy the same properties as the basic type operations, and provide a prototypical implementation of all new types in a UML tool with full support for OCL, thereby offering a proof of concept for our proposal. We have also evaluated our proposal according to different dimensions, including correctness, performance, reusability and interoperability.

All the supplementary materials and software artefacts related to this proposal are available from our Git repository [1] and website [6]. They include: the complete OCL specifications of the new datatypes and operations; their implementation in SOIL [13] (an OCL extension that enables the execution of OCL specifications for simulation purposes); the complete proofs of the algebraic properties of the extended operations; the two implementations in Java that we provide for the new datatypes, the choice of which depends on whether we assume values are independent and normally distributed (and therefore, a closed-form expression can be used for the calculation of the propagated uncertainty) or not (Monte-Carlo simulations are used if the variables follow arbitrary distributions or they are not independent); the source models of the examples and case studies shown in this paper; and the new version of the USE tool extended with the uncertain datatypes.

This paper is structured as follows: Section 2 introduces the related concepts to measurement uncertainty that will be used throughout the paper. Then, Section 3 describes our proposal, the algebra of operations on uncertain values and the implementations that we have developed for these operations. Tool support is described in Section 4. Section 5 presents various usage scenarios and applications of the proposal that we have used to evaluate its expressiveness and suitability. The evaluation of the proposal is discussed in Section 6. Finally, Section 7 compares our work to similar proposals and Section 8 presents the conclusions of the paper and an outlook on future work.

2 Background

Uncertainty is a quality or state that involves imperfect and/or unknown information. It applies to predictions of future events, estimates, physical measurements, or unknown properties of a system [34].

Various types of uncertainties may be considered when modeling a system [51,67,56]. For example, *aleatory* uncertainty refers to the inherent variation associated with the physical system under consideration, or its environment. In contrast, *epistemic* uncertainty refers to the potential inaccuracy in any phase of the modeling process that is due to the lack of knowledge [51]. In this paper, we are concerned with *measurement uncertainty*, which is a particular kind of aleatory uncertainty that refers to the inability to know with precision the value of a physical quantity. Depending on the type of quantity, we can express its measurement uncertainty in various ways. For quantities whose values are of numeric types (Real, Integer or Unlimited Naturals), measurement uncertainty can be expressed using, e.g., ranges or distribution probabilities describing the possible variations of the values. In our proposal, we will follow the ISO recommendations for representing measurement uncertainty [34,36], in which uncertainty is expressed in terms of the standard deviation of the measurements, for example, 13.5 ± 0.001 or 111.7 ± 1.5. This is the approach widely adopted in most engineering disciplines. For a Boolean value, a real number that is between 0 and 1 represents the confidence (i.e., the degree of belief) that we have on that value. Thus, possible uncertain Boolean values are $(\mathit{true}, 0.99)$ or $(\mathit{false}, 0.75)$. For Strings, which are sequences of characters, a Real number that is between 0 and 1 can also be used to represent the confidence we have on the values of a string: (`"Hello"`, 0.90). This is useful, for instance, when strings are obtained from unreliable or difficult-to-recognize sources such as handwritten manuscripts or images from low-precision cameras (see Section 3.6). Finally, for enumerations, variables should be able to store more than one literal, assigning a probability to each one. For example, if the literals of an enumeration type `TemperatureLevel` are `Low`, `Medium` and `High`, a possible value of that type is $\{(\texttt{Low}, 0.05), (\texttt{Medium}, 0.75), (\texttt{High}, 0.20)\}$.

2.1 Motivating example

To illustrate our approach, let us consider a system that represents a battle between robots and other unidentified objects in a planar surface. The elements of such a system are described by the metamodel that is shown in Figure 1. The robots are tasked with ensuring that no 'unidentified object' gets close to the area they protect. The position of each object is specified by a pair of coordinates, namely, `x` and `y`, and the object moves in the direction that is dictated by its `angle` attribute

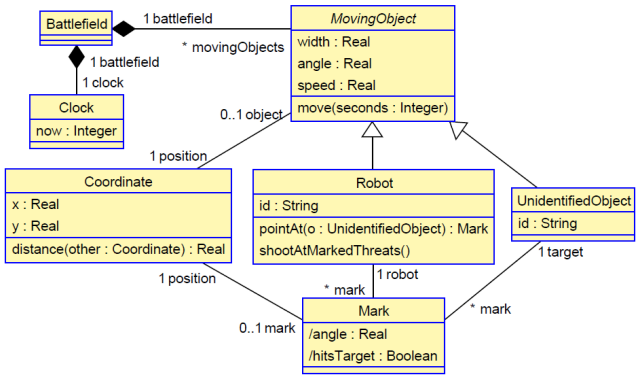


Fig. 1: A Robot Battle System.

(expressed in radians) with a specified `speed` (in m/s).[1] The size of each moving object is determined by its diameter (attribute `width`, also in meters). Movements are performed via the operation `move`, which updates the coordinates of the object based on its current angle and speed, and the number of elapsed seconds since the last movement. Times are expressed using the POSIX time convention, i.e., by the number of seconds since January 1, 1970 [32].

If a robot detects that an unidentified object is moving at a speed that exceeds 0.3 m/s and gets close to its position, the robot recognizes it as a threat and marks it as such. The mark is represented by an object of class `Mark`, whose coordinates coincide with the position of the marking robot, and whose attributes `angle` and `hitsTarget` are calculated as shown in Listing 1.

```
context Mark::angle : Real derive:
  ((target.position.y-robot.position.y) /
   (target.position.x-robot.position.x)).atan()
context Mark::hitsTarget : Boolean derive:
 let d:Real = robot.position.distance(
                                target.position) in
  (robot.position.x - target.position.x +
     d*self.angle.cos()).abs() <= target.width
  and (robot.position.y - target.position.y +
     d*self.angle.sin()).abs() <= target.width
```

Listing 1: Derived attributes of class `Mark`.

Operation `shootAtMarkedThreats()` shoots at all objects that have been marked by the robot as threats of a high level (closer to it than 10 m). Every time the robot or a marked object moves, the attributes of the associated `Mark` object also change.

These specifications miss however an essential aspect of the system: the uncertainty associated to some of its elements, and how it is propagated through the system operations. For example, positioning systems are not fully accurate, and therefore we should allow for some imprecision in the attributes of the `Coordinate`

[1] This is inspired by how various simple robots operate, in particular, `Ozobot` robots (`https://ozobot.com`).

objects. Similarly, measurement devices should include some uncertainty when estimating the `width`, `angle` and `speed` of any moving object. Clocks may not be fully accurate, so some tolerance in their measurements should also be considered. We could also obtain imprecise readings of the `id`'s of the unidentified objects, due to insufficient precision of the reading instruments, or just lack of visibility.

The imprecision of these values should also be propagated through the operations. For example, when computing the attributes of `Mark` objects, their `hitsTarget` attribute should have an associated probability now that its operands incorporate imprecision—representing the fact that we may miss the target—, and then operation `shootAtMarkedThreats()` may only shoot at objects that have been marked by the robot with a probability of hitting them that exceeds 0.9. Without incorporating these uncertainties in the attributes and in the operations, the system specifications would be very unrealistic and naïve, and hence of little value as a faithful representation of the system and its behavior.

Our goal is to address these concerns by capturing these uncertainties in the attributes of the system that represent physical values subject to imprecision, and properly propagating this uncertainty through the operations. To achieve this, we will use the new primitive datatypes `UReal`, `UBoolean`, `UInteger` and `UString`, which will simply replace the types of those attributes of the model that are subject to uncertainty. The advantage of this approach is that the existing OCL expressions in the model will not need to be modified, since the operations of the extended types will naturally take care of the propagation of uncertainty.

The following subsection briefly describe the various types of measurement uncertainties that we consider in this paper and their characteristics, as well as how the newly defined primitive datatypes represent them.

2.2 Measurement uncertainty of numeric values

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 [34] 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 ISO Guide to the Expression of Uncertainty in Measurement (GUM) [34] 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. This uncertainty is specified as a *confidence interval*, which can be expressed in terms of the *standard uncertainty*—i.e., the standard deviation of the measurements of the value. Therefore, a real number x becomes a pair (x, u), which is also noted $x \pm u$, that represents a random variable X whose average is x and standard deviation is u. For example, if X follows a normal distribution $N(x, u)$, 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, the choice of which depends 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 evaluations of measurement uncertainty, it is assumed that the distribution that best describes an input quantity X given repeated measured values of it (which were obtained independently) is a Gaussian distribution. Thus, 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.

In Type B evaluation, uncertainty can also be characterized by standard deviations, which are evaluated from assumed probability distributions based on experience or other information. For example, if we assume that the values of X follow a normal distribution $N(x, \sigma)$, then we set $u = \sigma$. 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 $u^2 = (b - a)^2/12$; hence, $u = (b - a)/(2\sqrt{3})$ [34].

2.3 Propagation of measurement uncertainty

In addition to the measurement or estimation of individual attributes, we typically need to combine them to produce an aggregated measure or calculate derived attributes. For example, to compute the area of a rectangle, we need to consider its height and width and combine them via multiplication. The individual uncertainties of the input quantities must also be combined to yield the uncertainty of the result. The GUM refers to this as the *law of propagation of uncertainty*, or *uncertainty analysis*.

Uncertainty analysis is challenging since combining the probability distributions of individual uncertainties is not a trivial task. The most common case is when the individual uncertainties follow normal or rectangular distributions; either analytical solutions exist or

they can be approximated based on a first-order Taylor series approximation of the combination function [34]. However, there are cases in which two quantities to be combined differ substantially; therefore, analytical solutions for the aggregated uncertainty cannot be easily computed. This is also the case when the linearization of the model provides an inadequate representation or the probability density function for the resulting quantity significantly differs from a Gaussian distribution or a scaled and shifted t-distribution (e.g., due to marked asymmetry). In these cases, the GUM recommends simulation via the Monte Carlo method, which is described in [35]. This approach is based on repeated random sampling from the probability density function of the input quantities $\{x_1, \ldots, x_n\}$ that must be aggregated and the combination of the samples to produce the samples of the derived quantity X. From these, the expected value x of X and its uncertainty u, are calculated as in Type A evaluation of uncertainty—i.e., x as the mean of the samples and u as its standard deviation. Operations on quantities that are specified in this way are performed on the samples since their distributions are unknown or their combinations do not admit analytical solutions. The number of samples depends on the shape and complexity of the probability density function; typically, 10^6 samples are expected to deliver a 95% coverage interval for the output quantity such that this length is correct to one or two significant digits (cf. [35]).

2.4 Uncertainty as confidence

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 insufficient. In this case, each operator must return a number that is between 0 and 1 instead, which represents the probability that one uncertain value is equal, less or greater than another [5]. This leads to the definition of *uncertain Booleans*, which are Boolean values that are accompanied by the level of confidence that we assign to them, namely, pairs (b, c) in which b is `true` or `false` and c is a Real number that is between 0 and 1. This is a proper supertype of `Boolean` and its associated operations (see Section 3.2).

