Chapter 1

Managing Quantities and Units of Measurement in Code Bases

Steve McKeever

Abstract

Quantities in engineering and the physical sciences are expressed as units of measurement (UoM). If a software system fails to maintain the algebraic attributes of a system's UoM information correctly when evaluating expressions then disastrous problems can arise. However, it is perhaps the more mundane unit mismatches and lack of interoperability that over time incurs a greater cost. Global and existential challenges, from infectious diseases to environmental breakdown, require high-quality data. Ensuring software systems support quantities explicitly is becoming less of a luxury and more of a necessity. While there are technical solutions that allow units of measurement to be specified at both the model and code level, a detailed assessment of their strengths and weaknesses has only recently been undertaken. This chapter provides both a formal introduction to managing quantities and a practical comparison of existing techniques so that software users can judge the robustness of their systems with regards to units of measurement.

Keywords: Units of measurement, Quantities, Dimension checking, Unit conversion, Libraries, Component based checking

1. Introduction

With ubiquitous digitalisation, and removal of humans in the loop, the need to faithfully represent and manipulate quantities in physical systems is ever increasing [39]. Popular programming languages allow developers to describe how to evaluate numeric expressions but not how to detect inappropriate actions on quantities. Consequently there have been infamous examples, such as the Mars Climate Orbiter [42], where units of measurement (UoM) conversion omissions led to catastrophic outcomes.

Humans have used local units of measurement since the days of early trade, enhanced over time to fulfil the accuracy and interoperable needs of science and technology. In the 19th century, James Clerk Maxwell [25] introduced the concept of a system of quantities with a corresponding system of units. This generalisation allowed scientists working with different measurement systems to communicate more easily, as unit names (such as inch or metre) are treated as numeric variables and can be interchanged through multiplication.

There are many ways in which manipulating physical quantities in a digital system can be made more robust. Thereby enabling the developer to depend on an automated validator, rather than trust, to ensure UoM are

handled correctly. The software engineering benefits of embracing quantity checking and automatic conversion support is beyond dispute. It is clear from the various known UoM failures that one size does not fill all. However, a general lack of awareness has ensured that developers often reinvent the wheel or forego any kind of checking. From a robustness perspective, the optimal solution would be to natively support quantities as this allows for efficient unit conversion and static checking. However none of the mainstream languages provide such support, nor is that level of rigour always required. A software library might seem to confer the desired functionality of adding a unit type to one's preferred language but they are inconvenient in practice, adding an extra layer to one's code base, and incur a performance cost. All popular programming languages have a multitude of freely available quantity libraries but we argue that these are best suited to applications in which UoM checking is required at run-time. Component based checking and black-box testing are two lightweight methods of providing a degree of robustness while sacrificing completeness. Component checking will only validate the interfaces between components, while testing will ensure known examples of methods dealing with quantities perform correctly. Neither are comprehensive.

Implicit in our presentation is that the various approaches have compromises. For users who need a degree of robustness when using quantities in their code bases, this discussion is very important. An approach that initially might seem suitable for a given project, could be too costly in terms of speed, too cumbersome in terms of use or too risky by creating an unnecessary dependency on some library. In Section 2 we discuss early attempts to incorporate UoM checking into programming languages, and recent attempts to incorporate quantities into modeling languages. In Section 3 we introduce a very simple assignment language to show how the various aspects of quantities are defined and validated. In Section 4 we discuss the various approaches to providing quantity support in programming environments, highlighting their strengths and weaknesses. We summarise the results of our comparative study in Section 5, providing suggestions for developers as to which method to choose depending on their requirements, along with enabling software users to access pertinent aspects of an implementation that claims UoM support.

2. Background

Dimensions are physical quantities that can be measured, while units are arbitrary labels that correspond to a given dimension to make it relative. For example a dimension is length, whereas a `metre` is a relative unit that describes length. Units of measure can be defined in the most generic form as either *base quantities* or *derived quantities*. The base quantities are the basic building blocks, and the derived quantities are built from these. For instance, the base quantity for time is `second` and that for length is `metre` in the International System of Units (SI), also known as the metric system. The SI system of measurement is based on seven base quantities for length, mass, time, electric current, temperature, quantity, and brightness [34]. Velocity, (`metre/second` or $\texttt{metre} \times \texttt{second}^{-1}$), is a derived quantity made from the two base quantities. Rather than representing UoM as a tree structure, a normal form exists which makes storage and comparison a lot easier. Any system of units can be derived from the base units as a product of powers of those base units: $\texttt{base}^{e_1} \times \texttt{base}^{e_2} \times \ldots \texttt{base}^{e_n}$, where the exponents $e_1, \ldots, e_n$ are rational numbers. Thus an SI unit can be represented as a 7-tuple $(e_1, \ldots, e_7)$

where e_i denotes the i-th base unit; or in our case e_1 denotes length, e_2 mass, e_3 time and so on.

Adding units to conventional programming languages originates in the 1970s [21] and early 80s with proposals to extend Fortran [15] and then Pascal [11]. However these efforts were heavily syntax based and required modifications to the underlying languages, reducing backwards compatibility and thus uptake. Ada's abstraction facilities, namely operator overloading and type parameterisation, allowed for a more versatile approach [18] to labelling variables with UoM features. With the appearance of practical object oriented programming languages, such as C++ and Java, developers began to implement UoM either by means of a class hierarchy of units and their derived forms, or via the Quantity pattern [13]. This has led to a veritable explosion in the number of UoM libraries available for all popular programming languages based on this pattern [30].

Software development typically begins at a more abstract level through diagrams and rules that focus on the conceptual model that is to be implemented. Extensions to the Unified Modeling Language (UML) have been proposed to support quantities. SysML, for instance, is defined as an extension of a subset of the UML to support systems engineering activities and has extensive support for quantities[1]. Unit checking and conversion can be undertaken before code is generated, either through a compilation workflow that leverages Object Constraint Language (OCL) expressions [26] or staged computation [2]. Unless the workflow has been created specifically, declaring quantities in a system specification language offers no guarantee that that the UoM information is supported in the eventual implementation. The aim of this chapter is to discuss the various ways in which quantity information can be transferred into software and errors detected either at compile-time or run-time. Motivated by a prior critique of UoM libraries and a survey of scientific coders [29], we assess various approaches based on their ease of use, execution speed, numeric accuracy, ease of integration and coverage of unit error detection capabilities.

