Adding Uncertainty and Units to Quantity Types in Software Models

Tanja Mayerhofer Manuel Wimmer
TU Wien, Austria
{mayerhofer,wimmer}@big.tuwien.ac.at

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

Abstract

Although several software modeling languages permit the representation of key aspects of physical entities, such as units, precision or measurement uncertainty, these aspects are not typically incorporated into their type systems. Therefore, operating with such properties is normally cumbersome and contrived. This paper shows how both data uncertainty and units can be captured in software models and naturally incorporated into their type systems. In particular, we present extensions of the UML/OCL type `Real` and define a set of operations on values of these new types. Furthermore, we show how they can be used in software models to carry out computations that consider measurement uncertainty and permit the detection of unit mismatches when trying to operate with their values.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques—Object-oriented design methods; D.3.3 [*Software Engineering*]: Language Constructs and Features—Data types and structures

Keywords model-based engineering, modeling quantities, measurement uncertainty, dimensions, units.

1. Introduction

The emergence of Industry 4.0 [28] and the proliferation of Cyber-Physical Systems (CPS) [35] are challenging current software models, which now need to faithfully represent key properties of physical world systems and of their elements, often referred to as digital twins, and to integrate them into the software modeling domain. Several authors [23, 37] have already warned about the lack of expressiveness in current software models for handling in an appropriate manner some key aspects of the real world, such as concurrency, units, precision, or real-time properties. For example, a first-class concept of a "physical" value seems to be missing in most programming and software modeling languages, and the problem is that user-defined types without behavioral contracts are not enough: a compiler would still not catch unit mismatches or know how to compare values [37].

To address this issue, in this paper we are concerned with the representation of *Quantities*, which are observable properties of objects, events or systems that can be measured numerically [12]. Quantities are determined by two main attributes: *kind* and *magnitude*. The first one identifies the sort of observable property being quantified, e.g., length, force, time. In turn, the magnitude of the quantity expresses its relative size compared to other quantities of the same kind. A quantity's magnitude and kind are both expressed by means of a *quantity value*, which is given by the product of a *numerical value* and a *unit of measure*.

Moreover, when dealing with objects of the physical world, numerical values need to consider not only the exact values of their attributes, but also some *measurement uncertainty*. As stated in [15], "a measurement result can only be considered complete when it is accompanied by a statement of the associated uncertainty".

The correct representation of numerical values and their units is an essential requirement for faithfully and precisely modeling systems. In fact, most modeling notations that include aspects of physical systems, such as MARTE [29] and SysML [32], already incorporate some elements for representing uncertainty and units. However, representing them in the models is not enough. It is even more critical to be able to carry out computations with them at the level of abstraction of the models—for example, to calculate the values of derived attributes, or to evaluate OCL expressions that represent preconditions on the operations. Otherwise, elements annotated with this information become mere descriptive (decorative) elements. Furthermore, we need to incorporate them into our type systems, in order to be able to, e.g., make calculations with uncertainty; compute the accumulated measurement uncertainty that is propagated when values are aggregated to compute derived measures and op-

SLE'16, October 31 – November 1, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4447-0/16/10...
http://dx.doi.org/10.1145/2997364.2997376

erations are chained; or to detect unit mismatches when trying to combine the values of two quantities.

This paper shows how both data uncertainty and units can be incorporated into software models. In particular, we present an extension of the UML/OCL type `Real` and define a set of operations defined on the values of these new types, and show how they can be used in models. Furthermore, we show how simulations (model executions) can be performed in fUML [26, 33] taking the units and measurement uncertainty into account in a natural manner. Our work builds on a previous paper [39] that shows how to express measurement uncertainty in software models. Here we extend that work by incorporating units and combining both extensions.

This paper is structured as follows. First, Section 2 briefly introduces the concepts related to quantities, values, and units that will be used throughout the paper. After that, Section 3 presents our proposal for representing units and quantities in models, the types we have defined, and introduces a running example that will illustrate our approach. Then, Section 4 describes the algebra of operations on quantities and units. Thereafter, Section 5 discusses the implementations we have developed for Java, OCL, and fUML. Finally, Section 6 compares our work to similar proposals and we conclude in Section 7 with an outlook on future work.

2. Background and Definitions

2.1 Quantities

A *Quantity Kind* (also called *Dimension*) is any observable property of an object or a system that can be measured and quantified numerically. Length, mass, time, force, energy, power and electric charge are examples of dimensions.

A *Quantity* is an observable property of an object, event or system that can be measured and quantified numerically [12]; for example its mass, speed or temperature.

The *Value* of a quantity is its magnitude expressed as the product of a number and a unit. The number multiplying the unit is referred to as the *numerical value* of the quantity expressed in that *unit* [38]; for example, 3.5 m/s.

These concepts are further explained below.

2.2 Units and Dimensions

In order to represent units, we need to determine first their possible dimensions (or *systems of units*), and for each dimension we need to determine the units.

The most widely used system of units is the International System of Units (SI) [38]. It defines seven *base dimensions*: **Length**, **Mass**, **Time**, **Electric Current**, **Thermodynamic Temperature**, **Amount of Substance** and **Luminous Intensity**. The SI determines seven *base units*, one for each dimension: Meter (m), Kilogram (kg), Second (s), Ampere (A), Kelvin (K), Mole (mol) and Candela (cd). The SI also defines 90 *derived dimensions* (Velocity, Force, etc.) and their corresponding units (m/s, *Newton*, etc.).

There is an additional supplementary dimension in the SI, for angles. The SI committee has not yet fully agreed on the nature of this angular dimension, because it is considered dimensionless. However, it is required to represent Angular Velocity (rad/s), Angular Acceleration (rad/s^2), Area Angle ($m^2 st$) and Power per Angle (W/st). Therefore we decided to incorporate it, treating angular units like normal base units. So we will consider eight base dimensions. The base unit for **Angle** is Radian (rad). There is also a derived unit for solid angle measurement, the Steradian (st), which corresponds to rad^2.

A fundamental property of any system of units, which we will heavily exploit in this paper, is that any unit can be derived from the base units it defines, as a product of powers of these base units: $B_1^{e_1} \cdot B_2^{e_2} \ldots B_n^{e_n}$, where the exponents $e_1, \ldots, e_n$ are rational numbers. Thus, in the SI, the representation of any unit can be univocally determined by an 8-tuple $\langle e_1, \ldots, e_8 \rangle$, where e_i is the rational number that represents the exponent of the i-th base unit [12]: Meter (m), Kilogram (kg), Second (s), Ampere (A), Kelvin (K), Mole (mol), Candela (cd) and Radian (rad).

For example, Linear Velocity is a derived dimension whose SI unit is m/s. Using the representation above, it can be expressed as $\langle 1,0,-1,0,0,0,0,0 \rangle$, with 1 in the Length dimension and -1 in the Time dimension; Acceleration, whose SI units are m/s^2, is represented as $\langle 1,0,-2,0,0,0,0,0 \rangle$; and Force, expressed in Newtons (mKg/s^2), is represented as $\langle 1,1,-2,0,0,0,0,0 \rangle$. Tuples for base units contain one value of 1 and the rest of the values are 0.

Dimensionless units (e.g., counts or ratios) are represented by a 8-tuple whose 8 components are 0. As mentioned above, radians (and steradians) are considered in the SI as dimensionless units, but they do have an identity as units. This is why in our proposal they have their own dimension. However, we will distinguish between *dimensionless* and *unit-less* units. Unit-less units are those represented by a 8-tuple whose eight components are 0. Dimensionless units include both unit-less units and angles, hence ensuring consistency with the SI definition of *dimensionless* unit.

2.3 Other Systems of Units

Apart from the SI, there are other systems of units which are used in different countries. For example, the *Centimeter-Gram-Second System* (CGS) is a variant of the metric system that has the same dimensions but uses centimeters, grams and seconds as base units. The *Imperial System* used in UK also defines the same dimensions as the SI, but uses several different units: miles, feet, inches, stones, pounds, Fahrenheit degrees, etc. In USA, the *United States Customary System* (also called USCS or USC) is a variant of the Imperial System that uses different units for fluids.

