Quantity Checking through Unit of Measurement Libraries, Current Status and Future Directions

Steve McKeever, Görkem Paçaci and Oscar Bennich-Björkman
Department of Informatics and Media, Uppsala University, Sweden

Keywords: Units of Measurement, Units Checking, Unit Libraries, Quantity Pattern.

Abstract: Unit errors are known to have caused some costly software engineering disasters, most notably the Mars Climate Orbiter back in 1999. As unit annotations are not mandatory for execution only dramatic events become newsworthy. Anecdotally however, there is evidence to suggest that these kinds of errors are recurrent and under-reported. There are an abundance of tools and most notably libraries to aid scientific developers manage unit definitions. In this paper we look in detail at how a number of prominent libraries in the most popular programming languages support units. We argue that even when these libraries are based on a sound design pattern, their implementation becomes too broad. Each library is distinct with varying features, lacking a core API, compromising both interoperability and thereby usage. We claim that further library or tool development is not needed to further adoption, but that a greater understanding of developers requirements is.

1 INTRODUCTION

In scientific applications, physical quantities and units of measurement are used regularly. However few programming languages provide direct support for managing them. The technical definition of a physical quantity is a "property of a phenomenon, body, or substance, where the property has a magnitude that can be expressed as a number and a reference" (Joint Committee for Guides in Metrology (JCGM), 2012). To explain this further, each quantity is declared as a number (the magnitude of the quantity) with an associated unit (Bureau International des Poids et Mesures, 2014). For example you could assert the physical quantity of length with the unit metre and the magnitude `10` (`10m`). However, the same length can also be expressed using other units such as centimetres or kilometres, at the same time changing the magnitude (`1000cm` or `0.01km`). Although these examples are all based on the International System of Units (SI), which is the most used and well known unit system, there exists several other systems that these physical quantities can be expressed in, each with different units for the same quantity. Other examples include the *Imperial system*, the *Atomic Units system*, and the *CGS* (centimetre, gram, second) system. These have evolved over time and branched off from each other.

Some well known and expensive errors have arisen due to unit inconsistencies when operating over values of differing unit systems or differing representations of dimensions. The most famous of which is the Mars Climate Orbiter (Stephenson et al., 1999). The orbiter had malfunctioned, causing it to disintegrate in the upper atmosphere. A later investigation found that the root cause of the crash was the incorrect usage of Imperial units in the probe's software. However for the most part, scientific developers have been able to manage their code without tool or static checking support. The burden of cost for such errors has been contained within the scope of their endeavours.

Several popular software modelling languages include representations of physical units, such as Modelica (Modelica, 2018) or VSL in the MARTE standard (Ribeiro et al., 2016), but these form part of the specification and are not a requirement for derived executables, much as cardinality annotations in UML diagrams. With greater interoperability, industrial use of computational simulations and penetration of digitalisation through cyber-physical systems; it seems pertinent to faithfully represent key properties of physical systems such as units of measurement in code bases. This can be achieved through the use of domain specific languages (Garny et al., 2008) or programming languages that support units of measurement, tools that detect unit inconsistencies or libraries that provide units to existing languages.

McKeever, S., Paçaci, G. and Bennich-Björkman, O.
Quantity Checking through Unit of Measurement Libraries, Current Status and Future Directions.
DOI: 10.5220/0007524704410447
In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*, pages 441-447
ISBN: 978-989-758-358-2

Adding units to conventional programming languages has a rich history going back to the 1970s and early 80s, with proposals to extend Fortran (Gehani, 1977) and then Pascal (Dreiheller et al., 1986). The pioneering foundational work was undertaken by Wand and O'Keefe (Wand and O'Keefe, 1991). They revealed how to add dimensions to the simply-typed lambda calculus, such that polymorphic dimensions can be inferred in a way that is a natural extension of Milner's polymorphic type inference algorithm.

In terms of initial library support for modular or object-oriented languages, Hilfinger (Hilfinger, 1988) showed how to exploit Ada's abstraction facilities, namely operator overloading and type parameterisation, to assign attributes for units of measurement to variables and values.

All of the aforementioned solutions require either migration to a new language or annotating the source code, both of which are burdens on the developers. A more lightweight methodology is presented in (Ore et al., 2017) that uses an initial pass to build a mapping from attributes in shared libraries to units. The shared libraries contain unit specifications so this mapping is used to propagate into a source programme and, as the authors show, detect inconsistencies.

