Specifying Quantities in Software Models

Loli Burgueño[a,b], Tanja Mayerhofer[c], Manuel Wimmer[d], Antonio Vallecillo[e]

[a] *Open University of Catalonia, IN3, Av. Tibidabo, 39-43. (08035) Barcelona, Spain*
[b] *Institut LIST, CEA, Université Paris-Saclay, Avenue de la Vauve. (91120) Palaiseau, France*
[c] *TU Wien, Institute of Information Systems Engineering, Business Informatics, Favoritenstraße, 9-11. (1040) Vienna, Austria*
[d] *Johannes Kepler University, CDL-MINT, Altenbergerstraße 69 (4040) Linz, Austria*
[e] *Universidad de Málaga, Atenea Research Group, Bulevar Louis Pasteur, 35. (29071) Málaga, Spain*

Abstract

Context: An essential requirement for the design and development of any engineering application that deals with real-world physical systems is the formal representation and processing of physical quantities, comprising both measurement uncertainty and units. Although solutions exist for several programming languages and simulation frameworks, this problem has not yet been fully solved for 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 introduce the main concepts and mechanisms needed for representing and handling physical quantities in software models. More precisely, we describe an extension of basic type Real, called Quantity, and a set of operations defined for the values of that type, together with a ready-to-use library of dimensions and units, which can be added to any modeling project.
Results: We show how our approach permits modelers to safely represent and operate with physical quantities, statically ensuring type- and unit-safe assignments and operations, prior to any simulation of the system or implementation in any programming language.
Conclusion: Our approach improves the expressiveness and type-safety of software models with respect to measurement uncertainty and units of physical quantities, and its effective use in modeling projects of physical systems.

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

1. Introduction

The formal representation of measurement uncertainty and units is an essential requirement for the design and development of any engineering application that deals with physical entities, e.g. in the automotive and aerospace domains. The failure to do so has led to disasters such as the Mars Climate Orbiter [1] and the Gimli Glider Incident [2]. Moreover, the emergence of Industry 4.0 [3] and the proliferation of Cyber-Physical Systems (CPS) [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 the tolerance of mechanical tools and devices.

Although different solutions exist for representing and manipulating units and measurement uncertainty in programming languages, this problem has not yet been fully solved in the case of software models [5]. Modeling notations that permit dealing with aspects of physical systems, such as the UML Profile for MARTE [6] or SysML [7], already incorporate some elements for representing units and measurement uncertainty. However, they only offer representation mechanisms, no means to perform computations. To be able to carry out computations with units and measurement uncertainty at model level it is important to, for instance, calculate the values of derived attributes, evaluate expressions that represent model invariants and pre-/postconditions of operations, and compute the accumulated measurement uncertainty that is propagated when values are aggregated. Without computation facilities, elements annotated with units and measurement uncertainty become mere descriptive (decorative) elements. Furthermore, measurement uncertainty and units have to be incorporated into the models' type systems if we are to statically detect unit mismatches when trying to combine values of two physical quantities or to compute the values and units of

derived attributes—at model level, in other words, before any implementation is developed or a simulation is carried out, and ensuring that the high-level models are correct and free from any unit-mismatch errors.

To address this issue, in this paper we concern ourselves with the representation of *physical quantities*, which are observable properties of objects, events or systems that can be measured numerically [8]. Values of such physical quantities are expressed by a numerical value and a unit of measurement (e.g., 50.0 m/s or 1.3 N). The unit characterizes the sort of observable property being quantified, i.e., its quantity kind (length, force, time, mass, etc.), and it may indicate its order of magnitude compared to other quantities of the same kind. Furthermore, when dealing with objects of the physical world, not only do exact values need to be considered, but also some measurement uncertainty due to, for example, the lack of precision in the measuring tools (and so a value may become 1.3 ± 0.01 N). As stated in [9], "a measurement result can only be considered complete when it is accompanied by a statement of the associated uncertainty."

This paper shows how both measurement uncertainty and units can be effectively incorporated into software models, becoming part of their basic type systems. We abstract the manner in which quantities are internally represented, providing system modelers with a model library of quantity kinds and units, together with an associated algebra of operations that permit operating with uncertain values, checking unit mismatches, automatic conversion between units, and propagation of uncertainty. The work described here builds on our previous work [10], which presented an extension of the UML and OCL type Real for the representation of and computation with measurement uncertainty and units. This paper extends that work as follows. First, we provide a conceptual model that defines the main concepts needed for representing physical quantities independently from a concrete modeling language. The purpose of this conceptual model is to provide a basis for incorporating support for the representation of physical quantities into any software modeling language. Second, we extend our initial proposal with new dimensions and units defined in the ISO/IEC 80000 standard [11]. Third, we provide a novel extension to support comparison operations between uncertain values, which additionally returns probabilities and not just logical values when comparing measurement results affected by uncertainty.

Finally, we evaluate the relevance, applicability and effectiveness of our proposal with the aim of answering the following research questions. *Relevance*: How relevant is the present approach for current software modeling languages? How can existing software modeling languages benefit from our proposal? *Applicability*: How extensively do current software modeling languages designed for modeling cyber-physical systems use quantities? *Effectiveness*: Can our approach be applied to such software modeling languages to allow the precise modeling of quantities? *Efficiency*: How much effort is needed to apply our approach to already existing software models?

We provide answers to these research questions by analyzing existing modeling proposals for cyber-physical systems, and by applying our approach to two real-case studies where we have employed our model library of quantities and units, showcasing its possibilities and serving as a validation exercise for our work. More details about the research questions used to motivate and drive our work are given in Section 2.6.

The remainder of this paper is structured as follows. First, Section 2 presents a motivating example to show the current shortcomings of existing software modeling languages, and to illustrate our goal. It also presents the research questions that motivate and direct our work in further detail. Then, Section 3 introduces the concepts related to physical quantities. Section 4 describes a computational kernel for performing computations on physical quantities comprising an internal representation format as well as an algebra of operations. Section 5 discusses how units and quantities can be integrated into modeling languages with examples of UML [12], OCL [13], and fUML [14]. In Section 6, we evaluate our approach with regard to the effort needed to integrate units and quantities into existing modeling languages and the usefulness of the obtained unit and quantities support. Finally, Section 7 compares our work to similar proposals and we conclude in Section 8 with an outlook on future work.

2. Motivation

In order to motivate the need for our proposal, and also for illustration purposes, this section describes a simple example of a system that requires measurement uncertainty and units. We will model this example with two of the most widely-used modeling notations for modeling physical systems, MARTE [6] and SysML [7], and show the shortcomings of these notations with respect to the representation of units and measurement uncertainty.

2.1. *Example Description*

The example 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`). Times are expressed using the POSIX time convention, i.e., the number of seconds elapsed since January 1, 1970 [15]. We wish to analyze the particle's movements, for which we use segments, each one defined by a starting and an end point, where observations are made. For each 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 measurement uncertainties which should also be represented and taken into account in the models.

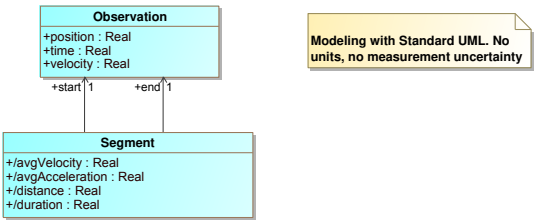

Figure 1: The moving particle example modeled using standard UML (model created with MagicDraw).

Figure 1 shows the representation of the system using standard UML in the UML modeling tool MagicDraw.[1] In standard UML there is neither support for modeling units nor the precision of the measurements. Note that this is the usual way in which these systems are modeled (see Section 7), namely using simple `Real` numbers and explaining, in the accompanying documentation, the unit in which each attribute should be expressed. Measurement uncertainty is normally ignored, or considered somewhere else in the models.

2.2. *Modeling Solution with MARTE*

Figure 2 shows the example modeled using the UML profile for MARTE [6], a notation with a priori support for units and measurement uncertainty. The exemplar model first needs to define the quantities to be used as extensions of the `NFP_Real` type, which is an «`NfpType`» 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 already defines a few quantity types, such as `NFP_Length`, `NFP_Duration` or `NFP_DataTxRate`, but the rest should be defined by the user (e.g. velocity or acceleration, as in this example). MARTE also provides the `precision` attribute in the `BasicNFP_Types` package, of type `Real`, to represent measurement uncertainty [6]—although it does not provide operations to perform computations with them.

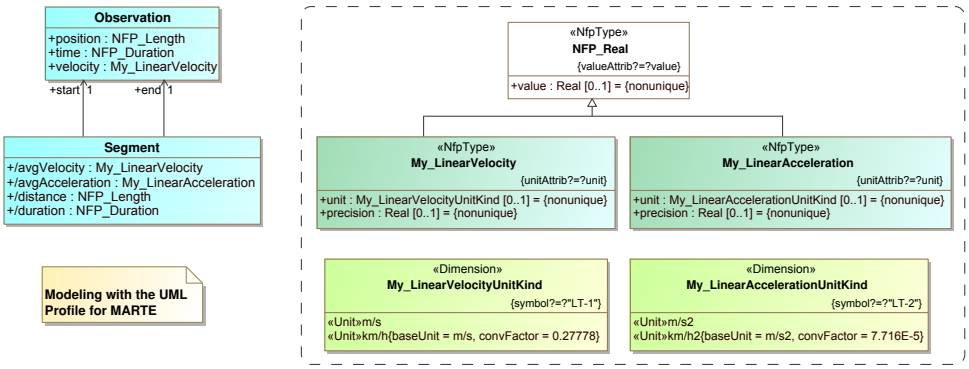

Figure 2: The moving particle example modeled using MARTE (model created with MagicDraw).