Since they define the same dimensions, conversions among these systems of units are possible by simply multiplying the quantity values by the corresponding conversion factors. In fact, any unit from any system can be expressed

in terms of SI units, and the conversion among them can be easily defined using multiplication factors and, in some cases, offsets. For example, to convert between miles and meters we only need to multiply by the conversion factor 1609.34. To convert from km/h to m/s the conversion factor is $1000/3600 = 0.277777$. To convert from Celsius to Kelvin the conversion factor is 1.0, but we need an offset of 273.15. From Fahrenheit to Kelvin both a conversion factor (0.55555555556) and an offset (255.372222222) are needed.

The problem, however, is not the conversion itself, but the fact that values expressed in different units can be mixed without any corresponding warning, because the units are not made explicit. This issue has been reported as the cause of some well-known disasters, such as the Mars Climate Orbiter crash [14].

2.4 Numerical Values and Measurement Uncertainty

When dealing with real-world entities, models need to take into account the inability to know, estimate or measure with complete precision the value of any quantity. For instance, in physical systems measurement uncertainty normally arises in partially observable and/or stochastic environments, or when the system properties are not directly measurable or accessible. On other occasions estimations are needed because the exact values are too costly to measure, or simply because they are unknown—for example, the duration of a given task in a software process or the life of a battery. Sometimes values are based on expert judgments and estimations. Such estimates normally feature ranges, or intervals, not exact values, which determine the possible lower and upper bounds for the exact values, or are given by a probability distribution that represents a range of its variation. This is why, in general, a measurement result that determines the value of a quantity is only complete when it is accompanied by a statement of the associated uncertainty [15, 16].

The "Guide to the Expression of Uncertainty in Measurement" (GUM) [15] defines the term *standard uncertainty* as "the uncertainty of the result of a measurement expressed as a standard deviation". This is why instead of giving a single number x to model a measurement result, engineers normally use $x \pm u$ to represent the result, u being the *associated standard uncertainty*. For example, if the measures of a given quantity X follow a Normal distribution with mean x and standard deviation $u = \sigma$, the interval $[x-\sigma, x+\sigma]$ represents a range of its variation, and we know that it will contain 68.3% of the possible values of X.

In the following, we will refer to x as the estimated value and u (or u_x when we want to refer to the precise variable) as its standard uncertainty, and therefore any value X for a quantity will be given by a pair (x,u) where x is the estimated value and u its standard uncertainty. In some cases we will also use the alternative representation $x \pm u$ to refer to the pair (x,u), to improve readability.

Given that normally the interval $[x-u, x+u]$ contains 68.3% of the expected values, the GUM also defines the *Extended Uncertainty*, which multiplies the associated uncertainty by a constant positive integer factor (the *coverage factor k*) to improve the coverage. Then, if we need to consider a wider coverage of that interval (in order to, e.g., account for more values of the measured quantity), we can take the extended uncertainty with $k = 2$ that will account, in case of a Normal distribution, for 95.4% of the values, or $k = 3$ that will account for 99.7% of them.

Finally, quantities are rarely used in isolation, but combined to produce aggregated measures or to calculate derived attributes. The individual uncertainties of the input quantities need to be combined too, to produce the uncertainty of the result. This is known as the *propagation of uncertainty*, or *uncertainty analysis*. This is in general a difficult problem since combining the probability distributions of the individual uncertainties is not a trivial task [15]. In fact, in the general case it does not permit analytical solutions but requires simulations [16].

This is why uncertain values admit two implementations: one that assumes that all the probability distributions of the individual uncertainties follow Normal or Uniform distributions allowing the application of analytic solutions to compute the aggregated uncertainty, and the other that deals with the general case where that assumption cannot hold requiring Monte Carlo simulations. Although being more specific, the first one is more efficient and represents the most usual case. The second one is more general, but requires more number crunching.

3. Representation

3.1 Representing Quantity Values in UML and OCL

Our goal is to extend UML and OCL with a new type that is able to represent and handle quantities in a natural manner. Main benefits include a platform-independent and high-level representation and manipulation of measurement uncertainty and units in UML and OCL models.

To properly model quantities we need to represent their values and their units, and to carry out operations with them. Fig. 1 shows our proposal that is in detail discussed in the following sections.

3.1.1 Modeling Uncertain Values

In the first place, to represent values with measurement uncertainty, we make use of the type `UReal` and the algebra of operations on the values of such a type that we defined in our previous work [39]. Basically, the values of `UReal` are pairs of `Real` numbers $X = (x,u)$. They determine the expected value (x) and associated standard uncertainty (u) of a quantity X, as defined in the previous section.

The major advantage is that this approach provides a natural extension to the UML and OCL type `Real`. The conversion between the subtype and supertype is defined by identifying a real number r with the `UReal` value $(r,0)$. Operations respect the subtyping relationship, i.e., they ensure

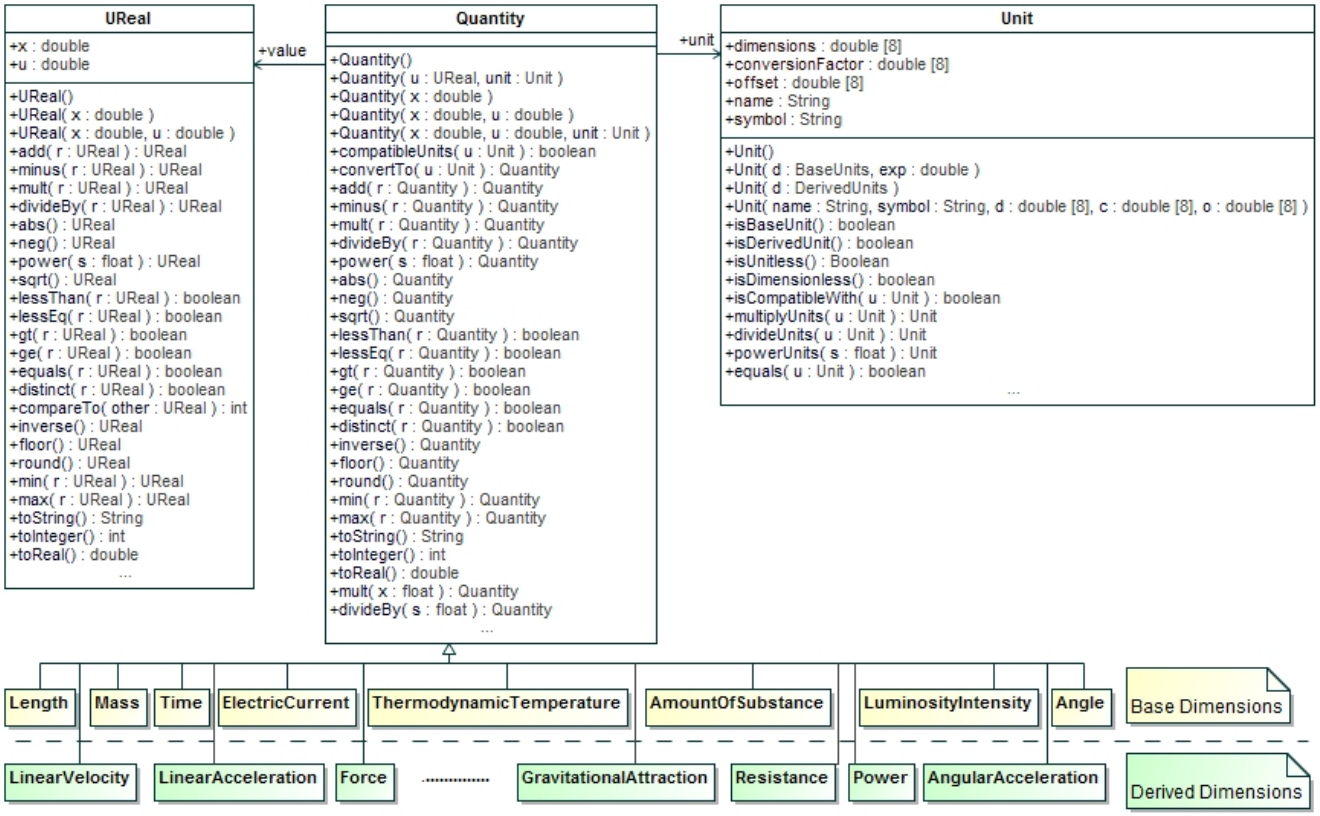

Figure 1. Representation of quantities with their values and units.

safe-substitutability. In other words, `UReal` operations work as before when fed with `Real` values, and operations defined for `Real` values do not need to be modified to work with `UReal` values: their behavior is the same but now incorporating the treatment of uncertainty and its propagation through the different operators as specified in [15].