An alternative pathway is to introduce physical units into an object oriented modelling platform, along with a compilation workflow that leverages OCL expressions (Mayerhofer et al., 2016) or staged computation (Allen et al., 2004) to derive units where possible at compile-time. These elegant abstractions lift the declaration and management of units into software models but do not solve the interoperability problem due to the lack of agreed conventions for scientific, medical and financial applications of the Quantity pattern.

Unfortunately we lack an authoritative estimate of how frequently unit inconsistencies occur or their cost. Anecdotally we can glean that it is not negligible from experiments described in certain papers. Cooper (Cooper and McKeever, 2008) developed a validation tool for CellML, a domain specific language for modelling biological systems. He applied it to the repository of CellML models and, of those that were found to be invalid, 60% had dimensionally inconsistent units. Similarly, the Osprey type system type system (Jiang and Su, 2006) provides an advanced unit checker and inference engine for C. In their paper they describe having applied it to mature scientific application code and found hitherto unknown errors. Unfortunately they do not describe the prevalence or magnitude of these errors. A more telling statistic is found in (Ore et al., 2017) where they apply their lightweight unit inconsistency tool to 213 open-source systems, finding inconsistencies in 11% of them.

There are many libraries that provide support for units in all of the popular programming languages. However no standard has emerged. In this paper we argue that the existing design pattern is too broad and open to interpretation. The rest of this paper is structured as follows. In Section 2 we provide an introduction to units of measurement and the Quantity pattern that provides a standard object oriented solution. In Section 3 we describe the underlying implementation strategy of some of the most used unit libraries in the most popular programming languages. In Section 4 we surmise as to why the Quantity pattern has failed to galvanise the scientific programming community, and suggest areas of future work required to promote better software engineering with regards to units of measurement. Finally in Section 5 we articulate our position and contextualise our study further.

2 DESCRIBING UNITS OF MEASUREMENT

Here, we will first introduce the problem by means of an example scenario. Two programmers are working on a system that manages physical quantities using a popular language such as C#, Java, or Python. The first programmer wants to create two quantities that will be used by the second at a later stage. Because the language does not have support for this type of construct, he or she decides to do it using integers and adding comments, like this:

```
int mass = 10; // in tonnes
int acceleration = 10; // in m.s^-2
```

Now the second programmer wants to use these values to calculate force using the well-known equation $F = m \times a$:

```
int force = mass * acceleration; // 100N
```

The variable `force` will now have the value of `100`, assumed to be `100 N`. The issue is that the variable mass is actually representing `10 tonnes`, not `10 kilograms`. This means that the actual value of the force should be `100000 N` (instead of `100 N`), off by a factor of one thousand. Because the quantities in this example are represented using integers there is no way for the compiler to know this information and therefore it is up to the programmers themselves to keep track of it.

The only reliable way to solve this is try to remove the human element by having a systematic means of checking the units of calculations to ensure that

they are handled correctly. This type of automatic check is potentially something that could be undertaken at compile time in a strongly typed language, but unfortunately very few languages have support for units of measurement and only one of those is in the top twenty most popular programming languages, according to the TIOBE index (TIOBE, 2018). In all other languages, it is up to the software developers to create these checks themselves.

One example of how this can be achieved is to make sure the compiler knows what quantities are being used by encapsulating this into a class hierarchy, with each unit having its own class. The scenario above would then look like this instead:

```
Tonne mass = 10;
Acceleration acceleration = 10; // in m.s^-2
Force force = mass * acceleration; // 100000N
```

Compared to the previous example, here the compiler now knows exactly what it is dealing with and thus the information that the mass is in tonnes is kept intact and the correct force can be calculated in the end. This type of solution not only means that differences in magnitude and simple conversions are taken care of but also that any erroneous units being used in an equation can be caught at compile time. However, making a class hierarchy similar to the one illustrated above could potentially involve hundreds of units and thousands of conversions.

A more abstract approach is to bind the value along with the unit in what is known as the Quantity pattern (Fowler, 1997).

```
class Quantity {
  private float value;
  private Unit unit;

  ....
}
```

We can include arithmetic operations to the `Quantity` class that ensures addition and subtraction only succeed when their units are equivalent, or multiplication and subtraction generate a new unit that represents the derived value correctly. Moreover we can include other useful behaviour such as printing and parsing to this class.

