Using Physical Quantities in Robot Software Models

Loli Burgueño
Universidad de Málaga, Atenea Research Group
Málaga, Spain
loli@lcc.uma.es

Tanja Mayerhofer
TU Wien, Business Informatics Group
Vienna, Austria
mayerhofer@big.tuwien.ac.at

Manuel Wimmer
TU Wien, CDL-MINT
Vienna, Austria
wimmer@big.tuwien.ac.at

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

ABSTRACT

One of the challenges of modeling any software application that deals with real-world physical systems resides in the correct representation of numerical values and their units. This paper shows how both measurement uncertainty and units can be effectively incorporated into software models, becoming part of their basic type systems, and illustrates this approach in the particular case of a robot language. We show how our approach allows robot modelers to safely represent and manipulate units and measurement uncertainties of the robots and their elements in a natural manner, statically ensuring unit-safe assignments and operations, as well as the propagation of uncertainty in the computations of derived attributes and operations.

KEYWORDS

Model-driven engineering, quantities, units, measurement uncertainty, cyber-physical systems, robotics

ACM Reference Format:
Loli Burgueño, Tanja Mayerhofer, Manuel Wimmer, and Antonio Vallecillo. 2018. Using Physical Quantities in Robot Software Models. In *RoSE'18: RoSE'18:IEEE/ACM 1st International Workshop on Robotics Software Engineering , May 28-June 28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3196558.3196562

1 INTRODUCTION

As any other cyber-physical systems, robots are typically designed and built as a network of interacting elements of different nature (software, hardware, communications, sensors, actuators, etc.) with physical input and output instead of as standalone devices, and with complex interactions among their internal elements, and also with their physical environment [15]. Model-Driven Engineering (MDE) [4] is a state-of-the-art approach for the design, development and maintenance of software applications, particularly well suited to deal with complex systems because it relies on two basic and key concepts: abstraction and automation [24]. MDE has been successfully used in many domains for modeling complex systems [11, 23], and software models have proved to be very useful for providing high-level formal descriptions of complex systems that permit their representation, specification, analysis and automated development.

When it comes to modeling robotic systems, and in order to faithfully represent and manipulate the key properties of physical world systems and their elements, software models need to be able to deal with correct numerical values and their units. This includes the representation of measurement uncertainty due to errors in physical measures or the tolerance of mechanical tools and devices, as well as the units in which these values are expressed. Ignoring uncertainty makes models too naïve, assuming that all measures are exact and that no deviation due to tolerance in the robot devices (gears, wheels, etc.) occur. Thus, we may be building a model of a system which does not provide a faithful representation for it.

Taking standard UML as a prominent example for a software modeling language, there is neither support for modeling units nor for modeling measurement uncertainty. Instead, mere `Real` numbers are used to specify attribute values that represent physical properties, explaining at most in the companion documentation the units in which each attribute value should be expressed (cf., for instance, [7]). Uncertainty is normally ignored, or considered somewhere else in the models.

Although some of the existing modeling languages, such as MARTE [20] and SysML [21], already provide mechanisms for describing these properties, such mechanisms are not integrated into their type systems and therefore they do not support operations for propagating uncertainty or for statically checking possible unit mismatches—which have already proved to be the cause of significant software disasters, such as the Mars Climate Orbiter [12] or the Gimli Glider Incident [16]. Furthermore, incorporating by hand units and measurement uncertainty into the models is far from trivial, and it can produce much more cumbersome models, significantly increasing the accidental complexity of the solution. In addition, both unit conversions and propagation of uncertainty need to be explicitly handled by users.

The authors of this paper have been recently working on an approach to deal with measurement uncertainty and units in software models [17, 18]. In particular, we have defined an extension of the UML and OCL type `Real` to represent physical properties, called `Quantity`, a set of operations for this new type, and type checks that impede the unit-mismatch problem.

This paper presents a concrete case study of the use of our approach in the case of a language for specifying robot movements. It is based on the `Ozoblockly` language (http://ozoblockly.com) for controlling `Ozobot` robots (https://ozobot.com). We show how physical properties of such robots and their environment can be effectively specified using the introduced `Quantity` type, how their units can be safely combined and computed, and how measurement uncertainties can be propagated when operating with them. We also show how USE/OCL [8] can be employed for analyzing and quickly prototyping modeled robot missions. Note that existing languages that could be employed for this application case do not cover all these features. For example, the robot family of languages defined by Davide et al. [7] does not consider units and uncertainty, and the `Ozoblockly` language does not take into account uncertainty.

