Native Support for UML and OCL Primitive Datatypes Enriched with Uncertainty in USE

Tool Paper

Victor Ortiz[1], Loli Burgueño[2,3], Antonio Vallecillo[1], and Martin Gogolla[4]

[1] University of Málaga, Spain (vikour92@gmail.com, av@lcc.uma.es)
[2] Open University of Catalonia, Spain (lburguenoc@uoc.edu)
[3] Institut LIST, CEA, Université Paris-Saclay, France
[4] University of Bremen, Germany (gogolla@uni-bremen.de)

Abstract. Measurement uncertainty, an essential property of any physical system, is gaining attention in the modeling community due to the need to specify these kinds of system properties. Some notations already provide support for representing this information, but there is also the need to count on tools with native support for datatypes enriched with measurement uncertainty, able to manage it in a natural and transparent manner. This paper describes the extension that we have developed for the tool UML-based Specification Environment (USE) for supporting the primitive UML/OCL datatypes with uncertainty information.

Keywords: UML/OCL datatype · Measurement uncertainty · UML/OCL tool.

1 Introduction

Physical systems operating in the real world are always subject to uncertainty. In most of the cases, measurement uncertainty [10, 11] is a property which cannot be disregarded, i.e., we cannot assume that the measures taken by physical components as well as their state (position, temperature, etc.) are precise values. For instance, mechanical arms need to take into account the precision of their movements, the tolerance of their components and their decalibration over time. Hence, when modeling and simulating physical systems, we need to capture and operate with their intrinsic uncertainty. Due to this need to specify and represent these kinds of system properties, *uncertainty* is gaining attention in the modeling community.

In previous works [2, 3, 5], we defined an extension to the UML and OCL primitive datatypes to represent and operate with measurement uncertainty. In particular, we extended the types `Real`, `Integer`, `UnlimitedNatural`, `Boolean` and `String` and created their corresponding datatypes `UReal`, `UInteger`, `UUnlimitedNatural`, `UBoolean` and `UString` as well as their operations. All these operations were created/adapted to allow the interoperability of compatible

datatypes. For instance, the infix operator +, originally defined for the numeric types `UnlimitedNatural`, `Integer` and `Real`, was adapted to operate with the numeric uncertain types, too. This way, a user can add, for example, an `Integer` number and a `UReal` number, and the result is then a `UReal` number. Furthermore, we also defined an extension for the OCL operations on collections (i.e., `select`, `forAll`, etc.) to account for the new types of elements. Finally, we provided operations to convert values between the new and old datatypes.

In [5, 3], we provided a Java library with the implementation of the uncertain datatypes as well as their operations, and implemented a proof of concept in the tool USE [9]. In the current contribution, we have completed the implementation and explain details of the approach that we followed, and how we have built our architecture. Although we have developed our extension for the tool USE, this approach could also be used to extend other UML/OCL tools.

The rest of the paper is structured as follows. Sect. 2 explains the related concepts to uncertainty in UML/OCL datatypes and the tool USE, and Sect. 3 details the technical details of the implementation in USE. Finally, Sect. 4 presents our conclusions and outlines future work.

2 Background

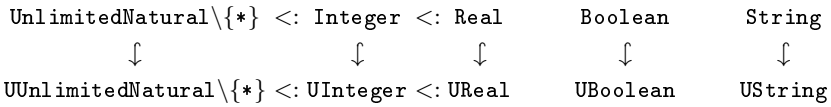
2.1 Uncertainty in UML/OCL Datatypes

In [3], to extend the OCL/UML primitive datatypes, we applied type embedding [4], which is one kind of *subtyping* [12]. We say that type A is a *subtype* of type B (denoted as $A <: B$) if all elements of A belong to B and operations of B, when applied to elements of A, behave the same as those of A [1]. If $A <: B$, then we also say that B is a supertype of A. For instance, Integer is a subtype of Real because every Integer number can be viewed as a Real number whose decimal part is zero. This concept is referred to as *injecting* `Integer` numbers into type `Real`. Furthermore, operations that are defined on the type `Real`, when applied to numbers of the type `Integer`, behave as those operations of type `Integer`.

Then, for extending a primitive OCL datatype `<T>`, we defined an embedding into a supertype `U<T>` that incorporates information about the uncertainty in the values of `<T>` and defines the operations for the extended type, which are also applicable to the base type.

The uncertainty information, both its meaning and the way in which it is propagated, varies depending on whether the values of the base type are numbers, Booleans or Strings. For example, UBoolean values are pairs (b, c), where b is a Boolean value and c is a Real number in the range $[0; 1]$ that represents the confidence in b; while a UReal number is a pair (x, u) where x is a Real number and u is its standard uncertainty as stated by the ISO Guide to the Expression of Uncertainty in Measurement (GUM) [10].