Units can be defined in the most generic form using an algebraic definition that includes two types, *base quantities* and *derived quantities*. The base quantities are the basic building blocks, and the derived quantities are built from these. The base quantities and derived quantities together form a way of describing any part of the physical world (Sonin, 2001). For example length (metre) is a base quantity, and so is time (second). If these two base quantities are combined they express velocity (metre/second or $\mathsf{metre} \times \mathsf{second}^{-1}$) which is a derived quantity. The International System of Units (SI) defines seven base quantities (length, mass, time, electric current, thermodynamic temperature, amount of substance, and luminous intensity) as well as a corresponding unit for each quantity (The National Institute of Standards and Technology, 2015).

These physical quantities are also organised in a system of dimensions, each quantity representing a physical dimension with a corresponding symbol (`L` for length, `M` for mass, `T` for time etc.).

```
type base = L | M | T ...
```

Any derived quantity can be defined by a combination of one or several base quantities raised to a certain power. These are called *dimensional exponents* (Bureau International des Poids et Mesures, 2014).

```
type derived = Base of base
             | Times of (derived * derived)
             | Exp of (derived * int)
```

Dimensional exponents are not a type of unit or quantity in themselves but rather another way to describe an already existing quantity in an abstract way. Using the same example of velocity as before, it can be expressed as:

```
Times (L, Exp (T,-1))
```

In concrete syntax this is instead expressed as $\texttt{L}\times\texttt{T}^{-1}$, where `L` represents length and $\texttt{T}^{-1}$ represents the length being divided by a certain time. Representing units in this manner is not always optimal as a normal form exists which makes storage and, more importantly, 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 $\langle e_1, \ldots, e_7 \rangle$ 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. Thus 3 Newtons would be represented as $\langle 1, 1, -2, 0, 0, 0, 0 \rangle$, or `3 kg.m.s^-2`. Dimensionless units are represented by a tuple whose 7 components are all 0. Interestingly any unit from any other system can be expressed in terms of SI units. Conversions can be undertaken using mostly multiplication factors, but in some case offsets are required too.

This suggests implementing units through the following class outline:

```
class Unit {
  private int [7] dimension
  private float [7] conversionFactor
  private int [7] offset
  private String name
  ...
  boolean isCompatibleWith (Unit u)
```

```
  boolean equals (Unit u)
  Unit multiplyUnits (Unit u)
  Unit divideUnits (Unit u)
}
```

The `dimension` array contains the 7-tuple of base unit exponentials. The attributes `conversionFactor` and `offset` enable conversions from this unit system to the SI units, while `name` is so that users can define their own unit system.

The class `Unit` also defines operations to compare and combine units. The method `isCompatibleWith` checks whether two units are compatible for being combined, such as miles and centimetres. While `equals` returns true if the units are exactly the same, which is used when adding or subtracting quantities. When two quantities are multiplied then `multiplyUnits` adds the two `dimension` arrays. Correspondingly, `divideUnits` subtracts each of the elements of the `dimension` array.

The idea behind a software design pattern is a general, reusable solution to a commonly occurring problem. Due to the lack of an agreed interface to the Quantity pattern, non-compatible domain specific instantiations have proliferated as we shall demonstrate in the next section.

3 ANALYSIS

A study of Unit of Measurement libraries (Bennich-Björkman and McKeever, 2018) has shown that there is a lot of reinvention and little collaboration. By analysing popular open-source repositories, the authors discovered close to 300 active libraries for the top twenty programming languages. A further reduction based on features, reputation and development status brought this number down to 82 as shown in Figure 1.

We have looked in more detail at a number of these prominent libraries to elucidate how they operate, their feature set and potential for interoperability. All of the libraries we analysed implemented the Quantity pattern, albeit in a number of different ways. How each library in each specific language implemented the pattern is detailed below.

3.1 Java

Two of the most prominent libraries written in Java are CaliperSharp and the JSR 385 project (`github.com/point85/CaliperSharp`, `github.com/unitsofmeasurement/unit-api`).

CaliperSharp implements the Quantity pattern through the use of the class `Quantity` in which the `Amount` is used to model the magnitude of the quantity and `UnitOfMeasure` is used to specify the unit for the quantity. In CaliperSharp, `UnitOfMeasure` is defined as an enumeration which contains all unit definitions that the library supports.

On the other hand JSR 385 implements the Quantity pattern through the generic interface `Quantity<T>` which each defined (specific) quantity then implements. For example:

```
Acceleration extends Quantity<Acceleration>
```

The quantity interface itself contains a definition of a unit through the use of another interface, `Unit`. This `Unit` interface also utilises the concept of physical dimensions to define the unit.