The structure of this paper is as follows. After this introduction, Section 2 briefly presents the concepts related to quantities, units and measurement uncertainty. After that, Section 3 presents the case study of the `Ozoblockly` language, while Section 4 describes some of the analysis we are able to conduct on the models. Finally, Section 5 compares our work to similar proposals, and we conclude in Section 6 with an outlook on future work. All the models and artefacts described in this paper are available from [19].

2 BACKGROUND

A *Quantity* is an observable property of an object, event or system that can be measured and quantified numerically [10]; for example its position, size, speed or temperature. By convention, physical quantities are organized in a *system of dimensions*. Examples of these *dimensions* are length, mass, time, force, energy, power and electric charge. These are expressed in *units*. The *Value* of a quantity is its magnitude expressed as the product of a number and a unit. The number multiplying the *unit* is referred to as the *numerical value* of the quantity expressed in that unit [25]; for example, $3.5\ m/s$.

2.1 Units and Dimensions

The most widely used system of dimensions is the International System of Units (SI) [25]. It defines seven *base dimensions*: Length, Mass, Time, Electric Current, Thermodynamic Temperature, Amount of Substance, and Luminous Intensity; with its corresponding *base units*: Meter (*m*), Kilogram (*kg*), Second (*s*), Ampere (*A*), Kelvin (*K*), Mole (*mol*) and Candela (*cd*). The SI also defines 90 *derived dimensions* (Area, Volume, Velocity, Force, etc.) and their corresponding units (m^2, m^3, m/s, *Newton*, etc.).

The ISO/IEC 80000:2009 standard [1] extends the International System of Units incorporating four new base dimensions (and their corresponding base units): *Data Storage Capacity* (*bit*), *Entropy* (*shannon*), *Traffic Intensity* (*erlang*) and *Level* (*decibel*). 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.

Apart from the SI, there are other systems of units which are used in different countries. For example, the *Centimeter-Gram-Second System* (CGS) is a variant of the metric system that has the same dimensions but uses centimeters, grams and seconds as base units. The *Imperial System* used in UK also defines the same dimensions as the SI, but uses different units: miles, feet, inches, stones, pounds, etc. In USA, the *United States Customary System* (also called USCS or USC) is a variant of the Imperial System that uses different units for fluids. Since they define the same dimensions, conversions among these systems of units are possible by simply multiplying the quantity values by the corresponding conversion factors. In fact, any unit from any system can be expressed in terms of SI units, and the conversion among them can be easily defined using multiplication factors and, in some cases, offsets. For example, to convert from Celsius to Kelvin the conversion factor is 1.0 and the offset 273.15.

2.2 Measurement Uncertainty

When dealing with real-world entities, models need to take into account the inability to know, estimate or measure with complete precision the value of any quantity. This is for instance needed to take into account the tolerance of a device when taking a measurement, or to consider cases where estimations are needed because the exact values are too costly to measure. 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" [13, 14].

The Guide to the Expression of Uncertainty in Measurement (GUM) [13] defines the term *standard uncertainty* as "the uncertainty of the result of a measurement expressed as a standard deviation".

Finally, quantities are rarely used in isolation, but combined to produce aggregated measures or to calculate derived attributes. The individual uncertainties of the input quantities need to be combined too, to produce the uncertainty of the result. This is known as the *propagation of uncertainty*, or *uncertainty analysis* [13].

2.3 Integration into UML and OCL

In [17] we defined an extension of the UML and OCL type `Real`, called `Quantity`, that provides an algebra of operations for specifying and performing computations with measurement uncertainty and units in attributes representing properties of entities of the physical world. We also provided a ready-to-use library of dimensions (Length, Mass, etc.) implemented in UML, Java, OCL and fUML, that can be added to any modeling project, and that permits modelers to safely represent and manipulate units and measurement uncertainties of physical systems in a natural and transparent manner. These libraries are available from [19].

Figure 1 shows the basic elements of our proposal, whereby a `Quantity` is composed of a `value` and a `unit`. The class `Quantity` is then refined for each dimension specified in the ISO 80000 standard. Thus, values of type `Quantity` are given by a `Real` number representing the measurement result; the unit in which it is expressed; and the standard uncertainty associated to the measurement (i.e., the precision). For example, $(1000.0, 0.0001, m)$ or $(352.44, 0.0, ft)$ are valid values of type `Length`.