3.1.2 Modeling Units

Units are modeled by the class `Unit`, which is shown at the right hand side of Fig. 1. It has several attributes to represent the properties of units. First, the `dimensions` array contains the 8-tuple with the exponents of the base units that determine the unit. The attributes `conversionFactor` and `offset` represent the corresponding conversion factors and offsets, respectively, for each base dimension. Finally, one `name` and one `symbol` attribute are required, too, so that users can define their own units.

One characteristic of SI units is that, when represented as units, all the elements of their `conversionFactor` vector are 1.0 and the elements of their `offset` vector are 0.0. For simplicity, in the following we will use the notation $\langle x \rangle$ to represent 8-tuples whose elements are all x.

The class `Unit` also defines a set of operations to investigate their nature (`isBaseUnit()`, `isDerivedUnit()`, `isDimensionless()`, `isUnitless()`, `equals()`) and to combine units (`multiplyUnits()`, `divideUnits()` and `powerUnits()`). These operations will be used when combining quantities. For example, when two quantities are multiplied, their units should be multiplied too. This is carried out by operation `multiplyUnits()`, which adds the two `dimensions` vectors (since their elements represent exponents). Similarly, the operation `divideUnits()` subtracts element by element the two `dimensions` vectors, and the operation `powerUnits(s)` multiplies each element of the vector by the scalar `s`.

The operation `isCompatibleWith()` checks whether two units are compatible for being combined or compared (e.g., miles and centimeters, Fahrenheit and Celsius). In our proposal, this is accomplished by simply checking that their `dimensions` vectors are equal (irrespective of their conversion factors and offsets).

The treatment of offsets requires a separate discussion. One of the benefits of using SI units is that they are all linear functions. This means that we can perform arithmetic operations on their values with no problem, since they all represent absolute values. However, units with offsets (such as Fahrenheit and Celsius) are *affine* (and hence non-multiplicative) units. These temperature units are expressed in a system with a reference point, and relations between temperature units include not only a scaling factor but also an offset. Thus,

it does not make sense to add or multiply two Celsius values [11, 24]. This is where *Delta* units come into play. They represent increments in affine values, and are obtained by simply considering the conversion factor of the unit and ignoring the offset. For example, if two Celsius values are subtracted, a DeltaCelsius value is obtained. Deltas can be added to affine units ($10F + 5\Delta F = 15F$), and delta units can be multiplied and divided (since they represent absolute values). With all this, our proposal imposes the following two requirements in order to provide sound results when operating with affine units: (a) we only allow at most one offset in any unit; (b) we do not allow quantities with a non-null offset in the following two cases: as the argument in addition and subtraction operations; or as any of the operands in `mult()`, `divideBy()`, `power()` and `sqrt()` operations.

Given that the SI specifies the names and symbols of the eight base units and the SI derived units, two auxiliary methods `name()` and `symbol()` provide the correct name of a unit, in case it is a base or derived unit. Finally, the class also supports constructors for easily creating instances of units (both base and derived units) using their symbols, too. For example, we can create an instance of *meter* by simply giving the String `"m"` or an instance of *Newton* by simply giving the String `"N"`. Figure 1 shows some of the constructors defined for this type. In addition, a class `CommonUnits` (not shown here) provides a set of static variables that represent the most commonly used units and their symbols (miles, feet, inches, kilometers, miles per hour, kilometers per hour, Celsius, Fahrenheit, angular degrees, days, hours, minutes, milliseconds, etc.).

3.1.3 Putting all Pieces Together

With all this, a quantity value is represented as a pair (x,u) that expresses the numerical `value` of the quantity and one `unit` which is expressed by three 8-tuples: one that expresses its SI units, one for the corresponding conversion factors for each dimension, and one for the offsets.

For example, a speed of 50 ± 0.0001 miles per hour is modeled by the pair $(50.0, 0.0001)$ that represents the value, and the three following 8-tuples $\langle 1,0,1,0,0,0,0,0\rangle$, $\langle 1609.34,1,3600,1,1,1,1,1\rangle$ and $\langle 0\rangle$. A speed of 3.0 km per minute is modeled by the pair $(3.0, 0.0)$ and the 8-tuples $\langle 1,0,-1,0,0,0,0,0\rangle$, $\langle 1000,1,60,1,1,1,1,1\rangle$ and $\langle 0\rangle$. Then, we can easily add them to obtain a resulting speed of 161.847 ± 10^{-4} miles per hour, or $4.341 \pm 2.68223 \cdot 10^{-6}$ km per minute (although addition is commutative, the units of the result are normally given in the units of the first operand). In SI units, the result of this operation is $72.35194 \pm 4.47038 \cdot 10^{-5}$ m/s. In turn, 36 ± 0.0001 degrees Fahrenheit is modeled by pair $(36.0, 0.0001)$ that represents the value, and the three 8-tuples $\langle 0,0,0,0,1,0,0,0\rangle$, $\langle 1,1,1,1,0.55555555556,1,1,1\rangle$ and $\langle 0,0,0,0,255.372222\ 222,0,0,0\rangle$.

Static type checking of the correct usage of units in operations that involve quantities is achieved by subclassing. Thus, a set of subclassses of class `Quantity` (`Length`, `Time`, `Force`, etc.; see Fig. 1) permits constraining the possible values of the superclass according to the values they are expected to represent, and coerce the types of the parameters of the operations and their return values. Theses classes are the ones that provide the static type checks needed to ensure that units are properly combined. For illustration purposes, Fig. 2 shows the classes `Length` and `LinearVelocity`. We can see that by multiplying two `Length` values an `Area` is obtained, and dividing a `Length` value by a `Time` value produces a `LinearVelocity` value. Similarly, a `Force` value is obtained when multiplying a `LinearVelocity` value by a `MassPerUnitTime` value. However, trying to add something that is not a `Length` to a `Length` would produce an error, which can be statically checked. Of course, values expressed in any unit that is compatible with the one of the class are allowed, which permits adding feet and meters with no problems (although the type system detects whether, for instance, it is tried to add feet and seconds). All valid combinations and their results in our models faithfully conform to the SI definitions [38].

3.2 Motivating and Running Example

As motivating and running example for this paper, we make use of a scenario that deals with the experimental measurement of the average velocity and average acceleration of a toy car as it is for instance described in [36]. The main setting of this example scenario is depicted in Fig. 3. Note that we decided to use this "simple" example as it already shows the strong need for enhanced types to deal with physical quantities, units, and uncertainty.

The running example considers a track, which is divided into different sections (`A`, `B`, etc. in Fig. 3). At the end of each section there is a time measurement procedure as well as a velocity measurement procedure installed. Thus, we can expect for each section to have the following information: (i) the initial and final positions of the section, which gives also the distance to traverse; (ii) the initial velocity (that coincides with the final velocity of the previous section, or with zero in the first section) and the final velocity of the car; and (iii) the duration, i.e., the time needed by the car to travel across the section.

While the information concerning point (i) is measured at design time with some fixed uncertainty of 1 millimeter, the information of points (ii) and (iii) is measured during the system operation and contains uncertainty coming from the applied measurement methods. For instance, the measured velocity at each section end has an associated uncertainty of 1% and the time measurement is uncertain to $\pm 2ms$. Having this measured data with associated uncertainty at hand, the aim is to compute the average velocity and average acceleration of the car for each section as well as for the complete track also considering the uncertainty of the measured data in the computations.