3.2 C++

The leading library for C++ is BoostUnits (`github.com/boostorg/units`). It is in effect the de facto standard and is actively developed, has very good documentation, supports many units as well as numerous different constants. BoostUnits is also part of a big development team (`Boost.org`). However Boost exploits the C++ template meta-programming library so it is more than just a library as it supports a staged computation model similar to MixGen (Allen et al., 2004).

None of the other prominent programming languages have this flexible compilation strategy so it is an anomaly and even though it implements dimensional analysis in a general and extensible manner, treating it as a generic compile-time meta-programming problem, this style of software development is radically different than a single stage compilation.

The key advantage of this staged approach is that with appropriate compiler optimisation, no runtime execution cost is introduced, encouraging the use of this library to provide dimension checking in performance-critical code. Nonetheless, the core feature set is not too distinct from other libraries.

3.3 Python

The Astropy library (`github.com/astropy/astropy`) has the single most commits out of all those presented in (Bennich-Björkman and McKeever, 2018), which implies that it is also one of the most well-developed. Like the previously presented libraries, Astropy has the Quantity pattern at its root and, as it has so many commits and contributors, it is feature rich. Much of the functionality is built on top of its well engineered core.

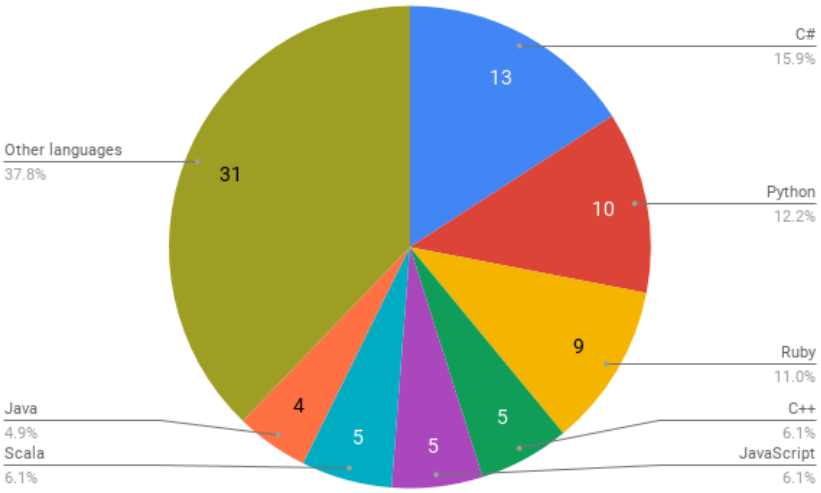

Figure 1: Leading unit libraries per language (Bennich-Björkman and McKeever, 2018).

Another popular Python library is Pint. (`github.com/hgrecco/pint`). It implements the Quantity pattern explicitly as the central unit in the library is the `Quantity` class with a magnitude and a unit of measurement and this is how all units are defined. For example:

```
>>> import pint
>>> ureg = pint.UnitRegistry()
>>> 3 * ureg.meter + 4 * ureg.cm
<Quantity(3.04, 'meter')>
```

3.4 C#

Looking at the number of commits, contributors, comprehensiveness of the documentation as well as adoption, UnitsNet (`github.com/angularsen/UnitsNet`) is one of the best libraries overall, not only for C#.

Although UnitsNet also implements the same pattern through the `IQuantity` interface, this is achieved in a slightly different manner to other libraries shown here. The type of the quantity and the value for the seven base dimensions are the only attributes that are defined in the interface. The magnitude (or value) of the quantity is added later as an operation for a specific unit that is calculated. The type for each quantity is defined as `QuantityType` which is an enumeration containing all the quantities that the library supports. This is similar to how it is handled in the CaliperSharp Java library mentioned earlier.

In UnitsNet each specific unit is defined as its own class and utilises operator overloading to convert between units. The library also employs automatic code generation to produce all the different conversions between different units.

Another capable library written in C# is Gu.Units (`github.com/GuOrg/Gu.Units`). Similar to UnitsNet it implements the Quantity pattern through the `IQuantity` interface which defines a value (`SiValue`) and a unit (`SiUnit`).

The unit of a quantity is defined through the interface `IUnit` which contains information about the symbol for the unit, the base unit and ways to convert the value of unit to its base value (kilometre to metre for example) or from the base value. Another similarity to UnitsNet is that Gu.Units also uses code generation which leverages these interfaces to make concrete quantities.

3.5 Javascript

A popular library for JavaScript is JS Quantities (`github.com/gentooboontoo/js-quantities`). With over 300 commits, it is one of libraries with the most commits for the JavaScript language. The library is also based upon another popular physical quantity library called Ruby Units (`github.com/olbrich/ruby-units`) which is also quite mature.

