Specifying Quantities in Software Models (Technical Report)

Tanja Mayerhofer\textsuperscript{a}, Manuel Wimmer\textsuperscript{b}, Loli Burgueño\textsuperscript{b}, Antonio Vallecillo\textsuperscript{c}

\textsuperscript{a}TU Wien, Institute of Information Systems Engineering, Business Informatics, Favoritenstraße, 9-11, (1040) Vienna, Austria
\textsuperscript{b}TU Wien, CDL-MINT, Favoritenstraße, 9-11, (1040) Vienna, Austria
\textsuperscript{c}Universidad de Málaga, Atenea Research Group, Bulevar Louis Pasteur, 35, (29071) Málaga, Spain

Abstract

Context: The correct representation of numerical values and their units is an essential requirement for the design and development of any engineering application that deals with real-world physical systems. Although solutions exist for several programming languages and simulation frameworks, this problem is not fully solved in the case of software models.

Objective: This paper shows how both measurement uncertainty and units can be effectively incorporated into software models, becoming part of their basic type systems.

Method: We describe an extension of UML and OCL type Real, called Quantity, and a set of operations defined on the extended values, together with a ready-to-use library of dimensions (Length, Mass, etc.) implemented in UML, Java, and OCL, that can be added to any modeling project.

Results: We show how our approach permits modelers to safely represent and manipulate units and measurement uncertainties of physical systems and their elements in a natural manner, ensuring statically type- and unit-safe assignments and operations, prior to any simulation or conversion to any programming language.

Conclusion: Our approach permits improving the expressiveness and type-safety of models regarding the units and measurement precision of system attributes, and its effective use in modeling projects of physical systems. This is especially relevant for these kinds of systems, in which more than 50\% of their attributes may require this type of information.

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

1. Introduction

The correct representation of numerical values and their units is an essential requirement for the design and development of any engineering application, particularly in automotive and aerospace domains. Failure to do so have led to disasters such as the Mars Climate Orbiter \cite{1} or the Gimli Glider Incident \cite{2}. Moreover, the emergence of Industry 4.0 \cite{3} and the proliferation of Cyber-Physical Systems (CPS) \cite{4} have made evident the need to faithfully represent and manipulate the key properties of physical world systems and their elements. These not only include units but also measurement uncertainty due to errors in physical measures or to the tolerance of mechanical tools and devices.

Although different solutions exist for representing and manipulating units and uncertainty in programming languages, this problem is not fully solved in the case of software models \cite{5}. In fact, modeling notations that permit dealing with aspects of physical systems, such as the UML Profile for MARTE \cite{6} or SysML \cite{7}, already incorporate some elements for representing units and measurement uncertainty. However, they just offer representation mechanisms, which is not enough. It is even more important to be able to carry out computations with them at the model level to, e.g., calculate the values of derived attributes; to evaluate OCL expressions that represent model invariants and preconditions on the operations, or to compute the accumulated measurement uncertainty that is propagated when values are aggregated. Otherwise, elements annotated with this information become mere descriptive (decorative) elements. Furthermore, uncertainty and units have to be incorporated into the models’ type systems if we have to be able to statically detect unit mismatches when trying to combine values of two quantities or to compute the resulting values and unit of derived attributes and operations—at the model level, i.e., before any implementation is developed or any simulation is conducted, ensuring that the high-level models are correct and free from any unit-mismatch errors.

Technical Report LCC-001-2018

January 13, 2018
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 [8]. 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, Mass, etc. 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. Furthermore, 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 [9], “a measurement result can only be considered complete when it is accompanied by a statement of the associated uncertainty.”

Thus, this paper shows how both data uncertainty and units can be effectively incorporated into software models, becoming part of their basic type systems. In particular, we present an extension of UML and OCL type Real and define a set of operations defined on the extended values, as well as type checks that impede the unit-mismatch problem. We also show how simulations (model executions) can be performed in fUML [10, 11] taking the units and measurement uncertainty into account in a natural manner.

Our work builds on a previous paper [12] that presented the initial proposal. This paper presents several extensions to that work. First, we discuss how to use our concepts from an API perspective at the model level, and how they can be incorporated in metamodels in order to generate quantity-aware domain-specific modeling languages, able to handle uncertainty and units in a natural manner. Second, we abstract away the manner in which quantities are internally represented, providing system modelers with a library of dimensions and units, together with an associated algebra of operations that permit operating with uncertain values, checking unit and dimension mismatches, automatic conversion between units, and propagation of uncertainty. Third, we also allow the specification of custom units and their conversion to existing ones, while we extend the initial proposal with the new dimensions and units defined in the ISO/IEC 80000 standard [13]. Fourth, we provide a novel extension to support comparison operations between uncertain values, which permit returning probabilities and not only logical values when comparing measurement results affected by uncertainty. Finally, we present a real case study where we have used our model library of quantities and units, showing its possibilities and serving as a validation exercise for our proposal.

This paper is structured as follows. First, Section 2 briefly introduces the concepts related to quantities, values, and units. After that, Section 3 presents a motivating example to show the current shortcomings of existing modeling languages, and to illustrate our approach. Then, Section 4 introduces the conceptual domain model we propose for representing units and quantities in models. Section 5 describes the internal representation of quantities, as well as the model type libraries (and associated operations) that realizes the conceptual model, together with the implementation of the algebra of operations on quantities and units. Section 6 discusses how units and quantities can be integrated into modeling languages and describes the integration in Java, UML, OCL, and fUML. In Section 7 we evaluate our approach concerning effort needed to integrate units and quantities into existing modeling languages and usefulness of the obtained unit and quantities support. Finally, Section 8 compares our work to similar proposals and we conclude in Section 9 with an outlook on future work.

2. Background and Definitions

2.1. Quantities

A Quantity is an observable property of an object, event or system that can be measured and quantified numerically [8], for example its position, size, speed or temperature. By convention, physical quantities are organized in a system of dimensions. Length, mass, time, force, energy, power and electric charge are examples of dimensions. These are expressed in units. 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 [14]; 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, and for each dimension we need to determine its possible units.
The most widely used system of units is the International System of Units (SI) [14]. 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 (Area, Volume, Velocity, Force, etc.) and their corresponding units (m², m³, 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/s²), Area Angle (m²st) and Power per Angle (W/st). Therefore we decided to incorporate it, treating angular units like normal base units. The base unit for Angle is Radian (rad).

There is also a derived unit for solid angle measurement, the Steradian (sr). We decided to incorporate it, treating angular units like normal base units. The base unit for Angle is Radian (rad).

Standard ISO/IEC 80000:2009 [13] extends the International System of Units incorporating four new base dimensions and their corresponding units. The new dimensions are Data Storage Capacity, Entropy, Traffic Intensity and Level, with corresponding base units bit, Shannon, Erlang, and Decibel. Other derived units are also defined, including byte for information storage, natural unit of information (nat) and hartley for entropy, and neper for level of sound.

The standard includes all SI prefixes as well as the binary prefixes kibi-, mebi-, gibi-, etc., originally introduced by the IEC to standardise binary multiples of byte, to distinguish them from their decimal counterparts such as megabyte (MB). Binary prefixes are not limited to units of information storage, and then a frequency ten octaves above one hertz, i.e., 2⁴₀ Hz (1024 Hz), is one kibihertz (1 KHz).

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. For example, Linear Velocity is a derived dimension whose SI unit is m/s. Using the representation above, it can be expressed as (1, 0, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) with 1 in the Length dimension and −1 in the Time dimension; Acceleration, whose SI units are m/s², is represented as (1, 0, −2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) and Force, expressed in Newtons (mKg/s²), is represented as (1, 1, −2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0). Tuples for base units contain one value of 1 and the rest of the values are 0.

Dimensionless units (e.g., scalars, counts, or ratios between quantities of the same dimension) are represented by a 12-tuple whose 12 components are 0.

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, 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 [11].

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” [9] [13].