We lack a definitive understanding of how frequently quantity errors occur in practice. Anecdotally we can infer that it is not negligible from experiments described in the literature. When applied to a repository of CellML models, a validation tool [8] found that 60% of the descriptions that were invalid had dimensionally inconsistent units. A spreadsheet checker [3] was applied to 22 published scientific spreadsheets and detected 3, nearly 14%, with errors. Ore [35] applied his lightweight C++ unit inconsistency tool to 213 open-source systems, finding inconsistencies in 11% of them. It must be noted that these figures are gleaned from post development studies and are not representative of a quantity adhering software discipline. Thus, it seems important to ensure UoM information existing in software models is supported in derived implementations.

3. Validating Quantities

Performing calculations in relation to quantities, dimensions and units is subtle and can easily lead to mistakes. We shall begin by looking at a very simple language of declarations and assignments so that we can untangle the various aspects involved in managing quantities correctly. A program will

[1] https://sysml.org

$$
\begin{array}{lcl}
\mathcal{E} \ : \ uexp \rightarrow (uv \rightarrow value) \rightarrow value & & \\
\mathcal{E}[\![uv]\!]_\rho & = & \rho\ uv \\
\mathcal{E}[\![uexp_1 + uexp_2]\!]_\rho & = & \mathcal{E}[\![uexp_1]\!]_\rho + \mathcal{E}[\![uexp_2]\!]_\rho \\
\mathcal{E}[\![r * uexp]\!]_\rho & = & r \times \mathcal{E}[\![uexp]\!]_\rho \\
\mathcal{E}[\![uexp_1 * uexp_2]\!]_\rho & = & \mathcal{E}[\![uexp_1]\!]_\rho \times \mathcal{E}[\![uexp_2]\!]_\rho
\end{array}
$$

Figure 1.
Rules for evaluating expressions.

consist of a sequence of UoM variable declarations, *uv*, followed by a sequence of assignment statements, *ustmt*. Unit arithmetic expressions, *uexp*, impose syntactic restrictions so that their soundness can be inferred using the algebra of quantities.

$$
\begin{array}{lcl}
ustmt & ::= & uv := uexp \\
uexp & ::= & uv \mid uexp_1 + uexp_2 \mid r * uexp \mid uexp_1 * uexp_2
\end{array}
$$

By creating a separate syntax for unit expressions we can distinguish between scalar values, such as *r*, and *unitless quantities* in which all the dimensions are zero, such as moisture content. Consider a simple program to calculate Newton's second law of motion:

```
begin
 f : float;
 m : float of 5.7;
 a : float of 3.2;
 ...
 f := m * a
end
```

We can use the evaluate function, $\mathcal{E}$ of Figure 1, with an environment consisting of values for `m` and `a` to calculate `f`:

$$
\begin{array}{lcl}
\mathcal{E}[\![\mathtt{m} * \mathtt{a}]\!]_{\{\mathtt{m}\mapsto 5.7, \mathtt{a}\mapsto 3.2\}} & = & \mathcal{E}[\![\mathtt{m}]\!]_{\{\mathtt{m}\mapsto 5.7, \mathtt{a}\mapsto 3.2\}} \times \mathcal{E}[\![\mathtt{a}]\!]_{\{\mathtt{m}\mapsto 5.7, \mathtt{a}\mapsto 3.2\}} \\
& = & (\{\mathtt{m} \mapsto 5.7, \mathtt{a} \mapsto 3.2\}\ \mathtt{m}) \times (\{\mathtt{m} \mapsto 5.7, \mathtt{a} \mapsto 3.2\}\ \mathtt{a}) \\
& = & 5.7 \times 3.2
\end{array}
$$

As is the case in nearly all programming languages, users have to assume that the mass (`m`) is given in kilograms, and the acceleration (`a`) is given in metres per second per second for the assignment to be correct. The reminder of this section explores the various aspects involved in handling quantities correctly, and how these aspects can be automated.

3.1 Dimensions

A dimensional analysis needs to ensure that (1) two physical quantities can only be equated if they have the same dimensions; (2) two physical quantities can only be added if they have the same dimensions (known as the *Principle of Dimensional Homogeneity*); (3) the dimensions of the multiplication of two quantities is given by the addition of the dimensions of the two quantities. If we only consider the three common dimensions of length, mass and time then

$$
\begin{array}{lcl}
\mathcal{DE} \ : \ uexp \rightarrow (uv \rightarrow dims) \rightarrow dims & & \\
\mathcal{DE}[\![uv]\!]_{\varrho} & = & \varrho \ uv \\
\mathcal{DE}[\![uexp_1 + uexp_2]\!]_{\varrho} & = & \mathcal{DE}[\![uexp_1]\!]_{\varrho} \ \hat{+} \ \mathcal{DE}[\![uexp_2]\!]_{\varrho} \\
\mathcal{DE}[\![r * uexp]\!]_{\varrho} & = & \mathcal{DE}[\![uexp]\!]_{\varrho} \\
\mathcal{DE}[\![uexp_1 * uexp_2]\!]_{\varrho} & = & \mathcal{DE}[\![uexp_1]\!]_{\varrho} \ \hat{\times} \ \mathcal{DE}[\![uexp_2]\!]_{\varrho}
\end{array}
$$

Figure 2.
Dimensional Analysis rules for expressions.

we can capture the rules for addition and multiplication.

$$
\begin{array}{lcl}
(l_1, m_1, t_1) \ \hat{+} \ (l_2, m_2, t_2) & = & (l_1, m_1, t_1), \text{ if } l_1 = l_2 \wedge m_1 = m_2 \wedge t_1 = t_2 \\
(l_1, m_1, t_1) \ \hat{\times} \ (l_2, m_2, t_2) & = & (l_1 + l_2, m_1 + m_2, t_1 + t_2)
\end{array}
$$

This allows us to rewrite the rules of Figure 1 by replacing the addition and multiplication operators to create a dimensional checker, shown in Figure 2. Scalar multiplication does not affect the dimensions of a quantity. The dimension of mass, `m`, is described as `(0,1,0)`, while acceleration is `length` $\times$ `time`$^{-2}$, or `(1,0,-2)` as a tuple. Our dimensional checker will compute with dimensions and attempt to ensure all assignments are correct. Consider our example program:

```
begin
 f : float of (1,1,-2);
 m : float of (0,1,0);
 a : float of (1,0,-2);
 ...
 f := m * a
end
```

