# From Quantities in Software Models to Implementation

Steve McKeever[a]
*Department of Informatics and Media, Uppsala University, Sweden*

Keywords: Units of Measurement, Quantity Pattern, Libraries, Component based Checking, Testing.

Abstract: In scientific and engineering applications, physical quantities expressed as units of measurement (UoM) are used regularly. If the algebraic properties of a system's UoM information are incorrectly handled at run-time then catastrophic problems can arise. Much work has gone into creating libraries, languages and tools to ensure developers can leverage UoM information in their designs and codes. While there are technical solutions that allow units of measurement to be specified at both the model and code level, a broader assessment of their strengths and weaknesses has not been undertaken. Inspired by a survey of practitioners, we review four competing methods that support unit checking of code bases. The most straightforward solution is for the programming language to Natively support UoM as this allows for efficient unit conversion and static checking. Alas, none of the mainstream languages provide such support. Libraries might seem compelling, and all popular programming languages have a myriad of options, but they're cumbersome in practice and have specific performance costs. Libraries are best suited to applications in which UoM checking is desirable at run-time. Lightweight methods, such as Component based checking or Black Box testing, provide many benefits of UoM libraries with minimal overheads but sacrifice coverage and thus robustness. By separating and analysing the various options, we hope to enable scientific developers to select the most appropriate approach to transferring UoM information from their software models to their programs.

## 1 INTRODUCTION

Ensuring numerical values that denote physical quantities are handled correctly is an essential requirement for the design and development of any engineering application. Notorious examples such as the Mars Climate Orbiter (Stephenson et al., 1999) or the Gimli Glider incident (Witkin, 1983) attest to this. With ever increasing digitalisation, and removal of humans in the loop, the need to faithfully represent and manipulate quantities in physical systems is ever increasing. There are many ways in which this can be achieved, allowing the designer or programmer to rely on the checker to ensure correctness and not on themselves or their colleagues. The software engineering benefits of adopting unit checking and automatic conversion support is indisputable. However, it is not always clear which approach is best suited for a given problem, coupled with a general lack of awareness of solutions, means that implementers often reinvent the wheel or forgo any kind of checking.

Providing unit support in spreadsheets (Antoniu et al., 2004), Wolfram Mathematica[1], simulation tools and modeling environments has been very effective. For dedicated workflows that support numerical applications, the cost of annotating cells or variables with unit information is typically ingrained in the design process and thus less intrusive than in a more general setting. Modelica (Modelica, 2020) supports UoM and checks descriptions before animating them. Domain specific languages for the curation and interchange of biological models[2] support UoM information and require unit validation before uploading to repositories.

For general purpose development we have to look before the advent of object oriented modeling languages or formal specification notations. Adding units to conventional programming languages goes back to the 1970s (Karr and Loveman, 1978) and early 80s with proposals to extend Fortran (Gehani, 1977) and then Pascal (Dreiheller et al., 1986). These efforts were mostly syntax based. A more viable starting point would be (Hilfinger, 1988) that showed how to exploit Ada's abstraction facilities, namely opera-

[a] https://orcid.org/0000-0002-1970-2884

[1] https://www.wolfram.com/mathematica/
[2] sbml.org, cellml.org

McKeever, S.
From Quantities in Software Models to Implementation.
DOI: 10.5220/0010247201990206
In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2021)*, pages 199-206
ISBN: 978-989-758-487-9

tor overloading and type parameterisation, to assign attributes for UoM to variables and values. The emergence of Object Oriented programming languages enabled developers to implement UoM either through a class hierarchy of units and their derived forms, or through the Quantity pattern (Fowler, 1997). There are a large number of libraries for all popular object oriented programming languages that support this approach (McKeever et al., 2019).