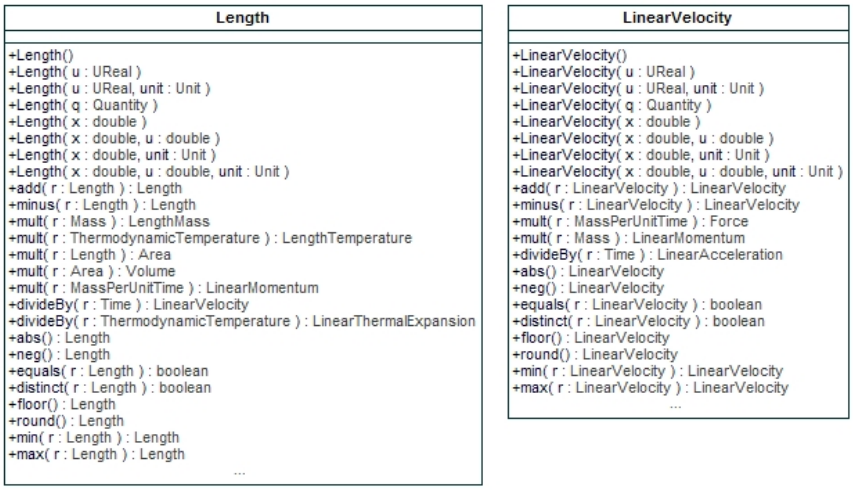

Figure 2. Excerpt of classes `Length` and `LinearVelocity`.

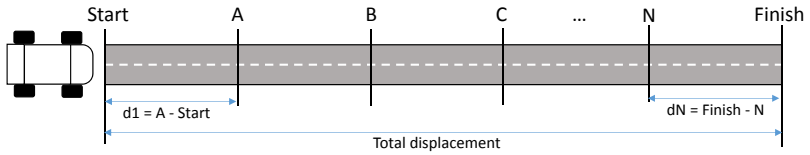

Figure 3. The Toy Car example.

Figure 4 shows a potential modeling solution for representing the introduced scenario on the type level by using a UML class diagram. The focus is on representing the data including the measurements of the discussed scenario as realistically as possible. In particular, the scenario is represented by two classes: one class is representing the vehicles used in the experiments, i.e., the toy cars, and the other class is used for capturing the measured data for each section. The ordered constraint on the reference end `am` is representing the sequential ordering of the different sections of the track.

As can be seen in Fig. 4, we already make use of the previously introduced type system to specify the types of the attributes contained in both classes. In particular, we used `Length` for the positions and for the distance attributes, `Time` for the duration attribute, and `LinearVelocity` and `LinearAcceleration` for the derived attributes concerned with the average velocity and acceleration, respectively, for each section and for the complete track.

Note that all attributes represent uncertain values due to measurement uncertainties. The measured attribute values have either absolute uncertainty values (in the case of position and duration) or relative uncertainty values (in the case of initial and final velocity).

Finally, we make use of OCL expressions to derive the values to be computed, such as the distance, average velocity and acceleration. While all computations have to deal with units, the computations considering the duration, velocity as well as acceleration also have to consider uncertain values. Important to highlight here is that the operations used in the OCL expressions are defined by the introduced types in the previous subsection. Thus, these operations not only take care of the value calculations as known for the `Real` type, but they are also able to deal with uncertainty and units. More information on how these operations are defined is presented in the next section.

A concrete experiment is modeled in Fig. 5 in terms of a UML object diagram instantiating the UML class diagram shown in Fig. 4. The measurement objects for two sections are given. Note that we are using a compact representation of quantities for illustration purposes. In particular, we show for quantity values their estimated value x, their standard uncertainty u and the symbol of their unit.

Now two important questions arise, which are currently not well-addressed by existing software models and their supporting tools[1]: (i) are the models consistent, i.e., is the UML class diagram consistent at all with respect to the used units? and (ii) how is the model interpreted, i.e., how do units and uncertainty influence the computation of values? We provide a solution in the next section, which is a basis to answer these questions.

[1] For instance, the static information about the used units and uncertainty may be also defined in MARTE and SysML, but the impact on computations is not further discussed in these standards.

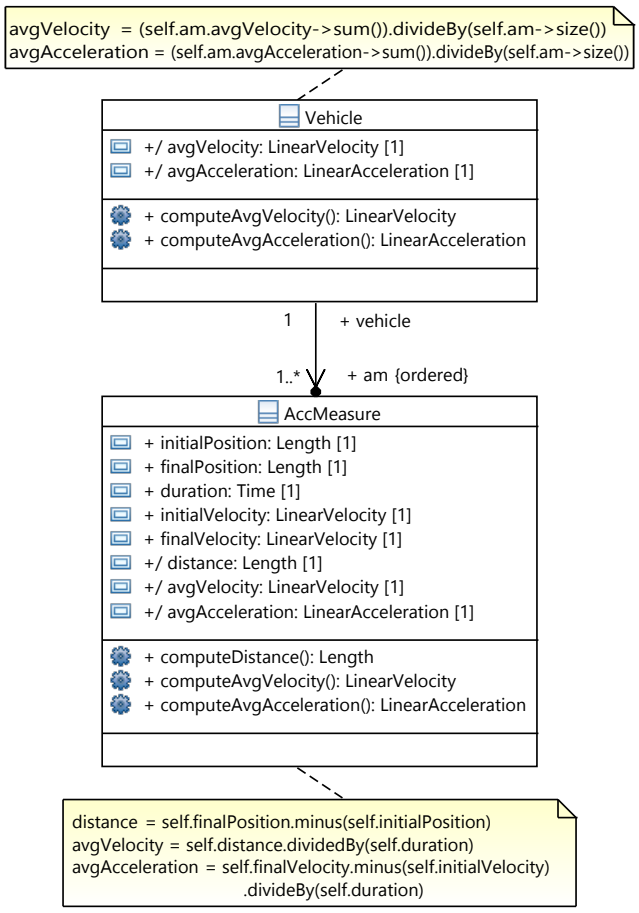

Figure 4. UML representation of the Toy Car system.

4. An Algebra for Operating with Quantity Values

The representation of numerical values and units is important, but it is even more important to be able to manipulate and carry out computations with them in the models, otherwise they become mere decorative elements. Furthermore, we need to incorporate them into our type system, in order to be able to take into account the accumulated measurement uncertainty that is propagated when operations with numbers involving uncertainty are chained; or to detect unit mismatches when trying to combine two numbers.

The first step is to specify the behavior of these types and their operations, independently from their further implementation in any programming language or model execution formalism. This is the goal of this section.

4.1 Operations with `Quantity` Objects

The previously introduced Fig. 1 shows the type `Quantity` and its operations. The type includes operations to interrogate the properties of its values, and to perform computations with them. This section describes their specifications in OCL, independently from any implementation.

The first operation `compatibleUnits()` permits deciding whether the units of two quantities are the same, to check their compatibility for carrying out sums and subtractions:

```
context Quantity::compatibleUnits(u :Unit) :Boolean
    post: result = self.unit.isCompatibleWith(u)
```

The second operation `convertTo()` permits converting the units of a quantity. It takes care of the conversion factors and offsets, as defined below. A precondition states that the two units must be compatible:

```
context Quantity::convertTo(u :Unit) :Quantity
  pre: self.compatibleUnits(x.unit)
  post: result.value =  self.value.mult(self.unit
      .factor()/u.factor()).add((self.unit.offset->sum()
      -u.offset->sum())/u.factor())
     and result.unit = u
```

The auxiliary operation `factor()` computes the aggregated conversion factor of a unit:

```
context Unit::factor() :Real -- reqired for conversions
  post: result = Sequence{1..8}->
       iterate(i : Integer; acc : Real = 1.0 |
          acc*(self.conversionFactor->at(i)).power(self.
              ↪dimensions->at(i)))
```