Checking would proceed as follows:

$$
\begin{array}{l}
\mathcal{DE}[\![\mathtt{m} * \mathtt{a}]\!]_{\{\mathtt{m} \mapsto (0,1,0),\ \mathtt{a} \mapsto (1,0,-2)\}} \\
\quad = \ \mathcal{DE}[\![\mathtt{m}]\!]_{\{\mathtt{m} \mapsto (0,1,0),\ \mathtt{a} \mapsto (1,0,-2)\}} \ \hat{\times} \ \mathcal{DE}[\![\mathtt{a}]\!]_{\{\mathtt{m} \mapsto (0,1,0),\ \mathtt{a} \mapsto (1,0,-2)\}} \\
\quad = \ (\{\mathtt{m} \mapsto (0,1,0),\ \mathtt{a} \mapsto (1,0,-2)\}\ \mathtt{m}) \ \hat{\times} \ (\{\mathtt{m} \mapsto (0,1,0),\ \mathtt{a} \mapsto (1,0,-2)\}\ \mathtt{a}) \\
\quad = \ (0,1,0) \ \hat{\times} \ (1,0,-2) \\
\quad = \ (1,1,-2)
\end{array}
$$

and as this dimension matches `f` we know that the assignment is correct. Most UoM checkers adopt this approach, extending the checking into the statements and function calls of typical programming language constructs. For instance, all branches of conditionals and case statements must have the same dimensions, while comparison operators can only operate on quantities of the same dimension. If the dimensions of all variables are known at compile-time then this process can be undertaken before the program runs.

3.2 Kinds of Quantities

Two values that share the same UoM might not represent the same *kinds of quantities* (KOQ) [12]. For example, torque is a rotational force which causes an object to rotate about an axis while work is the result of a force acting over some distance. Surface tension can be described as newtons per meter

$$
\begin{array}{lcl}
\mathcal{NE} : uexp \rightarrow (uv \rightarrow \texttt{quantname}) \rightarrow \texttt{quantname} & & \\
\mathcal{NE}[\![uv]\!]_\tau & = & \tau\ uv \\
\mathcal{NE}[\![uexp_1 + uexp_2]\!]_\tau & = & \mathcal{NE}[\![uexp_1]\!]_\tau \diamond \mathcal{NE}[\![uexp_2]\!]_\tau \\
\mathcal{NE}[\![r * uexp]\!]_\tau & = & \mathcal{NE}[\![uexp]\!]_\tau \\
\mathcal{NE}[\![uexp_1 * uexp_2]\!]_\tau & = & \mathcal{NE}[\![uexp_1]\!]_\tau \vartriangle \mathcal{NE}[\![uexp_2]\!]_\tau
\end{array}
$$

Figure 3.
Named quantity rules for unit expression.

or kilogram per second squared, and even though they equate, they represent different quantities.

We have recently developed a simple set of rules for arithmetic and function calls that allow quantities to be named and handled *safely* [28]. This is not as straightforward as preserving the names of quantities throughout the program text. Multiplication will generate a new quantity so it is very likely that information is lost in intermediate stages of a calculation. Moreover, not all quantity variables in a program will have a name such as `Torque` or `Work`. Some might denote an entity such as `length` that could be in metres or yards, while another might be a variable used to store some temporary value. Neither of these need to be named. Using an algebraic data type, we define named quantities as:

```
type quantname = Named of string | Noname
```

We can now define the rules for adding and multiplying named quantities. In both cases we assume that the unit expression is dimensionally correct, our concern is to define how named quantities conduct themselves. The operator $\diamond$ takes two named quantities and states the conditions under which they can be summed: *two named quantities can be added together only if they represent the same entity*, if one quantity is named but the other is not then it is necessary for the result to be named, and if both are unnamed then the result will be too:

$$
\begin{array}{llllll}
\texttt{Named}\ n_1 & \diamond & \texttt{Named}\ n_2 & = & \texttt{Named}\ n_1, & \text{if } n_1 = n_2 \\
\texttt{Named}\ n & \diamond & \texttt{Noname} & = & \texttt{Named}\ n & \\
\texttt{Noname} & \diamond & \texttt{Named}\ n & = & \texttt{Named}\ n & \\
\texttt{Noname} & \diamond & \texttt{Noname} & = & \texttt{Noname} &
\end{array}
$$

Our comparison rules cast upwards from `Noname` to `Named`, so as to assume a named quantity whenever possible. This is required to ensure named quantities behave correctly. For multiplication the rules are simpler. The operator $\vartriangle$ takes in two named quantities and defines how they behave over the multiplication operator. As *multiplication sums the dimensions of the two operands, the value will be different to either and so the result will always be* `Noname`. The rule is thus: `_ △ _ = Noname`. The rules for named quantity analysis of expressions are given in Figure 3.

Multiplication will generate a new quantity so it is very likely that information is lost in intermediate stages of a calculation. We propose that functions, whose return KOQ are known, are used to regain information when calculations use multiplication. In this manner a discipline of programming with quantities is suggested Nonetheless, we can exemplify how our approach behaves on a simple incorrect assignment where we try to add a value of torque to one of work:

```
begin
```

```
 sumt : float of Named Torque;
 t : float of Named Torque;
 w : float of Named of Work;
 ...
 sumt := t + w
end
```

Checking would start as follows:

$$
\begin{aligned}
&\mathcal{NE}[\![\texttt{t + w}]\!]_{\{\texttt{t}\mapsto\texttt{Named Torque, w}\mapsto\texttt{Named Work}\}}\\
&\quad = \mathcal{NE}[\![\texttt{t}]\!]_{\{\texttt{t}\mapsto\texttt{Named Torque, w}\mapsto\texttt{Named Work}\}} \diamond \mathcal{NE}[\![\texttt{w}]\!]_{\{\texttt{t}\mapsto\texttt{Named Torque, w}\mapsto\texttt{Named Work}\}}\\
&\quad = (\{\texttt{t} \mapsto \texttt{Named Torque}, \texttt{t} \mapsto \texttt{Named Work})\}\ \texttt{t})\\
&\qquad \diamond (\{\texttt{t} \mapsto \texttt{Named Torque}, \texttt{w} \mapsto \texttt{Named Work}\}\ \texttt{w})\\
&\quad = \texttt{Named Torque} \diamond \texttt{Named Work}
\end{aligned}
$$

and as we cannot proceed, we deduce that the assignment is unsafe. None of the current methodologies for managing quantities perform this kind of quantity checking.