The Guide to the Expression of Uncertainty in Measurement (GUM) [9] 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, civil, mechanical and industrial 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 $u$, 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 $u_a$ 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.

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

In Type A evaluation, if $X = \{x_1, \ldots, x_n\}$ is the set of measured values, then the estimated value $x$ is taken as the mean of these values, and the associated uncertainty $u$ as their experimental standard deviation, i.e., $u_x^2 = \frac{1}{n(n-1)} \sum_{i=1}^{n}(x_i-x)^2$ [9].

In Type B evaluation of uncertainty, only lower and upper bounds $[a,b]$ for the values of $X$ are known, without any further information about the possible values of $X$ within the interval. Thus, we can only assume a uniform or rectangular distribution of the values, and therefore $x$ is calculated as the midpoint of the interval, $x = (a+b)/2$, and its associated variance is taken as $u^2 = (b-a)^2/12$. Therefore, $u = (b-a)/(2\sqrt{3})$ [9].

With this, we can also calculate the uncertainty associated with variables whose variation is given in terms of percentages. For example, imagine that the estimated value $x$ is subject to a variation of $\varepsilon$% of its nominal value, i.e., the expected value of $x$ lies within the interval $[x(1-\varepsilon), x(1+\varepsilon)]$, and, then, the associated uncertainty is $u = x\varepsilon/\sqrt{3}$. Of course, in case there is information available about the distribution of the values within the $[a,b]$ interval (e.g., a Normal distribution if they symmetrically concentrate around the midpoint or a $\mathcal{U}$ distribution if they concentrate in the endpoints), the mean and standard deviation of such a distribution should be used instead.

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 [9]. In fact, in the general case it does not permit analytical solutions but requires simulations [13].

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. Motivating Example

For illustration purposes, this section describes a simple example of a system that requires units and measurement uncertainty, and represents it with two widely used modeling notations for modeling physical systems: SysML [7] and MARTE [6].

The system represents an object (e.g., a particle) moving along a linear path. The particle is periodically observed. Three measurements are taken at each observation: distance from origin (position), time, and current speed (velocity). We are interested in analyzing the particle’s movements, for which we use segments, each one defined by an starting and an ending point, where observations are made. For every segment, we want to know the total distance traversed by the particle in that segment, the duration of the movement, and the average speed and acceleration of the particle. Measuring instruments have some tolerance, hence incurring in measurement uncertainties which should also be represented and taken into account in the models.

First, Figure 1 shows the representation of the system using standard UML, where there is no support for modeling units nor the precision of the measurements. Note that this is the usual manner in which these systems are modeled (cf. Section 7), using mere real numbers and explaining in the companion documentation the units in which each attribute should be expressed. Precision is normally ignored, or considered somewhere else in the models.

Figure 2 shows the example modeled using the UML profile for MARTE, a notation with a priori support for units and measurement uncertainty. The exemplar model first defines the quantities to be used, as extensions of NFP_Real type, which is a «NfpType» type in MARTE. Units are defined in terms of unit kinds, specified by means of «Dimension» classes that determine the base and derived units used in the model. MARTE provides the precision attribute, of type Real, to represent measurement uncertainty [6].

In turn, Figure 3 shows the example modeled using SysML. This notation uses Blocks instead of Classes, and it provides two alternative representations for modeling quantities. The first one, shown on the left-hand side of the figure, is similar to the MARTE approach and defines the types of the quantities to be used in the model. Quantity
dimensions are defined using «valueType» types. Unlike MARTE, SysML provides a standard library for dimensions and units, which can be used to specify each Quantity. This is very useful and beneficial, since all SysML models can rely on the same library of units.

Figure 3: The Moving Particle example modeled using SYSML (two options).

An alternative representation provided by SysML permits specifying the units in which values are expressed as the types of the object attributes that represent these values. This is shown in the model on the right-hand side of Figure 3. SysML does not provide any standard means to represent measurement uncertainty and it has to be incorporated in an ad hoc manner by the modeler in terms of either additional attributes of the value types (as in option 1), or by making use of DistributedProperty stereotypes that specify that these values are distributed following a given probability distribution (e.g. «normal» in option 2). The parameters of such distributions (i.e., the mean and standard deviation in case of normal distributions, or min and max values in case of intervals) become tag values of the stereotypes.

Although expressive enough, these solutions are unsatisfactory, due to several reasons. In the MARTE and first SysML solutions, users are expected to define their own quantities. This is fine in case of the model is developed by one person, or within one single company in an isolated modeling environment, because these quantities (and their associated dimensions) need to be known and reused across all models. However, these two approaches hamper integration and interoperability between models independently developed by different companies or parties, each one defining its own quantities and dimensions, and without previous consensus among them.

The second SysML approach solves this problem because it uses the SysML standard library of units. This permits developing more compact, reusable and interoperable models. However, it introduces two further problems. First, units are used as types, when they shouldn’t. Dimensions should provide the types of the variables, and not their units. For example, two variables that represent a Length should be compatible, despite in which unit their values are expressed. As an analogy, suppose two Integer values: they should be compatible no matter if they are expressed in Hexadecimal or in Octal bases. The type of these two variables should be Integer, and not Hexa or Octal. Furthermore, using units as types would force users to explicitly deal with unit conversions, which could be another potential source of errors. Unit conversion between compatible units (i.e., those defined for the same Dimension) should be implicitly accomplished by the type system. The second problem of this SysML representation is that modeling uncertainty measurement is not possible in an standard manner, hence losing compactness and reusability.

Finally, both MARTE and SysML solutions have one additional problem: none of them provide language mechanisms for manipulating expressions neither with units nor with measurement uncertainty; in particular, they do not permit calculating the resulting unit or the propagation of uncertainty when computing the values of derived attributes.

Figure 4 shows an example of how we would like to model the Moving Particle system, using a reusable library of Quantities. We can see how every attribute is typed with the dimension of the measure property it represents. These are types in the sense that they define a set of values and a set of valid operations on them. Values are given by a Real number representing the measurement result; the unit in which it is expressed; and the standard uncertainty associated to the measurement (i.e., the precision). For example, (1000.0, 0.0001, m) or (352.44, 0.0, ft) are valid values of type Length. Each type has a set of associated operations, which define the valid operations on its values. These operations permit implementing static type checking mechanisms when assigning values to variables, or when defining...
expressions that compute the value of derived attributes. Figure 4 also shows the specification of such derivation expressions on the previous model, using standard OCL expressions. One important feature of these operations is that they take into account the units in which the operands are expressed, and convert them accordingly in order to avoid any unit-mismatch error. That is, they do not only check that the resulting type of the operation coincides with the type of the attribute, but also they make sure all operations are performed using the same (and correct) unit—no matter the units in which the operands are expressed (e.g., meters, yards, inches, etc.). Moreover, these operations on the Quantity types take into account the propagation of measurement uncertainty when computing the derived values.

The rest of the paper is devoted to describe in detail how these quantity types are defined and modeled, the algebra of operations defined for them, and how to use this reusable model library with models that require dealing with physical quantities.

4. Modeling Quantities

In this section, we present a domain model for representing quantities. The purpose of this domain model is to formalize the different constituents of quantities as well as their relationships with each other. Furthermore, this domain model is intended to serve as starting point for integrating quantities into software modeling languages. In fact, our UML extension with quantities presented in Section 6 is obtained through a model transformation that transforms the domain model into an UML model. The domain model is compatible with the International Vocabulary of Basic and General Terms in Metrology (VIM) [16, 17] as well as the Ontology of Units of Measure (OM) [18].

Figure 5 shows the main constituents of any quantity: A Quantity comprises a unit and a value, where for the value its MeasurementUncertainty is given. In the following, we describe these key concepts in more detail, starting with quantity kinds and then continuing with units, and measurement uncertainty. Finally, we show how the running example introduced in Section 3 can be represented by means of the presented domain model.