The subtyping ($<:$) and embedding ($\hookrightarrow$) relationships considering both the standard and the extended datatypes are as follows:



The example in Fig. 1 shows a model, taken from [6], in which we have applied our extended datatypes. The system represents an `Ozobot` robot (see 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` the head at a certain angle, and another one to `move` forward some distance. The mission determines the `target` position the robot is supposed to reach with the plan. Positions are given by `coordinates` in a planar surface. We are interested in analyzing a robot's behavior, and, in particular, whether the sequence of movements defined in its plan (which is given by a set of movements) fulfills the mission, i.e., it reaches the target position. For this, operation `coincide()` determines whether two coordinates are equal.

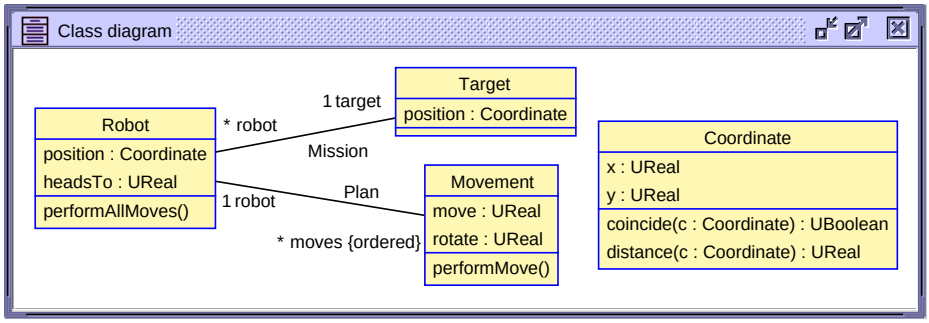


Fig. 1. Ozobot robots case study.

The code in Listing 1.1 shows the OCL expressions of the operations of class `Coordinate`. Note how the OCL expressions are the same as if dealing with the basic datatypes. However, now they become evaluated in the extended datatypes `UBoolean` and `UReal` and deal with the propagation of uncertainty, in a transparent way to the user.

Listing 1.1. Invariants for the Ozobot Robot case study.

```
1 context Coordinate::coincide(c:Coordinate):UBoolean =
2     self.x = c.x and self.y = c.y
3 context Coordinate::distance(c:Coordinate):UReal =
4     ((self.x-c.x)*(self.x-c.x) + (self.y-c.y)*(self.y-c.y)).sqrt()
```

Interestingly, the behavior of the system with uncertainty uncovers the fact that the precision of the robots movements has a significant impact on their plans, since it accumulates very soon and therefore the missions easily fail. Therefore the importance to have measurement uncertainty into account when modeling a system that deals with physical quantities; otherwise the models do not provide faithful descriptions of the systems they try to represent.

2.2 The UML-based Specification Environment (USE)

USE [9] is a modeling tool that allows the development and validation of UML models enriched with OCL expressions. The tool is open-source and distributed under a GNU General Public License.

USE provides mechanisms for the extension with new functionality by means of user-defined Ruby functions (placed in files under the folder `oclextensions`). All code available in this folder is loaded when USE is opened.

This extension mechanism enables the definition of new operations (and their behavior) for the predefined datatypes, but it does not allow the creation of new datatypes. In order to add new datatypes, the USE source code has to be forked and its native implementation has to be modified, which is what we did as a proof of concept in our previous work [3].

Internally, the USE tool uses ANTLR to define the grammars that it supports (for USE class models, OCL, SOIL [7], ShellCommands and other USE languages) and its source code is written in Java. In the following section, we explain how we have extended the grammars with the new uncertain datatypes and how we have developed their semantics in Java.

3 Extension of USE

Despite its graphical interface, the specification of models in USE is textual. Thus, the first step when adding new functionality to the tool is to update the grammar of its languages. In our case, we had to modify its OCL grammar which is located in its base file `org.tzi.use.parser.base.OCLBase.gpart`. Listing 1.2 shows the part of the grammar that we modified. In particular, we added the code in lines 7–18 and the new rule in lines 20–22.

Listing 1.2. Extended USE grammar.

```
1  literal returns [ASTExpression n] :
2        t='true'   { $n = new ASTBooleanLiteral(true); }
3      | f='false'  { $n = new ASTBooleanLiteral(false); }
4      | i=INT    { $n = new ASTIntegerLiteral($i); }
5      | r=REAL   { $n = new ASTRealLiteral($r); }
6      | s=STRING { $n = new ASTStringLiteral($s); }
7      | 'UString' LPAREN usve=additiveExpression COMMA
8          usue=additiveExpression RPAREN
9          { $n = new ASTUStringLiteral($usve.n,$usue.n); }
10     | 'UReal' LPAREN urve=additiveExpression COMMA
11         urue=additiveExpression RPAREN
12         { $n = new ASTURealLiteral($urve.n, $urue.n); }
13     | 'UBoolean' LPAREN ubve=conditionalImpliesExpression COMMA
14         ubpe=additiveExpression RPAREN
15         { $n = new ASTUBooleanLiteral($ubve.n, $ubpe.n); }
16     | 'UInteger' LPAREN uive=additiveExpression COMMA
17         uiue=additiveExpression RPAREN
18         { $n = new ASTUIntegerLiteral($uive.n, $uiue.n); }
19     ... ;
20 uncertaintyType returns [ASTType n] :
21     name=('UReal'|'UInteger'|'UBoolean'|'UString')
22     { $n = new ASTSimpleType($name); } ;
```