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 $(\texttt{true}, 0.05)$.

This approach should not be confused with *fuzzy logic*. Although both probability theory and fuzzy logic deal with states of uncertainty, fuzzy set theory uses the concept of fuzzy set membership and assumes an element may belong to different sets at the same time. In contrast, probability theory states the chance that an element belongs to each set, assuming it only belongs to one (see [40] for a discussion on the differences between probability theory and fuzzy logic). Uncertainty theory is an alternative approach to deal with belief degrees. However, it changes the way in which product measures are defined in probability theory [48].

2.5 Uncertainty in Strings

In special situations, it is also important to consider uncertainty in values of type `String`. As discussed above, measurement uncertainty is typically due to unreliable sources, symbol recognition problems, or errors in the reading instruments. For example, uncertainty in Strings can be due to mistakes in various characters when applying an Optical Character Recognition (OCR) reader to a handwritten manuscript or when texts in signs are viewed under poor visibility conditions (e.g., at night or with fog).

Measurement uncertainty will be assigned to values of type `String` via a Real value in the range $[0, 1]$ that expresses the confidence that we have in the correctness of the characters that compose that String. Such a confidence is defined using the Levenshtein distance [45], which estimates the number of character changes (additions, deletions and changes) between the current value of the String and its real value, and also takes into account the number of characters in the String. For example, if we are unsure of one character in the String "`Hello`", we will associate to it a confidence of $1/5 = 0.8$. Likewise, if we know that a String that represents a sentence of 500 characters has a confidence of 0.999, we should allow for two characters of the String to be mistaken, misplaced or missing. Operations on uncertain Strings should take into account this uncertainty and propagate it accordingly (see Section 3.6).

2.6 Uncertainty in Enumeration literals

Enumeration types allow modelers to define a bounded set of values (called literals) that can be taken by variables of such types. They are commonly used to represent *nominal* types, which differentiate between items or subjects based only on their names or categories and other qualitative classifications to which they belong.

For example, we can use enumeration types to represent the days of the week (`Monday-Sunday`); in grammar, the parts of speech (noun, verb, preposition, article, pronoun, etc.); or in software engineering, the type of faults (specification faults, design faults, and code faults). Equality between the literals of their values and listing them are the only operations that generically apply to the variables of a nominal type.

Measurement uncertainty can also be associated with the variables of enumeration types because at times we cannot be sure of the literal that we must assign to them. For instance, on some occasions, it is not clear whether a software bug is due to a faulty specification, an incorrect design, or an error in the code. In this case, the solution is to associate a probability to each literal, instead of a single value. Then, the values of variables of an uncertain enumeration type with n literals $\{l_1, ..., l_n\}$ are composed of sets 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 represent the probabilities that the variable takes each literal, with $\sum_{i=1}^{n} c_i = 1$. Then, instead of assigning a value `Spec`, `Design` or `Code` to a variable of enumeration type `SWFaultType`, we can assign the tuple $\{(\texttt{Spec}, 0.1), (\texttt{Design}, 0.3), (\texttt{Code}, 0.6)\}$. The operations of these uncertain enumeration types will be discussed later in Section 3.7.

2.7 Uncertainty in OCL Collections

There are two main dimensions in which measurement uncertainty can be considered in OCL collections (Sets, Bags, OrderedSets and Sequences). First, we can have basic OCL collections of uncertain elements; e.g., a Set of uncertain Reals. This implies the extension of the operations on collections to account for the new types of elements. For example, the operation that sums the values of a collection of `UReal` numbers should be able to return a `UReal` value. Moreover, the operations on collections that return a Boolean value (`forAll`, `exists`) or that select an element based on a Boolean predicate (`select`, `count`) need to be extended to be able to deal with uncertain Booleans.

As a second kind of extension, we can also consider uncertainty in the contents of the collection itself, i.e., when we are not certain of the elements that comprise it. To illustrate this with an example, the reader can think of a Set whose elements are the events produced by a sensor, and there is a possibility that some events are mistakenly inserted, lost, or modified, due to unreliable sources or network connections. Then, there is some degree of uncertainty about the presence of the elements that compose the collection, which may even affect its size. To represent this second kind of uncertainty, we use the same strategy that we applied for extending Strings with uncertainty: by means of associating a Real value in the range $[0, 1]$ to the collection, which represents the confidence that we have on its contents. Once again, we use the Levenshtein distance to specify the number of changes, additions, or deletions that are considered possible in the collection.

To differentiate the two previously mentioned kinds of uncertainty in collections, let us use the term *uncertain collection* (`UCollection`) to refer to the second case (i.e., when we are uncertain about the contents of the collection), and *collections with uncertain elements* to refer to the first one (i.e., when we are uncertain about the values of the elements of the collections). Note that both kinds of uncertainty can be combined since it is possible to define uncertain collections with uncertain elements, for instance, an uncertain Set of uncertain Reals. These two kinds of extensions are described later in Section 3.8.

2.8 Algebraic properties of operations under uncertainty

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 corresponds to the `equals` operation and that 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 $c = 0$).

3 Extension of OCL and UML Datatypes

Our main objective in this paper is to extend the OCL and UML languages by declaring new types that express the types of measurement uncertainties that were

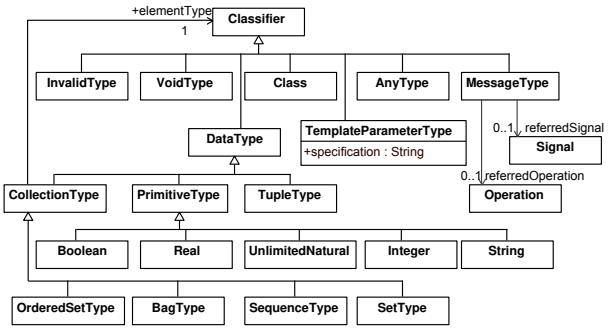


Fig. 2: OCL Types, from [52].

described in the previous section. The benefits are twofold: First, uncertainty can be expressed in software models; therefore, our approach enables the user to define and manipulate uncertainty in a high-level and platform-independent way. This includes model simulation and analysis using the uncertain types and the propagation of uncertainty through these types' operations. Second, information at the model level can be transferred to standard algorithms and tools so that they can also manage uncertainty by dealing with complex types in their computations.

We propose the extension of the OCL datatypes, which are illustrated in Figure 2, with measurement uncertainty information. This includes all primitive types (`Real`, `Integer`, `Boolean`, `String`, and `UnlimitedNatural`), collections (`Set`, `Bag`, `OrderedSet`, and `Sequence`) and enumeration types. Other OCL types, such as `Class`, `Tuple` and `TemplateParameter`, are user-defined and composed of heterogeneous types that will convey such information; hence, there is no need to extend them at this level. Similarly, types `oclInvalid`, `oclAny`, `oclVoid`, and `Message` are of different nature and do not need to transmit measurement uncertainty information. They could however be subject to other types of uncertainties, such as belief or occurrence uncertainty [11], but their treatment requires different notations and specific techniques, and falls outside the scope of this paper.

3.1 Extension strategy

To extend the OCL/UML primitive datatypes, we apply type embedding [7], which is one kind of *subtyping* [46]. Embedding, subtyping and inheritance are different concepts [17]. In broad terms, when we apply *inheritance* among classes, we say that objects of the subclass inherit the internal structure and code of the superclass and, in addition, can have new features (attributes, methods, relationships, etc.). In contrast, *subtyping* refers to the part of the objects' behavior that can be observed from the outside [4], namely, the operations that are applied on them. In algebraic terms, subtyping leads to a conceptual hierarchy that is based on behavioral specification. We say that type A is a *subtype* of type B (denoted as $A <: B$) if all elements of A belong to B and operations of B, when applied to elements of A, behave the same as those of A [3], i.e., they respect behavioral subtyping [46]. 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 viewed as a Real number whose decimal part is zero. Moreover, operations that are defined on the type `Real`, when applied to numbers of the type `Integer`, behave as those operations of type `Integer`.

There are some occasions where the subtyping relation between the types cannot be directly applied. For example, type `Boolean` can be viewed as a subtype of the type `Integer`, but this would imply defining its values as $\{0, 1\}$ instead of $\{\texttt{false}, \texttt{true}\}$. This is interesting because it would also result in many useful mathematical properties that are not available with the latter definition. Type *embedding* permits such an inclusion, by defining the corresponding (*injection*) isomorphism $\{\texttt{false}, \texttt{true}\} \leftrightarrow \{0, 1\}$ and then using subtyping. For example, Real numbers $\mathbb{R}$ can be embedded into Complex ones $\mathbb{C}$ by using the isomorphism $\mathbb{R} \leftrightarrow \mathbb{R} \times \{0\} <: \mathbb{C}$. We will denote such an embedding by $\mathbb{R} \hookrightarrow \mathbb{C}$. Similarly, OCL datatype `Real` can be embedded into the extended datatype `UReal` by considering that each `Real` number has an associated measurement uncertainty of 0, i.e., each Real number x corresponds to the `UReal` number $x \pm 0$. With respect to the behavior of the operations, the fact that $\leftrightarrow$ is an isomorphism and that $<:$ is a subtyping relation, ensures that the behavior of the operations of the embedded type is respected when *lifted* to the embedding supertype.

With this, for extending a primitive OCL datatype `<T>`, we will define an embedding supertype `U<T>` that incorporates information about the uncertainty in the values of `<T>` and defines the operations for the extended type, which are also applicable to the base type. This uncertainty information will vary depending on whether the values of the base type are numbers (types `Real`, `Integer` and `UnlimitedNatural`), Boolean values, Strings or Enumeration literals.

In the first case, the measurement uncertainty information will be expressed as specified in the ISO Guide to Uncertainty in Measurement (GUM) [34]. Thus, numbers of the extended types will be pairs (x, u), where u is the associated uncertainty (cf. Sections 3.3 to 3.5). Operations will respect the subtyping relationship, thereby ensuring safe-substitutability. In the case of Booleans

Table 1: New OCL datatypes and their operations.

Type	Operations
UReal	+,−,∗,/, abs(), neg(), power(), sqrt(), sin(), cos(), tan(), asin(), acos(), atan(), inv(), floor(), round(), <, ≤, >, ≥, =, <>, equals(), distinct(), min(), max(), toString(), toInteger(), toReal(), toUInteger()
UInteger	+, −, ∗, div, /, abs(), neg(), power(), sqrt(), inv(), mod(), <, ≤, >, ≥, =, <>, equals(), distinct(), min(), max(), toString(), toInteger(), toUReal(), toUInteger()
UUnlimitedNatural	+, ∗, div, /, mod, <, ≤, >, ≥, =, <>, equals(), distinct(), min(), max(), toString(), toInteger(), toUReal(), toUInteger()
UBoolean	not, and, or, xor, implies, equivalent, =, <>, equals(), distinct(), equalsC(), toString(), toBoolean(), toBooleanC()
UString	uSize(), uConcat(), uSubstring(), at(), uAt(), equals(), uEquals(), lt(), le(), gt(), ge(), indexOf(), uCharacters(), toString(), uEqualsIgnoreCase(), uToUpperCase(), uToLowerCase(), toBoolean(), uToBoolean(), toInteger(), toReal()
UEnum	equals(), uEquals(), literals()

and Strings, the uncertainty will be specified as a Real number that is between 0 and 1 and represents the assigned confidence (cf. Sections 3.2 and 3.6). Values of enumeration types will assign a probability (confidence) to each literal (Section 3.7).