Figure 5: Domain model of quantities.
4.1. Quantity Kinds

As defined in Section 2.1, a quantity is an observable property of an object, event or system that can be measured and quantified numerically [8]. Quantities that can be placed in order of magnitude relative to one another are of the same quantity kind [16]. For example, all lengths, such as diameters and wavelengths, are quantities of the same kind.

As defined in the VIM [16], and depicted in Figure 5, we need to distinguish between Base Quantities and Derived Quantities. Derived quantities are quantities that are derived from base quantities by means of a function. Base quantities cannot be derived from other quantities.

In our domain model, we defined the base and derived quantities of the International System of Units (SI) [14] and completed them with the ISO/IEC 80000:2009 standard [13], which are Length, Mass, Time, Electric Current, Thermodynamic Temperature, Amount of Substance, Luminous Intensity, Angle, Data Storage Capacity, Entropy, Traffic Intensity, and Level. Furthermore, we have defined 93 derived quantities as defined in ISO/IEC 80000:2009 [13], including the ones shown in Figure 5: Linear Velocity, Linear Acceleration, Force, and Power.

To represent dimensionless quantities, such as friction factors and mass fractions, our domain model provides the concept Dimensionless Quantity. They are quantities for which all the exponents of the factors corresponding to the base quantities in the representation of its dimension are zero. The values of dimensionless quantities are numbers (scalars).

Note that our domain model is extensible w.r.t. quantity kinds, i.e., additional base quantities and derived quantities can be added by introducing new subclasses of metaclasses BaseQuantity and DerivedQuantity, respectively.

4.2. Units

The value of a quantity is a product of a number and a unit. As shown in Figure 6, a Unit is identified by a name and a symbol. Units are always used to be able to compare quantities of the same kind. Thus, units are always related to a certain quantity kind, either to a base quantity, a derived quantity, or a dimensionless quantity. To represent this, our domain model defines the abstract metaclasses BaseQuantityUnit and DerivedQuantityUnit as subclasses of the metaclass Unit, as well as non-abstract subclasses for representing units associated with the different kinds of quantities. In particular, one concrete subclass of BaseQuantityUnit is defined for each of the twelve base dimensions (Length, Mass, etc.), and one concrete subclass of DerivedQuantityUnit is defined for each derived dimension (Linear Velocity, Linear Acceleration, etc.).

Furthermore, the domain model explicates the base quantity kinds (or dimensions) involved in the definition of a unit. As explained in Section 2.2, the dimensions of all units can be derived from the base dimensions as a product of powers of these base dimensions. Thus, a unit might define up to twelve exponents, i.e., at most one for each base dimension (see reference dimension and the subclasses of Dimension). For instance, the dimension Linear Velocity is a product of the base dimension Length to the power of 1.0 and the base dimension Time to the power of −1.0. Furthermore, the domain model explicates the function with which a unit can be derived from the base units. As explained in Section 2.2, each unit can be derived from the base units as a product of powers of these base units that are defined for the base quantity kinds or base dimensions. The following OCL constraint ensures the correct definition of the dimensions of units:

Note that our domain model is extensible w.r.t. quantity kinds, i.e., additional base quantities and derived quantities can be added by introducing new subclasses of metaclasses BaseQuantity and DerivedQuantity, respectively.
the product of powers of base units with the proportionality factor one. And finally, we also represent meter per second (m/s) as a derived unit, which are units formed from other units by multiplying (resp. dividing) by an integer greater than one. Derived units are units for derived quantities [16]. For example, the following OCL constraints prescribe that a Length Unit has to define exactly one Length Dimension with the exponent 1.0 and must not define any other dimension, and that a Linear Velocity Unit has to define exactly one Length Dimension with the exponent 1.0, one Time Dimension with the exponent -1.0, and must not define any other dimensions:

Since dedicated classes exist for all base quantity units and derived quantity units, and their dimensions are known, the dimensions can be further restricted. As an example, the following OCL constraints prescribe that a Length Unit has to define exactly one Length Dimension with the exponent 1.0 and must not define any other dimension, and that a Linear Velocity Unit has to define exactly one Length Dimension with the exponent 1.0, one Time Dimension with the exponent -1.0, and must not define any other dimensions:

To statically ensure that a quantity of a specific kind only specifies a unit related to this quantity kind, we have defined additional OCL constraints for all quantity kinds. For example, the following OCL constraint statically ensures that a Length quantity can only define a LengthUnit. For all other quantity kinds, similar OCL constraints are defined.

Units are defined within systems of units represented by the concept SystemOfUnits. A system of units is a conventionally selected set of base units and derived units, and also their multiples and submultiples, together with a set of rules for their use [16]. Thus, we distinguish between base units (reference baseUnits) and derived units (reference derivedUnits). Base units are units uniquely adopted for base quantities in a given system of quantities. For instance, the SI system of units defines the base units metre (m), kilogram (kg), second (s), ampere (A), kelvin (K), mole (mol), and candela (c) as its base units for the base quantities Length, Mass, Time, Electric Current, Thermodynamic Temperature, Amount of Substance, and Luminous Intensity. Derived units are units for derived quantities [16]. For instance, meter per second (m/s) is a derived unit for the derived quantity Linear Velocity. The VIM also defines coherent derived units, which are derived units that, for a given system of quantities and for a chosen set of base units, are the product of powers of base units with the proportionality factor one. And finally, we also represent multiples and submultiples of units, which are units formed from other units by multiplying (resp. dividing) by an integer greater than one.

Note as well that the same unit can be part of several systems of units. For instance, the unit second (s) is the base unit for the quantity kind Time in the SI system of units as well as in the Centimetre-Gram-Second (CGS) system of units. However, in any system of units there may be at most one base unit for each base dimension. This is validated by the following OCL constraint.
Furthermore, one particular unit of a system of unit, has to be either a base unit or a derived unit, but it cannot be a base unit and a derived unit at the same time. This is ensured by the following OCL constraint.

```
class SystemOfUnits inv EachUnitIsEitherBaseUnitOrDerivedUnit:
  self.derivedUnits -> intersection(self.baseUnits) -> isEmpty()
```

To convert values of quantities of the same kind between different units, Conversion Factors are needed. They comprise a multiplicator and optionally an offset. Thereby, conversion factors are always given relative to a base unit (reference baseUnit). To convert a quantity value to a base unit, it is multiplied by the specified multiplicator and then the given offset is added. For example, the unit kilometer would specify the multiplicator 1,000 and the offset 0 to convert values to the base unit meter. Since derived units might be derived from multiple base units, multiple conversion factors have to be specified, in particular, one relative to each involved base unit of the base dimensions, which are represented by the non abstract subclasses of the class ConversionFactor. Thus, a non base unit might comprise up to twelve conversion factors (reference conversionFactors), one for each of the supported base units and base quantity kinds (see subclasses of ConversionFactor). Taking again the example of the unit kilometer, it would specify one Length ConversionFactor with the multiplicator 1,000 and the offset 0 for the base unit meter. No further conversion factors are required for this unit. As an other example, the unit kilometer per hour would specify in addition to the Length ConversionFactor, a TimeConversionFactor with multiplicator 3,600 and offset 0 for the base unit second. Note that for enabling the conversion of units between different systems of units using different base units, conversion factors for all base units are required. For instance, to convert kilometer to foot, which is the base unit used in the United States Customary System (USCS) for Length, this unit has to define another Length-ConversionFactor with the multiplicator 3,280.80 and the offset 0. The following OCL constraints ensure that (1) for each dimension involved in the definition of a unit, a corresponding conversion factor is defined, and (2) conversion factors are defined for all relevant base units of the systems of units in which the unit is used (only excerpts of these constraints are shown).

```
class Unit inv ConversionFactorsCorrespondToDimensions:
  (self.dimensions -> select(d : Dimension | d.isKindOf(LengthDimension)) -> notEmpty() implies self.conversionFactors -> select(f : ConversionFactor | f.isKindOf(LengthConversionFactor))) -> notEmpty() and
  (self.dimensions -> select(d : Dimension | d.isKindOf(TimeDimension)) -> notEmpty() implies self.conversionFactors -> select(f : ConversionFactor | f.isKindOf(TimeConversionFactor))) -> notEmpty() and

context Unit inv UnitDefinesConversionFactorsForAllSystemsOfUnits:
  self.systemOfUnitsBase -> union(self.systemOfUnitsDerived) -> baseUnits -> select(s : Unit | s.isKindOf(LengthUnit) and s <> self) -> forAll(lu : LengthUnit | self.conversionFactors -> exists(f : ConversionFactor | f.isKindOf(LengthConversionFactor) and f.baseUnit = lu)) and
  self.dimensions -> exists(d : Dimension | d.isKindOf(LengthDimension)) implies self.systemOfUnitsBase -> union(self.systemOfUnitsDerived) -> baseUnits -> select(s : Unit | s.isKindOf(TimeUnit) and s <> self) -> forAll(du : TimeUnit | self.conversionFactors -> exists(f : ConversionFactor | f.isKindOf(TimeConversionFactor) and f.baseUnit = du)) and
```