[1] `https://www.nomagic.com/products/magicdraw`

2.3. Modeling Solution with SysML

Figure 3 shows the example modeled using SysML [7]. 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 kinds 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.

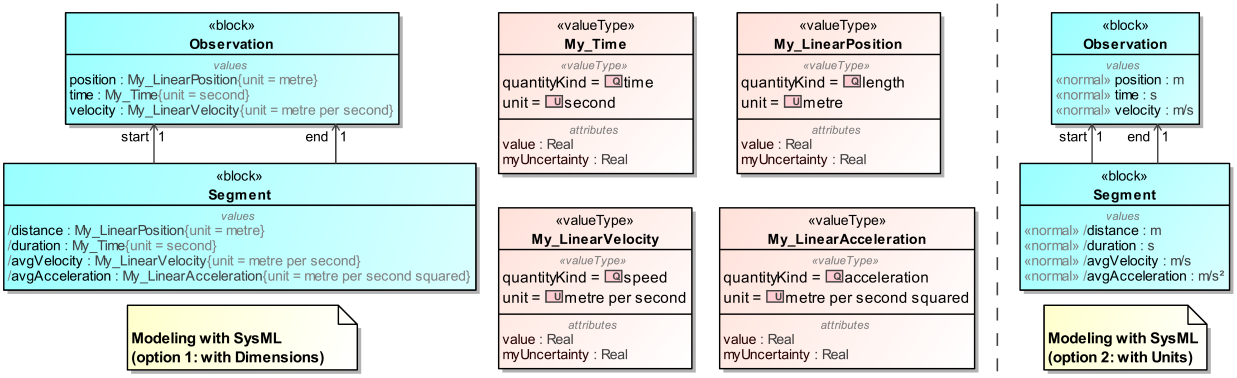

Figure 3: The moving particle example modeled in two options using SysML (model created with MagicDraw).

An alternative representation provided by SysML enables the units in which values are expressed to be specified as the types of the 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 thus has to be incorporated ad hoc, by the modeler in terms of either additional attributes of the value types (as in option 1), or by using `DistributedProperty` stereotypes, which specify that values of annotated attributes 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 the case of normal distributions, or min and max values in the case of intervals) become tag values of the stereotypes [7]. The problem is that, in many cases, including this example, these values are not constant but need to be derived from the values of other attributes of the model, and therefore they require the calculation of the propagation of uncertainty, a rather complex task that with this notation must be done by the software modeler.

2.4. Discussion

We argue that these solutions are not completely satisfactory, for several reasons. In MARTE and the first SysML solutions, users are expected to define their own quantity types (the MARTE library of pre-declared NFP types only contains a few, cf. [6, Annex D]). This is fine if the model has been 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, this hampers integration and interoperability between models developed independently 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 to ensure the compatibility of variables of the same quantity kind. For example, two variables that represent a length should be compatible, irrespective of the unit in which their values are expressed. As an analogy, think of two integer values: they should be compatible regardless of whether 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 is another potential source of errors. Unit conversion between compatible units (i.e., those defined for the same quantity kind) should be implicitly accomplished by the underlying type system. The second problem of this SysML representation is that modeling uncertainty measurement is not possible in a standard manner, hence losing compactness and reusability.

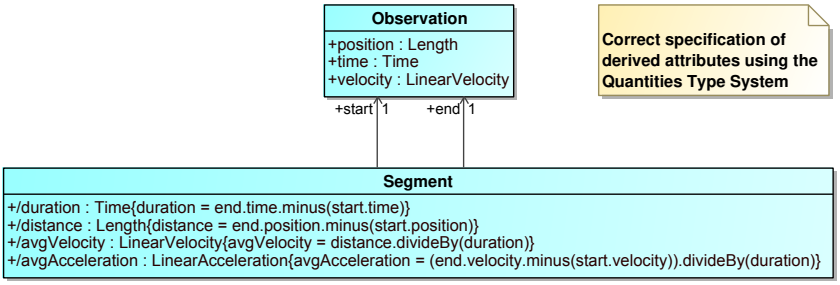

Figure 4: The moving particle example modeled using our approach (model created with MagicDraw).

Finally, both MARTE and SysML solutions have one additional problem: none of them provide language mechanisms for manipulating expressions with units or with measurement uncertainty. For instance, they do not permit calculating the resulting unit, or the propagation of uncertainty, when computing the values of derived attributes. Summarized, these solutions are not integrated with the basic type systems, and therefore the modeler cannot operate with the unit and uncertainty information.

Also the combination of SysML and MARTE is not free from problems. While this option is both possible and realistic, problems may occur when mixing concepts provided by SysML and MARTE for modeling physical quantities [16]. This discussion also shows that a general and neutral library for defining physical quantities (like the one presented in this paper) may represent a valid approach [17].

2.5. Targeted Modeling Solution

Figure 4 shows how we would like to model the moving particle system using a library of reusable quantity kinds (types) offering support for representing and computing with units and measurement uncertainty. We can see how each attribute is typed with the quantity kind of the 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 uncertainty associated with the measurement (i.e., the precision), and the unit in which the number is expressed. For example, $(1000.0, 0.0001, m)$ and $(352.44, 0.0, ft)$ are valid values of type `Length`.

Each type (`Length`, `Time`, etc.) has a set of associated methods, which define the valid operations on its values. They implement the static type checking mechanisms used 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 unit-mismatch errors. That is, they not only check that the resulting operation type complies with the type of the attribute, but also make sure that all the operations are carried out using the same (and correct) unit—irrespectively of the units in which the operands are expressed (e.g., meters, yards, inches, etc.). Moreover, these operations consider the propagation of measurement uncertainty when computing the derived values.

Our goal is to investigate how these quantity types can be defined and how they can be effectively incorporated into software models.

2.6. Research Questions

Now that the problem has been identified and the goal established, we discuss our research questions as introduced in Section 1 in detail in this section. These research questions served to drive our research and to determine the relevance, applicability, and effectiveness of our proposal for modeling languages concerned with cyber-physical systems, i.e., computer systems that incorporate physical processes or tasks.

2.6.1. RQ1: Relevance

The first research question, RQ1, is concerned with the relevance of our proposal and comprises the following two sub-questions:

- *RQ1a*: How relevant is the present approach for current software modeling languages?
- *RQ1b*: How can existing software modeling languages benefit from our proposal?

To see the use of units and measurement information in software models, we have conducted an analysis of literature reporting on the use of models and domain specific languages (DSL) to specify physical systems. We used the DBLP database[2] for an initial source of information, looking for papers that satisfied at least one of the following search strings: "UML robot", "UML physical", "UML IoT", "SysML robot", "SysML physical", "SysML IoT", "DSL robot", "DSL physical", "DSL IoT", "domain language robot", "domain language physical", and "domain language IoT". The search returned 84 papers, and then we subsequently classified them, trying to identify:

- The papers **related** to the target of our study, i.e., those that aimed to model physical quantities of systems. We discovered that many of the papers that we initially identified were not directly related to our study, because they did not focus on the models of the systems themselves, but rather on other aspects of their development, such as eliciting the system requirements, defining reference architectures, or generating code for different platforms.
- The papers that contained **examples** of one or more models with physical quantities of systems. Interestingly, many of the papers did not include models of a system, just partial examples showing some illustrative snapshots of unrelated systems, or no models at all.
- From the related papers, the ones that explicitly dealt with **units** or with **measurement uncertainty**.

While analyzing the 84 papers, we used backward snowballing [18] to identify further papers, by examining their references, as well as those of the newly found papers. This gave us a final total of 157 papers. Table 1 shows the main results of that literature review.

Table 1: Results of the literature review.

Category	Num. papers	Related	Example	Units	M.Uncertainty
Journals	17	5 (29%)	4 (80%)	0 (0%)	0 (0%)
Conferences	98	32 (33%)	25 (78%)	10 (31%)	4 (13%)
Workshops	42	12 (29%)	10 (83%)	4 (33%)	1 (8%)
Total	157	49 (31%)	39 (80%)	14 (29%)	5 (10%)

In the table, the 157 papers surveyed are categorized into journal, conference and workshop papers. From them, 49 papers (around one third of the total) deal with the representation of physical quantities in software models, and 39 of them (78% of the related papers) make use of at least one case study to illustrate or validate their proposal. In theory, they should use units and measurement uncertainty when expressing the values of the attributes of the models they specify. However, only 14 (less than 30% of the total of related papers, no matter whether they use one example or not) deal with units, and only 5 (10%) deal with information about measurement uncertainty.

The languages and notations used to represent the models in these papers include UML, OCL, SysML, some UML Profiles (such as MARTE, UML4IoT [19], or the UML Trajectory Profile [20]), and several domain specific languages.

Although in theory both MARTE and SysML support notations that permit modeling units, only one of the papers uses this feature of SysML—and it does so, to connect the models with Simulink in order to perform simulations. The rest of the models in the papers that incorporated units were written in ad-hoc DSLs. Two of these DSLs deserve particular attention: the Monticore language workbench[3], and mbeddr[4]. More than prototypical proposals that merely serve as proof-of-concept for some approaches, these are languages used in different industrial projects.

Very few modeling notations provide support for measurement uncertainty—this aspect seems to have received very little attention so far. According to our search, only five papers, from just two research groups, deal with the representation of measurement uncertainty in modeling languages (for instance, the UML profile for MARTE). Given

[2] `http://dblp.dagstuhl.de/`
[3] `http://www.monticore.de/`
[4] `http://mbeddr.com/`

the importance of representing units—in particular to avoid unit-mismatch errors—and the importance of representing the associated uncertainty to their values, we classify this as a surprising finding.