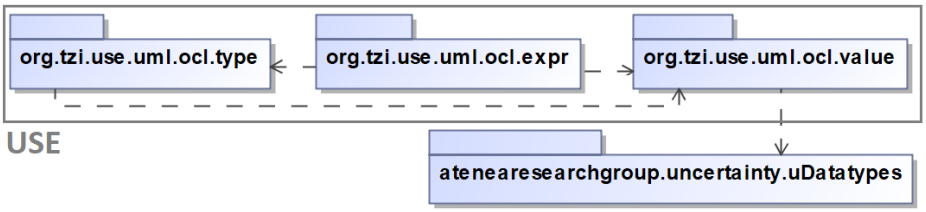


Fig. 2. USE packages for the implementation of OCL datatypes.

Then, using the ANTLR tools, the Java lexer, parser, tokens and listeners were automatically generated.

The implementation of datatypes in USE is done in a modular way. It distinguishes between *values* and *expressions*, both of which have a *type*. Accordingly, the Java source code that implements the datatypes is structured in three packages, namely, `org.tzi.use.uml.ocl.type`, `org.tzi.use.uml.ocl.value`, and `org.tzi.use.uml.ocl.expr`, which is shown in the top part of Fig. 2. The package at the bottom, called `atenearesearchgroup.uncertainty.uDatatypes` contains the Java library that we have developed with the behavior of the types `UUnlimitedNatural`, `UInteger`, `UReal`, `UBoolean` and `UString`, which is used by the other three packages.

The content of the extended USE package `org.tzi.use.ocl.type` is displayed in Fig. 3. The elements in white are the USE original classes, whilst the interfaces and the elements shaded in gray are the new classes that we have added. We created an abstract class called `UncertainType` from where all the uncertain types inherit. Each class defining a type contains the constructor of the type and the OCL methods for type checking. For instance, the class `UReal` contains its constructor and methods such as `isTypeOfUReal`, `isKindOfUReal`, `isKindOfNumber` and `isKindOfOclAny`. We have also modified the original USE classes to add the operations that allow the identification of the new datatypes as their supertypes of the original datatypes, when applicable. For instance, we have added to the class `RealType` the methods `isTypeOfUReal()` and `isKindOfUReal()`.

The package `org.tzi.use.ocl.value` contains a similar hierarchy of elements, as Fig. 4 shows. This package uses the Java library that implements the behavior of the uncertain datatypes [3].

The new classes that we have created in USE implement the Adapter design pattern [8]; they act as a wrapper for the classes in the library. This way, for instance, the new USE class `UIntegerValue` contains an attribute of the class `UInteger` from the library, and has to define only its comparison methods (`equals`, `compareTo`, etc.) to allow the interoperability with compatible values such as `Real` and `UReal`. Apart from the newly created classes, we also had to modify the abstract class `Value` to add a method with the signature `is<T>()` for each datatype, where `<T>` is the name of the new datatype (`isUReal()`, `isUInteger()`, etc.).

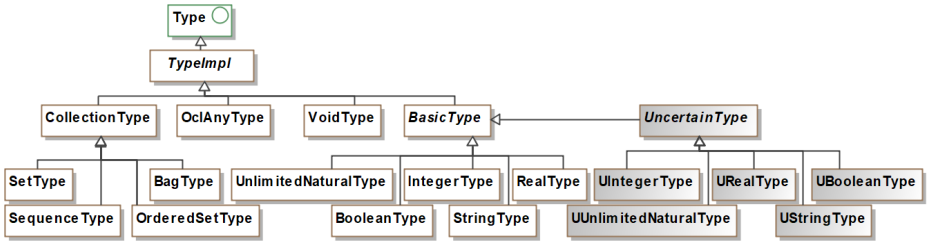


Fig. 3. Content of the extended `org.tzi.use.ocl.type` package.