Table 1 lists the newly defined types and their operations. In addition, the subtyping (<:) and embedding (↪) relationships among the numeric datatypes —both standard and extended—are as follows:

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

Furthermore, we have that Boolean↪UBoolean and String↪UString, which complete these relationships. To extend collections, we will specify them using the corresponding extended operations of their element types. The next sections describe these extensions in more detail. For simplicity, and without loss of generality, in the following we will omit the isomorphism, and directly talk about subtypes and supertypes to refer to embedded subtypes and embedding supertypes.

3.2 Extending type Boolean

Type UBoolean is the embedding 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 in b. Since this representation admits a canonical form (see Section 2.4), true and false are injected into the supertype as $(\texttt{true}, 1)$ and $(\texttt{true}, 0)$, respectively.

The operations supported by type UBoolean extend those of type Boolean, as defined by OCL [52]. 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, Listing 2 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 [52], relying on how the primitive OCL operations deal with these values.

We have preserved the semantics of the equals() and distinct() operations: two UBoolean values are the same if their confidences 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 comparison 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.

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 [35] 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. An excerpt of such a specification, which includes only the first two operations, is shown in Listing 3. An additional invariant, which is at the end of the listing, requests that all samples be of the same size. Similar specifications (and their corresponding implementations in Java) are also available for the rest of the extended types.

Regarding the algebraic properties of the operations for type UBoolean, since they can be equated to numbers that are in the range $[0, 1]$ in the canonical form

```
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
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
equivalent(b : UBoolean) : UBoolean = self.xor(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) or (self.b=not b.b) and ((self.c-1+b.c).abs()<=1-C)
distinct(b : UBoolean) : Boolean = not (self.equals(b))
distinctC(b : UBoolean, C: Real) : Boolean = not (self.equals(b),C)
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
```

Listing 2: Specification of `UBoolean` operations.

```
class UBoolean_A
-- canonical form: pairs (sample[],c), with:
-- sample[]: the set of measured values obtained
--          for self
-- c: the confidence that self is true (derived)
attributes
 sample : Sequence(Boolean)
 c : Real derive: self.sample->count(true)/
                  self.sample->size()
operations
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))))
... -- type invariant:
context UBoolean_A inv SameSampleSize:
UBoolean_A.allInstances->forAll(u,v|
                  u.sample->size=v.sample->size)
```

Listing 3: `UBoolean` specifications using samples.

of the `UBoolean` values, and due to the way in which we have defined the operations, it is easy to prove that the following properties hold for every A and B of type `UBoolean` (operator "$\doteq$" corresponds to the operarion `equals` of type `UBoolean`, see Sect. 2.8).

- `not(not(A))` $\doteq$ `A`
- `not(A or B)` $\doteq$ `not(A) and not(B)` (AND Morgan's Law)
- `not(A and B)` $\doteq$ `not(A) or not(B)` (OR Morgan's Law)
- Operation `and` is commutative, associative and its identity element is $(\texttt{true}, 1.0)$.
- Operation `or` is commutative, associative and its identity element is $(\texttt{false}, 1.0)$.

To show an example of how these properties have been proved, we present here the proof of the first one, i.e., `not(not(A))`$\doteq$`A`. First, if A`.oclIsUndefined` (i.e., $A =$ `null` or A`.oclIsInvalid`) then, by the specification of the OCL `not` operation [52], `not(`A`)`$\doteq A$. Therefore, `not(not(`A`))`$\doteq A$ and hence `not(not(`A`)` $\doteq$ A. Second, suppose A is well defined, i.e., $A = (\texttt{true}, a)$. Then, `not(`A`)` $\doteq (\texttt{true}, 1-a)$, by the specification of the `not` operation. Applying this again, `not(not(`A`))` $\doteq$ `not(true,`$1-a$`)` $\doteq (\texttt{true}, 1-(1-a)) \doteq (\texttt{true}, a) = A$. The rest of the properties can be proved similarly.

The "AND complement" property (`A and not(A)` $\doteq$ `(false,`1`)`) does not hold in this case. 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 $c = 0.5$). A similar result can be demonstrated 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 $c \geq 0.75$.

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 (`true`, 1).
- Similarly, `xor` is commutative, associative, and its identity element is (`false`, 1). However, (`A xor A`) $\doteq$ (`false`, $c(2-c)$).

To check whether `equivalent` is an equivalence relation, we must prove that it is reflexive, symmetric and transitive.

- It is reflexive since `A equivalent A` $\doteq$ (`true`, 1).
- Symmetry holds because the operation is commutative on `Boolean` values.
- Transitivity cannot be ensured: assuming $a \geqslant b \geqslant c$, then ($A$ `equivalent` B) `and` (B `equivalent` C) $\doteq$ (`true`, $(1-b+c)+(a-b)(b-c)$), while (A `equivalent` C) $\doteq$ (`true`, $(1-b+c)$. They coincide in case of `Boolean` values, or if any of the two values are equal. .

Analogous results are obtained for the relation defined by operation `xor`.

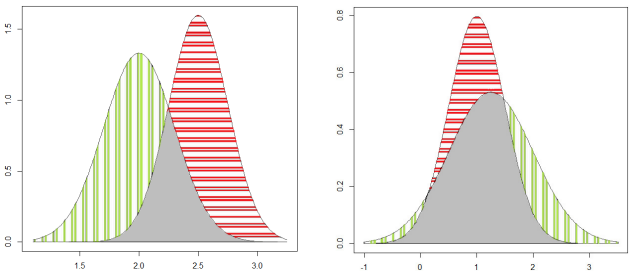
3.3 Extending type `Real`

To represent real values with measurement uncertainty, we use type `UReal` and the algebra of the operations defined for the values of that type, see Table 1. The values of `UReal` are pairs of `Real` numbers which are denoted as $X = (x, u)$. They determine the expected value (x) and associated standard uncertainty (u) of quantity X, as defined in Section 2. Real numbers x are naturally injected into type `UReal` and correspond to pairs $(x, 0)$.

We have specified in OCL and implemented in Java all the operations on the values of type `UReal` to enable modelers to use them to define derived attributes and specify operations and invariants in OCL and UML models. Furthermore, to validate our proposal, we have extended a tool, namely, USE, by implementing the new types as native ones—see Section 4.

As an example, Listing 4 presents the specifications of two of the `UReal` operations:[2]

[2] 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.). 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 the 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. 3: Graphical representations of `UReal` values.

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

Listing 4: Specification of $+$ and $*$ `UReal` operations.

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, in case we cannot assume that the variables are independent, or follow arbitrary distributions. An excerpt of this specification is shown in Listing 5.

```
class UReal_A
attributes
  x : Real derive: self.sample->avg()
  u : Real derive: self.sample->stdDev()
  sample : Sequence(Real)
operations
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)))
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)))
```

Listing 5: `UReal` operations using samples.

Comparison operations between uncertain reals ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$) should now return `UBoolean` values. To illustrate this need, consider the graphical representation of two pairs of uncertain reals that is shown in Figure 3. Indeed, there is 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, 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. 3(a)) is $(0.893, 0.106, 1.11 \cdot 10^{-16})$, which specifies 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 (Figure 3(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 3(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 shown in Listing 6.

To deal with dependent variables using closed-form expressions, we have also specified and developed an implementation of the operations that consider the covariance of the operands, according to the GUM [34]. For example, Listing 7 shows the OCL specification of the `add` and `mult` operations of type `UReal`, considering covariance.

```
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()
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()
```

Listing 7: `UReal` operations with covariance.

Table 1 listed the set of operations that are defined for type `UReal`, including conversion operations to other OCL datatypes (both standard and extended). The properties of those operations, namely, the algebraic properties of type `UReal` for each operation, are of interest. Using the specifications of these operations, the following properties of the four basic arithmetic operations can be proved:

- Operations `+` and `*` are commutative and associative and their identity elements are $(0.0, 0.0)$ and $(1.0, 0.0)$, respectively.
- The distributive property of the `+` and `*` operations only holds if the multiplicator is a `UReal` value. This is because the uncertainty of the sum is accummulated by the multiplication:
  $X_1 * (X_2 + X_3) = (x_1, u_1) * ((x_2, u_2) + (x_3, u_3))$
  $= (x_1, u_1) * (x_2 + x_3, \sqrt{u_2^2 + u_3^2})$
  $= (x_1x_2 + x_1x_3, \sqrt{u_1^2(x_2 + x_3)^2 + x_1^2(u_2^2 + u_3^2)})$
  $= (x_1x_2 + x_1x_3,$
  $\sqrt{u_1^2x_2^2 + u_1^2x_3^2 + \mathbf{2u_1^2x_2x_3} + x_1^2u_2^2 + x_1^2u_3^2})$
  $\neq (x_1x_2 + x_1x_3, \sqrt{(x_1^2u_2^2 + x_2^2u_1^2) + (x_1^2u_3^2 + x_3^2u_1^2)})$
  $= (x_1x_2, \sqrt{x_1^2u_2^2 + x_2^2u_1^2}) + (x_1x_3, \sqrt{x_1^2u_3^2 + x_3^2u_1^2})$
  $= X_1 * X_2 + X_1 * X_3.$
- 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.

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` (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 $(-x, 0.0)$.

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

```
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))
```

Listing 6: Specification of `UReal` comparison operations.

<> 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.

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

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; 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; however, we cannot guarantee that they are antisymmetric or transitive unless they are applied to `Real` numbers. 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.

3.4 Extending type `Integer`

Datatype `UInteger` is the embedding supertype of OCL datatype `Integer` that defines measurement uncertainty. It is needed, e.g., 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 embedding relationship between `Integer` and `UInteger`.

3.5 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 [52]. 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`$\setminus\{$`*`$\}$ <: `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 `*` (internally represented by "-1") are defined by lifting them to type `UInteger`. For illustration purposes, Listing 8 provides the OCL specifications of the comparison operations between `UUnlimitedNatural` values.

3.6 Extending type `String`

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 embedded into `UString` as $(S, 1.0)$.

To calculate the confidence of a string S we will use the Levenshtein distance [45], which is defined as

```
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: result = 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
```

Listing 8: `UUnlimitedNatural` comparison operations.

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

Listing 9: Calculation of `UString` confidence.