Far more important, from our point of view, is the fact that more than 70% of the proposals that represent physical quantities of systems, do not deal with units or with measurement information at all. The systems described in the papers surveyed are modeled with different kinds of notations and several domain specific languages—either developed ad-hoc, or with a more general purpose language, such as RobotML.[5] The problem is that these notations, which are those proposed in literature (and reported by the industry in these papers) for modeling physical quantities, seem to ignore units and uncertainty. Furthermore, 21 out of the 30 papers that did not take any of these aspects into account, were written in UML, UML profiles, or SysML. Conversely, 82% of the surveyed proposals for modeling physical systems that used UML or SysML did not deal with units or measurement uncertainty. Herein lies the relevance and need for our proposal.

2.6.2. RQ2-RQ4: Applicability, Effectiveness, and Efficiency

Another set of research questions aims to analyze the applicability, effectiveness, and efficiency of our proposal, and in particular, to answer the following questions:

- *RQ2 - Applicability:* How extensively do current software modeling languages designed for modeling cyber-physical systems use quantities?
- *RQ3 - Effectiveness:* Can our approach be applied to such software modeling languages in enabling the precise modeling of quantities?
- *RQ4 - Efficiency:* How much effort is needed to apply our approach to already existing software models?

To answer these questions, we have chosen two non-trivial case studies from the papers we surveyed. These models have sufficient detail to be modeled in full and do not deal with units or measurement uncertainty. A set of measures was used to answer questions RQ2-RQ4:

- *Application Rate (RQ2)*: Ratio between attributes representing quantities and all model attributes.
- *Coverage (RQ3)*: Ratio between the relevant quantities needed in the given languages that are successfully supported by our library and all quantities needed in the given language.
- *Application Cost (RQ4)*: Number of changes that are needed to introduce the quantity types into a unit-agnostic model.

The two case studies, as well as the results of the application of these metrics, are fully described in the evaluation section (Sect. 6).

The following sections describe our proposal in more detail, namely how quantity types can be defined and how they can be incorporated into software models.

3. Description of the Domain of Quantities

In this section, we describe the domain of physical quantities and introduce the different concepts that play a role in this domain. We have distilled these concepts from existing literature, such as in particular the ISO International Vocabulary of Basic Terms in Metrology (VIM) [21].

[5] `https://www.eclipse.org/papyrus/components/robotml/1.2.0/`

3.1. Quantities

A *physical quantity* (or simply a *quantity* from now on) is an observable property of an object, event or system that can be measured and quantified numerically [8], for example the object's position, size, speed or temperature. Quantities of the same kind are those that can be placed in order of magnitude relative to one another [21].

A *value of a quantity* (or *quantity value*) is composed by a *magnitude* (expressed as a *numerical value*) and a *unit*; e.g., 3.5 m/s. The unit represents the reference point in which the quantity value is described. The magnitude accompanying the unit is referred to as the *numerical value of the quantity* [21, 22], and should normally be accompanied by a statement of the associated uncertainty, e.g., 3.5 ± 0.001 m/s—particularly when it is taken from a measurement result that estimates the value of the quantity [9, 23].

Base and *derived* quantity kinds are distinguished in VIM. Base quantity kinds (e.g., length, mass) cannot be derived from other quantity kinds. Derived quantity kinds (e.g., force) are those that are derived from base quantity kinds by means of a function (e.g., $F = M * L/T^2$). We also need to be able to represent *dimensionless* quantity kinds, such as friction factors or mass fractions. Numerical values of dimensionless quantities are numbers (scalars), which refer to unit one.

3.2. Dimensions and Units

Quantities of the same kind within a given system of quantities have the same dimension, and can therefore be directly compared to each other (even if they are originally expressed in differing units). However, the relationship between quantity kind and dimension is not one-to-one. Different quantity kinds may share the same dimension. For example, moment of force and energy are quantities that, by convention, are not regarded as being of the same kind, although they have the same dimension. Similarly, heat has the energy dimension, diameter the length dimension, and duration the time dimension. A dimension is defined as "the dependence of a given quantity on the base quantities of a system of quantities, represented by the product of powers of factors corresponding to the base quantities" [21].

Units are defined by specific *systems of units*. 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 [21]. Each base and derived unit has an associated dimension, which is determined by a set of exponents of the base dimensions defined by the system of units, and the set of associated conversion factors between units. In addition to the multiplicative factor, as part of the conversion factors, we have included offsets in order to define affine conversion relationships between units that require both a conversion factor and an offset (e.g. between Celsius and Kelvin temperatures). Although not strictly considered in the ISO VIM, the use of offsets is the approach normally used in modeling notations such as MARTE [6], SysML [7] or the OMG Structured Metrics Metamodel [24] for dealing with these kinds of conversions.

Unit one (associated with scalar values, counts or ratios) can be considered as a special type of unit, which dimension is defined by having all the exponents of the base dimensions equal to 0. This unit is also referred to was dimensionless unit or unitless unit.

The most widely used system of units is the International System of Units (SI) [22]. It defines seven base quantities: length, mass, time, electric current, thermodynamic temperature, amount of substance, and luminous intensity. The SI also defines seven base units, one for each base quantity: metre (m), kilogram (kg), second (s), ampere (A), kelvin (K), mole (mol) and candela (cd). It also defines 90 derived quantities (area, volume, velocity, force, etc.) and their corresponding units (m^2, m^3, m/s, N, etc.).

Certain dimensionless quantities have units with special names and symbols, and they normally deserve special consideration. Plane angle and solid angle, for which the SI units are the radian (rad) and steradian (sr), respectively, are examples of such quantities.

Standard ISO/IEC 80000:2009 [11] extends the SI incorporating four new base quantities and their corresponding base units. The new quantity types 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 standardize 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, e.g., a frequency ten octaves above one hertz, i.e., 2^{10} Hz (1024 Hz), is one kibihertz (1 KiHz).

Apart from the SI, there are other systems of units. 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 the 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 volumes of fluids. Since these systems 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 conversions among them can be defined using multiplication factors and, in some cases, offsets. For example, to convert between miles and meters we 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 to convert absolute temperatures expressed by these units. 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 units are normally not made explicit in programming code.

3.3. Numerical Values and Measurement Uncertainty

When dealing with real-world entities, models need to take into account the inability to know, estimate or measure the value of any quantity with complete precision. 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. In 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, but 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].

Measurement uncertainty can be expressed in different ways [25], for example by means of a probability distribution associated with each uncertain variable, representing the distribution of the dispersion of its values. This is the approach used by, for instance, the UML Profile for MARTE [6]. However, this approach is somewhat limited when calculating the aggregated measurement uncertainty of the result of an operation that involves operands with different probability distributions. A more widely adopted approach among engineers of different disciplines, is defined by the The Guide to the Expression of Uncertainty in Measurement (GUM) [23], which associates a *standard uncertainty* with any uncertain value, defined by the standard deviation of the measurements for such a value. Therefore, a numerical value x of type Real becomes a pair (x,u), also noted $x \pm u$, which represents a random variable X whose average is x and its standard deviation is u. If X follows a normal distribution $N(x,u)$, we know that 68.3% of the values of X will be in the interval $[x-u, x+u]$.

The GUM framework also identifies two ways of evaluating the uncertainty of a measurement, depending on whether the knowledge about the quantity X is inferred from repeated measured values ("Type A evaluation of uncertainty"), or scientific judgment or other information concerning the possible values of the quantity ("Type B evaluation of uncertainty"). In the Type A evaluation of uncertainty, if $X = \{x_1, \ldots, x_n\}$ is the set of measured values, then the estimated value x is taken as the mean of these values, and the associated uncertainty u as their *experimental standard deviation*, i.e., $u^2 = \frac{1}{(n-1)} \sum_{i=1}^{n} (x_i - x)^2$ [9]. In the Type B evaluation, uncertainty can also be characterized by standard deviations, evaluated from assumed probability distributions based on experience or other information. For example, if we know or assume that the values of X follow a normal distribution, $N(x,\sigma)$, then we take $u = \sigma$. If we can only assume a uniform or rectangular distribution of the possible values of X, then x is taken as the midpoint of the interval, $x = (a+b)/2$, and its associated variance as $u^2 = (b-a)^2/12$, and hence $u = (b-a)/(2\sqrt{3})$ [9].

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. In these cases, one way to proceed is to follow the *law of propagation of uncertainty* proposed by the GUM, which computes this propagated uncertainty based on a first-order Taylor series approximation.

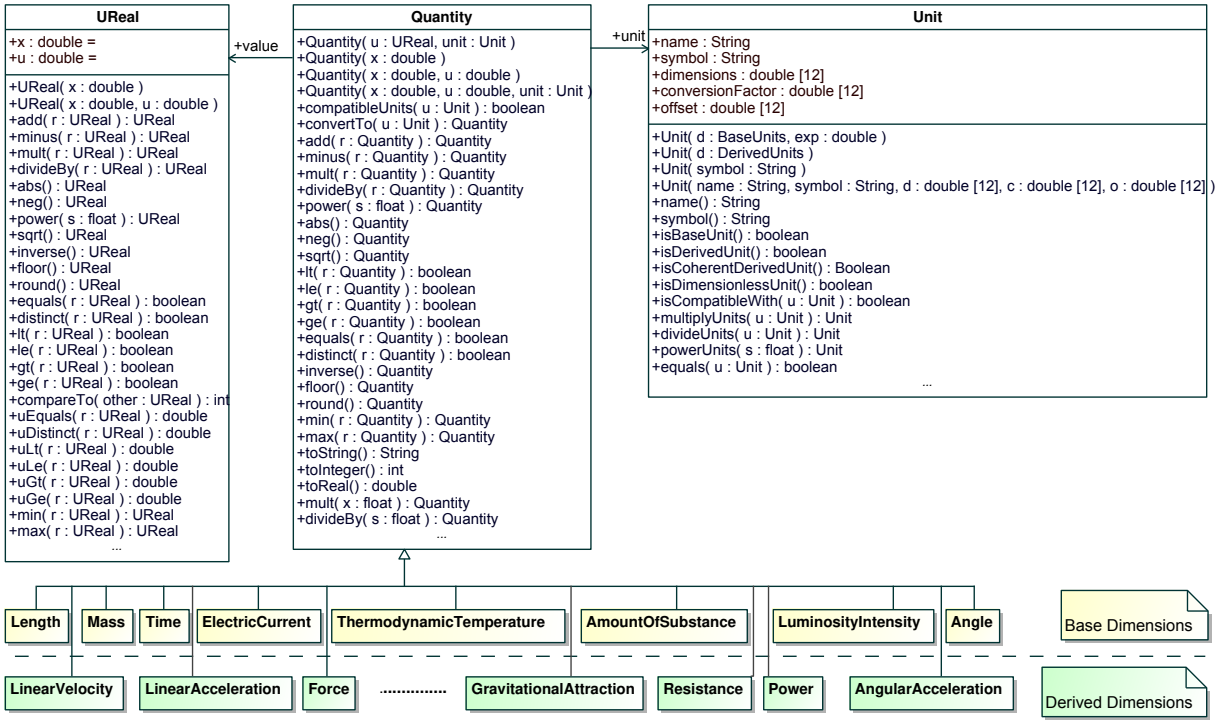

Figure 5: Kernel representation of quantities.