Software development typically begins at a more abstract level through diagrams and rules that focus on the conceptual model that is to be implemented. By extending the Unified Modeling Language (UML), quantities can be introduced into an object-oriented modeling platform. Unit checking and conversion can be undertaken before code is generated, either through a compilation workflow that leverages Object Constraint Language (OCL) expressions (Mayerhofer et al., 2016) or staged computation (Allen et al., 2004). Similarly (Gibson and Méry, 2017) add units to the Event-B modelling language and leverage the Rodin theorem prover to detect inconsistencies before compiling to Java. Comparable ideas have been presented for the formal specification language Z (Hayes and Mahony, 1995) and Maude (Chen et al., 2003). More specialised UML based systems modeling languages such as MARTE[3] and SysML[4] also have UoM support (Burgueño et al., 2019). These elegant abstractions lift the declaration and management of units into software models. However once the code has been generated, UoM information might very well be lost unless the workflow has been tailored explicitly. This is the research question that we address, namely *what is the most appropriate approach to transferring UoM information into software*. We consider aspects such as ease of use, execution speed, numeric accuracy, ease of integration and coverage of unit error detection capabilities. These aspects reflect our study of UoM libraries, survey of practitioners and preliminary design suggestions summarised in (McKeever et al., 2020).

We lack a definitive estimate of how frequently unit inconsistencies occur or their cost. Anecdotally we can glean that it is not negligible from experiments described in the literature. When applied to a repository of CellML models, a validation tool (Cooper and McKeever, 2008) found that 60% of the descriptions that were invalid had dimensionally inconsistent units. A spreadsheet checker (Antoniu et al., 2004) was applied to 22 published scientific spreadsheets and detected 3 with errors. A lightweight C++ unit inconsistency tool (Ore et al., 2017a) was applied to 213 open-source systems, finding inconsistencies in 11% of them. A further study (Ore et al., 2017c) using a corpus of robot software with 5.9M lines of code, found dimensional inconsistencies in 6% of repositories. Thus, it seems important to ensure UoM information existing in Software Models is supported in derived implementations.

The decision on which approach to choose, if at all, will depend more on the requirements of the project and whether the various components of the system can support UoM types. Very few programming languages provide Native support for quantities so this is rarely an option. UoM Libraries are not always the most effective solution. Lightweight methods, such as Component based checkers or Black Box testing, often provide 'good enough' detection without the drawbacks. UoM libraries are best suited to run-time checking where performance is not a key condition but correctness is.

The rest of this paper is structured as follows. In Section 2 we provide a brief background to UoM and the Quantity pattern. In Section 3 we describe the four means of supporting UoM information in implementations and in Section 4 we summarise the results of our comparative study, providing suggestions for developers as to which method to choose depending on their requirements.

## 2 BACKGROUND

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). Each quantity is declared as a number (the magnitude of the quantity) with an associated unit (Bureau International des Poids et Mesures, 2019). 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, such as the *Imperial system*.

Units 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. The base quantities and derived quantities together form a way of describing any part of the physical world (Sonin, 2001).

[3]https://www.omg.org/omgmarte/

[4]https://sysml.org

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 metre $\times$ 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 (NIST, 2015). Using an Object Oriented language we could create a class hierarchy for each base type and use a tree structure to construct derived types. However this would result in hundreds of units and thousands of conversions. Also showing two unit definitions to be equivalent would be non-trivial.