Each type has a set of associated operations, which define the valid operations on its values. These operations permit the implementation of static type checking mechanisms when assigning values to variables, or when defining expressions that compute the values of derived attributes. For example, the class Length offers an operation for multiplying a Length by another Length, giving an Area as result. Only valid operations are defined in every class, hence ensuring static type checking.

One important feature of these operations is that they take into account the units in which the operands are expressed, and convert them accordingly in order to avoid any unit-mismatch error. In this way, users do not need to worry about these issues: the type system takes care of them transparently.

In turn, the class UReal provides operations to operate with uncertain values, supporting the propagation of uncertainty in a natural and transparent manner. Also operations for comparing values with uncertainty are provided. These comparisons can return either a boolean ($a < b$), or they may also return a number between 0 and 1, indicating the probability of an uncertain number a being less (or equal) than another uncertain number b [17].

3 CASE STUDY: OZOBLOCKLY LANGUAGE

This section describes a case study of a system that requires the representation of both units and measurement uncertainty.

The studied system is an Ozobot robot (https://ozobot.com) that is able to move in the direction its head points to. These kinds of robots accept two type of commands: one to rotate its head at a certain angle, and another one to move forward some distance. Both commands can be combined, executing first the rotation and then the movement. A robot's Plan is composed of a sequence of movements. We can also assign a Mission to a given robot. The mission determines the target position it is supposed to reach with the plan. Positions are given by coordinates in a planar surface (i.e, they are points in a plane that represents the floor).

In this case study we are interested in analyzing a robot's behavior, and, in particular, whether the sequence of movements defined in its plan fulfills the mission, i.e., reaches the target position.

Figure 2 shows the representation of the system using UML. The class Robot offers the operation performAllMoves() that goes through all the movements in its plan, and performs them in sequence. Every movement calculates the target coordinate after the move, and requests the robot to advance to that position.

In order to check whether the robot has reached the final target position, the class Coordinate provides the operation coincide() that determines whether two coordinates are equal.

Suppose a scenario where a set of collaborating teams that are geographically distributed is controlling a robot. Each team defines some of the robot movements in order to reach the target. Suppose that each team is in a different country and uses a different system of units. For instance, one of them is in the UK and uses Imperial units while the other is in France and uses the SI. These issues will cause disasters.

Figure 3 shows how we can model the Ozobot robot system in UML, using our reusable library of Quantities. We can see how every attribute is typed with the dimension of the property it represents. In this example only two dimensions are used, Length and Angle. These are *types* in the sense that they define a set of values and a set of valid operations on them. Note that we have also defined some derived attributes in the classes Robot and Coordinate (e.g. /xyh, /xy and /u). These are not really needed for computational purposes, but they are rather useful for being able to visualize the values of the objects' attributes as we shall later see in Section 4.

Operations and derived values are specified in our approach in the usual manner. For example, the operation coincide() of class Coordinate can be specified as follows:

```
context Coordinate::coincide(c:Coordinate) : Boolean =
    self.x.equals(c.x) and self.y.equals(c.y)
```

Here the type system takes care of dealing with units of the values (internally performing the unit conversions, when required), and with the propagation of their measurement uncertainties.

Another operation that we have also included in this latter model is the method uCoincide() of the class Coordinate. Comparison operations for Real numbers return Boolean values. However, when comparing numbers with uncertainty there is always a degree of uncertainty in the comparison too, and this is why the result of a comparison may be given by a number between 0 and 1, indicating the probability with which two uncertain values are equal. Assuming the coordinates are independent, this probabilistic comparison is implemented by the method uCoincide(), which can be simply specified in OCL as follows, making use of the fuzzy comparison operators for UReal values:

```
context Coordinate::uCoincide(c:Coordinate) : Real =
    self.x.uEquals(c.x) * self.y.uEquals(c.y)
```

4 ANALYSIS

In this section we will discuss some of the possible analyses we can conduct on the presented UML models for Ozobot robots, namely simulation and verification. The advantage of these kinds of analyses is that they can be performed on high-level UML models of the system, well before the system is developed and deployed. Without claiming to be exhaustive, they are regarded as being useful for the early detection of structural problems or system-wide errors [27].

4.1 Simulation

SOIL [6] is a language supported by USE [8] to specify the behavior of OCL operations, and to execute models by means of creating object instances and invoking their operations.