4. A Computational Kernel for Quantities

The representation of measurement uncertainty and units in models is important, but it is even more important to be able to manipulate and carry out computations with them at model level, ensuring type safety and uncertainty analysis. To enable this, we have defined a computational kernel for quantities, comprising a compact representation for quantities and an algebra of operations, operating on their values. Our goal is to extend the basic type `Real` present in most modeling languages, with a type, `Quantity`, which enables measurement uncertainty and unit information to be incorporated in the values of the basic type.

4.1. Kernel Representation

Our kernel uses three main classes: `UReal` ("uncertain real") to represent values of quantities including measurement uncertainty, `Unit` to represent units, and `Quantity` to represent quantities. In this section, we use and define these concepts in more detail, explain how we have adopted and integrated definition from the standards in our approach and clarify how and why we deviate from the standards in some parts. The kernel representation is depicted in Figure 5.

To extend the primitive types of the base modeling language (e.g., UML, OCL or fUML), we apply type embedding [26], which is one kind of *subtyping* [27]. We say that type A is a *subtype* of type B (noted $A <: B$), if all elements of A belong to B, and the behavior of operations of B, when applied to elements of A, is the same as those of A [28], i.e., they respect behavioral subtyping [27]. If $A <: B$, we say that B is a supertype of A. It is important to clarify that, in this context, subtyping is not to be confused with inheritance [29]. In broad terms, when we apply inheritance among classes, we say that objects of the subclass get the internal structure and code of the superclass and, on top of that, they can have new features (attributes, methods, relationships, etc.). In contrast, subtyping refers to that part of the objects' behavior that can be observed from outside by sending messages to them [30], i.e., the operations that can be applied to them. In algebraic terms, subtyping leads to a conceptual hierarchy that is based on behavioral specification [13, A.2.7].

Thus, type UReal can be considered as a supertype of basic type Real, which adds measurement uncertainty information to it. Similarly, type Quantity is a supertype of UReal (and hence of Real) that adds information about the units in which a UReal value is expressed. In short, Real <: UReal <: Quantity. As we shall see below, values of type UReal are pairs (x,u), where x is a Real value and u its associated uncertainty.

Although a quantity can be represented by multiple units (e.g. length could be expressed in metres, centimetres, inches, etc.) and hence by different values, in our kernel, a quantity has only one value and one unit. In this way, we follow the single source of information principle. If a quantity has to be expressed in other units, we provide mechanisms for unit conversion (see Section 4.4).

All this is described in the following subsections.

4.2. Operations on Values with Measurement Uncertainty

To represent values with measurement uncertainty, we extend basic type Real with the new type UReal and the algebra of operations on the values of such a type, as defined in previous work [31]. Basically, the values of type UReal are pairs of Real numbers $X = (x,u)$. They determine the expected value (x) and associated standard uncertainty (u) of a quantity X, as previously mentioned.

The principal advantage is that this approach defines a natural extension to the UML and OCL type Real, whereby type UReal becomes a supertype of Real (i.e., Real <: UReal). The conversion between the subtype and supertype is defined by identifying a real number r with the UReal value $(r,0)$. The operations respect the subtyping relationship, i.e., they ensure safe-substitutability. In other words, UReal operations work as before when fed with Real values, and operations defined for Real values are extended to work with UReal values, i.e., to incorporate 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 UReal type operations, add and mult:

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

Comparison operations between Real numbers $(=,<,\leq)$ have been extended to deal with UReal values too, as defined in [9]. These operations return Boolean values, indicating whether an uncertain real is equal, less, or equal or less than another. However, when dealing with uncertain values, these comparisons are not that precise. This is why we also defined a second set of comparison operations that return a UBoolean value. This type extends type Boolean by adding a probability that represents the confidence (i.e., the degree of belief) that we have on base Boolean value [31]. To calculate the results of these comparisons, we have redefined the comparison operations so that, given two UReal values x and y, the result is the probability of x being less, equal or greater than y. Of course, it is always the case that the sum of these three probabilities is 1. For example, for values $x = 1.0 \pm 0.15$ and $y = 1.1 \pm 0.2$ we obtain that the result of $x < y$ is now UBoolean(true,0.248), the result of $x > y$ is UBoolean(true,0.003) and the result of $x = y$ is UBoolean(true, 0.749).

The complete list of operations that apply to these datatypes as well as their specifications and implementations in OCL/UML and Java is available from [32].

4.3. Precision and Rounding of Computed Values

To the best of our knowledge, 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 a length of 0.00000000039 m, and the machine/software in which it is coded only allows the representation of numbers with 10 digits. While this value in meters would have to be rounded to 0.000000001 m, instead, it could be represented more precisely as 0.38 nm. This is why modelers are encouraged to express quantity values in the unit in which they can be expressed with more significant digits.

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 sum is a better option. This is precisely what we do in our proposal. Although this does not solve the rounding problem in general [33], at least it does not introduce further errors, e.g., those due to missing digits as explained above.

4.4. Representing Units

Units are modeled by class `Unit`, which is shown on the right-hand side of Fig. 5. It has several attributes to represent the properties of units.

A fundamental property of any system of units, which we will heavily exploit in this paper, is that any unit can be derived as a product of the powers of the base units $B_1 \dots B_n$: $B_1^{e_1} \cdot B_2^{e_2} \dots B_n^{e_n}$, where the exponents $e_1, \dots, e_n$ are rational numbers. Thus, in ISO 80000, the representation of any unit can be univocally determined by a 12-tuple $\langle e_1, \dots, e_{12} \rangle$, where e_i is the rational number that represents the exponent of the i-th base unit [8]: metre (m), kilogram (kg), second (s), ampere (A), kelvin (K), mole (mol), candela (cd), bit (b), shannon (Sh), erlang (E), decibel (dB) and radian (rad).

For example, linear velocity is a derived dimension which SI unit is m/s. Using the representation above, it can be expressed as $\langle 1, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$, with 1 in the length dimension and -1 in the time dimension; acceleration, which SI units are m/s^2, is represented as $\langle 1, 0, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$; and force, expressed in newtons (m kg/s^2), is represented as $\langle 1, 1, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$. Tuples for base units contain one value of 1 and the rest of the values are 0. Dimensionless quantities (e.g., scalars, counts, or ratios between quantities of the same kind) are represented by a 12-tuple whose 12 components are 0, and its (dimensionless) unit is usually referred to as unit one.

We use a compact representation with arrays for each of the 12 base dimensions. First, the `dimensions` array contains the 12-tuple with the exponents of the base dimensions. The attributes `conversionFactor` and `offset` represent the corresponding conversion factors and offsets, respectively, for the unit with respect to its SI base unit.

We provide a set of operations to obtain the nature and properties of units (`isBaseUnit()`, `isDimensionless-Unit()`, `equals()`) and to combine units (`multiplyUnits()`, `divideUnits()` and `powerUnits()`). For example, when two quantities are multiplied, their units should also be multiplied. This is carried out by operation `multiply-Units()`, which adds the two `dimensions` vectors (since their elements represent exponents). Equally, operation `divideUnits()` subtracts element by element the two `dimensions` vectors, and operation `powerUnits(s)` multiplies each element of the vector by scalar `s`. No other operations are required, since adding and subtracting quantities do not change their units. Operation `isCompatibleWith()` checks whether two units are compatible for being combined or compared (e.g., miles and centimetres, degree Fahrenheit and degree Celsius). In our proposal, this is accomplished by simply checking that their `dimensions` vectors are equal (irrespective of their conversion factors and offsets). For illustration purposes, the following listing shows the specification of some of these functions:

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

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

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

context Unit::isDimensionlessUnit() :Boolean = (self.dimensions->count(0.0)=self.dimensions->size())
```

The treatment of offsets as needed for converting between interval scaled units as well as between interval and ratio scaled units requires a separate discussion. One of the benefits of using SI units is that we can perform arithmetic operations on their values with no problem, since they all represent absolute values. However, interval scaled units, i.e., 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 them include both a scaling factor and an offset [34]. Thus, it does not make sense to add or multiply two Celsius values [34]. This is where *delta* units are often used as a specific solution to deal with this problem on the computation level [34, 35, 36]. They represent increments in affine values, and are obtained by simply considering the conversion factor of the unit and ignoring the offset. For example, `DeltaCelsius` (ΔC) is a unit derived from `Celsius`, obtained when two Celsius values are

subtracted. Similar for `DeltaFahrenheit` (ΔF). Deltas can be added to affine units ($10F + 5\Delta F = 15F$), and delta units can be multiplied and divided (since they represent absolute values). With all this in mind, 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 each unit; (b) we do not allow units with a non-null offset in the following two cases: as the argument in addition and subtraction operations; or as any of the operands in `mult()`, `divideBy()`, `power()` and `sqrt()` operations.

4.5. Operations on Quantities

We can see that a Quantity has two components, its value and its unit. These two attributes are of types `UReal` and `Unit` as described in Fig. 5. The type also includes operations to fetch the properties of its values, and to perform computations with them. These operations are directly based on the corresponding operations of `Unit` and `UReal`, and their detailed specifications are available from [32]. Only two of the operations need to be specifically explained.

The first operation, `compatibleUnits()`, permits deciding whether the dimensions of the units of two quantities are the same, to check their compatibility. The second operation `convertTo()` permits converting between the units of quantities. A precondition states that the two units must be compatible:

```
context Quantity::convertTo(u :Unit) :Quantity
  pre:  self.compatibleUnits(u)
  post: result.value = self.convertFromSIBaseUnits(self.convertToSIBaseUnits().value,u)
        and result.unit = u
```

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

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

4.6. Static type checking

In general, users will never use the main class, `Quantity`, for typing the attributes of the model that represent physical quantities, but rather the corresponding subclass that represents the quantity kind of the attribute being modeled (e.g., `Length`, `Mass`, `LinearVelocity`, etc.). As shown in Figure 5, we have developed the classes corresponding to the whole set of base and derived quantity kinds defined in the ISO 80000 standard [11]. They are all subclasses of the class `Quantity`, and they are provided so that modelers can simply reuse them in their software models.

Static type checking of the correct usage of units in operations that involve quantities is achieved by subclassing. Thus, predefined subclasses of class `Quantity` (e.g., `Length`, `Time`, or `Force`—see Figure 5) allow the possible values of the superclass to be constrained 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, thereby, providing the static type checks needed to ensure that units are properly combined. For example, class `LinearVelocity` provides operations that allow its instances to be multiplied by objects of classes `Time` or `Permeability`, returning objects of classes `Length` or `Resistance`, respectively; or to be divided by a `Time`, returning an object of class `LinearAcceleration`. However, the class `Storage` does not provide operations for computations with incompatible types such as multiplication by `MolarEnergy` or the addition or subtraction from objects whose type is different from `Storage`, which prohibits its application.

5. Integrating Quantities into Software Modeling Languages

To validate the feasibility of realizing the quantities introduced in Section 3, as well as the algebra for operating with them introduced in Section 4, we have developed implementations of them for UML, Java, OCL, and fUML. These implementations are discussed in the following paragraphs and are available from [32].

5.1. UML

We have developed the UML classes corresponding to the whole set of base and derived quantity kinds defined in the ISO 80000 standard [11]. They are all subclasses of the abstract class `Quantity`. Given that we focus on the International System of Units for our internal representation, a separate UML class (`Units`) provides a complete set of units in several systems of units, including the SI, as well as their scaled values (tera-, peta-, mega-, etc.).

5.2. Java

We have also developed a Java implementation that fully executes the types introduced, together with their operations. More specifically, it provides an API to conveniently create quantities with their units and measurement uncertainty, and to perform any of the operations defined for quantities.

Due to the definition of subclasses of the general class `Quantity` dedicated to representing specific quantity kinds (base or derived), 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, which internally creates the corresponding quantity values, and to perform operations on them:

```
Length initialPos = new Length(0,0.0.001,Units.Metre);
Length finalPos = new Length(30,0.003,Units.Foot);
Length distance = finalPos.minus(initialPos);
```

Two implementations have been developed for Java `UReal` type operations, depending on whether or not the distribution of the values with uncertainty follow a Gaussian distribution [9, 31, 37]. 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, the use of samples and a Monte Carlo simulation method is required to implement the operations [37]. These two implementations for type `UReal` in Java enables types A and B of measurement uncertainty to be supported (see Section 3.3), and are fully described in [31], and available from [32].

The motivation behind the development of the Java API is 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. In summary, our contribution is twofold: apart from the kernel design and its set of operations to work at a model level (which users can use to model their systems), a Java library with the implementation of these concepts and operations is provided so that users can execute/simulate their models. Note that our design is independent of the underlying implementation. Thus, users could use either our Java library or create their own ones in their language of choice (e.g., Java [38], Python [34, 39], or Ruby [40]).

5.3. OCL

OCL is a declarative, non-executable language principally devised to write integrity constraints on software models, and to specify the behavior of model operations in terms of pre- and post-conditions, independently of any implementation. However, there are some executable extensions of OCL that enable quickly prototyping the system specifications. One of them is SOIL (Simple OCL-like Imperative Language) [41], which is part of the USE/OCL specification environment [42]. 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 develop prototypical implementations of the UML/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 tested and simulated 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 introduced in Section 2 in USE. It shows how instances of quantities and quantity values are created, and calculations with them are performed (using the operations specified here expressed in SOIL).

```
!new UReal('ip')
!ip.x :=0.0
!ip.u :=0.001
...
!new Quantity('initialPosition')
!new Quantity('finalPosition')
!new Quantity('distance')
!new Quantity('avrgVelocity')
...
!initialPosition.value := ip
!initialPosition.unit := metre
!finalPosition.value := fp
!finalPosition.unit := metre
```

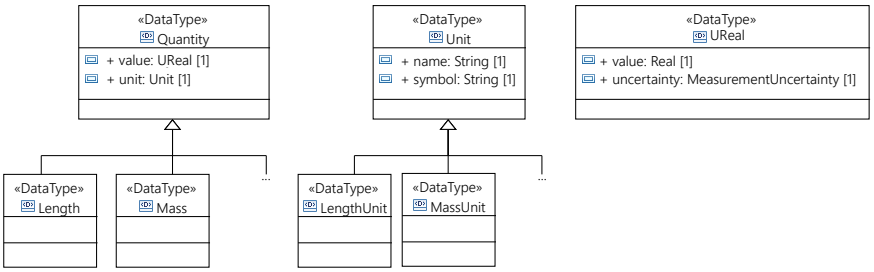

Figure 6: fUML datatypes for representing quantities (model created with Eclipse Papyrus).