"the minimum number of changes (insertion, deletion, or substitution of a single character) needed to transform one string into another." Then, if `dist` represents the number of changes that we estimate that a string S may have, and `len` is the size of the string, the operations shown in Listing 9 enable the calculation of the corresponding confidence of the string and vice-versa: given a confidence, the estimated distance of that string with respect to its real value.

For example, if S = `'Hello world!'`, a confidence of 1 would mean that the string is completely accurate; and 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).

Table 2 lists the newly defined operations for Type `UString`, in addition to those defined for OCL type `String`, which can also be applied to `UString` values. The former operations simply act on the string value, without changing its confidence. In contrast, the operations that are listed in Table 2 operate on both the string value and the confidence.

Listing 10 presents the specifications of some of these operations (the complete specifications of all operations

Table 2: Extended operations of 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

```
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
 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
```

Listing 10: Specification of `UString` operations.

are available from [1,6]). Using these specifications, it is easy to prove that the extensions to type `UString` of all operations that were originally defined in OCL for type `String` respect the original behavior when they are applied on values of type `String`.

The algebraic properties of the extended operations all 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 Listing 10).

```
-- Same substrings uncertainties
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
-- Different substrings uncertainties
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
```

Listing 11: Combining concat and substring operations.

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 in Listing 11.

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.

3.7 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.

```
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()
```

Listing 12: Specification of `UColor` `UEnum` class.

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`, Listing 12 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.

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)}`.

3.8 Extending OCL collections

As mentioned in Section 2.7, two kinds of uncertainties in OCL collections can be considered:

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

Table 3: 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()

```
use> ?Set{0.9,2.0,2}
-> Set{0.9, 2, 2.0} : Set(Real)
use> ?Set{0.9,2.0,2}->forAll(u|u>1.0)
-> false : Boolean
use> ?Set{UReal(0.9,0.1), UReal(2.0,0.1), 2.0}
-> Set{UReal(0.9,0.1),2.0,UReal(2.0,0.1)}:Set(UReal)
use> ?Set{UReal(0.9,0.1), 2.0, UReal(2.0,0.1)}->
         forAll(u|u>1.0)
-> UBoolean(true, 0.1586552596) : UBoolean
use> ?Set{UReal(0.9,0.1), 2.0, UReal(2.0,0.1)}->
         select(u|u>1.0)
<input>:1:40: Argument expression of select must have
        Boolean type, found UBoolean.
use> ?Set{UReal(0.9,0.1), 2.0, UReal(2.0,0.1)}->
         select(u|(u>1.0).confidence()>0.95)
-> Set{2.0,UReal(2.0, 0.1)} : Set(UReal)
```

Listing 13: Examples of operations with Collections.

each subtype: `Set`, `OrderedSet`, `Bag`, `Sequence`. Table 3 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 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 13 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.

Note that, due to the way in which `UBoolean` operations have been defined (see Section 3.2), 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 2.7. 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 [45]. 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 ob-

tained 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, Listing 14 presents the specifications of some of these operations.

```
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
```

Listing 14: Operations on uncertain collections.

4 Tool Support

4.1 USE

The UML-based Specification Environment (USE) [27] is a modeling tool that enables the specification and validation of UML and OCL models. The tool is open-source and distributed under a GNU General Public License. To validate our proposal and to develop a proof-of-concept for it, we extended USE by adding to it the uncertain datatypes defined in this paper, as well as their operations.

The USE tool provides an extension mechanism for adding new operations to the standard OCL primitive datatypes that it initially supports. Such new user-defined operations can be added to the language by implementing their behavior in the Ruby language, and incorporating the Ruby implementations in the file associated to the corresponding type in a USE folder called `oclextensions`. This was the approach we used, e.g. in [62], to add random operations to OCL. Nevertheless, this extension mechanism does not allow the creation of new datatypes, which is exactly what we needed here. Thus, our only choice was to directly modify the USE source code.

Internally, the USE tool uses ANTLR to define the grammars that it supports (USE, OCL, SOIL [13], Shell-Commands, and Generator), and to create the Java lexers and parsers for these languages. Thus, the first step was to extend these grammars (that included the definitions of the OCL primitive datatypes `Real`, `Integer`, etc.) with the new uncertain datatypes. Once the grammars were extended and built, the new lexers and parsers replaced the previous ones. This way, the uncertain datatypes and their operations form part of the language syntax and therefore they become readily available to any UML modeler as basic primitive datatypes of the language.

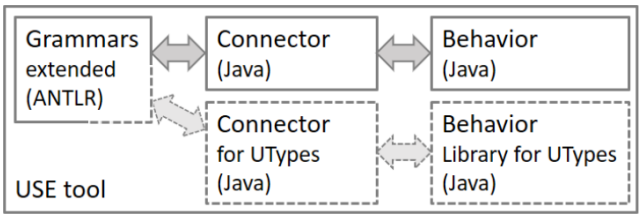


Fig. 4: USE internal architecture.

The constants of the new datatypes can be specified by the name of the type followed by the value. For example, `UReal(0.0,0.1)`, `UInteger(15,0.5)`, `UBoolean(true,0.9)`, or `UString('ABC',0.98)`. Uncertain enumeration values follow the same convention, with the name of the type followed by the corresponding sequence of pairs: `UColor({(#Red,0.9),(#Blue,0.1)})`. Operations on the extended types have been implemented to allow the infix notation, thereby enabling the modeler to use the natural notation in expressions that involve the new datatypes. For instance, it is possible to write `a+b` or `X or Y`, instead of `a.add(b)` or `X.or(Y)`. Operations that involve operands of different datatypes (e.g., `Real` and `UReal`), are always performed in the appropriate supertype.

The next step was to develop the semantics of the uncertain datatypes, i.e., the behavior of their constructors and operations. Since USE is implemented in Java, this library was implemented as a Java project, too. We built the required Java interfaces (i.e., connectors) for binding this library with the USE grammars. These connectors were, again, Java classes linking the syntax of the newly created datatypes with their behavior. For illustration purposes, Fig. 4 shows, at a very high-level of detail, the key components of the USE tool and of our extensions. Solid lines represent original blocks in the USE tool, while dashed lines represent the newly developed blocks. We tried to be as unobstrusive as possible, but in the case of the grammars, we had no other choice than modifying them. This is why the box *Grammars* is partly solid and partly dashed.

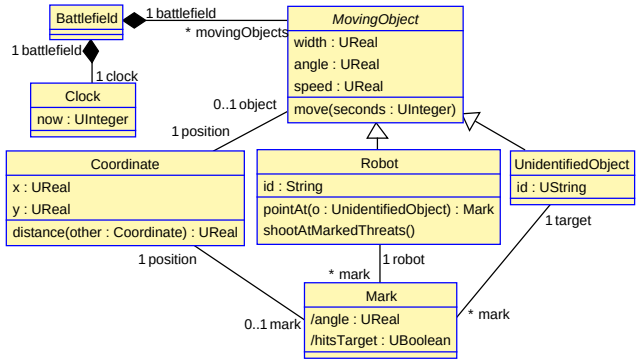


Fig. 5: A Robot Battle System.

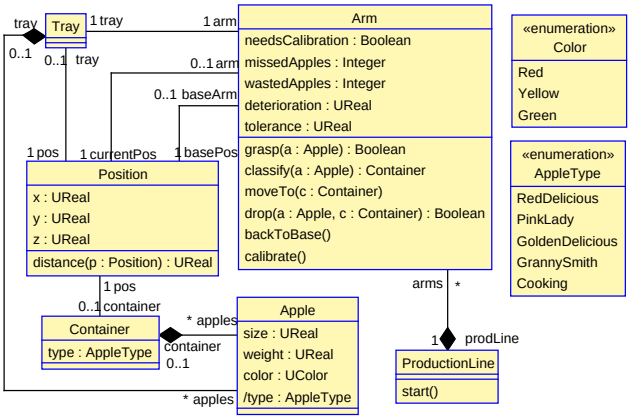


Fig. 6: The Mechanical Arm System.

This extension of the USE tool (available from [1]) has been used to model and simulate all the examples and case studies presented in this paper.

4.2 UML

In UML, a `PrimitiveType` defines a predefined datatype and there is an independent package, named `PrimitiveTypes`, which defines a set of reusable primitive types that are commonly used in the definition of metamodels [54]. UML does not define any substructure for such datatypes, but they do have an algebra and operations, which are defined outside of UML—for example, they can be mathematically expressed. The UML package `PrimitiveTypes` contains the same five primitive datatypes as OCL (`Boolean`, `Real`, `Integer`, `UnlimitedNatural` and `String`), and is specified in a separate XMI document [54, Annex E.3]. UML also permits defining new primitive types, using profiles. Thus, in addition to the extension of the USE tool, we have created a UML package that contains our newly defined datatypes. With it, modelers can incorporate them into their models using the importing and exporting facilities available in their modeling tools. The package was developed in MagicDraw, follows the OMG standard for UML 2.5 and is available in XMI format [1].

5 Applications

This section describes the set of case studies that we have developed to assess our proposal. Each one demonstrates a different aspect of it, illustrating the use and applicability of our extended types, as well as its expressiveness. These examples will be used later in Section 6 when we discuss how we have evaluated our proposal.

5.1 The Robot Battle system revisited

In Section 2.1 we introduced the `Robot Battle` system to motivate our proposal, and the need to faithfully capture the uncertainty of some of the system elements. Figure 5 shows the same model, but where some of the previous types have been replaced by the newly defined datatypes. In particular, the coordinates and properties of moving objects are now declared to be subject to measurement uncertainty; the clock can allow for some tolerance; the identifier of unidentified objects can be imprecise, and the derived values of the attributes the marks, namely, `angle` and `hitsTarget`, can also incorporate uncertainty. As we already mentioned in Section 2.1, there is no need to modify any of the OCL expressions defined in the original model; they are simply evaluated in the context of the extended types, and the uncertainty of the operands is automatically propagated through the numerical and comparison operators.

An interesting value added of this proposal is that we can also consider other system properties, and specify a more realistic system behavior. For example, we could exclude marking unidentified objects whose identifiers had a very low confidence level, in order to avoid shooting at our own drones in cases of low visibility and false readings. Likewise, we could avoid shooting at targets when the confidence of hitting them were low, e.g., less than 50%, in order not to waste ammunition.