Fortunately, 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 $\langle e_1, \ldots, e_7 \rangle$ 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. In Java this would be represented as:

```
class Unit {
  private int [7] dimension
  private float [7] conversionFactor
  private int [7] offset
  ...
  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. 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.

Using this representation for a unit we can construct 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 division generate a new unit that represents the derived value correctly. This pattern is the basis for many UoM Libraries (McKeever et al., 2019) but can also be described in UML.

The Quantity pattern provides a means of annotating variable declarations and method signatures with behavioural UoM specifications. Bearing in mind the annotation burden, (Ore et al., 2018) 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. In a Software Model, entities annotated in this manner are merely decorative. Tools can be written to ensure that the models use quantity information correctly, but they need to be rendered into the codebase to ensure implementations are robust with regards to UoMs. In the next Section we explore the various options for transferring UoM information faithfully into program code.

## 3 IMPLEMENTATION OPTIONS

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. However with UoM annotations the situation is more complex. Applying UoM annotations requires an advanced checker to ensure variables and method calls are handled soundly. Two units are compatible if they both can be represented as the same derived quantity. For instance degrees Celsius is compatible with Fahrenheit. Values in Celsius can be converted to values in Fahrenheit. Surface tension can be described as newtons per meter or kilogram per second squared, and even though they equate they represent different quantities. For complex derived units, showing compatibility is not so straightforward, nor is it clear whether such quantities should be aligned (Hall, 2020). Managing such quantities at run-time requires UoMs to become first-class citizens.

Two values can be added or subtracted only if their units are the same. Multiplication and division either add or subtract the two units product of power representations, assuming both values are compatible. Converting values to ensure compatibility can create round-off errors. Once a variable has been defined to be of a given unit, then it will remain as such. Checking that all annotated entities behave according to these rules ensures both *completeness* and *correctness* of the model or program. If a variable or method call is incorrectly annotated an error will arise. When

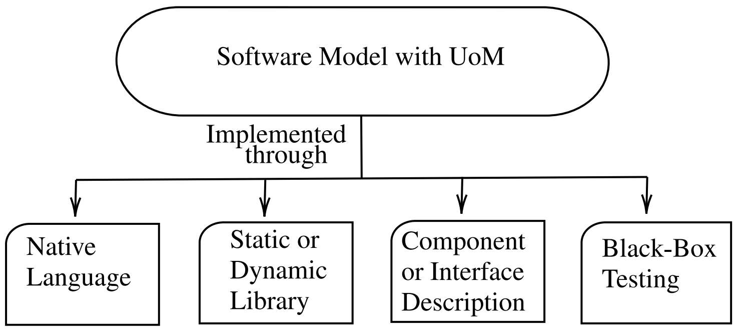

Figure 1: From a Software Model that includes UoM information to an implementation that supports them.

not all variables or method calls are annotated then neither completeness nor correctness can be guaranteed. All implementation options are affected by the following three concerns (McKeever et al., 2020):

**Lack of Awareness:** many developers are totally unaware of software solutions that deal with UoM. Inertia from developers stem from factors like tradition, fear of change and effort of learning something new.

**Technical Internal Factors:** many solutions are awkward and imprecise, introducing a loss of precision and struggling at times with dimensional consistencies.

**External Factors:** modern systems are not built in a vacuum but form part of an eco-system (Lungu, 2008). It is harder to argue for UoM annotations when values pass through numerous generic components that do not support them, such as legacy systems, databases, spreadsheets, graphics tools and many other components that are unlikely to support UoM without costly updates.

The rest of this section will focus on the technical factors of each approach.

## 3.1 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 (Wand and O'Keefe, 1991) showed 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 (Milner, 1978). The key feature of UoM is that they cannot be solved symbolically but equationally using the theory of Abelian groups (Kennedy, 1994). Providing UoM syntax and an equational checker is what distinguishes a language with native support. UoM information is checked at compile-time and can be removed from the generated run-time code. Moreover unit conversions can be minimised in order to reduce round-off errors (Cooper and McKeever, 2008).

The only language in the 20 most popular programming languages (TIOBE, 2020) that supports units of measurement is Apple's Swift language (Apple, 2020). The only other well-known language to support units of measurement is F# (Microsoft, 2020) which has the added benefit of allowing unit variables, namely undefined unit types, that the compile-time checker will attempt to resolve. If there is insufficient information then the program will be rejected. This feature allows a small degree of incompleteness in language definitions so the burden of annotation is mitigated somewhat, but the static checker will derive the missing UoM information so correctness is ensured. Nonetheless neither Swift nor F# are common in large software engineering projects.

C++ is still very popular (TIOBE, 2020) and has a de facto UoM extension that exploits the template meta-programming feature[5]. Consequently BoostUnits is more than just a library as it supports a staged computation model, similar to MixGen (Allen et al., 2004), that is more akin to a language extension and supports backwards compatibility. None of the other prominent programming languages have this flexible compilation strategy. This staged approach is very similar to native language support and, with appropriate compiler optimisation, no run-time execution cost is introduced. C++ with the BoostUnits library supports UoM checking in performance-critical code. In practice, however, the survey (Salah and McKeever, 2020) found both accuracy and usability issues with the use of this extension.

Translating a Software Model with UoM information into a native language with UoM support ensures that quantity information is maintained, and that the implementation more closely represents the specification. Even if the original software model was auto-

[5]github.com/boostorg/units

matically checked, the code will typically be modified once in use. Refactoring will have to maintain UoM information but the compile-time checker will guarantee correctness.

Native language support is best suited to safety-critical applications where a single unit system is employed, checking must be complete, performed at compile-time and have no run-time performance cost.

## 3.2 Static or Dynamic Library Support

As few programming languages support UoM natively, a second approach is to use a library that provides units of measurement. With advanced abstraction methods such as classes and generics, developing libraries that work well with existing code bases is possible nowadays. Consequently there are many libraries for all contemporary languages (Bennich-Björkman and McKeever, 2018). The core issue that these libraries aim to solve is something that is relatively easy to understand and familiar. However as Bekolay argued *"making a physical quantity library is easy but making a good one is hard"* (Bekolay, 2013). Trying to make a more complete library, including more units, more operators, effective conversions, good error messages, efficient and accurate is far more demanding than a robust implementation of the Quantity pattern (Fowler, 1997; Krisper et al., 2017).

In practice there are many problems with their use. They require too much boilerplate code, they're rarely idiomatic to their host language, provide poor error messages, lack support for user defined types and restrict underlying storage to a single floating point representation. Certain languages, like Ruby, are less affective by these shortfalls as they encourage duck typing and a flexible syntax that facilitates domain specific language creation. Nonetheless (McKeever et al., 2020), concludes that UoM types need to be as straightforward to use as arithmetic types or at least as close as possible for adoption to occur.

A strength of libraries is that they can support both compile-time and run-time error detection. Compile-time checking can be achieved through static overloading, or Java generic instantiations. While run-time checking is achieved through overriding. An example of both styles is shown in (McKeever et al., 2020). Combining the two semantics, even with the compact Quantity Pattern representation, would double the amount of syntax required and further complicate usage. Dynamically typed languages, such as Ruby or Python, will by definition perform quantity checking at run-time. Run-time support is a key requirement for certain projects: *"In our product line, our users may very well have one file whose units are "$kg \cdot m^3$", 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"* (Salah and McKeever, 2020).

A weakness of libraries compared to native language support is that variables of a Quantity class can be reassigned at run-time, due to their semantics being embedded within the `dimension` array, so that a meter could become a kilogram. Unit mismatch and conversion errors can be detected by UoM libraries but avoiding programming style errors requires further discipline that a conventional static checker provides. Such errors are caused by violations of standard type systems, such as when an intermediate variable is used with different units. These were found to account for 75% of inconsistencies in the study of 5.9M lines of code (Ore et al., 2017c).

Their core disadvantage, however, is that their implementation requires boxed values rather than the standard primitive entities. When units are not part of the language then there is a cost at both compile-time and run-time. For applications that carry out lots of calculations (such as matrix multiplications), their performance tends to matter more and boxed values with types would have unnecessary performance overheads. At first glance, a UoM library might seem easy to use and include in a software project, but the inner workings of the UoM library often increase the complexity of a project.

## 3.3 Component or Interface Description Support

Both Native language support and Libraries require all UoM variables and function definitions to have annotations. Lightweight Component or Interface based approaches aim to liberate the scientific programmer from the need to annotate each statement. Interfaces are the dividing line between the implementation of a particular functionality and its users. 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. A component implementer would want to further restrict access with semantic details such as UoM annotations. Ideally, an interface that solves a particular computational problem for meters would differ from one that solves the same problem for miles.

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. In