The next group of operations defines the basic operations on values of this type. Addition and subtraction require the units of the operands to be compatible. They also take into account the restrictions about the offsets described in Sect. 3.1.2. All operations make use of the corresponding operations of types `UReal` and `Unit` described later in Sect. 4.2 and 4.3. They also consider offsets, in different ways.

```
context Quantity::add(x :Quantity) : Quantity
  pre: self.compatibleUnits(x.unit) and
       x.unit.noOffset() -- operand should have no offset
  post: self.unit.noOffset() implies
            (result.value = self.value.add(x.convertTo(
                ↪self.unit).value)
               and result.unit =self.unit)
        and not (self.unit.noOffset()) implies
            (result = self.convertFromSIUnits(self.
                ↪convertToSIUnits().value.add(x.
                ↪convertToSIUnits())))
```

Operation `minus()` is more complex, since it has to take into account the existence (or not) of offsets. In fact, two units with offset can be subtracted, although the result will be a "delta" unit, i.e., with no offset. But if the subtrahend has an offset, the minuend should have it too.

```
context Quantity::minus(x :Quantity) : Quantity
  pre: self.compatibleUnits(x.unit) and (not x.unit.
      ↪noOffset() implies not self.unit.noOffset())
  post: (x.unit.noOffset() and self.unit.noOffset())
      ↪implies -- non of the two units have offsets
       (result.value = self.value.minus(x.convertTo(self
           ↪.unit).value)
         and result.unit =self.unit)
 and (x.unit.noOffset() and not self.unit.noOffset())
     ↪implies -- x has no offset Unit, but "self" has
       (result = self.convertFromSIUnits(self.
           ↪convertToSIUnits().value.minus(x.
           ↪convertToSIUnits())))
  and (not x.unit.noOffset() and not self.unit.noOffset()
      ↪) implies
  -- neither x nor self are Delta Units, but the result
  -- should be...then we convert to Delta"self" unit,
  -- no offset
  (result.value = self.value.minus(x.convertTo(self.unit)
      ↪.value) and
```

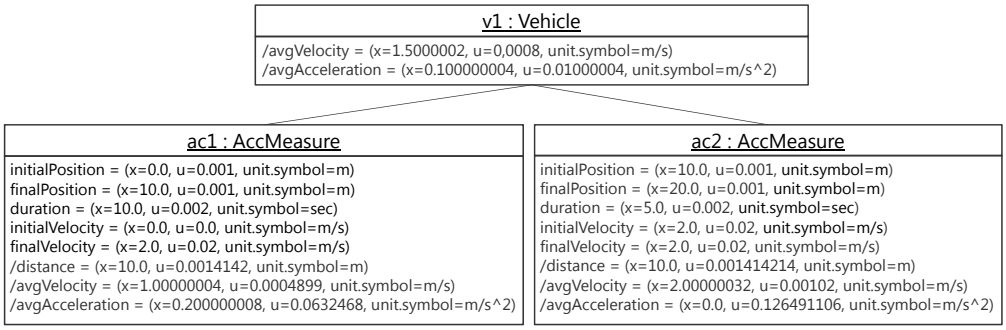

Figure 5. An object model for the Toy Car system.

```
  result.unit.dimensions = self.unit.dimensions and
  result.unit.conversionFactor = self.unit.
      ↪conversionFactor and
  result.unit.offset = Sequence{0,0,0,0,0,0,0,0} and
  result.unit.name = self.unit.name.concat('delta') and
  result.unit.symbol = self.unit.symbol.concat('delta'))
```

These operations make use of two auxiliary methods (not described here for brevity) for converting to/from SI units.

```
context Quantity::convertToSIUnits() :Quantity
context Quantity::convertFromSIUnits(val:UReal):Quantity
```

Another auxiliary method checks that a unit has no offset:

```
context Unit::noOffset() : Boolean
  post: result = (self.offset->count(0.0)=8)
```

The rest of the operations are easier to specify:

```
context Quantity::mult(x :Quantity) : Quantity
  pre:  self.compatibleUnits(x.unit)
        and x.unit.noOffset() and self.unit.noOffset()
        -- both operands should have no offset
  post: let one : Quantity = self.convertToSIUnits() in
        let other : Quantity = x.convertToSIUnits() in
        (result.value = one.value.mult(other.value) and
        result.unit = one.unit.multiplyUnits(other.unit))

context Quantity::divideBy(x :Quantity) : Quantity
  pre: ((x.value.x - x.value.u).max(0) > (x.value.x + x.
      ↪value.u).min(0)) --not x.value.equals(0,0)
       and self.compatibleUnits(x.unit)
       and x.unit.noOffset() and self.unit.noOffset()
       -- both operands should have no offset
  post: let one : Quantity = self.convertToSIUnits() in
        let other : Quantity = x.convertToSIUnits() in
      (result.value = one.value.divideBy(other.value) and
       result.unit = one.unit.divideUnits(other.unit))

context Quantity::abs() :Quantity
  post: result.value = (self.value).abs() and
        result.unit = self.unit

context Quantity::neg() :Quantity
  post: result.value = self.value.neg() and
        result.unit = self.unit

context Quantity::power(s : Real) :Quantity
  pre: s<>0 implies self.unit.noOffset()
  post: result.value = self.value.power(s) and
        result.unit = self.unit.powerUnits(s)
```

Given that the type `Quantity` should be a subtype of `oclAny`, it also has to implement the equal "=" and distinct "<>" comparison operations as defined in the following.

```
context Quantity::equals(x :Quantity) :Boolean
  pre:  self.compatibleUnits(x.unit)
  post: result=self.value.equals(x.convertTo(self.unit))

context Quantity::distinct(x :Quantity) :Boolean
  pre:  self.compatibleUnits(x.unit)
  post: result = not self.equals(x)
```

With the equality operation, we can then define the comparison operations, which also return a `Boolean` value. They also need to check that units match:

```
context Quantity::lessThan(x :Quantity) :Boolean
  pre:  self.compatibleUnits(x.unit)
  post: result = self.value.lessThan(x.convertTo(self.
      ↪unit))

context Quantity::lessEq(x :Quantity) :Boolean
  pre:  self.compatibleUnits(x.unit)
  post: result = self.lessThan(x) or self.equals(x)
```

With the comparison operations, maximums and minimums are easy to define (only `max()` is shown here):

```
context Quantity::max(x :Quantity) :Quantity
  pre:  self.compatibleUnits(x.unit)
  post: result = if self.lessThan(x) then x
                 else self endif
```

Finally, we also need to consider the multiplication and division with scalars (i.e., dimensionless quantities):

```
context Quantity::mult(x :Real) : Quantity
  post: result.value = self.value.mult(x) and
        result.unit.equals(x.unit)

context Quantity::divideBy(x :Real) : Quantity
  pre:  x <> 0.0
  post: result.value = self.value.divideBy(x) and
        result.unit.equals(x.unit)
```

4.2 Operations with `UReal` Objects

The operations described above for type `Quantity` make use of the corresponding operations on the `UReal` type for dealing with values that consider measurement uncertainty. These operations have already been described and fully specified in OCL in [39]. As an example, the following listing shows the specification of two of the `UReal` type operations:

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

4.3 Operations with `Unit` Objects

As mentioned earlier, values of type `Unit` are tuples with eight elements of type `Real`, with the exponents of the base units that define the unit. This type also provides a set of operations for dealing with its values. Their behavior can be specified in OCL as follows (in OCL, the 8-tuple is specified by a `Sequence` of `Real`):

```
context Unit::isCompatibleWith(u :Unit) : Boolean
  post: result = (self.dimensions = u.dimensions)

context Unit::equals(u :Unit) :Boolean
  post: result  = (self.dimensions = u.dimensions) and
    (self.conversionFactor = u.conversionFactor) and
    (self.offset = u.offset)

context Unit::isBaseUnit() :Boolean
  post: result = (self.dimensions->count(1.0)=1) and
    (self.dimensions->count(0.0)=7) and
    (self.noOffset()) and
    (self.conversionFactor->count(1.0)=8)

context Unit::isDimensionless() :Boolean
  post: result = (self.dimensions->count(0.0)=8)

context Unit::isUnitless() :Boolean
  post: result = (self.dimensions->subSequence(1,7)->
      ↪count(0.0)=7)

context Unit::multiplyUnits(Unit x) :Unit
  -- we add them because they are exponents
  -- conversion factors and offsets are not affected
  post: result = self.dimensions->sum(x.dimensions) and
        result.conversionFactor = self.conversionFactor
        and result.offset = self.offset

context Unit::divideUnits(Unit x) :Unit
  -- we subtract them because they are exponents
  -- conversion factors and offsets are not affected
  post: result = self.dimensions->minus(x.dimensions) and
        result.conversionFactor = self.conversionFactor
        and result.offset = self.offset

context Unit::powerUnits(Real s) :Unit
  -- conversion factors and offsets are not affected
  post: result.dimensions->size()=8 and
        Sequence{1..8}->forAll(i :Integer |
           result.dimensions->at(i) =
               s*(self.dimensions->at(i)) and
        result.conversionFactor = self.conversionFactor
        and result.offset = self.offset
```