5.2 Mechanical arm for classifying apples

This exemplar system is based on an existing project in the context of the Industry 4.0, of a mechanical arm for classifying parts in a production line. It has been characterized here differently for confidentiality reasons, although its main features have been maintained. The system models a production line that classifies apples

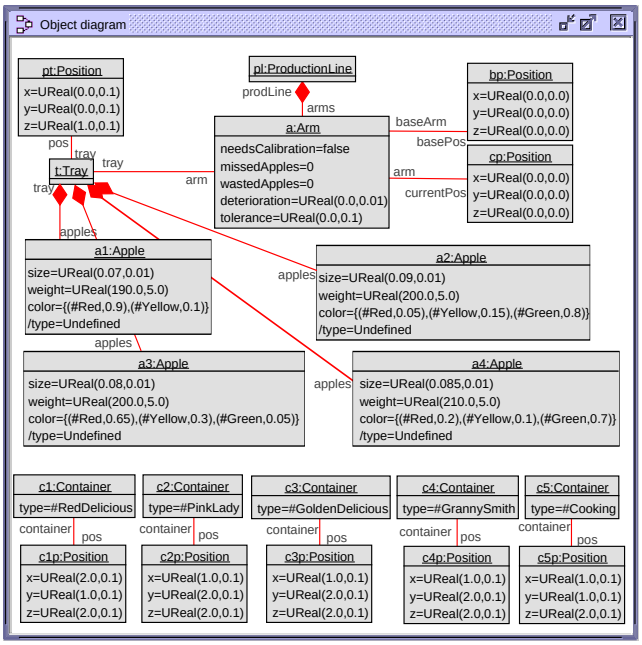


Fig. 7: Initial object diagram, Mechanical Arm system.

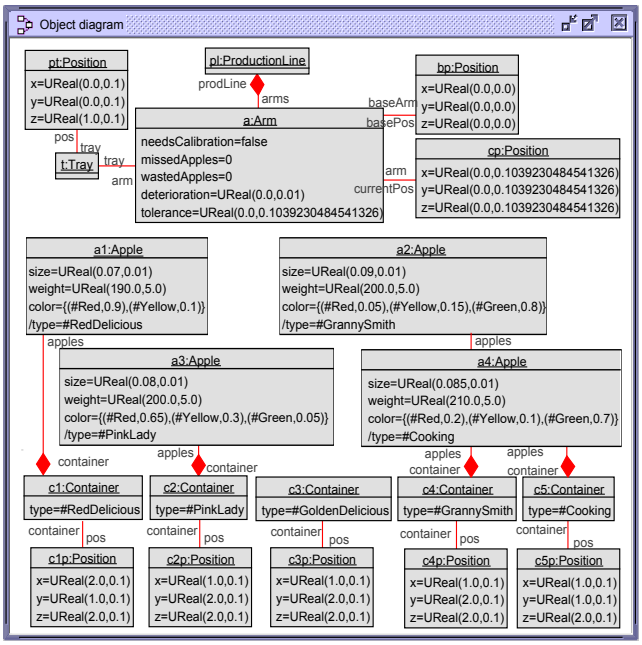


Fig. 8: Mechanical Arm system after 4 classifications.

into four categories, according to their color: *RedDelicious*, *PinkLady*, *GoldenDelicious* and *GrannySmith*. If an apple does not fit into any of these categories, it is considered to be an apple *for cooking*.

The production line is composed of mechanical arms. Each arm is associated with a tray with apples, and has a sensor that measures color and returns the amounts of red, yellow and green that are present in the apple skin. Figure 6 shows the metamodel for this case study. The arm has a base position where it waits to be signaled that there is an apple in the tray. Then, it grabs the apple from the tray and classifies it: if the color contains more than 90% red, the apple is *RedDelicious*; if it is 40% red or more, and less than 10% green, it is *PinkLady*; if it contains more than 80% yellow, it is *GoldenDelicious*; if it is more than 70% green, it is *GrannySmith*; otherwise, it is classified as for cooking. Once the apple has been classified, the arm moves to the corresponding container where the apples of that type are stored, and drops the apple there. Then, it moves back to its base position and waits for more apples to arrive. Every time that the arm returns to its base, its position is reset to the position of the base, which is known precisely.

Attribute `tolerance` indicates the arm positional accuracy. Normally, this parameter is not constant in any mechanical device; it increases with use: every time the arm moves—from its base position to the tray, from the tray to a container, or from a container back to its base position—the slack of its gears and mechanisms slightly increases, thereby producing a progressive degradation of the arm position accuracy. Attribute `deterioration` records this value, which is typically provided by the manufacturer. Although it is usually a very small value, it can significantly accumulate through repeated use. The `deterioration` value is added to the arm's `tolerance` with every movement. Due to this accumulated lack of accuracy, the arm may miss an apple when trying to fetch it. Similarly, it may miss the container when dropping the classified apple. These misses are captured by attributes `missedApples` and `wastedApples`, respectively. The tolerance and deterioration; and, hence, the numbers of missed and wasted apples, depends on the quality of the arm.

Finally, the arm can be calibrated by the manufacturer when its positional accuracy becomes very low. For instance, when the tolerance becomes 10 times higher than the deterioration, its attribute `needsCalibration` may become true, thereby, warning the workers in the plant that the arm must be sent to the manufacturer for recalibration.

Figure 7 shows an initial object diagram of a system with four apples. Figure 8 shows the same system after the apples have been classified. Propagation of uncertainty has been automatically performed via the datatype operations.

It is important to highlight that the type extensions have been able to represent some interesting properties of the system, that otherwise would have been transparent to the user—or that, if needed, would have required

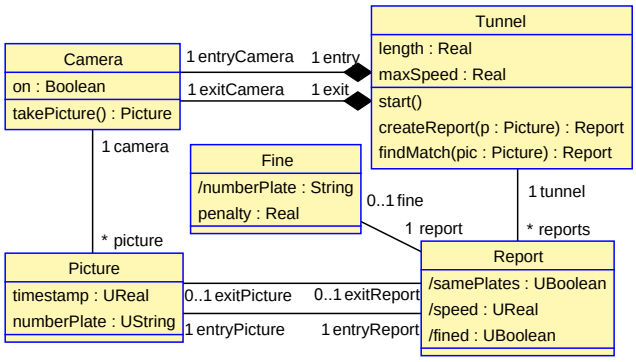


Fig. 9: The traffic control system.

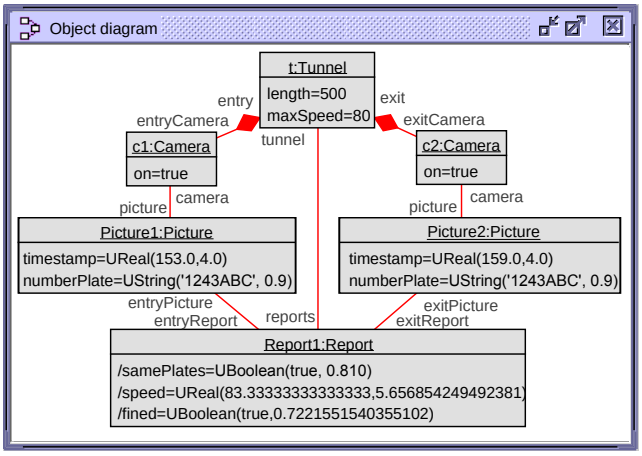


Fig. 10: Object diagram of the traffic control system.

non-trivial modeling efforts, and would have probably resulted in a more complex and cumbersome model. For example, handling the accuracy of the positions of the system elements, or the (imprecise) color assigned to the apples, is not easy with the existing modeling mechanisms available in UML and OCL. With our current proposal, we can also simulate the system with varying uncertainty values, being able to determine the influence of the precision of the individual elements in the overall performance and efficiency of the system; for example, we can conduct simulations to compute the the number of misses and the average time to require calibration, using similar techniques to the ones described in [12] for simulating robot trajectories.

5.3 Traffic control system

This case study models a SPECS average speed camera system for tunnels. For simplicity, we assume tunnels are one-way—the generalization of the system to two-way tunnels would be straightforward.

There is one camera at the entrance of each tunnel and another at its exit. Each time the system detects a movement at the entrance, the corresponding camera takes a picture and creates a report. The picture is processed by an automatic number plate recognition (ANPR) system and the text is stored as a `UString` value, with the confidence associated to it. For every picture, our system also stores, as a `UReal` value, the timestamp of the moment at which it was taken. Similarly, when a movement is detected at the end of the tunnel, the exit camera takes a picture.

The traffic control system checks whether there is an open report with the same number plate and a confidence that exceeds 0.8. If so, the relationship between this second picture and the report is established. The report contains information about the confidence that the two detected number plates are the same (i.e., they belong to the same vehicle), the average speed that is computed based on the timestamps and the length of the tunnel, and whether the vehicle should be fined or not (represented by a `UBoolean` value).

Traffic regulations state that a vehicle should be fined if its average speed exceeds the maximum allowed speed in the tunnel. Therefore, if the attribute `fined` in the Report states with a confidence that exceeds 0.85 that the vehicle should be fined, a `Fine` object is created, with a penalty of 500 if the speed exceeded the limit by less than 30 km/h and 800 otherwise. Figure 9 shows the class diagram for the system.

Figure 10 shows an object diagram, in which the entry camera took a picture of a plate with number `'1243ABC'`, with a confidence of 0.9, entering the tunnel at time 153 ± 4.0, and the exit camera took another of number plate `'1243ABC'` with confidence 0.9 leaving the tunnel at 159.0 ± 4.0.[3] Our system detected that these two plates correspond to the same vehicle with a confidence of 0.810. Since the confidence exceeds 0.8, our system assumes that both plates belong to the same vehicle and, for them, a report was created that indicates that the average speed was 83.33 ± 5.66. The probability that this speed exceeds 80 km/h is 0.72 (72%). Since this probability is lower than 0.85, the vehicle is not fined; therefore, the `Fine` object is not created.

Listing 15 shows how the derived attributes of class `Report` are calculated. Notice that, apart from the chan-

```
context Report::samePlates:UBoolean derive:
   entryPicture.numberPlate = exitPicture.numberPlate
context Report::speed:UReal derive: tunnel.length /
   (exitPicture.timestamp - entryPicture.timestamp)
context Report::fined:UBoolean derive:
   speed > tunnel.maxSpeed
```

Listing 15: Traffic Control System derived attributes.

[3] Although a confidence value of 0.9 is rather low, it may occur due to restricted visibility on foggy days, for instance.

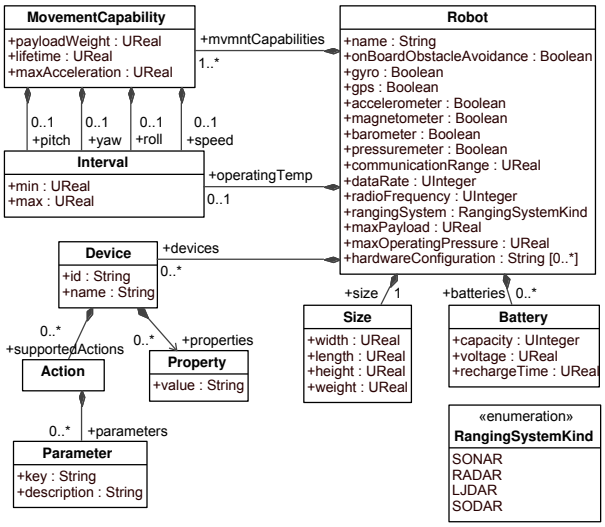


Fig. 11: Excerpt of the Robot language, from [15].