Another characteristic of a unit is its scale, i.e. its relationship to the corresponding base unit. We distinguish between ratio-scaled units and interval-scaled units (attribute scale of class Unit). Ratio scaled units have an absolute zero point, which is not the case for interval scaled units. Interval scaled units define equally sized intervals that have limiting interval points. Thereby, the difference between two interval scaled values is measureable and exactly equal. Most of the units are ratio scaled, however, there are a few interval scaled units to be considered. The most prominent examples of interval scaled units are units for thermodynamic temperature, namely fahrenheit (F) and celcius (C). The value of these attributes can be derived based on the conversion factors of a unit, since interval scaled units require offsets for their conversion. The derivation rule is given in the following.

```
class Unit::scale : Scale
  derive: if self.conversionFactors -> forAll(f : ConversionFactor | f.offset = 0.0) then ratio else interval
```

Units are intended to be pre-specified and collected in libraries of systems of units. We have already built up a library comprising 307 units as specified in ISO/IEC 80000:2009 [13]. This library can be extended with new units and additional unit libraries can be defined.
4.3. Measurement Uncertainty

As discussed in Section 2.4, the magnitude of a measured quantity can usually not be completely accurately determined. Instead, it is subject to uncertainty, e.g., due to the precision of measurement instruments, stochastic environments, or the need for relying on estimations. In such cases, the magnitude of a measured quantity is expressed by a pair of two values (see Figure 7): a single quantity value (attribute Quantity.value) and a measurement uncertainty (attribute MeasurementUncertainty.standardUncertainty). Thereby, we adopt the concept standard uncertainty from the Guide to the Expression of Uncertainty in Measurement (GUM) [9]. Note that a quantity value that is known exactly defines a standard uncertainty of 0.

To provide additional information about the measurement uncertainty, we have introduced abstract class MeasurementUncertaintyInformation and concrete subclasses for normal distributed quantities (NormalDistribution), quantities for that only an interval of quantity values can be determined (Interval), and quantity values retrieved through sampling (Sampling, Sample). Note that for normal distributed quantities, the mean value of the normal distribution (attribute NormalDistribution.meanValue) corresponds to the quantity value (attribute Quantity.value.value) and the standard deviation (attribute NormalDistribution.standardDeviation) corresponds to the standard uncertainty of the quantity value (attribute MeasurementUncertainty.standardUncertainty). Additional subclasses of MeasurementUncertaintyInformation could be introduced for providing additional or supplementary data about the measurement uncertainty of a quantity value.

4.4. Modeling the Motivating Example

Let us have a look into how the domain model can be used to model quantities. For this, we will revisit the motivating example introduced in Section 3.

The left-hand side of Figure 8 shows how the two classes Segment and Observation, and their attributes can be modeled with our domain model. Note that for the types of the attributes, the appropriate Quantity classes of our domain model are used. This specification is equivalent to the desired modeling solution presented in Section 3 (compare with Figure 4).

The right-hand side of Figure 8 shows how quantities are represented with our domain model. In particular, it shows how the position of an observation is specified: It defines a quantity value 0.0 with a measurement uncertainty of 0.001. As unit, the length unit meter specified in the International System of Units is used.

5. A Computational Kernel for Quantities

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.

Our representation makes use of three main classes: $U\text{Real}$ to represent values, $\text{Unit}$ to represent units, and $\text{Quantity}$ to represent quantities. They are depicted in Figure 9, and realize the entities of the conceptual domain model described in Section 4.

### 5.1. Values with uncertainty

In the first place, to represent values with measurement uncertainty, we make use of the type $U\text{Real}$ and the algebra of operations on the values of such a type, that we defined in our previous work [19]. Basically, the values of $U\text{Real}$ are pairs of $\text{Real}$ numbers $X = (x, u)$. They determine the expected value ($x$) and associated standard uncertainty ($u$) of a quantity $X$, as defined previously. Operations on these values are both specified in OCL and also implemented in Java to allow modelers to use them for defining derived attributes and for specifying operations and invariants in UML and OCL models.

The major advantage is that this approach provides a natural extension to the UML and OCL type $\text{Real}$. The conversion between the subtype and supertype is defined by identifying a real number $r$ with the $U\text{Real}$ value $(r, 0)$. Operations respect the subtyping relationship, i.e., they ensure safe-substitutability. In other words, $U\text{Real}$ operations work as before when fed with $\text{Real}$ values, and operations defined for $\text{Real}$ values do not need to be modified to work with $U\text{Real}$ values: their behavior is the same but now incorporating the treatment of uncertainty and its propagation through the different operators as specified in [9].

As an example, the following listing shows the specification of two of the $U\text{Real}$ type operations:

```java
context UReal::add(r : UReal) : UReal
post: result.x = self.x + r.x and
      result.u = (self.u*2 + r.u).sqrt()
context UReal::mul(r : UReal) : UReal
post: result.x = (self.x*r.x) and
      result.u = (r.u + self.u + self.u*r.x).sqrt()
```

In addition to the operations originally defined in [19], we have added several comparison operators for type $U\text{Real}$. Traditional comparison operations ($lt()$, $equals()$, $gt()$, etc.) return a Boolean value. This is fine with $\text{Real}$ numbers, and they can be naturally extended to deal with $U\text{Real}$ values too, as defined in [9]. In general there is no experiment that can performed to prove that two quantities are equal; the best that can be done is to measure them so precisely that a very close bound can be stated on their difference [20]. A generally accepted and more practical equality test between quantities with uncertainty is to check their intervals overlap, when expressed in the same units [21]. For example, let $a = 1.0 \pm 1.5$ and $b = 5.0 \pm 1.0$ be two $U\text{Real}$ numbers. To check whether $a < b$, it is enough to see if $1.0 + 1.5 < 5.0 - 1.0$. In turn, the fact that the intervals overlap means that we do not have enough precision to tell the two numbers apart, and therefore we can consider them equal. For example, suppose $c = 1.0 \pm 3.0$ and $d = 1.5 \pm 5.0$. 

---

Figure 8: The Moving Particle example modeled with the Quantities Domain Model.
With this, the traditional comparison operations between `UReal` numbers can be specified as follows.

```java
class UReal {
    x : double =
    u : double =
    +equals( r : UReal ) : boolean
    +round() : UReal
    +divideBy( r : UReal ) : UReal
    +ge( r : UReal ) : boolean
    +le( r : UReal ) : boolean
    +add( r : UReal ) : UReal
    +minus( r : UReal ) : UReal
    +sqrt() : UReal
    +abs() : UReal
    +distinct( r : UReal ) : boolean
    +uLe( r : UReal ) : double
    +uDistinct( r : UReal ) : double
    +lt( r : UReal ) : boolean
    +uGt( r : UReal ) : double
    +min( r : UReal ) : UReal
    +compareTo( other : UReal ) : int
    +value +unit
    ...}
```

Note that in these formulae we are using a 68\% confidence level. It would be possible to relax the equality test by increasing the interval to 2 or 3 times the standard deviation, using the Extended Uncertainty with a coverage factor of \( k = 2 \) or \( k = 3 \) that would cover, respectively, 95.4\% or 99.7\% of the possible values of these quantities. This would give more confidence on the comparison, but with an associated cost: we may be increasing the looseness. For instance, when trying to decide whether one mechanical piece fits into another, increasing the confidence level may result in increasing the slack between the two.