```
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 2: Java code and JUnit test case for simple addition of two kilometres, or kilometre and mile distances.

(Damevski, 2009), he argues that units of measurement should be inserted in software component interfaces. There is some anecdotal evidence in the many quotes of (Salah and McKeever, 2020) to support this approach. Damevski postulates that unit libraries are too constraining and incur an annotation or migration burden. His algorithm attempts to resolve UoM at run-time so that if the types of the called method's parameters are compatible with the arguments then unit conversions occur. Consider the C++ class Earth (Damevski, 2009):

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

It assigns and queries the earth's circumference using meters internally but can be called with kilometres and the return value bound to a variable of, say, type miles. Unlike libraries, within the class `Earth` no further annotations are required, nor will it be checked. The variable `circumference` will be assigned to a `double`.

Another lightweight methodology was presented by Ore (Ore et al., 2017a) that uses an initial pass to build a mapping from attributes in C++ shared libraries to units. The shared libraries contain UoM specifications so this mapping is used to propagate into a source program and detect inconsistencies at compile-time. Their algorithm leverages dimensional analysis rules for arithmetic operators to great effect (Ore et al., 2017b).

A Component based discipline means that the consequences of local unit mistakes are underestimated. On the other hand, it allows diverse teams to collaborate even if their domain specific environments or choice of quantity systems were, to some extent, dissimilar. More importantly it would have been sufficient to have caught the Mars Climate Orbiter error.