For example, the operation performMove() of the class Movement can be enriched with executable behavior as follows:

```
class Movement
attributes
 move:Length
 rotate:Angle
operations
 performMove()
 begin
   declare aux:Coordinate,sa:UReal,ca:UReal,dx:Length,dy:Length;
   -- we change the angle first (if we have to)
   if not self.rotate.oclIsUndefined() then
       self.robot.headsTo:=self.rotate
   end;
   -- and then we move (if we have to)
   if not self.move.oclIsUndefined() then
       ca:=self.robot.headsTo.cos();
       sa:=self.robot.headsTo.sin();
       dx:=self.move.sMult(ca);
```

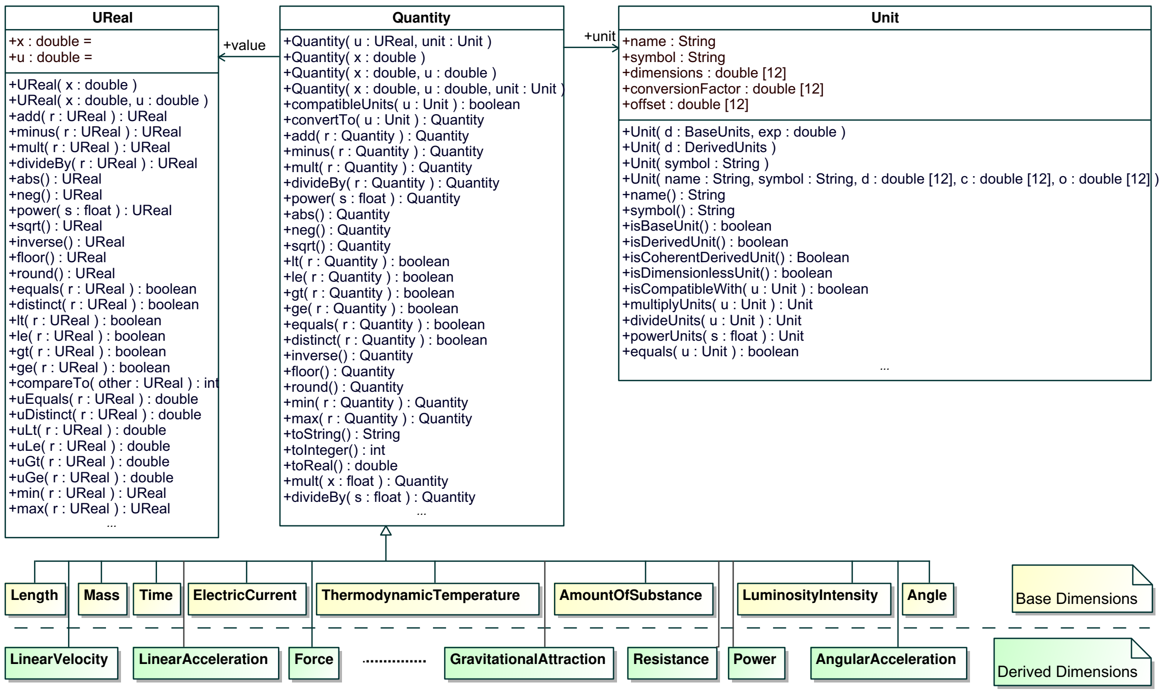
UReal
+x : double =
+u : double =
+UReal(x : double)
+UReal(x : double, u : double)
+add(r : UReal) : UReal
+minus(r : UReal) : UReal
+mult(r : UReal) : UReal
+divideBy(r : UReal) : UReal
+abs() : UReal
+neg() : UReal
+power(s : float) : UReal
+sqrt() : UReal
+inverse() : UReal
+floor() : UReal
+round() : UReal
+equals(r : UReal) : boolean
+distinct(r : UReal) : boolean
+lt(r : UReal) : boolean
+le(r : UReal) : boolean
+gt(r : UReal) : boolean
+ge(r : UReal) : boolean
+compareTo(other : UReal) : int
+uEquals(r : UReal) : double
+uDistinct(r : UReal) : double
+uLt(r : UReal) : double
+uLe(r : UReal) : double
+uGt(r : UReal) : double
+uGe(r : UReal) : double
+min(r : UReal) : UReal
+max(r : UReal) : UReal
...
+value
Quantity
+Quantity(u : UReal, unit : Unit)
+Quantity(x : double)
+Quantity(x : double, u : double)
+Quantity(x : double, u : double, unit : Unit)
+compatibleUnits(u : Unit) : boolean
+convertTo(u : Unit) : Quantity
+add(r : Quantity) : Quantity
+minus(r : Quantity) : Quantity
+mult(r : Quantity) : Quantity
+divideBy(r : Quantity) : Quantity
+power(s : float) : Quantity
+abs() : Quantity
+neg() : Quantity
+sqrt() : Quantity
+lt(r : Quantity) : boolean
+le(r : Quantity) : boolean
+gt(r : Quantity) : boolean
+ge(r : Quantity) : boolean
+equals(r : Quantity) : boolean
+distinct(r : Quantity) : boolean
+inverse() : Quantity
+floor() : Quantity
+round() : Quantity
+min(r : Quantity) : Quantity
+max(r : Quantity) : Quantity
+toString() : String
+toInteger() : int
+toReal() : double
+mult(x : float) : Quantity
+divideBy(s : float) : Quantity
...
+unit
Unit
+name : String
+symbol : String
+dimensions : double [12]
+conversionFactor : double [12]
+offset : double [12]
+Unit(d : BaseUnits, exp : double)
+Unit(d : DerivedUnits)
+Unit(symbol : String)
+Unit(name : String, symbol : String, d : double [12], c : double [12], o : double [12])
+name() : String
+symbol() : String
+isBaseUnit() : boolean
+isDerivedUnit() : boolean
+isCoherentDerivedUnit() : Boolean
+isDimensionlessUnit() : boolean
+isCompatibleWith(u : Unit) : boolean
+multiplyUnits(u : Unit) : Unit
+divideUnits(u : Unit) : Unit
+powerUnits(s : float) : Unit
+equals(u : Unit) : boolean
...
Length
Mass
Time
ElectricCurrent
ThermodynamicTemperature
AmountOfSubstance
LuminosityIntensity
Angle
Base Dimensions
LinearVelocity
LinearAcceleration
Force
....
GravitationalAttraction
Resistance
Power
AngularAcceleration
Derived Dimensions