3.3 Systems of Units

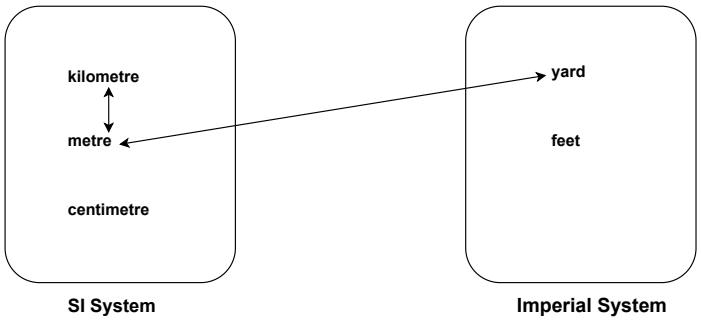

Figure 4.
Conversion within and in-between Systems of Units.

Dimension analysis would be sufficient if only one unit system, such as the SI system, was required. In such cases the base units of metre, gram and second could be implicit in implementations. There are, however, two further complications. A system of units provides a set of base units but they can be extended with prefixes. For instance, the SI system allows lengths to be described in terms of the base unit metre and prefixes, such as kilo- and centi-. While the Imperial system has the base unit feet on which an inch and a yard are constructed. Consequently we need to consider normalising values within a system of units and allow dimensionally faithful expressions to evaluate correctly with different units of measurement, as shown in Figure 4.

3.3.1 Normalising within a System of Units

We can extend our tuple for dimensions to include prefixes. They are used to avoid very large or very small numeric values. Speed is typically shown in kilometres per hour, which we could represent in our 3-tuple as `((kilometre,1),(gram,0),(hour,-1))`. This approach simplifies normalisation. A kilometre can be converted into metres by multiplying the

value by a 1000, and an hour is 3600 seconds. Consider the dimensionally correct assignment below where the mass, `m`, is given in grams:

```
begin
 f : float of ((metre,1),(kilogram,1),(second,-2));
 m : float of ((metre,0),(gram,1),(second,0));
 a : float of ((metre,1),(gram,0),(second,-2));
 ...
 f := m * a
end
```

A UoM analysis would reveal that force, `f`, is expected in Newtons so the value for `m` needs to be divided by 1000, `f:= (m/1000)*a`. It is straightforward to normalise within a given system of units by aligning the prefixes. However, we also need to take the exponent into account. For example, the density of iron is $7.86\,g/cm^3$, and can be converted into kg/m^3 as follows:

$$\begin{aligned} & 7.86 \times (C \text{ (gram, kilogram)}) \times (C \text{ (centimetre, metre)})^{-3} \\ &= 7.86 \times 0.001 \times 0.01^{-3} \\ &= 7860\ kg/m^3 \end{aligned}$$

where we assume the existence of a function C that takes in two prefixes and returns the conversion factor.

3.3.2 Converting between Two Systems of Units

A second issue is that of converting in-between two systems of units. This can be achieved by extending the function C to convert between known systems, such as from yards to metres and visa-versa. Consider a similar example to the Mars climate orbiter error, here we want to calculate momentum, where `p` is mass times velocity, in Newton seconds, or $m \cdot kg \cdot s^{-1}$. However our mass is in pounds:

```
begin
 p : float of ((metre,1),(kilogram,1),(second,-1));
 m : float of ((metre,0),(pound,1),(second,0));
 v : float of ((metre,1),(gram,0),(second,-1));
 ...
 p := m * v
end
```

The function conversion, C (pound, kilogram), yields 0.45 so we need to modify the assignment so that the result will be correct, namely `p := 0.45*m*v`.

A naive unit conversion algorithm will convert the right hand side's units to the left hand side's for each sub-expression when performing addition or multiplication, but this is rarely the most efficient. The algorithm described by Cooper [8] is applied at compile-time. It choose the least number of conversions by generating all possible valid UoM conversions from a given expression, and selecting the one with the fewest conversions. Finally, many non-SI units continue to be used in the scientific, technical, and commercial literature. Some units are deeply rooted in history and culture. We use the term week rather than 168 hours. These also need to be supported through conversion functions.

3.4 Levels of Measurement

A further aspect is that of the operations that might be applicable to a given quantity. To a physicist or applied mathematician, it is taken for granted that a quantity is used in the same manner as a unit-independent value, and that all arithmetic and comparative operators can be applied to it. However this is often not the case. For instance, what operations apply to a person's IQ? It does not makes sense to multiply scales of intelligence or personality traits.

Stevens [43] identified four categories of scale that places limits on the type of measurement that can be used to construct valid terms:

1. *Nominal Scales* represents the most unrestricted assignment of numerals. The only operations that applies to values in a nominal scale is that of *equality* and *inequality*, essentially numerals are only used as labels.

2. *Ordinal Scales* permits rank ordering. The classic example of an ordinal scale is the scale of hardness of minerals. Most of the scales used by psychologists are ordinal scales. Operations such as *greater-than* and *less-than* are also applicable to ordinal scales. In the strictest sense, statistical operations involving means and standard deviations should not to be used with these scales as they imply a knowledge of something more than the rank order of data. This is due to the successive intervals on the scale being unequal in size but the 'illegal' application of statistical functions is often very useful, such as the average IQ of a certain population.

3. *Interval Scales* are what we would consider quantitative and allow *addition* and *subtraction*, so all the usual statistical measures are applicable. The zero point on an interval scale is a matter of convention. Centigrade and Fahrenheit both represent volumes of expansion, with an arbitrary zero for each scale, such that a numerical value on one scale can be converted into a value on the other.

4. *Ratio Scales* are most commonly encountered in physics and include the previous three relations, along with *multiplication* and *division*. An absolute zero is always implied.

Currently no system employs levels of measurement checking, such that only meaningful operations are applied to certain quantities. We suspect that this is due to even less awareness of scale levels and an assumption that quantity checking is only necessary for ratio scales. Hall [16] has developed a UML class diagram that captures the inheritance relationship between the four levels, enabling operations to be restricted to the most general of a pair of operands. As discussed with the ordinal scale, there is a pragmatic context that needs to be considered when using levels of measurement.