```
...
!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} : Sequence(Real)
!avrgVelocity:=distance.divideBy(duration)
?avrgVelocity.value.x
-> 1.00000004 : Real
?avrgVelocity.value.u
-> 3.7416573867739413E-4 : Real
?avrgVelocity.unit.symbol
-> 'm/s' : String
?avrgVelocity1.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} : Sequence(Real)
```

5.4. fUML

Lastly, we have developed an implementation of the quantities domain model and computational kernel for Foundational UML (fUML) [14]. Foundational UML is an executable subset of UML, which is standardized by OMG. It comprises UML concepts for defining UML class diagrams to model system structures, and UML concepts for defining UML activity diagrams to model 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. The execution 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 only supports the primitive data types `Boolean`, `Integer`, `Real`, `String`, and `UnlimitedNatural`. We have extended the fUML's type system for the introduction of new datatypes as well as new methods operating on them. 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 6 shows the different quantity types (`Length`, `Mass`, etc.), unit types (`LengthUnit`, `MassUnit`, etc.), and the `UReal` type.They are implemented as new fUML datatypes. To enable computations on values of these new datatypes, the so-called fUML *function behaviors* are defined for each operation on these types. These fUML function behaviors define the names and parameters of the operations, as well as their behavior, which has to be implemented according to an interface prescribed by the fUML virtual machine. Each time an operation on these types is invoked, the fUML virtual machine will execute the corresponding behavior. Our implementations of the quantity operations use the Java implementation of the proposed computational kernel for quantities presented in Section 5.2. Note that to enable the static type checking of the correct usage of quantity operations, we not only need to define fUML function behaviors for the quantity operations of the abstract type `Quantity` shown in Fig. 5, but also specialized function behaviors for the subtypes of `Quantity` that constraint the parameters of the quantity operations (cf. discussion on static type checking in Section 4.6).

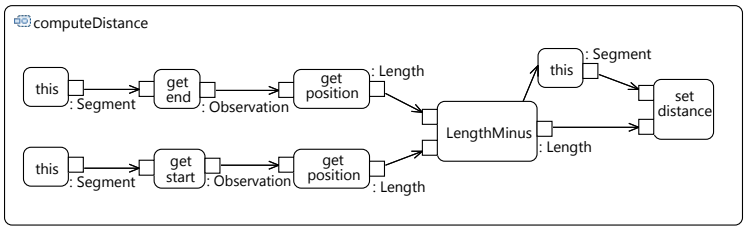

Figure 7: fUML example using quantities (model created with Eclipse Papyrus).

With our extensions, quantities can be used in fUML models as data types of class attributes and operation parameters, and operations on these quantities can be used for defining and executing computations on quantities. As an example, Figure 7 shows an fUML activity diagram for our motivating example computing the distance traveled by a particle within a segment. Note that for subtracting two positions for computing the distance, we use the function behavior `LengthMinus`, which is a specialization of the quantity operation `Minus` defined for the `Quantity` type of the kernel representation (see Fig. 5). `LengthMinus` specializes `Minus` by defining that the operation expects two parameters of type `Length` as input and provides a `Length` as output.

Our fUML implementation is integrated with the Eclipse Modeling Framework[6] and the fUML reference implementation[7], and has been created with Eclipse Papyrus.[8] Note that the current fUML implementation is only a proof-of-concept implementation showing how quantities can be integrated with fUML. As such, our fUML implementation currently provides implementations for just a few quantity types and operations.

6. Evaluation

In this section, we present the evaluation of our approach with respect to applicability, effectiveness, and efficiency.

6.1. Applicability (RQ2), Effectiveness (RQ3), and Efficiency (RQ4)

To answer the research questions concerning applicability (RQ2), effectiveness (RQ3), and efficiency (RQ4), defined in Section 2.6, we have used two case studies from the papers we surveyed when analyzing the relevance (RQ1) of our proposal (see Sec. 2.6.1). These models have sufficient detail to be modeled in full, do not deal with units or measurement uncertainty, and are specified in UML. Given that UML and SysML are well known and widely used for modeling physical systems, we focus on these notations, and especially on UML.

The following subsections fully describe these case studies, the metrics defined to assess the relevant properties of our approach, and the results of the application of those metrics.

6.1.1. Selected Case Studies

Case 1. As the first modeling language, we selected the family of DSLs for mobile multi-robot systems proposed by Ciccozzi et al. [43]. This 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 in the Eclipse Modeling Framework (EMF) using Ecore as the metamodeling language. Overall, the family of languages contains approximately 63 classes and 130 attributes.

To give the reader an idea of the type of language family that we are studying, Figure 8 (left) shows an excerpt of the robot language metamodel. The core of this metamodel is the `Robot` concept which permits modeling battery-operated mobile robots by specifying their devices and movement capabilities. The attributes of the metaclasses are typed using the standard Ecore data types: `Real`, `String`, `Integer` and `Boolean`. In our study, our aim is to find attributes that actually represent physical quantities, and therefore should be more precisely typed using quantity kinds. For this, we studied the metamodel definitions and the language description extracted from [43]. To illustrate

[6] `https://eclipse.org/modeling/emf`
[7] `https://github.com/ModelDriven/fUML-Reference-Implementation`
[8] `https://www.eclipse.org/papyrus/`

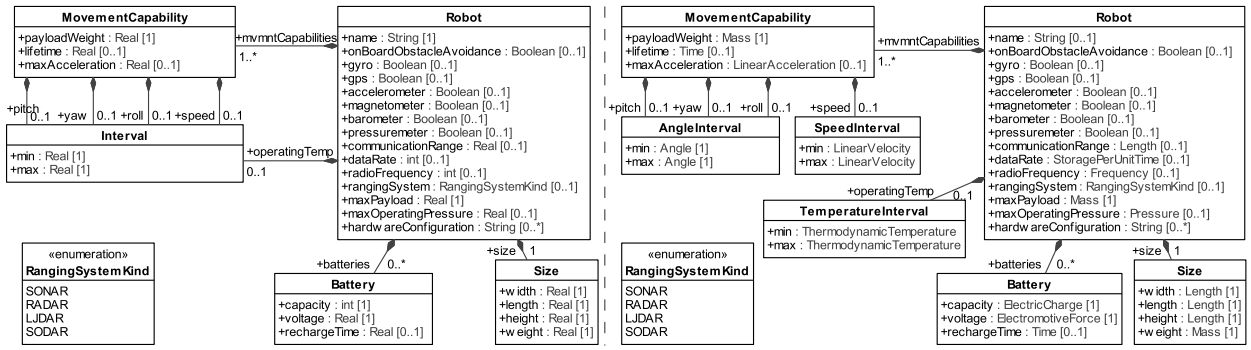

Figure 8: Excerpt of the robot language taken from [43]. The original version is on the left. On the right, the metamodel with quantity types.

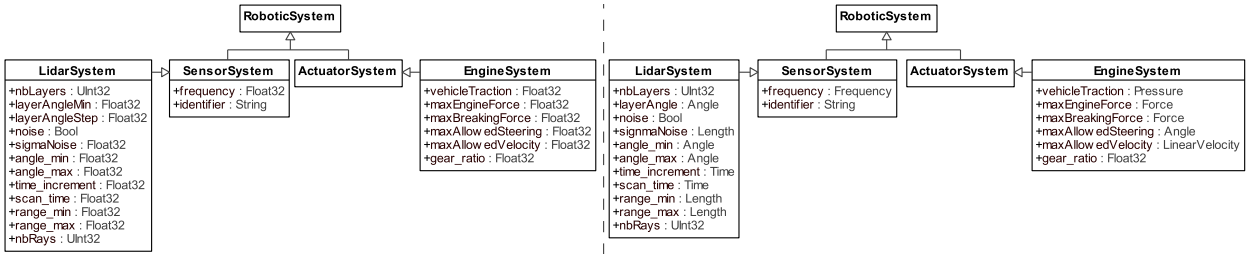

Figure 9: Excerpt of the RobotML language. The original version is on the left. On the right, the metamodel with quantity types.

how attributes are described in that paper, the following is the description given for attribute *radioFrequency*: "*...it is used to indicate the radio frequency used by the robot to communicate with the control station, expressed in MHz.*" The right of Figure 8 shows the `Robot` language metamodel after the types of the attributes representing quantities were replaced by their corresponding quantity kinds.

Case 2. For a second language we chose RobotML [44], a dedicated language for designing robotic applications. It permits both simulation and deployment to multi-target platforms. RobotML is implemented in Eclipse Papyrus, using UML profiles as the metamodeling language. Thus, RobotML is a domain-specific language based on UML. Overall, the RobotML language contains approximately 80 classes and 75 attributes.

To give the reader an idea of RobotML, Figure 9 (left) shows an excerpt from the language metamodel. The `RoboticSystem` is divided into different subtypes of systems, e.g., in `SensorSystems` and `ActuatorSystems`. These system types are further refined into more concrete types of sensors and actuators. As shown on the left-hand side of Figure 9, the attributes of the metaclasses are originally typed using types that are available in the individual robotic platforms: `UInt32`, `Float32`, `String` and `Boolean`. As before, the right-hand side of Figure 9 shows the model excerpt after the types of the attributes representing quantities have been replaced by their corresponding quantity kinds.

6.1.2. Measures

To evaluate the applicability (RQ2) of our approach, we use the application rate (AR) metric, which is defined as the ratio between attributes representing quantities and all the model attributes.

$$AR = \frac{|attributes->select(x|x.type.oclIsKindOf(Quantity))|}{|attributes|} \quad (1)$$

Concerning the effectiveness (RQ3) of our approach, we use the coverage (c) metric, which is defined as the ratio of the relevant quantities needed in the given languages that are successfully supported by our library.

$$C = \frac{|Q|}{|Q->intersection(Lib!Quantity.allInstances())|} \quad (2)$$

Table 2: Results concerning Application Rate and Effectiveness.

RobotFM Metamodel	#Class	#Atts	#QuantityAtts	#DiffQuantitiesUsed	AR	C	Dimensions Used
Behavior	24	22	10	3	0.45	1	Angle, Length, Time
Context	5	8	6	2	0.75	1	Angle, Length
Drone	8	48	31	14	0.65	1	Angle, Length, Time, Frequency, Storage, Mass, StoragePerTimeUnit, Power, ElectromagneticForce, ElectricCharge, LinearVelocity, LinearAcceleration, AngularVelocity, ThermodynamicTemperature
Mission	13	11	5	2	0.45	1	Angle, Length
Robot	8	34	21	12	0.62	1	Angle, Length, Time, Frequency, Mass, StoragePerTimeUnit, Pressure, ElectromagneticForce, ElectricCharge, LinearVelocity, AngularVelocity, ThermodynamicTemperature
Total	63	130	76	15	(avg) 0.58	(avg) 1	Angle, Length, Time, Frequency, Mass, StoragePerTimeUnit, Pressure, ElectromagneticForce, ElectricCharge, LinearVelocity, AngularVelocity, ThermodynamicTemperature, Storage, Power, LinearAcceleration

RobotML Metamodel	#Class	#Atts	#QuantityAtts	#DiffQuantitiesUsed	AR	C	Dimensions Used
Total	80	75	40	9	0.53	1	Angle, Length, Time, Frequency, Mass, LinearVelocity, LinearAcceleration, Force, Pressure

where

$$Q = attributes->select(x|x.type.oclIsKindOf(Quantity))->collect(x|x.type)->asSet()$$

To assess the efficiency (RQ4) of our approach, we had to evaluate the effort of applying it to a model. For this, we measured: (a) how many attributes may be simply re-typed by substituting the basic data type by a quantity kind; (b) how many additional atomic changes are needed to introduce the quantity types.

To compute the number of atomic changes, we built a difference model generated by comparing the initial model (iM) and the revised quantified model (qM). The difference model contains all changes (additions, deletions, and updates of elements) which can be detected between the two models. Based on the difference model, we computed the number of the differences reported, and related this number to the number of attributes to be re-typed. We used this ratio to indicate the application cost (AC) which is summarized in the following formula.

$$AC = \frac{|diff(iM, qM).elements|}{|attributes->select(x|x.type.oclIsKindOf(Quantity))|} \quad (3)$$

6.2. Results

The following subsections summarize the results of the application of the metrics defined above to the two selected case studies.

6.2.1. Applicability (RQ2)

Starting with the application rate (AR), Table 2 summarizes the results of applying our approach to the two selected robot languages. Columns 3 and 4 indicate the total number of model attributes (#Atts), and the number of attributes that represent quantities ("dimensional" attributes, #DimAtts), respectively. The rows under the RobotFM Metamodel correspond to the five languages that comprise it: Behavior, Context, Drone, Mission and Robot.

For the two modeling languages, we obtain an application rate of 55% on average, which means that more than half of the language attributes actually represent quantities in both languages.

6.2.2. Effectiveness (RQ3)