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

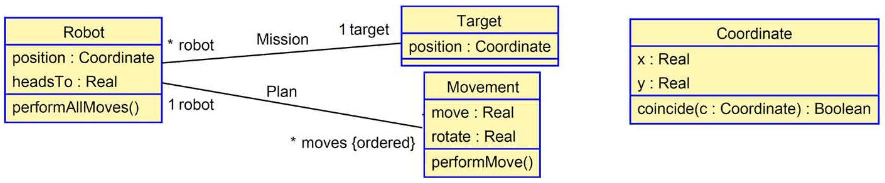

Figure 2: Initial Class diagram for the Moving Robot example, using plain UML.

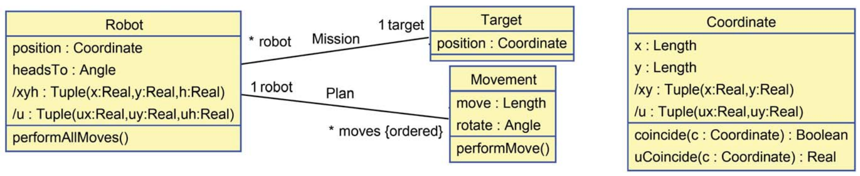

Figure 3: Class diagram for the Moving Robot example, with Quantities.

```
       dy:=self.move.sMult(sa);
       aux := new Coordinate;
       aux.x := self.robot.position.x.add(dx);
       aux.y := self.robot.position.y.add(dy);
       self.robot.position:=aux;
    end;
  end
end
```

Since all operations on attributes are performed using the enriched type system, they transparently take into account units and uncertainty. Similarly, the specification of the class `Movement` can be enriched with behavior as shown below.

```
class Robot
attributes
  position : Coordinate
  headsTo : Angle
operations
  performAllMoves()
  begin
    for m in self.moves do
          m.performMove()
    end
  end
end
```

Once the behavior of objects is specified with SOIL, the USE/OCL environment supports commands for creating objects and invoking operations on them.

Let us define the following list of movements we want the robot to perform. These movements request the robot to: move 10 meters in its pointing direction (initially 0); then rotate $\pi/2$ radians; advance 10m; rotate 225 degrees and advance $10\sqrt{2}$ ft; and finally rotate $\pi/4$ rad and move forward another $10\sqrt{2}$ ft. Starting from coordinate (0,0), this plan should lead it to its target position, (10,10). The precision of all measurements is supposed to be 1E-3 m.