### 3.4 Black-Box Testing

A final means of ensuring UoM checking is through automated testing. Black Box Testing mainly focuses on input and output of software applications and it is entirely based on software requirements and specifications. It seeks to spot incorrect or missing functions, interface errors, initialisation errors and errors in data structures. Creating Black Box unit tests from Software Models is another lightweight approach. The testing will not be exhaustive, as the focus would be on the initialisation of variables, the correctness of assignments and method calls. There have been many efforts to automatically generate unit tests from UML descriptions (Cavarra et al., 2002; Hartmann et al., 2005; Ali et al., 2010; Mussa et al., 2009), either through behavioural diagrams or with rule based approaches. However these techniques are seen to be costly and non-trivial in practice (Kasurinen et al., 2010).

Nonetheless, it is fairly common nowadays to manually develop tests alongside models, not only for the purpose of test driven development but also to ensure maintainability through refactoring. Including UoM tests requires no extra tool support and will not affect the eco-system as shown in Figure 2. Spending time writing unit tests would equate to adding unit annotations without the introduction of a library: *"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."* (Salah and McKeever, 2020). However, the UoM knowledge will be localised to each particular unit so the slight implementation cost comes at the expense of potentially average checking.

## 4 CONCLUSION

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 (Selic, 2015). Beginning with quantity annotated Software Models that could be diagrammatic or equational, we sought to elucidate the most appropriate method to migrate this information into code. 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. 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 are able to distinguish be-

Table 1: Contrasting alternative methods of Implementing Unit of Measurements in Software Projects.

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

tween quantities which are of the same kind and quantities that are of different kinds but have the same units. Component based techniques can be undertaken at both compile-time and run-time. They forgo completeness and thus correctness to ensure ease of adoption. Black Box testing based approaches are currently undertaken manually but offer many of the advantages of compile-time component based techniques without additional syntax. However, UoM information will then be embedded within the unit tests and not part of the code base.

UoM annotations are initially costly for the developer but relatively stable to program reorganisation. Refactoring will rarely require UoM annotation changes unless the underlying data structures are also modified, thus ensuring maintainability and scalability within potentially safety-critical code. Approaches that leverage the compiler to optimise unit conversions without boxed values, such as native support or static lightweight solutions, will ensure greater numeric accuracy.

We can summarise the benefits and drawbacks of each approach in Table 1 using some of the key factors that we have focused on. Presented in this manner it becomes clear that static UoM Libraries offer few compelling advantages over Component or Black Box based testing. Native language support can offer distinct advantages as the checking will have greater coverage and can be equational, thereby resolving unit variables, with the added benefit of compiler optimisations to reduce round-off errors and increase runtime performance. Although, complex modern software tends to favour more lightweight solutions that integrate seamlessly into existing eco-systems.

Beginning with a Software Model that includes UoM information we have attempted to show the various tradeoffs involved in deciding how best to support their implementation when taking their software ecosystem into account. Stand-alone safety critical applications, where all unit information will be known and supported at compile-time, are very different to the needs of a rapidly evolving on-line application that expects to deal with varying UoM input at run-time.

## REFERENCES

Ali, S., Hemmati, H., Holt, N., Arisholm, E., and Briand, L. (2010). 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.

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

Antoniu, T., Steckler, P. A., Krishnamurthi, S., Neuwirth, E., and Felleisen, M. (2004). Validating the unit correctness of spreadsheet programs. In *Proceedings of Software Engineering*, ICSE '04, pages 439–448, Washington, DC, USA. IEEE Computer Society.

Apple (2020). Swift open source. Online https://swift.org. Last Accessed on 15th April 2020.

Bekolay, T. (2013). A comprehensive look at representing physical quantities in python. In *Scientific Computing with Python*.

Bennich-Björkman, O. and McKeever, S. (2018). The next 700 Unit of Measurement Checkers. In *Proceedings of Software Language Engineering*, SLE 2018, page 121–132, NY, USA. Association for Computing Machinery.

Bureau International des Poids et Mesures (2019). SI Brochure: The International System of Units (SI), 9th Edition, Dimensions of Quantities. Online https://www.bipm.org. Last Accessed 15th April, 2020.

Burgueño, L., Mayerhofer, T., Wimmer, M., and Vallecillo, A. (2019). Specifying Quantities in Software Models. *Information and Software Technology*, 113:82 – 97.