3.5 Summary of Quantity Checking

Through the use of an illustrative expression language we have shown how to ensure programs correctly manipulate the dimensions, the named quantities and the UoM of values. In doing so we have raised two important issues relating to the implementation of quantities: at what point checking and conversions can occur, and how extensive the coverage will be. If all UoM

variables are annotated, or their annotations inferred, then both dimension checking and unit conversions can be undertaken by the compiler. This means that programs with UoM errors will be detected early, before the system is put in place, creating a strongly UoM typed language. Moreover, the code can be optimised so as to have the least number of conversions, reducing rounding errors, and increasing the accuracy of its calculations. The technique can be extended to include assertions on allowable unit conversions. If UoM are only known at run-time, or their design is embedded within the host language, then dimension checking and unit conversions will be undertaken at run-time. Programs will still manipulate UoM correctly but with a performance penalty and errors will only be detected once running. One must also consider the annotation burden, [38] found subjects choose a correct UoM annotation only 51% of the time and take an average of 136 seconds to make a single correct annotation.

4. Implementing Quantities

Many constructs in Software Models can be directly translated into code. A UML class diagram can be used to build the class structure of an object oriented implementation. The situation for UoM annotations is more complex. UoM values are neither primitive nor reference types in modern object oriented terminology. As described in Section 3, they require an advanced checker to ensure variables and method calls are handled soundly by the compiler.

In this Section we shall look at the four practical methods that support unit checking of code basis. All implementation options are affected by the following three concerns [29]: *lack of awareness*, *cumbersome implementations* and *lack of support* from the given software eco-system. Software rarely lives in a vacuum so even if it has been designed and developed with one of these methods, associated components are unlikely to support UoM, such as legacy libraries, databases and spreadsheets. Nonetheless as sources of data are migrated to quantity aware formats, the software must be able to follow suit.

4.1 Native Language Support

Adding unit checking to conventional imperative, object-oriented and even functional languages using syntactic sugaring is beyond the algorithmic scope of their underlying type checkers. The pioneering foundational work of Wand and O'Keefe [45] demonstrated how the simply-typed lambda calculus could be extended with dimensions. Milner's polymorphic type inference algorithm [32] is symbolic, so in order to include UoM variable resolution one has to provide an equational solver based on the theory of Abelian groups [23]. A programming language with native support will include additional syntax for UoM and an enhanced static analyser that calculates or infers the validity of annotations. When UoM annotations are validated prior to compilation, errors can be detected early and generated arithmetic code can minimise the number of conversions, mitigating round-off errors.

The only language in the 20 most popular programming languages [44] that supports units of measurement is Apple's Swift language [4]. Microsoft's F# [31] is a general purpose functional language with a full implementation of UoM, including unit variables that the static checker will seek to determine at compile-time [23]. This property permits valid UoM programmes to be written in which not all the quantity variables are annotated, thereby reducing the

annotation burden and allowing greater reusability. However, large software projects rarely use either Swift or F#.

4.1.1 C++ Boost Library:

C++ is still very popular [44] and has a de facto UoM extension that exploits the template meta-programming feature[2]. Consequently BoostUnits is more than just a library as it supports a staged computation model, similar to MixGen [2], which has the benefit of providing a language extension while still supporting backwards compatibility. C++ is quite unique in terms of being a popular programming language that supports this adaptable compilation strategy. Dimensional analysis is treated as a generic compile-time meta-programming problem, and delivers features and performance comparable to native language support. As no run-time penalty is incurred, BoostUnits supports UoM checking in performance-critical code. However, the survey [40] found both usability and accuracy issues with the use of this library.

4.1.2 Unit of Measurement Validators:

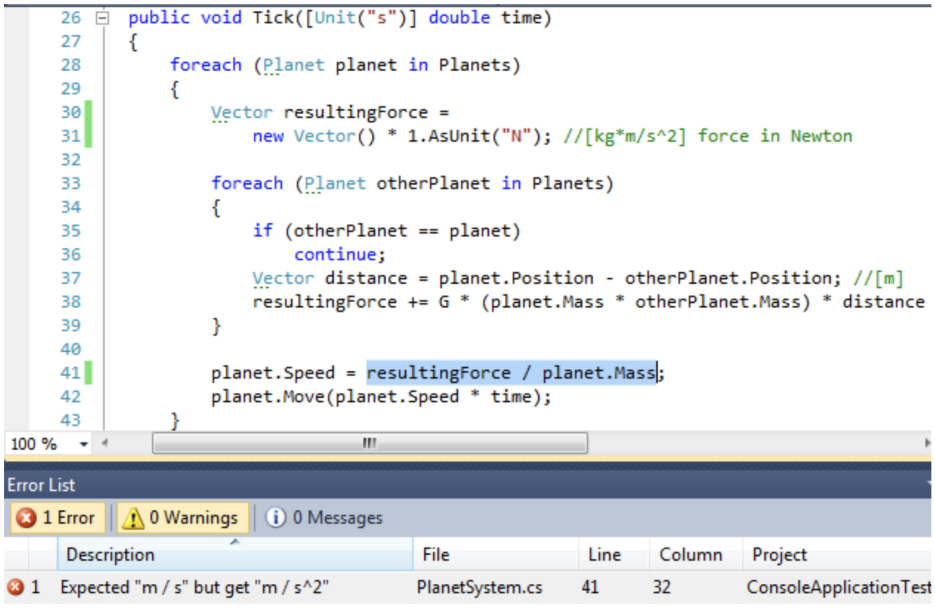

Figure 5.
Example of Unit of Measurement Validator for C#.

Creating a new compiler feature for an existing language is contentious, non-trivial and likely to become outmoded. An alternative static approach is to define UoM through comments or attributes and to build a tool that attempts to perform as much scrutiny as possible. Such a validator checks at compile-time for unit violations without adding a new syntax or changing the run-time behaviour of the code. Figure 5 shows an example of the Unit of Measurement Validator for C# [10]. The Osprey [20] system is a C front end

[2] https://github.com/boostorg/units

that automatically checks for all potential quantity errors. UoM are annotated with a `$` and are modelled as types, reducing dimensional analysis to type checking, with Gaussian elimination to resolve unspecified UoM variable exponents. Simiarly [19] allows one to express relationships between the units of function parameters and return values. Ensuring the validity of unit conversions can be specified. PUnits [47] is a Java front end, or pluggable type system, that has many additional features. It can be used in three modes: checking the correctness of a program, solving UoM type variables, and annotating a program with units. This last feature is the most novel and allows human inspection which is useful since having a valid typing does not guarantee that the inferred specification expresses design intent, and allows KOQs to be expressed. These approaches are lightweight and scalable but they need to be supported for users to feel that they are credible. Alas few of these tools have lifespans outside of their original research project.