The following listing shows an excerpt of a sequence of commands that we have issued to simulate this scenario. Lines 1–9 create `UReal` numbers, specifying their values and associated uncertainty. Then, lines 11–19 define some values of quantities of types `Length` and `Angle`. Note that some values are expressed in meters (line 13), others in feet (line 16).

```
1  Robot.soil> !new UReal('x10U')
2  Robot.soil> !x10U.x :=10.0
3  Robot.soil> !x10U.u :=0.001
4  Robot.soil> !new UReal('xy1010U')
5  Robot.soil> !xy1010U.x:=1.41421356*10.0
6  Robot.soil> !xy1010U.u:=0.001
7  Robot.soil> !new UReal('h90U')
8  Robot.soil> !h90U.x :=1.5707963267948965
9  Robot.soil> !h90U.u :=0.001
10 Robot.soil> ...
11 Robot.soil> !new Length('x10')
12 Robot.soil> !x10.value:=x10U
13 Robot.soil> !x10.unit:=m
14 Robot.soil> !new Length('xy1010')
15 Robot.soil> !xy1010.value:=xy1010U
16 Robot.soil> !xy1010.unit:=ft
17 Robot.soil> !new Angle('h90')
18 Robot.soil> !h90.value :=h90U
19 Robot.soil> !h90.unit:=rad
20 Robot.soil> ...
21 Robot.soil> !new Coordinate('initial')
22 Robot.soil> !initial.x:=x00
23 Robot.soil> !initial.y:=y00
24 Robot.soil> !new Coordinate('target')
25 Robot.soil> !target.x:=x10
26 Robot.soil> !target.y:=y10
27 Robot.soil> !new Robot('robot')
28 Robot.soil> !robot.position:=initial
29 Robot.soil> !robot.headsTo:=h0
30 Robot.soil> !new Movement('m1')
31 Robot.soil> !m1.move:=x10
32 Robot.soil> !new Movement('m2')
33 Robot.soil> !m2.rotate:=h90
34 Robot.soil> !new Movement('m3')
35 Robot.soil> !m3.move:=x10
36 Robot.soil> !new Movement('m4')
37 Robot.soil> !m4.rotate:=h225
38 Robot.soil> !m4.move:=xy1010
39 Robot.soil> !new Movement('m5')
40 Robot.soil> !m5.rotate:=h45
41 Robot.soil> !m5.move:=xy1010
42 Robot.soil> !insert(robot,m1) into Plan
43 Robot.soil> !insert(robot,m2) into Plan
44 Robot.soil> !insert(robot,m3) into Plan
45 Robot.soil> !insert(robot,m4) into Plan
46 Robot.soil> !insert(robot,m5) into Plan
47 Robot.soil> !robot.performAllMoves()
48 Robot.soil> !r:=robot.position.uCoincide(target)
49 Robot.soil> ?r
50 -> 0.03479626661931806 : Real
```

Lines 21–41 create instances of objects of types `Coordinate`, `Robot` and `Movement`, and assign values to their attributes. In turn, lines 42–46 create the instances (i.e., links) of the association `Plan` corresponding to the movements we want the robot to perform.

Line 47 asks the robot to perform all these movements. Once this is done, the resulting object model can be visualized with an object diagram in USE, as shown in Figure 4. We can easily see the resulting position of the robot and the values of its properties thanks to its derived attributes `/xyh`, `/xy` and `/u`.

Line 48 asks whether the robot position coincides with its target. Although both values are centered at (10,10), the robot's final position after the sequence of movements is off by one centimeter (1.0985E-2 m). However, the precision we are requesting for the target position is 1 millimeter. This is why the calculated probability of coincidence is just 3.4% (line 50). In other words, there is a high (96.6%) risk that the robot misses the target, at least with the required precision. Should we want to improve that situation, we could either request more precision to the robot's movements, or relax the requested uncertainty of the mission's target position to centimeters.

4.2 Adding Model Invariants

We can also specify some system invariants to be checked during the execution. For example, we could ask for the accumulated uncertainty of the robot's current position, making sure it does not go above a given threshold (e.g. 1E-2 m).

```
context Robot inv PrecisionUnderControl :
   self.position.x.u < 0.01 and self.position.y.u < 0.01
```

Furthermore, the behavior of operations is normally specified by determining their pre- and post-conditions. They can be used to check, for example, whether after given movements towards a coordinate we have deviated too much from it. In this way, we will be able to detect, during the execution of the system, any significant deviation of the robot. Let us assume as well an extended scenario where we consider obstacles that the robot cannot go through. A precondition for the operation `performMove()` can also check that the target coordinate is a reachable point, i.e. there are not obstacles preventing the robot from reaching its expected target position when moving forward.