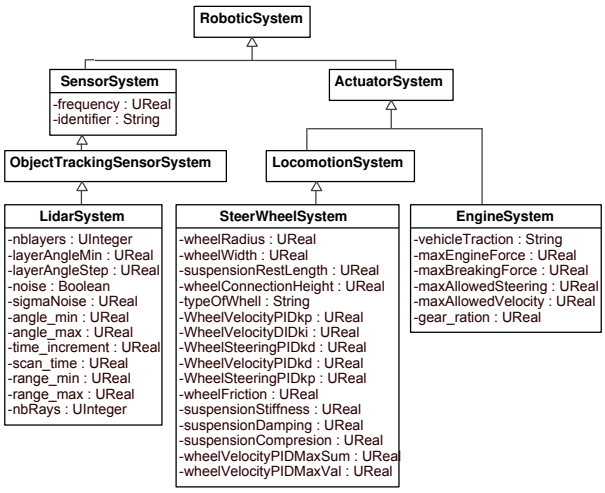


Fig. 12: Excerpt of the RobotML language.

ges in the basic datatypes, the OCL expressions have not changed (i.e., they are exactly the same as they were when dealing with certain values). The extended types take care of calculating the uncertainty and of propagating it though the operations.

Again, the explicit representation of uncertainty enables the specification and analysis of some interesting system properties that otherwise would require a more complex modeling approach. In particular, we now could compute the accuracy of the system depending on the precision of the cameras, or decide changes in the values of the thresholds, or even in the devices, if most of the reports were produced with low confidence.

5.4 A Multi-Robot System for Civilian Missions

This case study describes the family of Domain Specific Languages (DLS) for mobile multi-robot systems proposed by Ciccozzi et al. [15]. The family comprises five languages for modeling: (a) robot missions, (b) the contexts of these missions, (c) robot behavior, (d) robot structure and capabilities, and (e) specific language extensions for modeling particular robot types, such as drones. Overall, the family of languages contains 63 classes and 130 attributes, that together provide a modeling framework that can be reused and extended to model different kinds of robotic applications. In practice, they have been used to model critical missions of autonomous multicopters and unmanned underwater vehicles [15].

To give the reader an idea of the type of language family that we are describing here, Figure 11 shows an excerpt of metamodel of one of these DSLs: the `Robot` language. The core of this metamodel is the `Robot` concept, which permits modeling battery-operated mobile robots by specifying their devices and movement capabilities. The attributes of the classes were originally typed using the primitive UML datatypes: `Real`, `String`, `Integer` and `Boolean`.

In our study, our aim is to identify those attributes that actually represent physical quantities, and that need to account for measurement uncertainty. For this, we studied the attributes of all classes, deciding which ones should be typed with the extended datatypes. For example, Figure 11 shows the `Robot` language metamodel after the types of the attributes representing quantities were replaced by their corresponding uncertain types. MagicDraw UML was used in this case for modeling the system, employing the UML library with the extended uncertain datatypes that we have developed to support measurement uncertainty in UML [1].

5.5 The RobotML modeling language for robotic systems

RobotML [38] is a dedicated language for designing industrial robotic applications, developed as a Papyrus project that uses UML profiles as the metamodeling language. In other words, RobotML is a domain-specific language based on UML. It enables both simulation and deployment to multi-target platforms. Overall, the RobotML language contains 80 classes and 75 attributes.

Figure 12 depicts a small excerpt of the RobotML language metamodel. The `RoboticSystem` is divided into different subtypes of systems, including, e.g., `SensorSystems` and `ActuatorSystems`. These system types

are further refined into more concrete types of sensors and actuators. The figure shows the attributes of the metaclasses already typed using our extended datatypes. As a matter of fact, we realized that more than half of the attributes defined in RobotML metaclasses represent features and properties of physical or mechanical elements, which can be subject to measurement uncertainty. Hence the need and relevance of our proposal, particularly in the realm of systems that deal with physical elements and devices.

6 Evaluation

The evaluation we have conducted to assess our proposal studies its correctness, performance, reusability and interoperability [33] following the evaluation process defined in [44], which in turn is inspired in Adzic's adaptation of Moslow pyramid of human needs to software quality [2]. In the following, we introduce the criteria that we have used to evaluate of our proposal, in terms of the corresponding quality characteristics, and the tests we have conducted to evaluate each criterion. We finish by identifying some of the current limitations of our proposal, including a discussion on other quality aspects such as expressiveness or effectiveness.

First, we followed the ISO Quality Model [33] to identify the quality characteristics to use as evaluation criteria for our proposal. From the ones defined by ISO, the following subset was selected because they were the most appropriate ones in our context:

Correctness: The degree to which a product provides the correct results with the needed degree of precision (ISO/IEC 25010 Functional Correctness [33]).

Performance: The degree to which the response and processing times and throughput rates of a product, when performing its functions, meet requirements (ISO/IEC 25010 Time Behavior [33]).

Reusability: Degree to which an asset can be used in more than one system, or in building other assets [33].

Interoperability: The degree to which two or more systems, products or components can exchange information and use the information that has been exchanged [33].

6.1 Evaluating Correctness

Correcteness represents the extent to which the product complies with its specification and produces correct results [33]. It is specifically concerned with properties that involve the relations between the subprogram's inputs and outputs, as opposed to other properties such as running time or memory consumption.

Firstly, we carried out an internal evaluation of the Java and SOIL implementations that we have developed. A suite of unit tests was defined for executing all their operations using different input values, checking that the results obtained in each case were the expected ones.

The external validity was evaluated by comparing the results of our implementations with the ones returned by two of the existing external mathematical libraries for dealing with measurement uncertainty in programming languages, namely, OpenTurns [49] and the Python's uncertainties package [42]. With these tests, we checked that the operations defined on uncertain types produce correct results, and propagate the uncertainty properly.

We also evaluated our implementations using 'precise' values (i.e., those from the standard OCL and UML datatypes), checking that they produced the same results as the original SOIL and Java implementations, hence respecting the safe substitutability principle [46].

Finally, we endowed our SOIL implementation with the pre and postconditions of the extended operations, as defined in Section 3, so they could be automatically checked in any operation execution. Since the Java and SOIL specifications produce the same results, we indirectly checked that the Java implementations we have developed are also correct.

6.2 Evaluating Performance

To evaluate the overhead introduced by the incorporation of measurement uncertainty in the values of model attributes, we carried out a series of tests for the different operations defined for each type, comparing them with the primitive types in Java. Each test consisted of the repeated execution of the different operations of each type, in order to measure their execution time. Operations were carried out both on the extended types (with uncertainty) and on the basic datatypes, to compare their performance. Each operation was executed 10^6 times to scale out the resulting figures, and each test was repeated 5 times, calculating the median. The tests were conducted on a computer with an Intel Core i7-3770 @ 3.40 GHz (8 CPUs) processor, 8 GB of RAM, and Windows 10 Pro.

Table 4 shows a summary of the overhead introduced by the use of uncertainty. The performance of the operations of types `UInteger` and `UUnlimitedNatural` is the same as those of type `UReal`, and therefore they have been omitted from the table.

Table 4: Performance overhead of operations when incorporating uncertainty.

Type	Operations	Overhead	Time w/o Uncertainty	Time with Uncertainty
UReal	Arithmetic (+, −, /, ...)	×10	$\sim 10^{-7}$ms	$\sim 10^{-6}$ms
UReal	Exponential (power, sqrt)	×100	$\sim 10^{-7}$ms	$\sim 10^{-5}$ms
UReal	Comparison (=, <, ≤, ...)	×300-700	$\sim 10^{-7}$ms	$\sim 10^{-4}$ms
UBoolean	Logical ops. (and, or, ...)	×5	$\sim 10^{-7}$ms	$\sim 10^{-6}$ms
UString	Basic (size)	×10	$\sim 10^{-7}$ms	$\sim 10^{-6}$ms
UString	Concat (+, substring, ...)	×2	$\sim 10^{-5}$ms	$\sim 10^{-5}$ms
UString	Search (index, at, ...)	×15	$\sim 10^{-5}$ms	$\sim 10^{-4}$ms
UString	Comparison (=, ...)	×1	$\sim 10^{-5}$ms	$\sim 10^{-5}$ms

Notice how the propagation of uncertainty has an associated cost, particularly in the case of the comparison operations between numeric values. This is due to the need to calculate the intersection areas of the two Normal curves with the distribution of the possible variation of the numbers to compare, in order to compute the ratio of equality.

Of course, we cannot (and do not want to) compete with the existing software libraries for some programming languages, such as the aforementioned OpenTurns or the Python library. Should a user need to perform intensive computations with uncertain values, it would be probably better to do it from a lower level of abstraction. In any case, the performance figures that we have obtained seem to be acceptable enough for dealing with them in OCL and UML models.

6.3 Evaluating Reusability

Reusability is defined by ISO as the degree to which an asset can be used in more than one system, or in building other assets [33]. To assess the reusability of our proposal, we evaluated how much effort is needed to apply our approach to already existing software models.

In the first place, if our modeling tool incorporates our extended type system, the only effort comes from the identification of the attributes that need to be equipped with measurement uncertainty, and the change of their types. There is no need to modify any OCL expression or constraint, given that the extended types are part of the OCL type system.

In case our modeling tool does not support our extended OCL type system, the new types (and their operations) should be defined in a library and added to the model so that they could be used by the model classes. For this purpose, we developed a UML library in MagicDraw with the new datatypes, as described in Section 4, together with the OCL specifications and the Java implementation. These artefacts were precisely defined to facilitate the reusability of our proposal in other modeling tools. For example, both the RobotML language for the design of robotic applications (Section 5.4) and the the Robot DSL family of languages (Section 5.5) have been easily extended with uncertainty using our UML library, by simply importing it.

Finally, we also wanted to assess how difficult it would be for a tool vendor to implement these extended types in its OCL/UML tool. Of course, it will greatly depend on the tool internal architecture, how extensible it is, and the strategy used to implement the extended types and the subtyping relations among them. In particular, the extension strategy described in Section 3.1 can be implemented in different ways, depending on how the OCL primitive types are already implemented in the tool. If the existing tool defines the primitive types using classes and inheritance among them to implement subtyping (i.e., type `Real` inherits from type `Integer` to implement the subtyping relation `Integer <: Real`), the most natural and less intrusive manner to implement the new types is by extending the existing inheritance tree, using the family polymorphism strategy defined for type groups [10,21], more recently explored in the context of MDE in [30]. Alternatively, if the OCL primitive types are not implemented by the tool as classes but as datatypes (since according to UML the run-time instances of primitive types are just values [54, §10.2.3.2]), the new types can be defined as datatypes too, and embedded into the existing type hierarchy using the strategy described in Section 3.1.

To evaluate this aspect, we measured the effort required to extend the USE tool with the newly defined primitive datatypes, as described in Section 4. USE implements the OCL primitive types as datatypes, so the new ones we have introduced have been implemented in the same way. Roughly, the extension was developed by one person in two months, including the design, implementation and testing of the new tool.