4.1.3 Domain-Specific Languages (DSLs) with Quantities:

DSLs are languages specialised to a particular application domain. They are often written in a mark-up language such as XML or JSON which facilitates their use with general purpose languages as they can be easily parsed. They might have originally been designed for the purpose of curation, such as CellML and SMBL where the intention was to build biological repositories of computational models. A model will include the constants, variables and equations denoting a particular biological system. Thus, translating them into a programming language with the aid of a differential equation solver means that they can be readily simulated [14]. If the DSL contains UoM declarations then separately analysing source files can be undertaken before they are uploaded to a repository and translated into the run-time system [8]. Quantity checking coverage might be limited to the application domain but one can easily make the argument that this is where the vulnerabilities would lie.

4.2 Static or Dynamic Library Support

Adding a feature to an existing programming language is usually undertaken by developing a library which implements the desired behaviour. It is written in terms of the language, using advanced abstraction methods such as classes and generics, and delivers a well-defined interface. There are many UoM libraries for all modern programming languages [6]. The basic idea behind dimensional analysis, as shown in Section 3.1, is relatively straightforward and familiar to scientific coders. However, *"making a physical quantity library is easy but making a good one is hard"* [5]. The reasons behind this lie in the subtleties of implementation: providing non-standard UoM, offering many new operators, supporting conversions, creating helpful error messages, while being efficient. These aspects are far harder to code and maintain than a faithful implementation of the Quantity pattern [13, 24].

Experience has shown that quantity libraries add an extra layer to an existing application through boilerplate code that is rarely idiomatic to the host language. The Quantity pattern ensures that values are hard-wired to have a single floating point representation, requiring further conversions to other internal formats and accuracy issues. Poor error messages are also a frequent complaint. Certain modern languages, such as Ruby, provide a special syntax for adding features to the language which exploit duck typing, enabling lightweight libraries to be built. The main conclusion is that for adoption to

occur, UoM must be included in such a way that they are almost as easy to use as standard arithmetic types [29].

The standard technique for implementing the Quantity pattern is through exploiting overriding, making UoM checking a run-time activity. However UoM can be implemented through static overloading, or Java generic instantiations, ensuring compile-time checking. An example of both styles is shown in [29]. Merging the two techniques would double the amount of notation required and significantly add to the burden of adoption. Languages such as Python or Ruby are dynamically typed by definition, so quantity checking will occur at run-time regardless of how UoM are defined. Run-time support is a key reason why there are so many libraries for these two languages when typically one would assume quantities require high-performance executables. From a user survey of UoM libraries: *"In our product line, our users may very well have one file whose units are 'kg · m^3', another whose units are 'g_cc' and a third whose units are 'degrees Celsius'. We therefore need to be able to operate on units at run-time, not compile-time"* [40].

An additional limitation of libraries versus a native language or a pluggable type solution is that variables of a Quantity class can be reassigned at run-time. The 7-tuple that represents dimensions can be modified such that a kilometre could become a newton. Dimensional homogeneity and conversion errors can be caught by a library implementation but to avoid such programming style errors necessitates discipline. A dimensionally aware static checker ensures that all instances of a quantity declaration have the same dimensions. Errors caused by this violation were found to account for 75% of inconsistencies in the study of 5.9M lines of code [37].

Implementing UoM through a data type or a class requires values to be boxed at run-time, incurring both speed and memory penalties. Native language solutions can perform the checking at compile-time so that generated executables contain no further UoM annotations and values are represented by primitive types. For scientific applications that perform many calculations, such as matrix multiplications, this unnecessary performance overhead is unacceptable. A UoM library might seem attractive and undemanding initially, but the subtle burden that they inflict will often increase the complexity of a project.

4.3 Component or Interface Description Support

Encapsulating implementation details, interfaces are a collection of the externally visible entities and function signatures of a component. They are used by the compiler to ensure access is handled correctly. Libraries, native language support and pluggable type systems usually require all quantity variable and function declarations to have UoM annotations, while component or interface based approaches only require certain functions to be annotated. This drastically reduces the annotation burden, supports legacy code but at the expense of robustness.

A component based approach seeks to add UoM information to the interface in order to enforce unit consistency when composing components and thereby reduce dimensional mismatch errors. There is some anecdotal evidence in the many quotes of [40] to support this approach. Damevski [9] hypothesises that UoM libraries are too restricting by requiring complete coverage, incurring an annotation or migration burden. His technique performs dimensional analysis on component interfaces at run-time, and if the calling parameters

are compatible with the arguments dimensions then unit conversions occur. Consider the C++ class Earth [9]:

```
class Earth {
 void setCircumference(in Metre circumference);
 Metre getCircumference();
}
```

It assigns and queries the earth's circumference using `Metre` internally but can be called with `Kilometre` and the return value bound to a variable of, say, type `Mile`. Unlike libraries, within the class `Earth` no further annotations are required, nor will internal declarations be checked. This is a dynamic component based approach, units are converted at run-time.

Another lightweight methodology was presented by Ore [35] which performs an initial pass to build a table mapping attributes in C++ shared libraries to UoM. The shared libraries have been specifically enhanced to include UoM annotations. The table is used to disseminate UoM information into a source program and detect errors at compile-time. The algorithm successfully exploits dimensional analysis rules for arithmetic operators within components [36]. This is an example of a static component based approach, and through this process of static UoM propagation checking, manages to perform a greater coverage than Damevski's run-time method.

A component based discipline means that the consequences of local unit mistakes are underestimated. The analysis of local assignment expressions will not occur in Damevski's scheme and will be limited in Ore's. On the other hand, checking at the component level allows diverse teams to collaborate even if their domain specific environments or choice of quantity systems were, to some extent, dissimilar. More importantly, either a static or dynamic component implementation would have been effective at correcting the Mars Climate Orbiter error.

4.4 Black-Box Testing

The last method that we will look at for checking quantities are managed correctly is known as black box automated testing. This method is usually applied to detect incorrect or missing methods, initialisation errors, interface errors and errors in data structures. By extrapolating use cases from the requirements specification to create unit tests, which are then systematically applied to the program, errors can be discovered before the code is put into operation. In the case of UoM tests, the resulting testing will not be comprehensive. Unit testing will focus on the correct initialisation of variables, the UoM correctness of assignments and method calls. This approach is also considered lightweight as no annotations are required to the program, however creating sufficient unit tests by hand is tedious. Efforts to generate unit tests from UML descriptions automatically [1, 7, 17, 33], either through behavioural diagrams or with rule based approaches, are seen to be costly and non-trivial in practice [22].