5 RELATED WORK

This work uses the ideas presented in [9], incorporating not only uncertainty but also units, following [17, 18]. Our paper is closely related to those works that focus on the specification of robotic systems, specially those that deal with their behavior. Here the discussion happens between those that propose the use of separate views of the system, using independent domain-specific languages, and those that try to use general purpose modeling languages. One of the major problems with the former approach is the combination of the languages, both at the same level of abstraction (i.e., horizontally—see, e.g. [26]) and at different level of abstraction (i.e. vertically—one example of this kind of vertical combination for robotic systems is [3]). Among the latter, the most widely known ones use high-level component-based architectures with the functional decomposition of the robotic systems, using block-diagrams and/or UML components. Examples include SafeRobots [22], RobotML (http://robotml.github.io/), SmartSoft (http://smart-robotics.sourceforge.net/), V3CMM [2], BCM (http://www.best-of-robotics.org/bride/bcm.html), and HyperFlex [5]. Our approach is defined at a higher level of abstraction, even before the architecture of the system needs to be considered—just its basic functionality—and hence many of the details can be abstracted

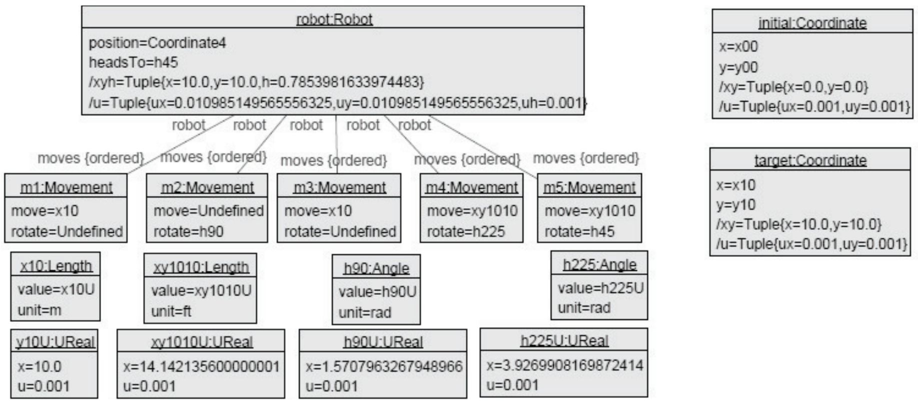

Figure 4: Object diagram for the Moving Robot example, after execution.

away. More importantly, none of these approaches permit dealing with units and measurement uncertainty in the models.

6 CONCLUSIONS AND FUTURE WORK

This paper presents, by means of the example of a robot language, how units of measurement and measurement uncertainty can be integrated into software models. Following our approach, dimensions (such as Length and Angle) can be considered primitive data types, whose values incorporate units and uncertainty. Dimensions also integrate the required mechanisms to deal with unit-safe assignments, unit conversions, and the propagation of uncertainty. We have also presented how our robot models are simulated and how to perform some analysis on them.

Our current plans for future work include extending other robot languages with Quantities, such as the ones defined in [7], and also conducting further experiments with real robots in order to validate the precision and accuracy of our estimations with respect to the robots actual behavior. We also plan to extend our approach to represent the precision of a value as a function.