Finally, some invariants specify the integrity constraints of the type `Unit`:

```
context u : Unit inv eightBaseDimensions:
    u.dimensions->size() = 8
context u : Unit inv threeArraysSameLength:
   (u.offset->size() = 8) and
   (u.conversionFactor->size() = 8)
context u : Unit inv atMostOneOffset:
    u.offset->select(x | x<>0.0 )->size() <= 1
```

5. Implementation

To validate the feasibility of realizing the type system for quantities introduced in Sect. 3, as well as the algebra for operating with them introduced in Sect. 4, we have developed implementations of them for Java, OCL, and fUML. These implementations are discussed in the following and are openly available at our project repository [27].

5.1 Java Implementation

Our Java-based implementation completely realizes the introduced type system and algebra. In particular, it provides an API for conveniently creating quantities with their units and measurement uncertainty, and for performing any of the operations defined for quantities.

Due to the definition of subclasses of the general class `Quantity` dedicated to representing values of specific dimensions (base dimensions or derived dimensions), the compatibility of quantity values for performing operations can be statically checked. As a result, incompatible types used in computations result in compile-time errors.

The following example shows how the Java API is used to instantiate quantities and perform operations on them:

```
Length initialPos = new Length(0,0.0.001,CommonUnits.M);
Length finalPos = new Length(10,0.001,CommonUnits.M);
Length distance = finalPos.minus(initialPos);
```

Two implementations have been developed for Java `UReal` type operations, depending on whether the distribution of the values with uncertainty follow a Gaussian distribution or not [15, 17, 39]. If they do, analytic solutions exist and the implementation is straightforward [15]. If the values to aggregate follow different distributions, a Monte Carlo simulation method is required to implement the operations [17]. These two implementations for type `UReal` in Java are fully described in [39] and available from [27].

Note that the intention behind the development of the Java API was to provide a reference implementation that is easily accessible for software engineers and can be consulted when implementing our proposal for its integration with different modeling languages and modeling frameworks.

5.2 OCL Implementation

OCL is a declarative, non-executable language mostly devised to write integrity constraints on software models, and to specify the behavior of model operations in terms of pre- and post-conditions, independently from any implementation. However, there are some executable extensions of OCL that permit quickly prototyping the specifications. One of them is SOIL (Simple OCL-like Imperative Language) [3], which is part of the USE OCL specification environment [10]. The benefit of this approach is that SOIL specifications can be executed. Although they do not provide a full-fledged execution environment for OCL specifications, and hence are insufficient as a complete computation framework, they can be easily used to have prototypical implementations of OCL specifications. We have used them as a proof-of-concept of our OCL specifications and, thus, as a first step towards the Java and fUML implementations.

As an example, the following listing shows a fragment of the USE commands used to simulate the Toy Car example in USE. You can see how instances of uncertain values and quantities are created, and calculations with them are performed (using the operations specified here expressed in SOIL). Values are displayed with the '?' command.

```
!new URea l('ip1')
!ip1.x :=0.0
!ip1.u :=0.001
...
!new Quantity('initialPosition1')
!new Quantity('finalPosition1')
...
!initialPosition1.value := ip1
!initialPosition1.unit := meter
!finalPosition1.value := fp1
!finalPosition1.unit := meter
...
!distance1:=finalPosition1.minus(initialPosition1)
?distance1.value.x
?distance1.value.u
?distance1.unit.symbol
?distance1.unit.dimensions
```

5.3 fUML Implementation

Besides the Java and OCL implementations of quantities, we have also developed a proof-of-concept implementation for Foundational UML (fUML) [33]. Foundational UML is an executable subset of UML standardized by OMG that comprises UML class modeling concepts for defining static aspects of systems, and UML activity modeling concepts for modeling dynamic aspects of systems. The execution semantics of fUML is defined by the so-called *fUML execution model* that specifies a virtual machine for executing fUML models, in particular, fUML-compliant UML activities referred to as *fUML activities* in the following. Thanks to this virtual machine, it is possible to execute fUML activities and, hence, execute computations with values allowing for the performance of model-level system analyses.

However, the type system of fUML supports only the primitive data types `Boolean`, `Integer`, `Real`, `String`, and `UnlimitedNatural` and operations on their values, but supports neither the representation of uncertain values and units, nor the execution of computations with such values. To overcome this limitation, we extended fUML with support for the newly defined types and the algebras of operations defined for them. To do so, we use fUML's built-in extension mechanism dedicated to extending fUML's type system with custom types and operations on their values suitable for a particular application domain.

In particular, we extended fUML's type system with the new data types `UReal`, `Unit` and `Quantity` as shown in Fig. 6. Please note that this is a model-level extension and not an extension of the fUML/UML language itself, i.e., the new data types are defined in a dedicated fUML (library) model (just as the primitive data types `Boolean`, `Integer`, `Real`, `String`, and `UnlimitedNatural` predefined by the fUML standard) that can be reused for modeling systems involving uncertain values and units.

To define operations on values of data types and make them available for the execution of an fUML model, fUML requires (i) to define their signatures in an fUML library model by means of so-called *function behaviors* that are instances of the UML metaclass `FunctionBehavior` and (ii) to register implementations of these operations at the fUML virtual machine. Note that implementations of operations on data types have to implement a dedicated interface defined by the fUML specification requiring each operation to be implemented in an own class. Thus, even though the reference implementation of the fUML virtual machine is written in Java, our Java-based implementation of quantities cannot be directly reused. More details on how to extend fUML's type system may be found in Section 8.2.2.1 of the fUML specification [33] under the clause "Primitive Behaviors and Primitive Types". Please note that extending the type system of fUML is a built-in extension mechanism that is foreseen in the fUML specification to offer additional custom types.

With our extensions, uncertain values and units can be used in fUML models in a natural way. As an example, Fig. 7 shows an fUML activity computing the distance traveled by a toy car as introduced in our motivating example. The fUML activity first retrieves the initial and final positions of a toy car and then subtracts them by calling the operation `minus()` implemented for the data type `Quantity` as defined above, i.e., taking into account the unit and uncertainty of the values assigned to the attributes `initialPosition` and `finalPosition`. Please note that our implementation for quantity types allows to define attributes that are quantities just like any other attribute that is of a primitive type by assigning the attribute's type to the newly introduced data type `Quantity` or one of its subtypes. Similarly, calls to operations on quantity values (e.g., `Quantity::minus()`) are defined just like calls to any operation defined for any of the primitive data types predefined for fUML (e.g., `IntegerMinus`) by referencing the function behaviors defined for the data type `Quantity` from an fUML call behavior action.

Our fUML implementation is integrated with the Eclipse Modeling Framework [2] and the fUML reference implementation[3]. The fUML library models defining the quantity types and the operations on their values (one of them is shown in Fig. 6), as well as the example model shown in Fig. 7 are defined with the Eclipse Papyrus UML editor[4].

Please note that the developed fUML implementation is only a proof-of-concept implementation showing how quantities can be integrated with fUML. As such, our fUML implementation provides only implementations for a few operations on quantities.

6. Related Work

With respect to the contribution of this paper, we discuss two threads of related work: (i) modeling physical quantities and (ii) measurement uncertainty.