6.4 Interoperability

To evaluate the interoperability of our proposal, we checked how the proposed extensions could be adjusted for data export with interchange data standards. As mentioned in Section 4, the UML model that we developed with the extended datatypes is available in the XMI 2.5 format, following the persistence requirements and format stated by UML to represent primitive datatypes [54, Annex E.3]. With this, any modeling tool can import and use our library of extended types with no major problems.

6.5 Further issues

In addition to the potential benefits of our proposal, this section discusses some issues that we have found during its evaluation.

Table 5: Applicability measures.

Case study	Classes	Attributes	U. Attributes
Robot Battle	7	10	9 (90%)
Mechanical Arm	6	13	8 (62%)
Traffic Control	5	10	5 (50%)
RobotDSL (complete)	63	130	73 (56%)
Behavior	26	24	11 (46%)
Context	7	11	9 (82%)
Drone	9	48	31 (65%)
Mission	13	11	5 (45%)
Robot	8	32	17 (53%)
RobotML	80	75	40 (53%)
Total	161	234	135 (58%)

Effectiveness. Effectiveness is defined in the ISO Quality Model [33] as the accuracy and completeness with which users achieve specified goals. In this respect, we wanted to estimate how extensively current software modeling languages designed for modeling cyber-physical systems make use of attributes that are affected by measurement uncertainty. For this, we computed the percentage of all attributes of our case studies that had to consider measurement uncertainty, and thus required the use of datatypes endowed with this information. Table 5 shows the results for all the case studies presented in this paper. Columns 2 and 3 list the number of classes and attributes of the metamodels of each case study. Column 4 counts the number of attributes that represent physical quantities and thus require the use of measurement uncertainty. For the modeling languages that we have considered in our study, we obtain an application rate of 58% on average, which means that more than half of the languages attributes actually represent quantities, and may significant benefit from capturing uncertainty information. Attributes not referring to quantities are mainly used in these models to introduce identifiers, names and configuration information such as the availability of measurement devices. Although we do not claim that these results can be generalized to all models of physical systems, they provide an indication of the potential effectiveness of our proposal.

Domain expert implication. Related to the use of our new types, it is important to highlight that, when incorporating measurement uncertainty information into a model, sometimes it is difficult to identify the attributes that need to be subject to uncertainty. For this, the judgment of the domain expert is essential, and the communication with them is needed to clarify which attributes should be endowed with this kind of information.

Expressiveness. There are different ways of representing measurement uncertainty, especially for the uncertainty associated to numeric values. They include ranges, probability distributions of the values, or the standard deviation of the variability of the measured attribute. From all the available alternatives, we decided to adopt the ISO VIM recommended representation and management of measurement uncertainty, as defined in the GUM, which is also the notation used in most engineering disciplines. Ranges and other kinds of possible expressions of the measurements deviations can be reduced to this representation, as discussed earlier in this paper (Section 2.2). Similarly, Bayesian Probability [25], Fuzzy logic [68] or uncertainty theory [48] can be used to assign confidence to Boolean values and to Strings. All these theories have advantages and limitations (see, e.g., [19,40,48]) but, as previously mentioned, we decided to use Bayesian Probability, which is the one that, in our opinion, is the most well known and easily understood by software engineers.

Uncertainty in other UML datatypes. We have incorporated measurement uncertainty to the OCL and UML primitive datatypes. What happens with the representation of uncertainty in other datatypes present in UML models, such as date, time, float, double or char? Note, however, that none of these types are defined in the OCL standard, whose type system covers just the primitive datatypes used in this work, but in other UML libraries. In this proposal, we decided to focus only in the basic datatypes (the ones defined in the OCL standard) leaving the extension of other datatypes outside the scope of this work. In any case, the same extension approach that we used here could be employed to extend other datatypes.

Precision and rounding. At this level of abstraction we have not considered precision and rounding, but depending on the particular application, further studies

may be required. In particular, technical aspects such as precision of floating point numbers [28] may have to be investigated. In this respect, requirements from particular domains or applications have to be elicited in more detail.

Dependent variables. In this work we have made some assumptions regarding the independence of the attributes when operating with their associated uncertainty using the closed-form solution. If such an independence cannot be ensured, we provide two ways to deal with dependent (i.e., correlated) variables. First, if we know their covariances, both our specifications and implementations support close-form expressions of the operations with uncertainty in case of dependent variables. However, the values of such covariances are rarely known by users. An alternative solution consists of using the implementation of the operations based on samples (i.e., Type A evaluation of uncertainty). This is also the approach proposed by ISO, which is very general and powerful. However, it may have a significant impact on the performance of the evaluation of the operations, given that they have to be applied to the samples, hence introducing an overhead proportional to the sample size.

7 Related Work

As model-based software engineering is being used to specify IoT and cyber-physical systems, the need to represent and manipulate physical values in software models is becoming evident. In particular, units, real-time properties and the measurement uncertainty of the properties of such systems have been identified as essential aspects that must be modeled [60]. Since timing values are uncertain by nature (they are very often estimates or measured via monitoring instruments), the real-time community is used to representing probability distributions and intervals for timing properties and their influence is clear in the MARTE Profile [53] and the SysML [55] notation. However, neither MARTE nor SysML support operations for performing calculations with imprecise values; their models mainly remain at the descriptive level.

Similarly, in [66], the authors propose a conceptual model, which is called Uncertum, that is supported by a UML profile (the UML Uncertainty Profile; UUP) and enables the inclusion of uncertainty into test models. Uncertum is based on the U-Model [67], and extends it for testing purposes. UUP is a very complete profile that covers many types of uncertainties, in particular, measurement uncertainty. Their focus, namely, testing, differs slightly from ours and they only must represent uncertainty, but do not perform operations with it. Other works on business process models (e.g., [37]) also consider uncertainty when modeling the arrival times of clients, the availability of resources or the durations of tasks. These works use probabilistic mass functions to model the values of the corresponding attributes. We have preferred to use the approach that was defined by the GUM [34,35]. Apart from being the method that has been widely adopted by other engineering disciplines, it has the main benefit of permitting operations on variables that do not follow any particular probabilistic distribution. This occurs, for instance, when defining derived attributes that are obtained via the combination of several quantities that follow diverse, or even unknown, distributions.

The work in [65] defines an XML-based modeling language for measurement uncertainty evaluation that is based on the GUM and a simulation framework for it. This work can be, in principle, considered closely related to our proposal; however, it not being integrated with the type system of any mainstream modeling language (such as OCL or UML), and its low-level syntax (based on plain XML) hinders its usability. Similarly, the work in [31] defines a datatype that incorporates measurement uncertainty and provides libraries for performing computations with its values. The integration of these works with OCL/UML models is not straightforward; therefore, their adoption and use by UML modelers might be limited. To the best of our knowledge, these works are more closely related to existing mathematical libraries and tools for operating with uncertain values, than to our present work.

Representing and reasoning about confidence can be done using different theories. In this work we have used Probability theory, although other authors have proposed other approaches including Possibility theory (based on on fuzzy logic), Plausibility (a measure in the Dempster-Shafer theory of evidence [61]) or Uncertainty theory [48]. The comparison among these theories falls out of the scope of this paper. Our decision was based again on simplicity: probability theory is well-known and understood by most domain experts, who could more easily use it to represent confidence in their model elements. In contrast, the complexity of the other approaches could hinder their correct application and, therefore, could risk their potential benefits.

Other works consider model uncertainty, but focus on different aspects from the ones that we have described here, for instance, on the uncertainty on the models themselves and on the best models to use according to the system properties that one wants to capture [47]. Other works deal with the uncertainty of the design decisions, of the modeling process, or of the domain that is being modeled [20,22,24,26,58,23]. We de-

part from them since we are concerned with the uncertainty of the values of the quantities that are being measured, which is a different problem.

Our work is also related to those approaches that propose extensions to OCL to incorporate new features, or to add required operations, such as aggregation functions [14,57], temporal logic [18], aspects [39], or randomness [62]. These approaches suggest the addition of new operations, but not extensions to the OCL primitive datatypes. Other group of works proposes changes and extensions to the underlying type system of OCL, e.g., [41]. This is not our case because we do not propose to modify the OCL type system, but to incorporate some new datatypes to it.

Finally, several mathematical libraries and tools exist for propagating measurement uncertainty and operating with uncertain values, such as the aforementioned OpenTurns [49] and the Python's uncertainties package [42] (see [64] for a comprehensive list). Their current integration with software models is however limited, since they sit at a lower level of abstraction than software models, as we have already discussed.

8 Conclusions and Future Work

Cyber-physical and IoT systems are promising applications for MBSE methods; however, several challenges may hinder realizing their potential needs. One of them is being able to faithfully represent and operate with measurement uncertainty in the attributes of high-level and platform-independent models.

In this paper, we have focused on representing and managing measurement uncertainty in OCL and UML software models. We have extended the OCL and UML primitive datatypes and their related operations with uncertainty information. OCL and Java libraries have been developed for implementing the type and its operations in MDE settings, and in particular the USE tool already implements the primitive datatypes described here.

This work opens several interesting lines of research that we would like to explore next. First, 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 platform-independent (and, hence, lighter-weight) specifications in OCL/UML can be progressively refined into more concrete, complete (and more complex) platform-specific specifications that use specialized analysis and simulation languages and tools, such as MATLAB/Simulink. Additionally, we would like to further validate our proposal with more case studies, and to test more thoroughly our Java libraries and the USE implementation, which still is a prototype. Another line of future work is the specification and analysis of different strategies for the propagation of the uncertainty, which would lead to parameterized operations. For instance, addition and subtraction operations could have different propagation of the uncertainty according to different (optimistic or pessimistic) heuristics. The analysis of such alternative behaviors, the comparison with the one proposed in this paper (which comes from the ISO VIM), and the study of the properties of these new operations represent interesting lines of future research. Modelers who have used our approach find it easy to use and sufficiently expressive for their purposes; however, further empirical evaluations of its usability and applicability are planned. Finally, the study of further kinds of uncertainties that our models or their elements may be subject to (such as *belief* or *occurrence* uncertainty [11]) are also of interest, and constitute future lines for extensions to the work presented in this paper, which is focused on measurement uncertainty.

Acknowledgements This work was partially funded by the Spanish Research rojects TIN2014-52034-R, TIN2016-75944-R and PGC2018-094905-B-I00. We are really thankful to the reviewers of this paper, for their insighful comments and suggestions.