In Table 2, we also report on the quantity types used. Overall, our proposed approach is suitable to precisely describe all dimensions used for all attributes of the two modeling languages. Most of them are base quantities, but there are also derived quantities such as *StoragePerTimeUnit*. Attributes not referring to quantities are mainly used to introduce identifiers, names and configuration information such as the availability of measurement devices. As we could not find an attribute that we could not precisely type with our proposed library, we obtained a coverage of 100% for both languages.

6.2.3. Efficiency (RQ4)

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—the one needed to change the type of the attribute from a basic type to a quantity type. In some specific cases, additional changes were necessary, which we explain below. Overall, we had to perform 125 atomic changes for 116 attributes. This gives us an application cost of 1.07 changes per attribute. Table 3 summarizes the results of applying our approach in relation to its cost.

Table 3: Results concerning Application Cost.

Metamodel	**#QuantityAtts**	**#Differences**	**AC**	**Comments**
Behavior	10	10	1	Only type substitution
Context	6	6	1	Only type substitution
Drone	31	31	1	Only type substitution
Mission	5	5	1	Only type substitution
Robot	21	30	1.43	Interval class specialization
RobotML	40	40	1	Only type substitution
Total	116	125	(avg) 1.07	

Taking a closer look at the attributes that require more than a single change, we observed a recurring pattern. These attributes normally refer to an `Interval` class, which establishes the minimum and maximum values for the attribute values. Such a class can normally be reused by different attributes when the type of its min and max values is `Real`. However, when the referencing attributes are properly typed (to represent angles, speeds or temperatures), the interval class also needs to be retyped accordingly. This is shown in Figure 8. Note how `Interval` class on the left-hand side model is separated into three different interval classes in the model on the right. Even so, the application cost is still very low in this case.

Finally, note that in our proposal, quantity types come equipped with measurement uncertainty information. This is an additional benefit, because otherwise language engineers are forced either to ignore this kind of information, or to add further attributes to the classes when indeed required.

6.3. Limitations

In addition to the potential benefits described in the previous sections, during the development and application of our proposal to different system models, we have identified some limitations, which we report here.

Firstly, the integration of our domain model with other modeling languages may not be straightforward, if they do not follow an object-oriented paradigm. We assume we are able to extend the base data types of the language as well as static typing support to reason about correctness.

Secondly, when incorporating quantity types to an existing model that does not use them, sometimes it may be difficult to identify the correct quantities. For this, communication between and incorporation of domain experts is needed to clarify which dimensions and units to use.

Thirdly, with regard to precision and rounding, further studies are required. On the one hand, technical aspects such as precision of floating point numbers [33] or the usage of other base data types such as integers have to be investigated. On the other hand, requirements from particular domains have to be elicited in more detail. For instance, domain specific problems that need domain specific knowledge to be solved are required to deal with currencies. Just consider the facts that currencies have a different maximum number of digits after the decimal separator or that different currency increments have to be used because of country-specific laws.

Finally, there are also limitations concerning the current evaluation of our approach. As for any case study, the results are specific to the presented cases and may not be generalizable to other cases. We have focused on the cyber-physical system modeling domain and have used two prominent protagonists which are concerned with robotics. Other languages focusing on other aspects of cyber-physical systems or languages for completely different domains may show different results.

7. 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.

7.1. Modeling Physical Quantities

The two most prominent existing software modeling languages for modeling physical quantities are MARTE [6] and SysML [7], which we have already extensively discussed in Section 2.

The ISO VIM [21, 37] defines a complete set of concepts for modeling quantities and units. In this sense, the main difference between the VIM and our approach is that, while the VIM considers two concepts, namely `Quantity` and `QuantityValue`, we only use `Quantity`, which plays the role of the VIM's `QuantityValue`. This is because, in our approach, classes `Length`, `Mass`, `LinearVelocity`, etc., used to type the attributes of UML classes, inherit from `Quantity`. Following the approach of VIM (instead of ours) would have implied that they would have been called `LengthValue`, `MassValue`, `LinearVelocityValue`— i.e., while `Length`, `Mass`, etc. are quantity types in VIM, the ranges of the attributes should be typed `QuantityValue`. We deviate from VIM in order to remain close to the way in which datatypes are usually modeled in MARTE and SysML notations. Therefore, our proposal is in line with these notations, and it would allow solving the problems mentioned in the introduction without introducing larger changes in the way quantities and units are modeled in the OMG standards. We currently support quantities with only one value and its associated unit but we will consider as a future extension to allow for more values (and units). Eventually, as part of our future work, we would like to propose a design fully in accordance with the VIM.

Besides MARTE and SysML, there are languages specifically devised for modeling physical systems that provide dedicated support for units. For instance, Modelica [45] provides SI unit support [46] as well as different reasoning techniques for the correct and user-friendly usage of units [47, 48, 49]. This is also the case for Mathematica, which provides enhanced support for units [50]. Finally, other modeling languages for particular domains, such as biology [51] and meteorology [52], also provide support for units.

The Ontology of Units of Measure (OM) [53, 54] and QUDT [8] are two examples of units-of-measure ontologies, which are in line with VIM. They play an interesting part in leveraging units of measure concepts and the like in Semantic Web approaches for modeling physical systems and quantities. It would be recommended to explore linking to these initiatives in the future.

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 [38]), Python (e.g., see the packages Numericalunits, Pint, Unit, and Uncertainties) [34, 39], Ruby [40] and F# [55, 56]. Units are also implemented for Eiffel [57], although that work is discontinued. 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. Our prime interest is to provide modelers with a mechanism that allows them to faithfully represent both the dimensions of their quantities and their associated precision in their high-level models, 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 of any simulation or implementation, represents a significant step ahead with regard to existing modeling solutions.

Language-independent design patterns have also been proposed to represent different types of quantities, such as the Quantity Pattern [58] as well as idioms for nominally typed object-oriented programming languages [59]. They permit specifying the quantity kinds, but they are not so effective for performing operations with them at the model level, since unit conversions need to be defined between each other. The internal representation used in our approach, which uses a tuple with the base quantities, facilitates the conversions and also the operations to query the properties of units—e.g., that two units are compatible.

Finally, note that in this paper we have focused only on physical units, without considering other kinds of units, such as money. Although in principle similar matters, it incorporates two issues that induce problems of different matter, as clearly explained by Martin Fowler in [60]. 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.

7.2. Modeling Measurement Uncertainty

Regarding the consideration of measurement uncertainty in software models, several authors have identified the need of mechanisms to represent and manipulate physical values in software models [5], in particular units or real-time properties. For example, some work on Business Process Models (e.g., [61]) 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 approaches use probabilistic mass functions for modeling the values of the corresponding attributes, instead of fixed values. We use the way defined by the GUM [9, 23], which uses the standard deviation of the possible variation of the quantity values. Despite losing some generality, we gain some key benefits; in particular we are able to operate with uncertain values and to propagate their uncertainty in an effective manner. For example, in this way we are able to combine several quantities that follow diverse, or even unknown, distributions, and for which a closed form solution does not exist. This approach to represent measurement uncertainty is also widely used by the rest of the engineering disciplines.

Similarly, the definition and management of uncertainty in measurements is widespread in other domains like real-time systems where, indeed, timing values are by their very nature uncertain (they are very often estimates and/or measured by means of monitoring). The real-time community is accustomed 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 tag definition of a 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 an 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 definition of operations to deal with measurement uncertainty and units, and its integration with fUML. Model transformations can provide the relationship between MARTE and SysML and our proposal transparently and clean.

8. Summary 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. Some of the existing modeling languages, such as MARTE or SysML, already provide mechanisms for describing these properties. However, these mechanisms are not integrated into their type systems and therefore do not support operations for propagating uncertainty or for statically checking possible unit mismatches. Every (software) engineer recognizes these problems, 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. Also provided is a ready-to-use library of dimensions (Length, Mass, etc.) implemented in UML, Java and OCL, that can be added to other modeling projects, and that permits modelers to naturally and safely represent and manipulate units and measurement uncertainties of physical systems.

The work presented here paves the way towards several interesting lines of research that we would like to explore next. First, it would be interesting to provide mappings to and from other modeling notations, such as MARTE or