Modern agile software development practices rely heavily on manually developing tests for enabling refactoring but also to support test driven development (TTD). Unit testing for UoM errors does not require any extra tool support, and will not alter any other parts of the system as shown in Figure 6, where we show the testing required for a simple UoM based addition function. If `t` is true then both arguments are in kilometres, alternatively the second argument is in miles and converted accordingly. The two test cases capture this intention.

```
class Distance {
 public double add_km(boolean t,
              double a, double b) {
   return ((t)? a+b : a+(b*1.609));}}
...
public class DistanceTest {
 public void test_add_km() {
   Distance d = new Distance();
   assert(d.add_km(true,10.0,10.0)==20.0);
   assert(d.add_km(false,10.0,10.0)==26.09);}}
```

Figure 6.
Java code and JUnit test case for simple addition of two kilometres, or kilometre and mile distances [27].

Many developers would argue that time spent becoming familiar with a UoM library, and updating their programs accordingly, might be better spend writing unit tests: *"I could use the same time to write tests and that would really find and prevent errors and at the same time not introduce a crazy complicated library every other developer in my team would have to deal with."* [40]. However, the UoM knowledge will be localised to each particular unit so the slight implementation cost comes at the expense of potentially average checking.

5. Conclusions

Table 1.
Contrasting alternative methods of Implementing Units of Measurement in Software Projects, extended from [27].

Technique	Programming Ease of Use	Execution Speed	Numeric Accuracy	Ease of Integration	Unit Error Detection
Native Support	High	Very High	Excellent	Low	Very High
Validator	High	High	Very Good	Average	Average
DSL	Low	High	Very Good	Low	Average
Static Library	Low	Average	Good	Low	High
Dynamic Library	Average	Low	Good	Low	High
Static Component	High	High	Very Good	High	Average
Dynamic Component	High	Average	Good	High	Low
Black Box Testing	Average	High	Very Good	Very High	Average

Recent initiatives such as the FAIR data principles [46] emphasis machine-actionability of scientific data. With greater interoperability, industrial use of computational simulations and penetration of digitalisation through cyber-physical systems; there is an urgency to faithfully represent key properties of physical systems in code bases [39, 41].

We have endeavoured to provide the necessary background for software users to choose the most appropriate method for enabling quantity checking in code bases. Alas native language support is not available for popular programming languages. This situation is unlikely to change as it would require new language definitions and expensive compiler rewrites, with an important criteria of ensuring backwards compatibility with existing code. Validators solve some of these issues but require assurances that tool support will be maintained. DSL

checkers are very effective for their given domain but lack generality. It is clear that even the best libraries currently cause significant performance issues while not being relevant for most developers. However some of the dynamic libraries include a lightweight syntax that makes UoM annotations significantly easier. A strength of component based techniques is that they can be undertaken at either compile-time or run-time with little overhead. They cede UoM checking completeness for a low annotation burden and ease of adoption. Approaches based on manual black-box testing frameworks offer many of the benefits of static component based techniques without requiring extra syntax. The drawback is that the UoM information will be implanted within the unit tests and not as a form of documentation within programs.

Annotating quantities in code bases is costly for developers [38] but relatively durable to software evolution. Refactoring does not change the external behaviour of the software, it will rarely require quantity annotation modifications unless there are changes to the core data structures. Techniques that manage UoM at compile-time, such as native language support or static lightweight solutions, allow unit conversions to be undertaken before code is created and values to be represented in unboxed form, thus resulting in better accuracy and efficiency.

The pros and cons of UoM techniques are listed in Table 1 using features that are relevant to users. Native language or pluggable type based techniques offer many benefits, such as an equational UoM checker that is capable of resolving UoM type variables, and unit conversion optimisation to improve run-time behaviour. Static component and black box based testing provide some of the benefits of static UoM libraries with less coverage but greater versatility. This is in tune with contemporary software development methods that favours lightweight techniques which integrate into existing digital platforms.

Software users who require a degree of robustness have had little say over how much validation was undertaken on their code to ensure UoM are handled correctly. This chapter aims to inform users and developers of the various options that exist to check that quantities are handled properly, along with the implications of these choices with regards to their software eco-system. The needs of a reactive and responsive on-line application, with assorted UoM input, are very different to a fully quantity specified stand-alone safety critical application. Nonetheless software-intensive systems are prevalent in our daily lives, with complex functionality and strong interconnection. The need to ensure quantity values behave correctly are greater than ever before.

Author details

Steve McKeever
Department of Informatics and Media, Uppsala University, Sweden

References

[1] S Ali, H Hemmati, NE Holt, E Arisholm, and LC Briand. Model transformations as a strategy to automate model-based testing-a tool and industrial case studies. *Simula Research Laboratory, Technical Report (2010-01)*, pages 1–28, 2010.

[2] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy L. Steele, Jr. Object-oriented units of measurement. In *Proceedings of Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 384–403, NY, USA, 2004. ACM.

[3] Tudor Antoniu, Paul A. Steckler, Shriram Krishnamurthi, Erich Neuwirth, and Matthias Felleisen. Validating the unit correctness of spreadsheet programs. In *Proceedings of Software Engineering*, ICSE '04, pages 439–448, Washington, DC, USA, 2004. IEEE Computer Society.

[4] Apple. Swift open source, 2022. Last Accessed on May 19th 2022.

[5] Trevor Bekolay. A comprehensive look at representing physical quantities in python. In *Scientific Computing with Python*, 2013.

[6] Oscar Bennich-Björkman and Steve McKeever. The next 700 Unit of Measurement Checkers. In *Proceedings of Software Language Engineering*, SLE 2018, page 121–132, NY, USA, 2018. Association for Computing Machinery.

[7] Alessandra Cavarra, Charles Crichton, Jim Davies, Alan Hartman, Thierry Jeron, and Laurent Mounier. Using UML for Automatic test Generation. *Proceedings of ISSTA*, 01 2002.

[8] Jonathan Cooper and Steve McKeever. A Model-Driven Approach to Automatic Conversion of Physical Units. *Software: Practice and Experience*, 38(4):337–359, 2008.

[9] Kostadin Damevski. Expressing measurement units in interfaces for scientific component software. In *Proceedings of Component-Based High Performance Computing*, CBHPC '09, pages 13:1–13:8, NY, USA, 2009. ACM.

[10] Dieterichs Henning. Units of Measurement Validator for C#. Last Accessed on May 19th 2022.

[11] A. Dreiheller, B. Mohr, and M. Moerschbacher. Programming pascal with physical units. *SIGPLAN Notes*, 21(12):114–123, December 1986.