REFERENCES

[1] ISO/IEC 80000:2009. 2011. *Quantities and Units.* https://www.iso.org/standard/30669.html
[2] D. Alonso, C. Vicente-Chicote, F. Ortiz, J. Pastor, and B. Alvarez. 2010. V3CMM: A 3-view component meta-model for model-driven robotic software development. *Journal of Software Engineering for Robotics* 1, 1 (2010), 3–17.
[3] Colin Atkinson, Ralph Gerbig, Katharina Markert, Mariia Zrianina, Alexander Egurnov, and Fabian Kajzar. 2014. Towards a Deep, Domain Specific Modeling Framework for Robot Applications. In *Proc. of MORSE'14) (CEUR WS Proceedings).* 1–12. http://ceur-ws.org/Vol-1319/#morse14_paper_01
[4] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-Driven Software Engineering in Practice* (2 ed.). Morgan & Claypool Publishers.
[5] Davide Brugali and Luca Gherardi. 2016. HyperFlex: A Model Driven Toolchain for Designing and Configuring Software Control Systems for Autonomous Robots. In *Robot Operating System (ROS): The Complete Reference*, Vol. 1. 509–534.
[6] Fabian Büttner and Martin Gogolla. 2014. On OCL-based imperative languages. *Sci. Comput. Program.* 92 (2014), 162–178.
[7] Federico Ciccozzi, Davide Di Ruscio, Ivano Malavolta, and Patrizio Pelliccione. 2016. Adopting MDE for Specifying and Executing Civilian Missions of Mobile Multi-Robot Systems. *IEEE Access* 4 (2016), 6451–6466. https://doi.org/10.1109/ACCESS.2016.2613642
[8] Martin Gogolla, Fabian Büttner, and Mark Richters. 2007. USE: A UML-Based Specification Environment for Validating UML and OCL. *Sci. Comput. Program.* 69 (2007), 27–34.
[9] Martin Gogolla and Antonio Vallecillo. 2017. (An Example for) Formally Modeling Robot Behavior with UML and OCL. In *Proc. of the MORSE Workshop at STAF'17 (LNCS).* Springer, 1–15.
[10] Ralph Hodgson, Paul J. Keller, Jack Hodges, and Jack Spivak. 2014. *QUDT – Quantities, Units, Dimensions and Data Types Ontologies.* TopQuadrant, Inc. and NASA AMES Research Center. http://qudt.org/.
[11] John Edward Hutchinson, Jon Whittle, and Mark Rouncefield. 2014. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Sci. Comput. Program.* 89 (2014), 144–161.
[12] D. Isbell and D. Savage. 1999. *Mars Climate Orbiter Failure Board Releases Report, Numerous NASA Actions Underway in Response. NASA Press Release 99-134.* http://nssdc.gsfc.nasa.gov/planetary/text/mco_pr_19991110.txt
[13] JCGM 100:2008. 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.
[14] JCGM 101:2008. 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.
[15] Edward A. Lee. 2008. Cyber Physical Systems: Design Challenges. In *Proc. of ISORC'08.* IEEE, 363–369.
[16] George H. Lockwood. 1985. *Final report of the Board of Inquiry investigating the circumstances of an accident involving the Air Canada Boeing 767 aircraft C-GAUN that effected an emergency landing at Gimli, Manitoba on the 23rd day of July, 1983.* Government of Canada [Ottawa]. vi, 199 p. ; pages.
[17] Tanja Mayerhofer, Manuel Wimmer, Loli Burgueño, and Antonio Vallecillo. 2018. Specifying Quantities in Software Models. *Submitted* (2018). Technical report available from http://atenea.lcc.uma.es/index.php/Main_Page/Resources/DataUncertainty.
[18] Tanja Mayerhofer, Manuel Wimmer, and Antonio Vallecillo. 2016. 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, 118–131.
[19] Tanja Mayerhofer, Manuel Wimmer, and Antonio Vallecillo. 2016. *Computing with Quantities: the Java Project.* https://github.com/moliz/moliz.quantitytypes
[20] Object Management Group. 2011. *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Version 1.1.* OMG Document formal/2011-06-02.
[21] Object Management Group. 2016. *OMG Systems Modeling Language (SysML), version 1.4.* OMG Document formal/2016-01-05.
[22] Arunkumar Ramaswamy, Bruno Monsuez, and Adriana Tapus. 2014. Model-driven software development approaches in robotics research. In *Proc. of MISE'14.* ACM, 43–48.
[23] Davide Di Ruscio, Richard F. Paige, and Alfonso Pierantonio. 2014. Guest editorial to the special issue on Success Stories in Model Driven Engineering. *Sci. Comput. Program.* 89 (2014), 69–70.
[24] Douglas C. Schmidt. 2006. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer* 39, 2 (2006), 25–31. https://doi.org/10.1109/MC.2006.58
[25] Barry N. Taylor and Ambler Thompson. 2008. *The International System of Units (SI).* NIST. http://www.nist.gov/pml/pubs/sp811/.
[26] Antonio Vallecillo. 2010. On the Combination of Domain Specific Modeling Languages. In *Proc. of ECMFA'10 (LNCS)*, Vol. 6138. Springer, 305–320.
[27] Pamela Zave. 2010. *Lightweight modeling of network protocols: The case of Chord.* Technical Report. AT&T LaboratoriesâĂŤResearch.