In addition to these operations, a recurrent requirement that we obtained from modelers when using our type library with Quantities was the need to count of comparison operations on `UReal` values that returned a probability, and not a `Boolean`. To illustrate this need, consider the graphical representation of the two pairs of numbers described above, which is shown in Figure 9. We can see that in both cases there is indeed an overlap (represented by the gray area): it constitutes the probability that the two values are equal.

Then, given two `UReal` values \( x \) and \( y \) we define three real numbers \((l, e, g)\) that represent, respectively, the probability of \( x \) being less, equal or greater than \( y \). Of course, it is always the case that \( l + e + g = 1 \). For example, the triplet that we obtain for values \( a \) and \( b \) (Figure 10a) is the following: \((0.893, 0.106, 1.11 \times 10^{-16})\). This means that \( a < b \) with probability 0.893; \( a = b \) with probability 0.106, and \( a > b \) with a probability \( 1.11 \times 10^{-16} \). Similarly, the triplet for \( c \) and \( d \) (Figure 10b) is: \((0.152, 0.754, 0.094)\). Note that these 3 numbers correspond to the 3 areas in which the curve that represent the first of the values can be divided (this is clearer in Figure 10b).
Adding and subtracting quantities do not change their units. In OCL, these operations can be specified as follows:

\[
\begin{align*}
\text{Tuple}(l: \text{Real}, e: \text{Real}, g: \text{Real}) \rightarrow (l \times e + g)
\end{align*}
\]

Similarly, operation \(\text{multiplyUnits()}\) and \(\text{divideUnits()}\) have been implemented here as operations. These operations will be used when operating with vectors, and operations \(\text{addDimensions()}\) and \(\text{subtractDimensions()}\) add and subtract dimensions of the vector by the scalar \(a\). No other operations are required, since adding and subtracting quantities do not change their units. In OCL, these operations can be specified as follows:

```plaintext
--- Query operations for derived attributes of Unit
context Unit::isDimensionlessUnit() : Boolean = (self.dimensions->count(0, 0)=self.dimensions->size())
class Unit::isBaseUnit() : Boolean = (self.dimensions->count(1, 0)=1) and (not self.isDimensionlessUnit()) and (not self.isConversionFactors())
class Unit::isCoherentDerivedUnit() : Boolean = (not self.isDimensionlessUnit()) and (not self.isBaseUnit()) and (not self.isOffset()) and (not self.isConversionFactors())
class Unit::isDerivedUnit() : Boolean = (not self.isOffset()) or (not self.isConversionFactors())
class Unit::isRatioScaled() : Boolean = self.isOffset() and (not self.isConversionFactors())
```
In these specifications, we have made use of two auxiliary methods that check whether a unit has no offset or all its conversion factors are 0:

```
context Unit::noOffset() : Boolean
  post: result = (self.offset->count(0.0)=self.dimensions->size())
context Unit::noConversionFactors() : Boolean
  post: result = (self.conversionFactor->count(1.0)=self.dimensions->size())
```

For example, Linear Velocity is a derived dimension whose SI unit is m/s. Using the representation above, it can be expressed as \( \langle 1, 0, -1, 0, 0, 0, 0 \rangle \), with 1 in the Length dimension and -1 in the Time dimension; Acceleration, whose SI units are \( m/s^2 \), is then represented as \( \langle 1, 0, -2, 0, 0, 0, 0 \rangle \); and Force, expressed in Newtons (\( mKg/s^2 \)), is represented a \( \langle 1, 1, -2, 0, 0, 0, 0 \rangle \). Tuples for base units contain one 1 and the rest of the values are 0. Dimensionless units (e.g., counts, ratios or plane angles) are represented by a 9-tuple whose seven components are 0.

Focusing on the specification of SI units, we use the compact representation of a 12-tuple with the exponents of units (miles, feet, inches, kilometers, miles per hour, kilometers per hour, Celsius, Fahrenheit, angular degrees, days, hours, minutes, milliseconds, etc.).

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 \([24] [25]\). 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 (10°F + 5°F = 15°F), 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 \( \text{mul()} \), \( \text{divideBy()} \), \( \text{power()} \) and \( \text{sqrt()} \) operations.

Given that the SI specifies the names and symbols of the twelve base units and the SI derived units, two auxiliary methods \( \text{name()} \) and \( \text{symbol()} \) 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 \text{meter} by simply giving the String "m" or an instance of \text{Newton} by simply giving the String "N". Figure \[Fig.2\] shows some of the constructors defined for this type. In addition, a class \text{Units} (not shown here) provides a set of static methods 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.).

Finally, some invariants specify the integrity constraints of the type \text{Unit}:
5.3. Quantity types

Figure 9 also shows type Quantity and its operations. Elements of this class comprise a value (of type UReal) and a unit (of type Unit). Apart from the getter and setter methods, this type also 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. These operations are directly based on the corresponding operations of Unit, and their detailed specifications are available from [23]. Only two deserve particular explanation.

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:

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

```ocp
context Quantity::convertTo(u : Unit) : Quantity
  pre: self.compatibleUnits(u)
  post: result = self.value * (self.unit.factor() / u.factor()) + (self.unit.offset() - u.offset() - sum()) / u.factor() and
        result.unit = u
```

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

```ocp
context Unit::factor() : Real -- required for conversions
  post: result = Sequence({1, self.dims->size()}) - iterate(i : Integer; acc : Real = 1.0 | acc = (self.conversionFactor->at(i)) * power(self.dims->at(i))
```

Two other methods permit translating a Quantity to/from SI units:

```ocp
context Quantity::convertToSIUnits() : Quantity
  post: let auxUnit : Unit is
         auxUnit.dims = self.dims and
         auxUnit.factor = Sequence({1,1,1,1,1,1,1}) and
         auxUnit.offset = Sequence({0,0,0,0,0,0,0}) and
         auxUnit.name = auxUnit.defaultName() and
         auxUnit.symbol = auxUnit.defaultSymbol()

context Quantity::convertFromSIUnits(val : UReal) : Quantity
  post: let auxQ : Quantity is
         auxQ.dims = self.dims and
         auxQ.factor = Sequence({1,1,1,1,1,1,1}) and
         auxQ.offset = Sequence({0,0,0,0,0,0,0}) and
         auxQ.value = val and
         result = auxQ.convertTo(self.unit)
```

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 previously described. All operations make use of the corresponding operations of types UReal and Unit described above. They also consider offsets, in different ways.

```ocp
context Quantity::add(x : Quantity) : Quantity
  post: if self.unit.noOffset() then
         result.value = self.value.add(x.convertTo(self.unit).value) and result.unit = self.unit
      else
         result = self.convertToSIUnits(self.convertToSIUnits().value.add(x.convertToSIUnits()))
  endif
```

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

The rest of the operations can be specified straightforward:

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.

Note that in the `equals()` operation we make use of the fact that the internal representation of the quantity values uses the SI units, and therefore it is enough to check that the units are the same—i.e., without the need of any conversion (which should have happened in a separate stage).