References

1. Atenea research group Git repository (2018). `https://github.com/atenearesearchgroup/uncertainty`. Accessed: May 20, 2019
2. Adzic, G.: Redefining software quality (2012). `https://gojko.net/2012/05/08/redefining-software-quality/`. Accessed: May 2019
3. America, P.: Inheritance and subtyping in a parallel object-oriented language. In: Proc. of ECOOP'87, pp. 234–242. Springer (1987)
4. America, P.: A behavioural approach to subtyping in object-oriented programming languages. pp. 173–190. John Wiley and Sons (1991)
5. Bertoa, M.F., Moreno, N., Barquero, G., Burgueño, L., Troya, J., Vallecillo, A.: Expressing measurement uncertainty in OCL/UML datatypes. In: Proc. of ECMFA'18, *LNCS*, vol. 10890, pp. 46–62. Springer (2018)
6. 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`. Accessed: May 20, 2019
7. Boute, R.T.: A heretical view on type embedding. SIGPLAN Not. **25**(1), 25–28 (1990)
8. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers (2012)
9. Broy, M.: Challenges in modeling Cyber-Physical Systems. In: Proc. of ISPN'13, pp. 5–6. IEEE (2013)

10. Bruce, K.B., Vanderwaart, J.: Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. Electr. Notes Theor. Comput. Sci. **20**, 50–75 (1999). DOI 10.1016/S1571-0661(04)80066-5
11. Burgueño, L., Bertoa, M.F., Moreno, N., Vallecillo, A.: Expressing confidence in models and in model transformation elements. In: Proc. of MODELS'18, pp. 57–66. ACM (2018). DOI 10.1145/3239372.3239394
12. Burgueño, L., Mayerhofer, T., Wimmer, M., Vallecillo, A.: Using physical quantities in robot software models. In: Proc. of the 1st International Workshop on Robotics Software Engineering (RoSE'18), pp. 23–28. ACM (2018). DOI 10.1145/3196558.3196562
13. Büttner, F., Gogolla, M.: On OCL-based imperative languages. Sci. Comput. Program. **92**, 162–178 (2014)
14. Cabot, J., Mazón, J., Pardillo, J., Trujillo, J.: Specifying aggregation functions in multidimensional models with OCL. In: Proc. of ER'10, *LNCS*, vol. 6412, pp. 419–432. Springer (2010)
15. Ciccozzi, F., Ruscio, D.D., Malavolta, I., Pelliccione, P.: Adopting MDE for specifying and executing civilian missions of mobile multi-robot systems. IEEE Access **4**, 6451–6466 (2016)
16. Ciccozzi, F., Seceleanu, T., Corcoran, D., Scholle, D.: UML-based development of embedded real-time software on multi-core in practice: Lessons learned and future perspectives. IEEE Access **4**, 6528–6540 (2016)
17. Clerici, S., Orejas, F.: GSBL: An algebraic specification language based on inheritance. In: S. Gjessing, K. Nygaard (eds.) Proc. of ECOOP'88, pp. 78–92. Springer (1988)
18. Dou, W., Bianculli, D., Briand, L.C.: OCLR: A more expressive, pattern-based temporal extension of OCL. In: Proc. of ECMFA'14, *LNCS*, vol. 8569, pp. 51–66. Springer (2014)
19. Dubois, D., Prade, H.: Fuzzy sets and probability: Misunderstandings, bridges and gaps. In: Proc. of the IEEE Conf. on Fuzzy Systems, pp. 1059–1068. IEEE (1993). DOI 10.1109/FUZZY.1993.327367
20. Eramo, R., Pierantonio, A., Rosa, G.: Managing uncertainty in bidirectional model transformations. In: Proc. of SLE'15, pp. 49–58. ACM (2015)
21. Ernst, E.: Family polymorphism. In: Proc. of ECOOP'01, *LNCS*, vol. 2072, pp. 303–326. Springer (2001). DOI 10.1007/3-540-45337-7_17
22. Esfahani, N., Malek, S.: Uncertainty in self-adaptive software systems. In: Software Engineering for Self-Adaptive Systems II, no. 7475 in LNCS, pp. 214–238. Springer (2013)
23. Famelis, M., Rubin, J., Czarnecki, K., Salay, R., Chechik, M.: Software product lines with design choices: Reasoning about variability and design uncertainty. In: Proc. of MODELS'17, pp. 93–100 (2017)
24. Famelis, M., Salay, R., Chechik, M.: Partial models: Towards modeling and reasoning with uncertainty. In: Proc. of ICSE'12, pp. 573–583. IEEE Press (2012)
25. de Finetti, B.: Theory of Probability: A critical introductory treatment. John Wiley & Sons (2017). DOI 10.1002/9781119286387
26. Garlan, D.: Software Engineering in an Uncertain World. In: Proc. of the FoSER Workshop at FSE/SDP 2010, pp. 125–128. ACM (2010)
27. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comp. Prog. **69**, 27–34 (2007)
28. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. **23**(1), 5–48 (1991)
29. Greengard, S.: The Internet of Things. MIT Press (2015)
30. Guy, C., Combemale, B., Derrien, S., Steel, J.R.H., Jézéquel, J.M.: On model subtyping. In: Proc. of ECMFA'12, *LNCS*, vol. 7349, pp. 400–415. Springer (2012)
31. Hall, B.D.: Component interfaces that support measurement uncertainty. Computer Standards & Interfaces **28**(3), 306–310 (2006)
32. IEEE Std 1003.1-2008: The Open Group Base Specifications. Issue 7, Sect. 4.16, Seconds Since the Epoch (2016)
33. ISO/IEC 25010:2011: Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. ISO/IEC (2011)
34. JCGM 100:2008: Evaluation of measurement data – Guide to the expression of uncertainty in measurement (GUM). Joint Committee for Guides in Metrology (2008). URL `http://www.bipm.org/utils/common/documents/jcgm/JCGM_100_2008_E.pdf`
35. 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`
36. JCGM 200:2012: International Vocabulary of Metrology – Basic and general concepts and associated terms (VIM), 3rd edition. Joint Committee for Guides in Metrology (2012). URL `http://www.bipm.org/utils/common/documents/jcgm/JCGM_200_2012.pdf`
37. Jiménez-Ramírez, A., Weber, B., Barba, I., del Valle, C.: Generating optimized configurable business process models in scenarios subject to uncertainty. Information & Software Technology **57**, 571–594 (2015)
38. Kchir, S., Dhouib, S., Tatibouet, J., Gradoussoff, B., Simoes, M.D.S.: Robotml for industrial robots: Design and simulation of manipulation scenarios. In: Proc. of ETFA'16, pp. 1–8 (2016)
39. Khan, M.U., Arshad, N., Iqbal, M.Z., Umar, H.: AspectOCL: Extending OCL for crosscutting constraints. In: Proc. of ECMFA'15, *LNCS*, vol. 9153, pp. 92–107. Springer (2015)
40. Kosko, B.: Fuzziness vs. Probability. International Journal of General Systems **17**(2–3), 211–240 (1990)
41. Kyas, M.: An extended type system for OCL supporting templates and transformations. In: Proc. of FMOODS'05, *LNCS*, vol. 3535, pp. 83–98. Springer (2005)
42. Lebigot, E.O.: Uncertainties: a Python package for calculations with uncertainties (2017). URL `https://pythonhosted.org/uncertainties/`. Accessed: May 2019
43. Lee, E.A.: Cyber Physical Systems: Design Challenges. In: Proc. of ISORC'08, pp. 363–369. IEEE (2008)
44. Letier, E., Stefan, D., Barr, E.T.: Uncertainty, risk, and information value in software requirements and architecture. In: Proc. of ICSE'14, pp. 883–894. ACM (2014)
45. Levenshtein, V.: Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Soviet Physics Doklady **10**, 707 (1966)
46. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (1994)
47. Littlewood, B., Neil, M., Ostrolenk, G.: The role of models in managing the uncertainty of software-intensive systems. Reliability Engineering & System Safety **50**(1), 87 – 95 (1995)
48. Liu, B.: Uncertainty Theory, 5 edn. Springer (2018)

49. Michaël Baudin Anne Dutfoy, B.I.A.L.P.: Openturns: An industrial software for uncertainty quantification in simulation (2015). URL `http://www.openturns.org/`
50. Mosterman, P.J., Zander, J.: Industry 4.0 as a cyber-physical system study. Software and System Modeling **15**(1), 17–29 (2016)
51. Oberkampf, W.L., DeLand, S.M., Rutherford, B.M., Diegert, K.V., Alvin, K.F.: Error and uncertainty in modeling and simulation. Reliability Engineering & System Safety **75**(3), 333–357 (2002)
52. Object Management Group: Object Constraint Language (OCL) Specification. Version 2.2 (2010). OMG Document formal/2010-02-01
53. Object Management Group: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Version 1.1 (2011). OMG Document formal/2011-06-02
54. Object Management Group: Unified Modeling Language (UML) Specification. Version 2.5 (2015). OMG document formal/2015-03-01
55. Object Management Group: OMG Systems Modeling Language (SysML), version 1.4 (2016). OMG Document formal/2016-01-05
56. Object Management Group: Structured Metrics Metamodel (SMM) Specification. Version 1.2 (2018). OMG Document formal/18-05-01
57. Pardillo, J., Mazón, J.N., Trujillo, J.: Extending OCL for OLAP querying on conceptual multidimensional models of data warehouses. Information Sciences **180**(5), 584–601 (2010)
58. Salay, R., Chechik, M., Horkoff, J., Sandro, A.: Managing requirements uncertainty with partial models. Requirements Eng. **18**(2), 107–128 (2013)
59. Selic, B.: The Pragmatics of Model-driven Development. IEEE Software **20**(5), 19–25 (2003)
60. Selic, B.: Beyond Mere Logic – A Vision of Modeling Languages for the 21st Century. In: Proc. of MODELSWARD 2015 and PECCS 2015, pp. IS–5. SciTePress (2015). URL `http://cescit2015.um.si/Presentations/KN_Selic.pdf`
61. Shafer, G.: A Mathematical Theory of Evidence. Princeton University Press (1976)
62. Vallecillo, A., Gogolla, M.: Adding random operations to OCL. In: Proc. of MoDeVVa'17, no. 2019 in CEUR Workshop Proc., pp. 324–328 (2017)
63. Vallecillo, A., Morcillo, C., Orue, P.: Expressing measurement uncertainty in software models. In: Proc. of QUATIC'16, pp. 1–10 (2016)
64. Wikipedia: List of uncertainty propagation software (Accessed: May 20, 2019). URL `https://en.wikipedia.org/wiki/List_of_uncertainty_propagation_software`
65. Wolf, M.: A modeling language for measurement uncertainty evaluation. Ph.D. thesis, ETH Zurich (2009)
66. Zhang, M., Ali, S., Yue, T., Norgren, R., Okariz, O.: Uncertainty-wise cyber-physical system test modeling. Software and System Modeling **18**(2), 1379–1418 (2019). DOI 10.1007/s10270-017-0609-6
67. Zhang, M., Selic, B., Ali, S., Yue, T., Okariz, O., Norgren, R.: Understanding uncertainty in cyber-physical systems: A conceptual model. In: Proc. of ECMFA'16, *LNCS*, vol. 9764, pp. 247–264. Springer (2016)
68. Zimmermann, H.J.: Fuzzy Set Theory – and Its Applications, 4 edn. Springer Science+Business Media (2001)