[12] Marcus Foster and Sean Tregeagle. Physical-type correctness in scientific python, 2018.

[13] Martin Fowler. *Analysis Patterns: Reusable Objects Models*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[14] Alan Garny, David Nickerson, Jonathan Cooper, Rodrigo Weber dos Santos, Andrew Miller, Steve McKeever, Poul Nielsen, and Peter Hunter. Cellml and associated tools and techniques. *Philosophical Transactions of the Royal Society, A: Mathematical, Physical and Engineering Sciences*, 366, 2008.

[15] Narain Gehani. Units of measure as a data attribute. *Computer Languages*, 2(3):93 – 111, 1977.

[16] Blair Hall. The problem with 'dimensionless quantities'. In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development - MODELSWARD,*, pages 116–125, Portugal, 2022. INSTICC, SciTePress.

[17] J. Hartmann, M. Vieira, H. Foster, and A. Ruder. A UML-based approach to system testing. *Innovations in Systems and Software Engineering*, 1:12–24, 2005.

[18] Paul N. Hilfinger. An Ada Package for Dimensional Analysis. *ACM Trans. Program. Lang. Syst.*, 10(2):189–203, April 1988.

[19] Hills M, Chen Feng, and Roşu Grigore. A Rewriting Logic Approach to Static Checking of Units of Measurement in C. *Electronic Notes in Theoretical Computer Science*, 290:51–67, 2012.

[20] Lingxiao Jiang and Zhendong Su. Osprey: A Practical Type System for Validating Dimensional Unit Correctness of C Programs. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 262–271, New York, NY, USA, 2006. ACM.

[21] Michael Karr and David B. Loveman. Incorporation of Units into Programming Languages. *Commun. ACM*, 21(5):385–391, May 1978.

[22] Jussi Kasurinen, Ossi Taipale, and Kari Smolander. Software Test Automation in Practice: Empirical Observations. *Advances in Software Engineering*, 2010, 01 2010.

[23] Andrew Kennedy. Dimension Types. In Donald Sannella, editor, *Programming Languages and Systems—ESOP'94*, volume 788, pages 348–362, Edinburgh, U.K., 11–13 1994. Springer.

[24] Michael Krisper, Johannes Iber, Tobias Rauter, and Christian Kreiner. Physical Quantity: Towards a Pattern Language for Quantities and Units in Physical Calculations. In *Proceedings of Pattern Languages of Programs*, EuroPLoP '17, pages 9:1–9:20, NY, USA, 2017. ACM.

[25] James Clerk Maxwell. *A treatise on electricity and magnetism [microform] / by James Clerk Maxwell*. Clarendon Press Oxford, 1873.

[26] Tanja Mayerhofer, Manuel Wimmer, and Antonio Vallecillo. Adding uncertainty and units to quantity types in software models. In *Software Language Engineering*, SLE 2016, pages 118–131, NY, USA, 2016. ACM.

[27] Steve McKeever. From Quantities in Software Models to Implementation. In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development - MODELSWARD*, pages 199–206, Portugal, 2021. INSTICC, SciTePres.

[28] Steve McKeever. Discerning quantities from units of measurement. In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development - MODELSWARD,*, pages 105–115, Portugal, 2022. INSTICC, SciTePress.

[29] Steve McKeever, Oscar Bennich-Björkman, and Omar-Alfred Salah. Unit of measurement libraries, their popularity and suitability. *Software: Practice and Experience*, 2020.

[30] Steve McKeever, Görkem Paçaci, and Oscar Bennich-Björkman. Quantity Checking through Unit of Measurement Libraries, Current Status and Future Directions. In *Model-Driven Engineering and Software Development*, MODELSWARD, January 2019.

[31] Microsoft. F# software foundation, 2020. Last Accessed on May 19th 2022.

[32] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[33] Mohamed Mussa, Samir Ouchani, Waseem Sammane, and Abdelwahab

Hamou-Lhadj. A Survey of Model-Driven Testing Techniques. *Proceedings - International Conference on Quality Software*, pages 167–172, 08 2009.

[34] NIST. International System of Units (SI): Base and Derived, 2015. Last Accessed May 19th, 2022.

[35] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. Lightweight Detection of Physical Unit Inconsistencies Without Program Annotations. In *Proceedings of International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 341–351, NY, USA, 2017. ACM.

[36] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. Phriky-Units: A Lightweight, Annotation-Free Physical Unit Inconsistency Detection Tool. In *Software Testing and Analysis*, ISSTA 2017, page 352–355, NY, USA, 2017. Association for Computing Machinery.

[37] John-Paul Ore, Sebastian Elbaum, and Carrick Detweiler. Dimensional inconsistencies in code and ROS messages: A study of 5.9m lines of code. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 712–718, 2017.

[38] John-Paul Ore, Sebastian Elbaum, Carrick Detweiler, and Lambros Karkazis. Assessing the Type Annotation Burden. In *Automated Software Engineering*, ASE 2018, pages 190–201, NY, USA, 2018. ACM.

[39] Robert Hanisch et al. Stop squandering data: make units of measurement machine-readable. *Nature*, 605:222–224, May 2022.

[40] Omar-Alfred Salah and Steve McKeever. Lack of Adoption of Units of Measurement Libraries: Survey and Anecdotes. In *Proceedings of Software Engineering in Practice*, ICSE-SEIP '20, NY, USA, May 2020. ACM.

[41] B. Selic. Beyond mere logic: A vision of modeling languages for the 21st century. In *Pervasive and Embedded Computing and Communication Systems (PECCS)*, pages IS–9–IS–9, 2015.

[42] Arthur Stephenson, Lia LaPiana, Daniel Mulville, Frank Bauer Peter Rutledge, David Folta, Greg Dukeman, Robert Sackheim, and Peter Norvig. Mars Climate Orbiter Mishap Investigation Board Phase 1 Report, 1999. Last Accessed on May 19th, 2022.

[43] S. S. Stevens. On the theory of scales of measurement. *Science*, 103(2684):677–680, 1946.

[44] TIOBE. The importance of being earnest index. Online https://www.tiobe.com/tiobe-index/, 2022. Last Accessed on 6th of July.

[45] Mitchell Wand and Patrick O'Keefe. Automatic Dimensional Inference. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 479–483, 1991.

[46] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. The fair guiding principles for scientific data management and stewardship. *Scientific data*, 3, 2016.

[47] Tongtong Xiang, Jeff Y. Luo, and Werner Dietl. Precise Inference of Expressive Units of Measurement Types. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.