6.1 Modeling Physical Quantities

The need for physical quantities in software models has been discussed in several previous work, e.g., [29–31, 37]. The

[2] `https://eclipse.org/modeling/emf`

[3] `https://github.com/ModelDriven/fUML-Reference-Implementation`

[4] `https://eclipse.org/papyrus/`

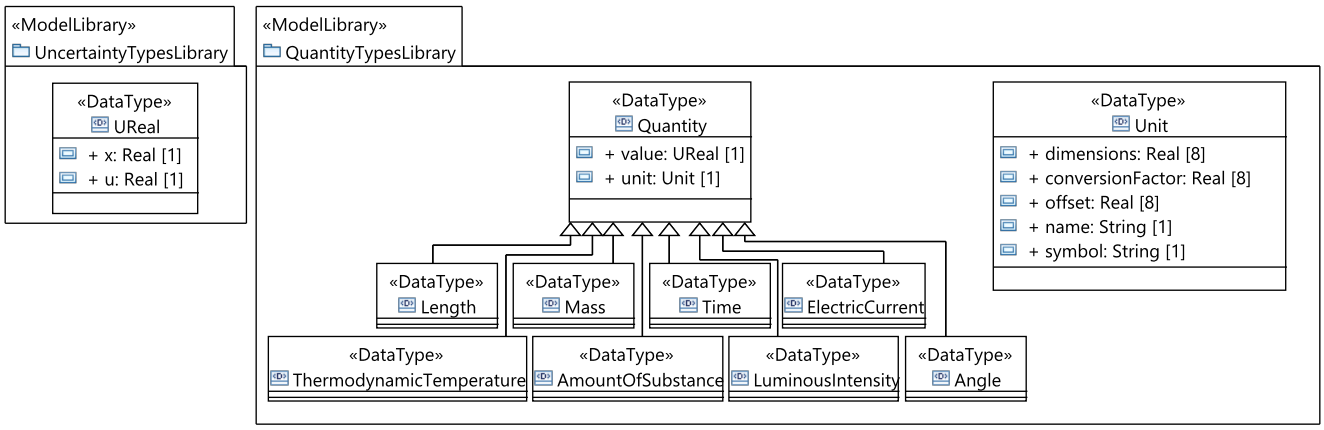

Figure 6. fUML data types for representing quantities.

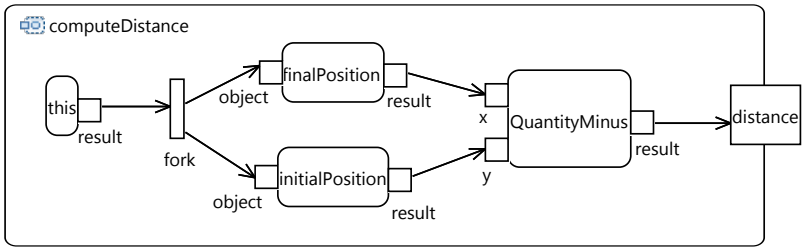

Figure 7. fUML activity computing the distance traveled by a toy car.

representation of such information resulted in several standardization efforts concerning the establishment of guidelines, vocabularies, and ontologies [12, 15, 17, 38, 39].

Concrete approaches for introducing physical quantities in the context of modeling languages are MARTE [29] and SysML [32]. Especially, during the evolution of SysML, different schemata for representing such information have been used, and a model library for Quantities, Units, Dimensions, Values (QUDV) has been established [12]. Furthermore, it is also discussed that the combination of SysML and MARTE, which is a possible and realistic option, may lead to problems when having a mixed usage of the modeling features of SysML and MARTE concerning physical quantities [6]. This discussion also shows that a general library for defining physical quantities may be a valid approach as presented in this paper without requiring this support from specific extensions of UML. Physical systems oriented modeling languages also provide dedicated support for units. For instance, Modelica [9] provides SI unit support[5] as well as different reasoning techniques for the correct and user-friendly usage of units [2, 5, 25]. We have to notice that dedicated support for physical quantities is supported also by commercial products, such as Mathematica [41], which provides very enhanced support for units[6]. Finally, several systems modeling languages for specific domains, such as biology [13] and meteorology [40] provide support for units.

In the context of programming languages, dedicated support for physical quantities is currently under development for different languages, such as Java (e.g., the JSR 363: Units of Measurement API [4]), Python (e.g., see the packages Numericalunits, Pint, Unit, and Uncertainties) [11, 22], Ruby [34] and F# [20, 21]. Units were also implemented, although discontinued, for Eiffel [19]. In addition, language independent design patterns have been proposed to deal with different types of quantities, such as the Quantity Pattern [8] as well as idioms for nominally typed object-oriented programming languages [1]. Furthermore, with the emergence of the Internet of Things (IoT), the need for being able to cope with units and uncertainty is becoming much more evident. If models and programs need to be connected and synchronized to fully achieve MDD, transformations between modeling and programming languages using physical quantities need to be in place.

6.2 Modeling Measurement Uncertainty

Regarding the consideration of measurement uncertainty in software models, several authors have also identified the need of counting on mechanisms to represent and manipu-

[5] see, e.g., `https://build.openmodelica.org/Documentation/Modelica.SIunits.html`

[6] `https://reference.wolfram.com/language/guide/Units.html`

late physical values in software models [37], in particular units or real-time properties. For example, some works on Business Process Models (e.g., [18]) and even some modeling languages also consider uncertainty when modeling the arrival time of clients, the availability of some resources or the duration of some tasks. These works use probabilistic mass functions for modeling the values of the corresponding attributes, instead of fixed values. We have preferred to use the way defined by the GUM [15, 16].

Similarly, the definition and management of uncertainty in measurements is widespread in other domains like real-time systems where, indeed, timing values are by nature uncertain (they are very often estimates and/or measured by means of monitoring). The real-time community is used to exploit probability distributions and intervals for timing properties, and their influence is clear in the MARTE UML Profile [29], which defines `precision` as a tagged definition of an stereotype that can be used to annotate model element attributes with information about the standard uncertainty of their values. However, MARTE does not offer any algebra of operations for making calculations with these stereotyped values. This lack of a neat integration with the type system hinders its usability and ease of use when having to define and compute derived attributes or to perform computations that deal with uncertainty in OCL. In fact, the use of stereotypes significantly complicates the specification of OCL expressions, invariants and operations over the model elements. In this respect, our work could be used to complement the MARTE or SysML standards with a computing kernel that allows the natural definition of operations to deal with measurement uncertainty and units, and its integration with fUML. Model transformations can easily provide the relationship between MARTE and SysML and our proposal in a transparent and clean manner.

In this paper we have focused only on physical units, without considering other kinds of units, such as compound units or money. Incorporating the first one (i.e., being able to express time as 03h:30m:15s, for instance) could probably lead to an over-engineered solution, incorporating something that could be better considered as a representational concern, and thus addressed in a separate (and hence more modular) manner. The second one, although in principle similar, incorporates two issues that induce problems of different matter, as clearly explained by Martin Fowler in [7]. First, the conversion factors are not constant but depend on the daily exchange rate between currencies. Second, and more importantly, money requires a different representation and different implementations of operations. This is because only two decimal digits are used (which makes an Integer representation more suitable) and also because special care should be taken with divisions because of rounding. In fact 10.00 divided by 3 does not result in three quantities of 3.33, but in two quantities of 3.33 and one of 3.34. Otherwise, one cent would be lost in the calculations and this could cause a huge alteration in bank operations that move billions of Euros every day. Similarly, information capacity units (bit, byte, kilobyte, etc.) are also considered for future extensions.

7. Conclusions and Future Work

This paper has presented an approach to deal with measurement uncertainty and units in software models, which is an essential requirement for the representation of elements of physical systems. Our proposal is the definition of the type `Quantity` that provides an algebra of operations for specifying and performing computations with measurement uncertainty and units in attributes representing properties of entities of the physical world. We have defined the proposal and discussed how it can be integrated with fUML for performing computations in a natural and automated way. OCL and Java libraries have also been developed to implement the type and its operations in MDE settings.