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:

With the comparison operations, maximum and minimums are easy to define. The propagation of errors in case of unit mismatches is automatically performed by the OCL type system [26].
We should also consider further operations on Quantity values:

<table>
<thead>
<tr>
<th>Context</th>
<th>Quantity :: min(x : Quantity) : Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>self compatibleUnits(x.unit)</td>
</tr>
<tr>
<td>Post</td>
<td>result = if self.lessThan(x) then self else x endif</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Context</th>
<th>Quantity :: floor() : Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Post</td>
<td>result.value = self.value.floor() and</td>
</tr>
<tr>
<td></td>
<td>result.unit.equals(x.unit)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Context</th>
<th>Quantity :: round() : Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Post</td>
<td>result.value = self.value.round() and</td>
</tr>
<tr>
<td></td>
<td>result.unit.equals(x.unit)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Context</th>
<th>Quantity :: mult(x : Real) : Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>x &lt;&gt; 0.0</td>
</tr>
<tr>
<td>Post</td>
<td>result.value = self.value.mult(x) and</td>
</tr>
<tr>
<td></td>
<td>result.unit.equals(x.unit)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Context</th>
<th>Quantity :: divideBy(x : Real) : Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>x &lt;&gt; 0.0</td>
</tr>
<tr>
<td>Post</td>
<td>result.value = self.value.divideBy(x) and</td>
</tr>
<tr>
<td></td>
<td>result.unit.equals(x.unit)</td>
</tr>
</tbody>
</table>

5.4. Precision and rounding of computed values

As far as we know there is no way to control the precision and rounding in a platform-independent way. The easiest way to deal with the problem is the use of magnitudes to avoid data loss. For instance, let us assume that we want to represent 0.00000000039 meters and the machine/software in which it is represented only allows the representation of numbers with 10 digits. While this value in meters would have to be rounded to 0.000000001 meters, instead, it could be represented more precisely as 0.38 nanometers.

Another scenario where this problem arises is when making calculations with numbers with different magnitudes. For instance, when adding two values one of them represented in hectometers and the other in kilometers. Instead of converting all the values involved in the operation to the SI unit—in this case meters—they can be converted to one of the magnitudes of the values involved. Since converting both values to meters may cause data loss, converting either to hectometers or kilometers before computing the addition is a better option.

6. Integrating Quantities into Software Modeling Languages

To validate the feasibility of realizing the type system for quantities introduced in Section 4 as well as the algebra for operating with them introduced in Section 5, we have developed implementations of them for UML, Java, and OCL, and fUML. These implementations are discussed in the following and are openly available from [23].

6.1. UML

The computational kernel described in Section 5 provides the specification of the basic classes of our library: UReal. Unit and Quantity. However, users will rarely use the main class, Quantity, but the corresponding subclass that represent the dimension of the attribute being modeled—e.g., Length, Mass, LinearVelocity, etc.

We have developed the UML classes corresponding to the whole set of base and derived dimensions defined in the ISO 80000 standard [13]. They are all subclasses of class Quantity.

Static type checking of the correct usage of units in operations that involve quantities is achieved by subclassing. Thus, a set of subclasses of class Quantity (Length, Time, Force, etc.; see Figure 9) 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. Thus, only valid and type-safe operations are allowed on values of these classes, hence, providing the static type checks needed to ensure that units are properly combined.

For illustration purposes, Figure 11 shows classes Length, Time, LinearVelocity and Mass. We can see that, e.g., 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, attempting 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 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 [14, 13].

Note that we are focusing on the International System of Units (SI) for our internal representation. A separate class (Units) provides a complete set of units in other systems of units, as well as their scaled values (tera-, peta-, mega-, etc.). In this way we can separate the internal representation of the quantities (that uses this compact expression) from its presentational issues, using conversions to and from different units or systems of units.

6.2. Java Implementation

We have also developed a Java implementation that completely realizes the introduced type system and algebra of operations. 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:

```java
Length initialPos = new Length(0, 0, 0.001, Units.Meter);
Length finalPos = new Length(10, 0.001, Units.Meter);
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 [19, 9, 17]. If they do, analytic solutions exist and the implementation is straightforward [9]. If the values to aggregate follow different distributions or the variables are not independent, a Monte Carlo simulation method is required to implement the operations [17]. These two implementations for type UReal in Java are fully described in [19] and available from [23].
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. Furthermore, the UML and Java implementations are fully aligned and synchronized by model transformations: changes in one of them are automatically reflected in the other.

6.3. 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) [22], which is part of the USE/OCL specification environment [27]. 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 the 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. We have also been able to test and simulate all the OCL specifications of the operations defined for the UReal, Unit and Quantity classes.

As an example, the following listing shows a fragment of the SOIL commands used to simulate the Moving Particle system 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). Assignments and operations are executed with the ‘! ’ command, while values are displayed with the query (‘?’) command.

```plaintext
new UReal('ip')
  ip x := 0.0
  ip u := 0.01

new Quantity('initialPosition')
new Quantity('finalPosition')
new Quantity('distance')
new Quantity('avgVelocity')

... initialPosition.value := ip
initialPosition.unit := meter
finalPosition.value := fp
finalPosition.unit := meter

distance := finalPosition minus(initialPosition)

distance value.x
  -> 10.0 : Real

distance value.u
  -> 0.001 : Real

distance unit : symbol
  -> 'm' : String

distance unit : dimensions
  -> Sequence[1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0] : Sequence(Real)

avgVelocity := distance.divideBy(duration)

avgVelocity value.x
  -> 1.00000004 : Real

avgVelocity value.u
  -> 3.741657366739413E-4 : Real

avgVelocity unit : symbol
  -> 'm/s' : String

avgVelocity unit : dimensions
  -> Sequence[1.0,0.0,-1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0] : Sequence(Real)
```

6.4. fUML

Lastly, we have developed an implementation of the quantities domain model and computational kernel for Foundational UML (fUML) [10]. Foundational UML is an executable subset of UML, which is standardized by OMG. It comprises UML concepts for defining UML class diagrams allowing the modeling of system structures, and UML concepts for defining UML activity diagrams allowing the modeling of system behavior. The modeling concepts for defining UML activities comprise a subset of UML’s action language consisting of predefined actions for expressing object manipulations, computations with values, and communications between activities. Besides actions, fUML includes also control nodes for expressing control flows and object nodes for expressing data flows. The execution...
Figure 12: fUML data types for representing quantities.

Figure 13: fUML example using quantities.

The semantics of fUML is defined by the so-called fUML execution model that specifies a virtual machine for executing fUML-compliant UML models. Thanks to this virtual machine, it is possible to execute fUML activities and, hence, perform computations with values assigned to instances of classes defined in UML class diagrams.

Currently, the type system of fUML supports only the primitive data types Boolean, Integer, Real, String, and UnlimitedNatural. With this, however, it is currently not possible to represent the measurement uncertainty and units associated with quantities. To overcome this limitation, we extended fUML’s type system with the quantity type system specified in the domain model presented in Section 4. This comprises the introduction of new data types, which correspond to the classes of the quantity domain model, as well as the introduction of new methods operating on these data types, which correspond to the operations offered by the computational kernel presented in Section 5.

To achieve this extension of fUML’s type system, we used fUML’s built-in extension mechanism, which is intended exactly for this purpose.

Figure 12 shows the different quantity types (Length, Mass, etc.), unit types (LengthUnit, MassUnit, etc.), and the quantity value type and measurement uncertainty type are defined as new fUML data types. 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 allow computations on values of these new data types, so-called fUML function behaviors are defined corresponding to the operations of the presented computational kernel. These fUML function behaviors define the names and parameters of the operations. Their behavior has to be implemented according to an interface prescribed by the fUML virtual machine. These implementations will be invoked by the fUML virtual machine whenever the corresponding function behaviors have to be executed for performing computations on quantities. Our implementations of the quantity operations rely on the Java implementation of the proposed computational kernel for quantities presented in Section 6.2.

With our extensions, quantities can be used in fUML models as data types of class attributes and operations on such quantities can be used for defining and executing computations on quantities. As an example, Figure 13 shows the class diagram of our motivating example using the newly introduced quantities data types and an fUML activity computing the distance traveled by a particle within a segment.

Alternatively to the graphical representation of fUML activities in activity diagrams, the action language for fUML
(Alf) can also be used. Alf is a standardized textual representation of fUML activities. The Alf-based representation of the velocity calculation is shown below:

```java
package Particles {
    public class Observation {
        private position: Length [1..1];
        private time: Time [1..1];
        private velocity: LinearVelocity [1..1];
        getPosition(): Length {
            return self position;
        }
    }

    public class Segment {
        private avgVelocity: LinearVelocity [1..1];
        private avgAcceleration: LinearAcceleration [1..1];
        private distance: Length [1..1];
        private duration: Time [1..1];
        computeDistance() {
            this.distance = this.'end'.getPosition() - this.'start'.getPosition();
        }
    }

    public assoc SegmentStartObservation {
        public 'start': Observation[1..1];
        public 'startSegment': Segments[1..1];
    }

    public assoc SegmentEndObservation {
        public 'end': Observation[1..1];
        public 'endSegment': Segments[1..1];
    }
}
```

Our fUML implementation is integrated with the Eclipse Modeling Framework and the fUML reference implementation. The models presented in Figure 12 and Figure 13 are defined with the Eclipse Papyrus UML editor.

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 currently only implementations for a few operations on quantities.

7. Evaluation

In this section, we discuss several concerns related to the potential usage of our approach. In particular, our aim is to quantitatively assess the relevance and applicability of our approach for modeling languages concerned with cyber-physical systems.

7.1. Research Questions

By conducting this evaluation, we aim at answering the following research questions (RQs).

- **RQ#1: Relevance**—*How relevant is the presented approach for current software modeling languages?* In particular, we are interested in observing how many attributes actually refer to quantities in current modeling languages for cyber-physical systems.

- **RQ#2: Applicability**—*How much effort is needed to apply the presented domain model of quantities in already existing software models?* In particular, we are interested in observing if it is sufficient to re-type attributes, or more intrusive changes are needed for introducing quantity types to existing models.

https://eclipse.org/modeling/emf
https://github.com/ModelDriven/fUML-Reference-Implementation
https://eclipse.org/papyrus/
7.2. Evaluation Design

As an appropriate input to this study, we require a modeling language that targets a cyber-physical systems domain, since we are interested in Quantities. In addition to the definition of the modeling language, i.e., its metamodel, we also require documentation and examples of this language; otherwise we would not be able to reason precisely about the quantity types which are likely to be used. Finally, we also require to have access to the creators of the selected language to check if our introduced quantity types follow their intentions.

7.2.1. Setup

As language under study, we selected the family of domain-specific languages (DSLs) for mobile multi-robot systems proposed by Ciccozzi et al. [29]. This language family comprises five languages for modeling (i) robot missions, (ii) the contexts of these missions, (iii) robot behavior, (iv) robot structure and capabilities, and (v) specific language extensions for modeling particular robot types, such as drones. These DSLs are all implemented based on the Eclipse Modeling Framework (EMF) using Ecore as the metamodeling language.

Overall, the family of languages contains about 63 classes and 130 attributes. We obtained access to the language specification by the language engineers as well as by means of its documentation and example models.

To give the reader an impression of what type of language family we are studying, we introduce an excerpt of the robot DSL which is illustrated in Figure 14 (left side). The core of this DSL is the Robot concept which allows to model battery-operated, mobile robots by specifying their devices and movement capabilities. The attributes of the illustrated classes are typed based on the standard Ecore data types: Float, String, Integer and Boolean. In our study, we aim for finding potential attributes which are actually referring to quantities, and thus, should be more precisely typed to reflect this aspect. For this, we studied the metamodel definitions as well as the documentation for the attributes which are extracted from [29]. For instance, in this article, one finds descriptions such as for the radioFrequency attribute: “...used to indicate the radio frequency used by the robot to communicate with the control station, expressed in MHz.” The right side of Figure 14 shows the metamodel of the language after the attributes’ types representing quantities were replaced by their corresponding quantity types.

7.2.2. Measures

For measuring the relevance of our approach (RQ#1), we use the application rate (AR) metric. AR is defined as the ratio between attributes which refer to quantities and all attributes in the software model.

\[
AR = \frac{|\text{attributes} - > \text{select}(x|x\text{.typeoclIsKindOf(Quantity))}|}{|\text{attributes}|}
\]  

For measuring the applicability of our approach, we measure how many attributes may be simply re-typed by substituting the given standard data type by a quantity type. Furthermore, we also measure how many additional atomic changes are necessary to introduce the quantity types and report which type of coarse-grained changes are required for quantifying a model. For computing the amount of atomic changes, we build on a difference model computed for the initial model (iM) and the revised quantified model (qM). The difference model contains all changes.
which can be detected between two model versions which are essentially additions, deletions, and updates of elements. Based on the difference model, we compute the number of the differences reported and relate this number to the number of attributes to be re-typed. We use this ratio to indicate the application cost (AC) which is summarized in the following formula.

\[
AC = \frac{|\text{diff}(iM, qM).\text{elements}|}{|\text{attributes} \rightarrow \text{select}(\{x|x\text{.type}.\text{oclIsKindOf}(\text{Quantity})\})|}
\]  

(2)

7.3. Results

Starting by the application rate (AR), Table 1 summarizes the results of applying our approach to the selected family of languages. Columns 3 and 4 are of particular importance for answering RQ#1: the number of all attributes (#Atts), and the number of dimensionable attributes (#DimAtts).

Table 1: Results concerning RQ#1.

<table>
<thead>
<tr>
<th>Metamodel</th>
<th>#Class</th>
<th>#Atts</th>
<th>#DimAtts</th>
<th>#DiffDimUsed</th>
<th>Dimensions Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behaviour</td>
<td>24</td>
<td>22</td>
<td>10</td>
<td>3</td>
<td>Angle, Length, Time</td>
</tr>
<tr>
<td>Context</td>
<td>5</td>
<td>8</td>
<td>6</td>
<td>2</td>
<td>Angle, Length</td>
</tr>
<tr>
<td>Drone</td>
<td>8</td>
<td>48</td>
<td>31</td>
<td>14</td>
<td>Angle, Length, Time, Frequency, Storage, Mass, StoragePerTimeUnit, Power, ElectromagneticForce, ElectricCharge, LinearVelocity, LinearAcceleration, AngularVelocity, ThermodynamicTemperature</td>
</tr>
<tr>
<td>Mission</td>
<td>13</td>
<td>11</td>
<td>5</td>
<td>2</td>
<td>Angle, Length</td>
</tr>
<tr>
<td>Robot</td>
<td>8</td>
<td>34</td>
<td>21</td>
<td>12</td>
<td>Angle, Length, Time, Frequency, Mass, StoragePerTimeUnit, Pressure, ElectromagneticForce, ElectricCharge, LinearVelocity, AngularVelocity, ThermodynamicTemperature</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>63</strong></td>
<td><strong>130</strong></td>
<td><strong>76</strong></td>
<td><strong>15</strong></td>
<td></td>
</tr>
</tbody>
</table>

For the given case, we finally get an application rate of 58%, which means that more than half of the language attributes actually represent quantities. Most of them are base quantities, but there are also derived quantities such as `StoragePerTimeUnit`. Attributes not referring to quantities are mainly used for introducing identifiers, names and configuration information such as the availability of measurement devices.

**Answering RQ#1.** As we observed an application rate greater than 50% for the given case, we conclude that there is a high relevance of having a dedicated modeling approach for quantities and their usage for software modeling languages in the cyber-physical systems domain.

Concerning the application cost (AC) of our approach, we observed that for some attributes the introduction of quantity types did not correspond to a single atomic change which is needed to exchange the type definition of the attributes. For some specific cases, additional changes have been necessary. Overall, we had to perform 85 atomic changes for 76 dimensionable attributes. This gives us an application cost of 1.09 changes per dimensionable attribute. Table 2 summarizes the results of applying our approach concerning its cost.

Table 2: Results concerning RQ#2.

<table>
<thead>
<tr>
<th>Metamodel</th>
<th>#QuAtts</th>
<th>#Diffs</th>
<th>#AC</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behaviour</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>Only datatype substitution</td>
</tr>
<tr>
<td>Context</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>Only datatype substitution</td>
</tr>
<tr>
<td>Drone</td>
<td>31</td>
<td>31</td>
<td>1</td>
<td>Only datatype substitution</td>
</tr>
<tr>
<td>Mission</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>Only datatype substitution</td>
</tr>
<tr>
<td>Robot</td>
<td>21</td>
<td>30</td>
<td>1.43</td>
<td>Interval class specialized</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>76</strong></td>
<td><strong>85</strong></td>
<td><strong>(avg) 1.09</strong></td>
<td></td>
</tr>
</tbody>
</table>