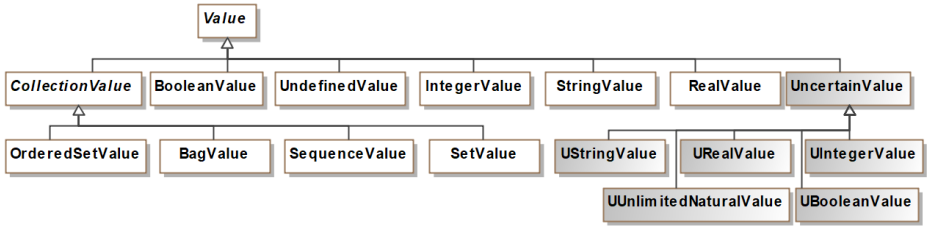


Fig. 4. Content of the extended `org.tzi.use.ocl.value` package.

Finally, once the types and value of the new datatypes were created, the last step was to make them available for their use inside OCL expressions and make their operations available, too. Regarding the latter, we decided to overload the existing operators such as "+". Then, a user can perform the operation "+" with any numeric parameter independently whether it is uncertain or not.

In USE, this is implemented in the package `org.tzi.use.ocl.expr`, which contains a class for each datatype. Then, for each uncertain datatype, we have created a new class with the following signature: `StandardOperations<T>` where `<T>` is the name of the new datatype—for instance, `StandardOperationsUBoolean`. Furthermore, we had to register all the operations in the USE class `OpGeneric`.

Regarding the operations on collections, we distinguish between those which return a boolean value (such as `forAll`, `exists` and `includes`) and those which return objects (such as `select`, `any` and `count`). The former operations are overloaded and only when they contain uncertain values, they return a `UBoolean`—in the rest of the cases, they stick to their original behavior and return a `Boolean` value. For the latter operations, we decided to create new versions of them with the prefix "u" for dealing with uncertain values—for instance, `uSelect`. Each operation is defined in a Java class that inherits from `ExpQuery`.

Finally, we followed a test-driven methodology when extending use. Thus, we extensively used the testing facilities that USE provides for unit and system testing. Following the same folder structure and style in which all the tests were developed, we included our tests under the folder `src/test`. We created new test classes for the new functionality that checked both its grammar and its behavior—, and executed them in batch using ANT.

4 Conclusions and Future Work

We have presented an extension of the tool USE to enable the application of native uncertain types for capturing measurement uncertainty, and we have shown how we structured and implemented the extension. Although the aim of this work was to extend the UML/OCL primitive datatypes that are supported by USE with our library of uncertain types, the same approach could be used to extend other UML/OCL tools, if they have an internal organization for supporting the datatypes. Our uncertainty extension has been already applied in several middle-sized models — see, e.g., http://bit.ly/UncertainContracts-Examples.

There are several lines of research that we plan to address soon. First, we would like to check and improve (if needed) the efficiency of the execution of operations and to extend the evaluation browser with aspects of uncertainty. We also plan to check how far other OCL evaluators can be extended in this way, and study the effort required to do so.

References

1. America, P.: Inheritance and subtyping in a parallel object-oriented language. In: Proc. of ECOOP'87. pp. 234–242. Springer (1987)
2. Bertoa, M.F., Moreno, N., Barquero, G., Burgueño, L., Troya, J., Vallecillo, A.: Expressing measurement uncertainty in OCL/UML datatypes. In: Proc. of ECMFA'18. LNCS, vol. 10890, pp. 46–62. Springer (2018)
3. Bertoa, M.F., Moreno, N., Burgueño, L., Vallecillo, A.: Incorporating measurement uncertainty into OCL/UML primitive datatypes. Software and System Modeling (2019). https://doi.org/10.1007/s10270-019-00741-0
4. Boute, R.T.: A heretical view on type embedding. SIGPLAN Not. **25**(1), 25–28 (Jan 1990)
5. Burgueño, L., Bertoa, M.F., Moreno, N., Vallecillo, A.: Expressing confidence in models and in model transformation elements. In: Proc. of MODELS'18. pp. 57–66. ACM (2018). https://doi.org/10.1145/3239372.3239394
6. Burgueño, L., Mayerhofer, T., Wimmer, M., Vallecillo, A.: Using physical quantities in robot software models. In: Proc. of RoSE'18. pp. 23–28. ACM (2018)
7. Büttner, F., Gogolla, M.: On OCL-based imperative languages. Sci. Comput. Program. **92**, 162–178 (2014)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman (1995)
9. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comp. Prog. **69**, 27–34 (2007)
10. JCGM 100:2008: Evaluation of measurement data – Guide to the expression of uncertainty in measurement (GUM). Joint Committee for Guides in Metrology (2008), http://www.bipm.org/utils/common/documents/jcgm/JCGM_100_2008_E.pdf
11. JCGM 200:2012: International Vocabulary of Metrology – Basic and general concepts and associated terms (VIM), 3rd edition. Joint Committee for Guides in Metrology (2012), http://www.bipm.org/utils/common/documents/jcgm/JCGM_200_2012.pdf
12. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (Nov 1994)