This work opens several interesting lines of research that we would like to explore next. First, larger case studies should give more feedback about the expressiveness and applicability of the proposal. Second, we are currently integrating our proposal with existing modeling tools, such as Papyrus and USE [10]. In particular, its connection with fUML [26, 33] provides interesting advantages in this respect. Similarly, the integration of this type and its operations into the simulation and analysis tools that we use to reason about the behavior of systems and their properties represents an interesting challenge. Likewise, the connection of our types with existing mathematical tools for dealing with measurement uncertainty and units could add more powerful computing capabilities to our approach. Likewise, we could also replace our current Java prototypical implementation with other implementations of uncertain values and units available in other programming languages, such as Python [11, 22], Java [4], Ruby [34] or F# [20, 21].

Our proposal clearly provides a *modeling kernel* for dealing with quantities. In addition to these *computational* capabilities, we are also working on enhancing the *presentational* aspects, using more compact representations. In this sense, we are also considering the use of the model-view-controller pattern to enable that MARTE or SysML models (that permit the specifications of these aspects but do not have a proper type system to perform computations and type checking) can be connected to our proposal.

Currently our proposal allows users to define new units. Thus, another interesting extension to our work is the possibility of defining Systems of Units, able to aggregate certain sets of pre-defined units.

Acknowledgments

This project is partially funded by EU Cost Action IC 1404 (MPM4CPS), Spanish Project TIN2014-52034-R, and by the Christian Doppler Forschungsgesellschaft and the BMWFW, Austria.

References

[1] E. E. Allen, D. Chase, V. Luchangco, J. Maessen, and G. L. Steele Jr. Object-oriented units of measurement. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 384–403. ACM, 2004.

[2] P. Aronsson and D. Broman. Extendable Physical Unit Checking with Understandable Error Reporting. In *Proceedings of the 7th International MODELICA Conference*, 2009.

[3] F. Büttner and M. Gogolla. On OCL-based imperative languages. *Sci. Comput. Program.*, 92:162–178, 2014.

[4] J.-M. Dautelle, W. Keil, and L. Lima. *Java JSR 363: Units of Measurement API*, 2016. URL `https://www.jcp.org/en/jsr/detail?id=363`.

[5] K. L. Davies and C. J. Paredis. Natural Unit Representation in Modelica. In *Proceedings of the 9th International MODELICA Conference*, 2012.

[6] H. Espinoza, D. Cancila, B. Selic, and S. Gérard. Challenges in Combining SysML and MARTE for Model-Based Design of Embedded Systems. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, pages 98–113, 2009.

[7] M. Fowler. *Quantity: Represent dimensioned values with both their amount and their unit*. URL `http://martinfowler.com/eaaDev/quantity.html`.

[8] M. Fowler. *Analysis Patters: Reusable Object Models*. Addison-Wesley, 1997.

[9] P. Fritzson and V. Engelson. Modelica – a unified object-oriented language for system modeling and simulation. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 67–90, 1998.

[10] M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.*, 69:27–34, 2007.

[11] H. E. Grecco. *Temperature Conversions*, 2016. URL `http://pint.readthedocs.io/en/0.7.2/nonmult.html`.

[12] R. Hodgson, P. J. Keller, J. Hodges, and J. Spivak. *QUDT – Quantities, Units, Dimensions and Data Types Ontologies*. TopQuadrant, Inc. and NASA AMES Research Center, 2014. `http://qudt.org/`.

[13] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden, et al. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.

[14] D. Isbell and D. Savage. *Mars Climate Orbiter Failure Board Releases Report, Numerous NASA Actions Underway in Response. NASA Press Release 99-134*, 1999. URL `http://nssdc.gsfc.nasa.gov/planetary/text/mco_pr_19991110.txt`.

[15] JCGM 100:2008. *Evaluation of measurement data – Guide to the expression of uncertainty in measurement (GUM)*. Joint Committee for Guides in Metrology, 2008. `http://www.bipm.org/utils/common/documents/jcgm/JCGM_100_2008_E.pdf`.

[16] JCGM 101:2008. *Evaluation of measurement data – Supplement 1 to the "Guide to the expression of uncertainty in measurement" – Propagation of distributions using a Monte Carlo method*. Joint Committee for Guides in Metrology, 2008. `http://www.bipm.org/utils/common/documents/jcgm/JCGM_101_2008_E.pdf`.

[17] JCGM 200:2012. *International Vocabulary of Metrology – Basic and general concepts and associated terms (VIM), 3rd edition*. Joint Committee for Guides in Metrology, 2012. `http://www.bipm.org/utils/common/documents/jcgm/JCGM_200_2012.pdf`.

[18] A. Jiménez-Ramírez, B. Weber, I. Barba, and C. D. Valle. Generating optimized configurable business process models in scenarios subject to uncertainty. *Information & Software Technology*, 57:571–594, 2015.

[19] M. Keller. *Eiffel Units*, 2002. URL `http://se.inf.ethz.ch/old/projects/markus_keller/EiffelUnits.html`.

[20] A. J. Kennedy. Relational parametricity and units of measure. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 442–455. ACM, 1997.

[21] A. J. Kennedy. Types for units-of-measure: Theory and practice. In *Proceedings of the Third Summer School on Central European Functional Programming (CEFP'09)*, volume 6299 of *LNCS*, pages 268–305. Springer, 2010.

[22] E. O. Lebigot. *Uncertainties package*, 2016. URL `https://pythonhosted.org/uncertainties/`.

[23] E. A. Lee. Cyber Physical Systems: Design Challenges. In *Proceedings of 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE, 2008.

[24] Mathworks. *Thermal Unit Conversions*, 2016. URL `http://www.mathworks.com/help/physmod/simscape/ug/thermal-unit-conversions.html`.

[25] S. E. Mattsson and H. Elmqvist. Unit Checking and Quantity Conservation. In *Proceedings of the 6th International MODELICA Conference*, 2008.

[26] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel. xMOF: Executable DSMLs Based on fUML. In *Proceedings of the 6th International Conference on Software Language Engineering (SLE)*, volume 8225 of *LNCS*, pages 56–75. Springer, 2013.

[27] T. Mayerhofer, M. Wimmer, and A. Vallecillo. *Computing with Quantities: the Java Project*, 2016. URL `https://github.com/moliz/moliz.quantitytypes`.

[28] P. J. Mosterman and J. Zander. Industry 4.0 as a cyber-physical system study. *Software and System Modeling*, 15(1): 17–29, 2016. doi: 10.1007/s10270-015-0493-x.

[29] Object Management Group. *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Version 1.1*, June 2011. OMG Document formal/2011-06-02.

[30] Object Management Group. *Object Constraint Language (OCL) Specification. Version 2.4*, Feb. 2014. OMG Document

formal/2014-02-03.

[31] Object Management Group. *Unified Modeling Language (UML) Specification. Version 2.5*, Mar. 2015. OMG Document formal/2015-03-01.

[32] Object Management Group. *OMG Systems Modeling Language (SysML), version 1.4*, Jan. 2016. OMG Document formal/2016-01-05.

[33] Object Management Group. *Semantics Of A Foundational Subset For Executable UML Models (FUML), version 1.2.1*, Jan. 2016. OMG Document formal/2016-01-05, `http://www.omg.org/spec/FUML/1.2.1/PDF/`.

[34] K. C. Olbrich. *Ruby Units*, 2016. URL `https://github.com/olbrich/ruby-units`.

[35] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-Physical Systems: The Next Computing Revolution. In *Proceedings of the 47th Design Automation Conference (DAC)*, pages 731–736. ACM, 2010.

[36] C. Redeker. *Determination of speed and acceleration of a toy car*, Oct. 2007. `http://www.lectures4you.de/pdf/chris_prot/speed-accleration.pdf`.

[37] B. Selic. Beyond Mere Logic – A Vision of Modeling Languages for the 21st Century. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages IS–5, 2015.

[38] B. N. Taylor and A. Thompson. *The International System of Units (SI)*. NIST, 2008. `http://www.nist.gov/pml/pubs/sp811/`.

[39] A. Vallecillo, C. Morcillo, and P. Orue. Expressing measurement uncertainty in software models. In *Proceedings of the 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 1–10, 2016.

[40] M. Wolf. *A Modeling Language for Measurement Uncertainty Evaluation*. ETH, 2009.

[41] Wolfram Research Inc. *Mathematica 10*, 2016. URL `http://www.wolfram.com`.