Taking a closer look at the attributes which require more changes to be typed by quantity types, we observed a recurring pattern. If classes contain attributes which are referenced by different classes, the references may define a specific context which results in a specific quantity type. If just basic data types are used, e.g., `float` as in the initial version, this is not reflected in the model and the reuse can be simply done. However, if the quantity types are introduced, a separation of the referenced class may be needed for incompatible references with respect to the required
quantities. We have explored such a case for so-called *Interval* class which just defines an interval for Real values. Of course, such a class may be reused for angle intervals or for speed intervals just to mention a few possibilities. Figure 14 shows this concrete example before and after the application of quantity types. Please note the duplicated interval related classes on the right part of the figure which concretize the intervals for the specific quantity types.

We also have to note that this case comes up due to the lack of having interval types for metamodels defined in Ecore. If such an interval type would have been available, a more concise representation is possible by simply using attributes which also allows to easily exchange quantity types as well as does not require to model additional classes for intervals.

Answering RQ#2. As we observed an application cost greater than 1 for the given case, we conclude that there may be some more complicated applications than just retyping attributes. However, given that the application cost is still very low, we assume that our approach is efficient enough to retype existing models with quantity types.

As an additional benefit of our proposal, quantity types come equipped with information about the measurement uncertainty of the represented values. In general this has been perceived as valuable, because otherwise language engineers are forced either to ignore this information, or to add further attributes to the classes when indeed required.

7.4. Threats to Validity

In this subsection, we elaborate on several factors that may jeopardize the validity of our results and how we mitigated the risk.

Internal validity—are there factors which might affect the results in the context of our study? We may have wrongly classified the quantity types for the given attributes with respect to completeness and correctness. However, we mitigate this risk by having an additional review cycle with the language creators.

External validity—to what extent is it possible to generalize the findings for modeling languages in general? We have selected one family of languages for mobile robots. We may explore other languages in different domains which may require quantities such as industrial plants, networked systems, or e-commerce. However, we expect the results to be similar because these languages tend to rely on the basic types of the host language (in this case, Ecore) and hence they normally omit unit and uncertainty information. The integration of our domain model may be more difficult for other languages following a paradigm different to the object-oriented paradigm.

8. 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.

8.1. Modeling Physical Quantities

The need for physical quantities in software models has been discussed in several previous work, e.g., [6, 30, 26, 5]. The representation of such information resulted in several standardization efforts concerning the establishment of guidelines, vocabularies, and ontologies [9, 17, 19, 14, 8].

Concrete approaches for introducing physical quantities in the context of modeling languages are MARTE [6] and SysML [7]. 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 [8]. 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 [31]. 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 [32] provides SI unit support [4] as well as different reasoning techniques for the correct and user-friendly usage of units [33, 34, 35]. We have to notice that dedicated support for physical quantities is supported also by commercial products, such as

\[\text{see, e.g., } \url{https://build.openmodelica.org/Documentation/Modelica.SIunits.html} \]

25
Mathematica [36], which provides very enhanced support for units [1]. Finally, several systems modeling languages for specific domains, such as biology [37] and meteorology [38] provide support for units.

In the context of programming languages, dedicated support for physical quantities is available, or currently under development, for different languages, such as Java (e.g., the JSR 363: Units of Measurement API [39]), Python (e.g., see the packages Numericalunits, Pint, Unit, and Uncertainties) [24, 40], Ruby [41] and F# [42, 43]. Units were also implemented, although discontinued, for Eiffel [44]. In addition, language independent design patterns have been proposed to deal with different types of quantities, such as the Quantity Pattern [45] as well as idioms for nominally typed object-oriented programming languages [46]. We have also described here the Java implementation of our modeling proposal, not only for validation purposes but also for counting on an execution platform for it. Anyway, we see all these libraries at the programming level as complementary to ours, which sits at the modeling level. Our prime interest is to provide modelers with a mechanism that allows them to faithfully represent in their high-level models both the dimensions of their quantities and their associated precision, and to ensure static type- and unit-safe assignments and operations on their attributes—prior to any simulation or conversion to any of these programming language solutions. Being able to check that the high-level models are correct and free from unit-mismatch errors, independently from any simulation or implementation, represents a significant step ahead with regard to existing solutions.

8.2. 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 manipulate physical values in software models [5], in particular units or real-time properties. For example, some works on Business Process Models (e.g., [47]) 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 [9, 15]. Apart from being simpler and widely adopted by the rest of the engineering disciplines, it has the main benefit of permitting the operations on variables that do not follow any particular probabilistic distribution. This happens for instance when defining derived attributes obtained by the combination of several quantities that follow diverse, or even unknown, distributions.

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 [6], 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 [48]. 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.

9. Conclusion 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. Although some of the existing modeling languages, such as MARTE or SysML, already provide individual mechanisms for describing these properties, such mechanisms are not integrated into their type systems and therefore do not support operations for propagating uncertainty or for statically checking possible unit mismatches—which have already proved to be the cause of significant software failures. 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, together with a ready-to-use library of dimensions (Length, Mass, etc.) implemented in UML, Java and OCL, that can be added to any modeling project, and that permits modelers to safely represent and manipulate units and measurement uncertainties of physical systems in a natural manner.

This work opens several interesting lines of research that we would like to explore next. First, it would be interesting to provide mappings (automated bridges) to and from other modeling notations, such as MARTE or SysML. Our solution will provide the computational kernel they need, a type system for quantities, while existing models developed using these standard notations could still be used.

Similarly, the mapping of quantity-aware models to existing programming languages (such as Python [24, 40], Java [39], Ruby [41], F# [42, 43]) and analysis tools (Simulink, Matlab, Modelica [34, 33]) that provide support for units and tolerance, and can provide implementations of our models for specific platforms or applications. 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.

We also plan to propose extensions to existing model-based solutions for the specification and verification of critical systems (e.g., [49]) that do not take units or measurement uncertainty into account. Enriching these notations with our proposal can provide very interesting benefits—at a very low cost now that we have a ready-to-use library.

Finally, in addition to the computational capabilities of our proposal, we can also work on enhancing its presentation aspects, using more compact representations, in particular for issues such as precision, magnitudes, rounding, etc. For example, it is more natural to express an age attribute in years than in seconds, and without decimals.

Acknowledgments. This work is funded by the Spanish Research Project TIN2014-52034-R, the EU COST Action IC1404 (MPM4CPS), the National Foundation for Research, Technology and Development (CDG), and the Austrian Federal Ministry of Science, Research, and Economy (BMWFW Austria).

References

URL http://nssdc.gsfc.nasa.gov/planetary/text/mcs_pr_19991110.txt


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

Appendix A. The Robot Family of Languages, with Quantities

The following diagrams show the family of Robot languages defined in [29], specified using Quantities as types for the languages’ attributes.

Figure A.15: The Context Language, specified with Quantities.

Figure A.16: The Mission Language, specified with Quantities.
Figure A.17: The Behaviour Language, specified with Quantities.

Figure A.18: The Robot Language, specified with Quantities.
Figure A.19: The Drone Language, specified with Quantities.