Similar to Pint, JS Quantities implements the Quantity pattern through a general class called `Qty`, which represents a generic quantity. A user can then create whatever quantity they want through this class.

```
qty = Qty('1m');
qty = Qty('1 N*m');
qty = Qty('1 m/s');
qty = Qty('1 m^2/s^2');
qty = Qty('1 m^2 kg^2 J^2/s^2 A');
qty = Qty('1 attoparsec/microfortnight');
```

In the example above, the use of dimensions and dimensional exponents in JS Quantities is also show-

cased through the use of the `Qty` class. Similar to other libraries, all the available quantities are defined in an enumeration.

4 EVALUATION AND FUTURE WORK

In this section we postulate as to why we have arrived at this state of affairs when the underlying computing science is well understood and Fowler (Fowler, 1997) provided an effective object model for units of measurement some years ago. The Quantity pattern has been shown to be applicable to mathematical calculations, medical observations and financial conversions. A more detailed and specialised version of this pattern is provided by the Physical Quantity pattern (Krisper et al., 2017) that looks in more depth at the requirements of the physical and mathematical sciences, presenting interfaces to enforce the use of explicit quantity types. The key aspect is that the *research challenges* are not technical in nature, more work is required to create robust interfaces from the Quantity pattern that engage the respective communities and gather traction. Even if these well engineered interfaces existed, we suggest two other reasons that hamper uptake.

The first reason was put forward by Damevski (Damevski, 2009) and is that scientific programmers should not be burdened by units at each statement in their programs, but that units should be present in software component interfaces. This makes sense when you consider that research groups in, say, physics and chemistry departments have evolved their own conventions, methodologies and ontologies. Problems can arise when they try to collaborate over energy conversions, for instance. While the physicist works in terms of electron volts per formula unit, the chemist thinks in terms of kilojoules per mole. In such cases not only are the units different but so are the magnitudes. In this mode, each research group should be free to develop their codes without unit annotations but when they come to combining their codes with others, the conversions need to be explicitly introduced.

The Logic of Collective Action (Olson, 2009) develops a theory of political science and economics of concentrated benefits versus diffuse costs. Its central argument is that concentrated minor interests will be overrepresented and diffuse majority interests due to a *free-rider problem* that is stronger when a group becomes larger. This is perhaps an explanation as to why some of the libraries have continued to exist and prosper when there are equally good ones for that particular language. Once developed and a user base has accrued, due to the open-source nature of the endeavour, it becomes necessary to keep supporting the library as vital code has become dependent on its existence.

There are two central avenues of further work in this area. We will interview scientific, medical and financial developers to elucidate their requirements. Some users might be content with very lightweight support, as envisaged by Damevski (Damevski, 2009). This would allow diverse teams to collaborate even if their domain specific environments or choice of unit systems were to some extent incompatible. However other users might require a more robust environment in which all variables are given an explicit unit or are dimensionless, and a checker will ensure operational unit correctness. In-between we might allow static or dynamic unit conversions, the ability to define new domain specific unit systems and a degree of flexibility that unit variables allow. Whether all of these features need to be supported, and to what extent, is an open question that we hope to address in the near future so that the Quantity pattern can be enriched with an effective API for modelling and implementing units of measurement.

A second avenue that is urgently required is an understanding of the true cost of unit of measurement errors. The big disasters grab the headlines but it is the more mundane day to day outlay of defective code, leading to incorrect results that could be more costly yet is not reported. We mentioned earlier some informal figures describing the extent of the problem but a more accurate understanding is needed. An open source repository of defective unit error code would allow researchers to explore techniques to detect, assist and possibly recover from some form of unit errors.

5 CONCLUSION

Our initial study of unit libraries (Bennich-Björkman and McKeever, 2018) highlighted an abundance of similar contributions without a clear standard emerging. Unit libraries allow developers to faithfully represent quantities in their code and provide assurances over correctness. These features are desirable from both model and software engineering perspectives as they reduce errors and encourage maintainability. However they are not a requirement for creating executable code.

In order to escape from this impasse we need to know how people actually work with units. Are unit libraries sufficient? How can we ensure that

they are used unless mandated by the organisation. Modern development workflows might favour separate unit inferencing tools or unit checking to be undertaken through testing instead. The needs of modern commercial systems, deployed on a vast number of distinct devices, developed with a plethora of languages, evolving daily through continuous integration is rather different to those of an academic research group.