Cavarra, A., Crichton, C., Davies, J., Hartman, A., Jeron, T., and Mounier, L. (2002). Using UML for Automatic test Generation. *Proceedings of ISSTA*.

Chen, F., Rosu, G., and Venkatesan, R. P. (2003). Rule-Based Analysis of Dimensional Safety. In *RTA*.

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

Damevski, K. (2009). 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. ACM.

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

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

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

Gibson, J. P. and Méry, D. (2017). Explicit modelling of physical measures: from Event-B to Java. In *International Workshop on Handling IMPlicit and EXplicit knowledge in formal system development*.

Hall, B. D. (2020). Software for calculation with physical quantities. In *2020 IEEE International Workshop on Metrology for Industry 4.0 IoT*, pages 458–463.

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

Hayes, I. J. and Mahony, B. P. (1995). Using Units of Measurement in Formal Specifications. *Formal Aspects of Computing*, 7(3):329–347.

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

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 15th April 2020.

Karr, M. and Loveman, D. B. (1978). Incorporation of Units into Programming Languages. *Commun. ACM*, 21(5):385–391.

Kasurinen, J., Taipale, O., and Smolander, K. (2010). Software Test Automation in Practice: Empirical Observations. *Advances in Software Engineering*, 2010.

Kennedy, A. (1994). Dimension Types. In Sannella, D., editor, *Programming Languages and Systems—ESOP'94*, volume 788, pages 348–362, Edinburgh, U.K. Springer.

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 Pattern Languages of Programs*, EuroPLoP '17, pages 9:1–9:20, NY, USA. ACM.

Lungu, M. (2008). Towards reverse engineering software ecosystems. In *2008 IEEE International Conference on Software Maintenance*, pages 428–431.

Mayerhofer, T., Wimmer, M., and Vallecillo, A. (2016). Adding uncertainty and units to quantity types in software models. In *Software Language Engineering*, SLE 2016, pages 118–131, NY, USA. ACM.

McKeever, S., Bennich-Björkman, O., and Salah, O.-A. (2020). Unit of measurement libraries, their popularity and suitability. *Software: Practice and Experience*.

McKeever, S., Paçaci, G., and Bennich-Björkman, O. (2019). Quantity Checking through Unit of Measurement Libraries, Current Status and Future Directions. In *Model-Driven Engineering and Software Development*, MODELSWARD.

Microsoft (2020). F# software foundation. Online https://fsharp.org. Last Accessed on 15th April 2020.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

Modelica (2020). Modelica and the Modelica Association. Online https://www.modelica.org. Last Accessed on 15th April 2020.

Mussa, M., Ouchani, S., Sammane, W., and Hamou-Lhadj, A. (2009). A Survey of Model-Driven Testing Techniques. *Proceedings - International Conference on Quality Software*, pages 167–172.

NIST (2015). International System of Units (SI): Base and Derived. Online https://physics.nist.gov/cuu/Units/units.html. Last Accessed October 2nd, 2019.

Ore, J.-P., Detweiler, C., and Elbaum, S. (2017a). 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. ACM.

Ore, J.-P., Detweiler, C., and Elbaum, S. (2017b). Phriky-Units: A Lightweight, Annotation-Free Physical Unit Inconsistency Detection Tool. In *Software Testing and Analysis*, ISSTA 2017, page 352–355, NY, USA. Association for Computing Machinery.

Ore, J.-P., Elbaum, S., and Detweiler, C. (2017c). Dimensional inconsistencies in code and ROS messages: A study of 5.9m lines of code. In *Intelligent Robots and Systems*, IROS, pages 712–718. IEEE.

Ore, J.-P., Elbaum, S., Detweiler, C., and Karkazis, L. (2018). Assessing the Type Annotation Burden. In *Automated Software Engineering*, ASE 2018, pages 190–201, NY, USA. ACM.

Salah, O.-A. and McKeever, S. (2020). Lack of Adoption of Units of Measurement Libraries: Survey and Anecdotes. In *Proceedings of Software Engineering in Practice*, ICSE-SEIP '20. ACM.

Selic, B. (2015). 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.

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. Last Accessed on October 1st, 2019.

TIOBE (2020). The importance of being earnest index. Online https://www.tiobe.com/tiobe-index/. Last Accessed on 1st of October.

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

Witkin, R. (1983). Jet's Fuel Ran Out After Metric Conversion Errors. *The New York Times*.