SysML, using model transformations. 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, it provides the mapping of quantity-aware models to existing programming languages (such as Python [34, 39], Java [38], Ruby [40], F# [55, 56]) and analysis tools (Simulink, Matlab, Modelica [47, 48]) 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, the need for being able to cope with units and uncertainty is becoming even more evident. If models and programs need to be connected and synchronized to fully achieve model-driven development, 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 that do not take units or uncertainty into account presently, e.g., [62]. Enriching these notations with units and uncertainty can bring to them all the interesting benefits that we have described in the paper while the reuse of our library could significantly reduce the cost.

In addition to the computational capabilities of our proposal, we can also work on enhancing its presentational aspects, using more compact representations.

Finally, in the context of inter-disciplinary engineering where models have to be exchanged between different disciplines, including the usage of engineering models as well as scientific ones, a potential alignment with VIM seems beneficial in the future. As discussed before, the presented approach and the current versions of the modeling language standards such as SysML, UML/MARTE deviate from VIM. However, with currently ongoing standardization efforts concerning SysML v2[9]—where systems of quantities as well as enhanced representation of values and value types are both stated as requirements in the request for proposals (RFP) for the next-generation system modeling language—the modeling community has the unique opportunity to discuss a potentially stronger alignment with VIM.

Acknowledgments. We would like to sincerely thank the reviewers for their insightful comments and suggestions, which significantly helped improving the manuscript. This work is funded by the Spanish Research Projects TIN2014-52034-R and TIN2016-75944-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

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

[2] W. H.Nelson, The Gimli Glider Incident (1997).
URL https://en.wikipedia.org/wiki/Gimli_Glider

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

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

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

[6] OMG, UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Version 1.1, OMG Document formal/2011-06-02 (Jun. 2011).

[7] OMG, OMG Systems Modeling Language (SysML), version 1.4, OMG Document formal/2016-01-05 (Jan. 2016).

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

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

[10] T. Mayerhofer, M. Wimmer, A. Vallecillo, Adding uncertainty and units to quantity types in software models, in: Proc. of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2016), ACM, 2016, pp. 118–131.

[11] ISO/IEC 80000:2009, Quantities and Units (2011).
URL https://www.iso.org/standard/30669.html

[12] OMG, Unified Modeling Language (UML) Specification. Version 2.5, OMG Document formal/2015-03-01 (Mar. 2015).

[13] OMG, Object Constraint Language (OCL) Specification. Version 2.4, OMG Document formal/2014-02-03 (Feb. 2014).

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

[9] http://www.omgsysml.org/SysML-2.htm

[15] IEEE Std 1003.1-2017, The Open Group Base Specifications. Issue 7, Sect. 4.16, Seconds Since the Epoch (2017).
[16] H. Espinoza, D. Cancila, B. Selic, S. Gérard, Challenges in Combining SysML and MARTE for Model-Based Design of Embedded Systems, in: Proc. of ECMDA-FA'09, 2009, pp. 98–113.
[17] D. Flater, Architecture for Software-Assisted Quantity Calculus. NIST Technical Note 1943 (Dec. 2016).
URL https://doi.org/10.6028/NIST.TN.1943
[18] C. Wohlin, Guidelines for snowballing in systematic literature studies and a replication in software engineering, in: Proc. of EASE'14, ACM, 2014, pp. 38:1–38:10.
[19] K. Thramboulidis, F. Christoulakis, UML4IoTâĂŤA UML-based approach to exploit IoT in cyber-physical manufacturing systems, Computers In Industry 82 (2016) 259–272.
[20] W. Oueslati, J. Akaichi, A trajectory UML profile for modeling trajectory data: A mobile hospital use case, CoRR abs/1102.4429. arXiv:1102.4429.
URL http://arxiv.org/abs/1102.4429
[21] ISO/IEC Guide 99:2007, International vocabulary of metrology – Basic and general concepts and associated terms (VIM), Joint Committee for Guides in Metrology, http://www.bipm.org/utils/common/documents/jcgm/JCGM_200_2012.pdf (2012).
[22] B. N. Taylor, A. Thompson, The International System of Units (SI), NIST, http://www.nist.gov/pml/pubs/sp811/ (2008).
[23] JCGM 101:2008, Evaluation of measurement data – Supplement 1 to the "Guide to the expression of uncertainty in measurement" – Propagation of distributions using a Monte Carlo method, Joint Committee for Guides in Metrology, http://www.bipm.org/utils/common/documents/jcgm/JCGM_101_2008_E.pdf (2008).
[24] OMG, OMG Structured Metrics Metamodel (SMM), version 1.2, OMG Document formal/18-05-01 (Jun. 2018).
[25] M. Zhang, S. Ali, T. Yue, R. Norgren, O. Okariz, Uncertainty-wise cyber-physical system test modeling, Software & Systems Modelingdoi:10.1007/s10270-017-0609-6.
URL https://doi.org/10.1007/s10270-017-0609-6
[26] R. T. Boute, A heretical view on type embedding, SIGPLAN Not. 25 (1) (1990) 25–28.
[27] B. H. Liskov, J. M. Wing, A behavioral notion of subtyping, ACM Trans. Program. Lang. Syst. 16 (6) (1994) 1811–1841. doi:10.1145/197320.197383.
[28] P. America, Inheritance and subtyping in a parallel object-oriented language, in: Proc. of the European Conference on Object-Oriented Programming, ECOOP'87, Springer, 1987, pp. 234–242.
URL http://dl.acm.org/citation.cfm?id=646147.679032
[29] S. Clerici, F. Orejas, Gsbl: An algebraic specification language based on inheritance, in: S. Gjessing, K. Nygaard (Eds.), ECOOP'88 European Conference on Object-Oriented Programming, Springer, 1988, pp. 78–92.
[30] P. America, Inheritance hierarchies in knowledge representation and programming languages, John Wiley and Sons Ltd., Chichester, UK, 1991, Ch. A Behavioural Approach to Subtyping in Object-oriented Programming Languages, pp. 173–190.
URL http://dl.acm.org/citation.cfm?id=120539.120551
[31] M. F. Bertoa, N. Moreno, G. Barquero, L. Burgueño, J. Troya, A. Vallecillo, Expressing measurement uncertainty in OCL/UML datatypes, in: Proc. of ECMFA'18, Vol. 10890 of LNCS, Springer, 2018, pp. 46–62.
[32] T. Mayerhofer, M. Wimmer, A. Vallecillo, Computing with Quantities: the Java Project (2016).
URL https://github.com/moliz/moliz.quantitytypes
[33] D. Goldberg, What every computer scientist should know about floating-point arithmetic, ACM Comput. Surv. 23 (1) (1991) 5–48.
[34] H. E. Grecco, Temperature Conversions (2016).
URL http://pint.readthedocs.io/en/0.7.2/nonmult.html
[35] Mathworks, Thermal Unit Conversions (2016).
URL http://www.mathworks.com/help/physmod/simscape/ug/thermal-unit-conversions.html
[36] Wolfram, Temperature Units (2018).
URL https://reference.wolfram.com/language/tutorial/TemperatureUnits.html
[37] JCGM 200:2012, International Vocabulary of Metrology – Basic and general concepts and associated terms (VIM), 3rd edition, Joint Committee for Guides in Metrology, http://www.bipm.org/utils/common/documents/jcgm/JCGM_200_2012.pdf (2012).
[38] J.-M. Dautelle, W. Keil, L. Lima, Java JSR 363: Units of Measurement API (2016).
URL https://www.jcp.org/en/jsr/detail?id=363
[39] E. O. Lebigot, Uncertainties package (2016).
URL https://pythonhosted.org/uncertainties/
[40] K. C. Olbrich, Ruby Units (2016).
URL https://github.com/olbrich/ruby-units
[41] F. Büttner, M. Gogolla, On OCL-based imperative languages, Sci. Comput. Program. 92 (2014) 162–178.
[42] M. Gogolla, F. Büttner, M. Richters, USE: A UML-based specification environment for validating UML and OCL, Sci. Comput. Program. 69 (2007) 27–34.
[43] F. Ciccozzi, D. D. Ruscio, I. Malavolta, P. Pelliccione, Adopting MDE for specifying and executing civilian missions of mobile multi-robot systems, IEEE Access 4 (2016) 6451–6466. doi:10.1109/ACCESS.2016.2613642.
URL https://doi.org/10.1109/ACCESS.2016.2613642
[44] S. Kchir, S. Dhouib, J. Tatibouet, B. Gradoussoff, M. D. S. Simoes, Robotml for industrial robots: Design and simulation of manipulation scenarios, in: 21st IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, 2016, pp. 1–8.
[45] P. Fritzson, V. Engelson, Modelica – a unified object-oriented language for system modeling and simulation, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1998, pp. 67–90.
[46] Modelica, Modelica SI Units (2011).
URL https://build.openmodelica.org/Documentation/Modelica.SIunits.html
[47] K. L. Davies, C. J. Paredis, Natural Unit Representation in Modelica, in: Proc. of the 9th International MODELICA Conference, Modelica

Association, Linköping University Electronic Press, 2012, pp. 801–808.
[48] S. E. Mattsson, H. Elmqvist, Unit Checking and Quantity Conservation, in: Proc. of the 6th International MODELICA Conference, Modelica Association, Linköping University Electronic Press, 2008, pp. 13–20.
[49] P. Aronsson, D. Broman, Extendable Physical Unit Checking with Understandable Error Reporting, in: Proc. of the 7th International MODELICA Conference, Modelica Association, Linköping University Electronic Press, 2009, pp. 890–897.
[50] Wolfram Research Inc., Mathematica. Support for Units (2016).
URL https://reference.wolfram.com/language/guide/Units.html
[51] M. Hucka, A. Finney, et al., The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models, Bioinformatics 19 (4) (2003) 524–531.
[52] M. Wolf, A Modeling Language for Measurement Uncertainty Evaluation, ETH, 2009.
[53] H. Rijgersberg, M. Wigham, J. Top, How semantics can improve engineering processes: A case of units of measure and quantities, Advanced Engineering Informatics 25 (2) (2011) 276 – 287. doi:https://doi.org/10.1016/j.aei.2010.07.008.
[54] H. Rijgersberg, M. van Assem, J. L. Top, Ontology of units of measure and related concepts, Semantic Web 4 (1) (2013) 3–13.
[55] A. J. Kennedy, Relational parametricity and units of measure, in: Proc. of POPL'97, ACM, 1997, pp. 442–455.
[56] A. J. Kennedy, Types for units-of-measure: Theory and practice, in: Proc. of CEFP'09, Vol. 6299 of LNCS, Springer, 2010, pp. 268–305.
[57] M. Keller, Eiffel Units (2002).
URL http://se.inf.ethz.ch/old/projects/markus_keller/EiffelUnits.html
[58] M. Fowler, Analysis Patters: Reusable Object Models, Addison-Wesley, 1997.
[59] E. E. Allen, D. Chase, V. Luchangco, J. Maessen, G. L. Steele Jr., Object-oriented units of measurement, in: Proc. of OOPSLA'04, ACM, 2004, pp. 384–403.
[60] M. Fowler, Quantity: Represent dimensioned values with both their amount and their unit.
URL http://martinfowler.com/eaaDev/quantity.html
[61] A. Jiménez-Ramírez, B. Weber, I. Barba, C. D. Valle, Generating optimized configurable business process models in scenarios subject to uncertainty, Information & Software Technology 57 (2015) 571–594.
[62] I. Dragomir, I. Ober, C. Percebois, Contract-based Modeling and Verification of Timed Safety Requirements Within SysML, Software and System Modeling 16 (2) (2017) 587–624.