Numerous stakeholders, from developers upwards to project managers in both small and large organisations need to be interviewed. Rather than focusing on unit library or tool support, the purpose of our current ongoing research is to delve into these broader topics using questionnaires to understand the underlying issues and causes.

REFERENCES

Allen, E., Chase, D., Luchangco, V., Maessen, J.-W., and Steele, Jr., G. L. (2004). Object-oriented units of measurement. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 384–403, New York, NY, USA. ACM.

Bennich-Björkman, O. and McKeever, S. (2018). The next 700 unit of measurement checkers. In *Proc. SLE*, pages 121–132. ACM.

Bureau International des Poids et Mesures (2014). SI Brochure: The International System of Units (SI), 8th Edition, Dimensions of Quantities. Online https://www.bipm.org/en/publications/si-brochure/chapter1.html. Last Accessed July 2nd, 2018.

Cooper, J. and McKeever, S. (2008). A model-driven approach to automatic conversion of physical units. *Softw. Pract. Exper.*, 38(4):337–359.

Damevski, K. (2009). Expressing measurement units in interfaces for scientific component software. In *Proceedings of the 2009 Workshop on Component-Based High Performance Computing*, CBHPC '09, pages 13:1–13:8, New York, NY, USA. ACM.

Dreiheller, A., Mohr, B., and Moerschbacher, M. (1986). Programming pascal with physical units. *SIGPLAN Not.*, 21(12):114–123.

Fowler, M. (1997). *Analysis Patterns: Reusable Objects Models*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Garny, A., Nickerson, D., Cooper, J., dos Santos, R. W., Miller, A., McKeever, S., Nielsen, P., and Hunter, P. (2008). Cellml and associated tools and techniques. *Philosophical Transactions of the Royal Society, A: Mathematical, Physical and Engineering Sciences*, 366.

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

Hilfinger, P. N. (1988). An ada package for dimensional analysis. *ACM Trans. Program. Lang. Syst.*, 10(2):189–203.

Jiang, L. and Su, Z. (2006). 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. ACM.

Joint Committee for Guides in Metrology (JCGM) (2012). International Vocabulary of Metrology, Basic and General Concepts and Associated Terms (VIM). Online https://www.bipm.org/en/about-us/. Last Accessed November 20th, 2018.

Krisper, M., Iber, J., Rauter, T., and Kreiner, C. (2017). Physical quantity: Towards a pattern language for quantities and units in physical calculations. In *Proceedings of the 22Nd European Conference on Pattern Languages of Programs*, EuroPLoP '17, pages 9:1–9:20, New York, NY, USA. ACM.

Mayerhofer, T., Wimmer, M., and Vallecillo, A. (2016). Adding uncertainty and units to quantity types in software models. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 118–131, New York, NY, USA. ACM.

Modelica (2018). Modelica and the Modelica Association. Online https://www.modelica.org. Last Accessed on November 12th, 2018.

Olson, M. (2009). *The Logic of Collective Action: Public Goods and the Theory of Groups, Second printing with new preface and appendix*, volume 124. Harvard University Press.

Ore, J.-P., Detweiler, C., and Elbaum, S. (2017). Lightweight detection of physical unit inconsistencies without program annotations. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 341–351, New York, NY, USA. ACM.

Ribeiro, F. G. C., Rettberg, A., Pereira, C. E., and Soares, M. S. (2016). An analysis of the value specification language applied to the requirements engineering process of cyber-physical systems. *IFAC-PapersOnLine*, 49(30):42 – 47. 4th IFAC Symposium on Telematics Applications TA 2016.

Sonin, A. A. (2001). The physical basis of dimensional analysis. Technical report, Massachusetts Institute of Technology.

Stephenson, A., LaPiana, L., Mulville, D., Peter Rutledge, F. B., Folta, D., Dukeman, G., Sackheim, R., and Norvig, P. (1999). Mars Climate Orbiter Mishap Investigation Board Phase 1 Report. Online https://llis.nasa.gov. Last Accessed on November 20th, 2018.

The National Institute of Standards and Technology (2015). International System of Units (SI): Base and Derived. Online https://physics.nist.gov/cuu/Units/units.html. Last Accessed July 2nd, 2018.

TIOBE (2018). Tiobe (The Importance of Being Earnest) company index for November, 2018. Online https://www.tiobe.com/tiobe-index/. Last Accessed on November 20th, 2018.

Wand, M. and O'Keefe, P. (1991). Automatic dimensional inference. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 479–483.