Expressing Measurement Units in Interfaces for Scientific Component Software

Kostadin Damevski
Virginia State University
1 Hayden Dr.
Petersburg, Virginia, 23806, USA
kdamevski@vsu.edu

ABSTRACT

Scientific computing programs rarely use measurement units to express quantities, even though such units are commonplace in pencil and paper calculation. Tools that enable expressing measurement units and enforce dimensional safety have been made available in popular programming languages for some time, but they still lack a significant following. We posit that scientific programmers should not be burdened by units at each statement in their programs, but that units should be inserted in software component interfaces. This coarser grain approach enables the prevention of errors that occur when components written by different teams are composed, which we believe is a major cause of dimensional errors. We present our design of adding measurement units to interfaces in the CCA component model, which is targeted at the scientific computing community, in order to provide dimensional safety, an extensible set of measurement units and automatic conversion between commensurable quantities.

Categories and Subject Descriptors

D.2.12 [**Interoperability**]: Interface definition languages

General Terms

Design, Reliability

Keywords

Units, Dimensions, Runtime Checking

1. INTRODUCTION

Ensuring the consistency of software with respect to measurement units (e.g. meters, seconds, etc.), also known as dimensional analysis, is a topic that has received research attention for a number of years. Contrary to the availability of software tools, using measurement units is not common practice in scientific computing or other areas of computer science. We believe that the reason for the inability of tools that provide this capability within existing languages (e.g. in C++ [4], Java [3], Ada [7], Pascal [6] etc.) to gain a solid foothold in the scientific computing community is that they are too constraining and may present a significant runtime performance overhead. Individual programmers tend not to make such trivial mistakes as dimensional mismatch at the small scale, and adding unit descriptions to each program line can be tedious and adversely impact performance. Instead, dimensional mismatch errors tend to occur as mistaken assumptions between two programmers or two teams: NASA's Mars Orbiter crashed into the surface of Mars in 1999 due to a software navigation error traced to one team using English units while another used metric units [8]. In this work, we propose to integrate dimensional analysis at the software architecture level - in software components, where individual components are often implemented separately and composed to form an application.

Component technology is becoming increasingly popular for large-scale scientific computing in helping to tame the software complexity required in coupling multiple disciplines, multiple scales, and/or multiple physical phenomena. The Common Component Architecture (CCA) [1] is a component model that was designed to fit the needs of the scientific computing community by imposing low overhead and supporting parallel components. CCA has already been used in several scientific domains, creating components for large simulations involving accelerator design, climate modeling, combustion, and accidental fires and explosions [10]. As a component model that targets scientific computing, where computations are often dimensional, the CCA provides an opportune place to address the problem of dimensional safety.

Interfaces define the boundary between a provider of a particular functionality and its users. Intended to encapsulate implementation details of the providing component, interfaces are often simple and often lack enforceable semantic information about their use, which leads to their misuse by users that may not have read the accompanying documentation. A component implementer wants to express semantic information in the interface (more than just integer and floating point method parameters) in order to specify the characteristics of his or her implementation. In this way, an interface that solves a particular computational problem for meters would differ from one that solves the same problem for miles. In the state of the art in component software these two interfaces would be alike, leading to difficulty in selecting the proper component. To address these problems,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CBHPC'09, Nov. 15-16, Portland, OR, USA
Copyright 2009 ACM 978-1-60558-718-9/09/11 ...$10.00.

we propose adding measurement unit information in the interface in order to enforce unit consistency in component composition and reduce dimensional mismatch errors. As a proof of concept, we build a system to specify measurement units into the CCA component model. To our knowledge, this is the first attempt to add measurement units at the software architecture level. Our objectives in designing a system to express and enforce interface level dimensions are to:

1. Clearly express measurement unit information in interfaces.

2. Components that use a particular interface (port) should be able to express unit information programatically in order to interact with an interface containing units.

3. Automatically convert commensurate quantities (e.g. miles into meters).

4. Provide a convenient way to express derived units (such as meters per second) and to extend the system with new units.

The first two objectives enable specifying dimensions in interfaces and selecting interfaces based on this information. It is important to separate these two since the mechanism for the first one is to embed unit information in the interface, while the interface user, in the second objective, needs a selection system. This selection system, for instance, may take the form of a location service, use the parameters of a method invocation to specify measurement units, or both. Objectives 3 and 4 add value to the system design by enabling automatic unit conversion to be performed by the system, once commensurability between the units has been established, and by providing extendability to new units.

We organize the discussion of our dimensional analysis system as follows. Section 2 contains related work and background of the problem. In Section 3 we consider three possible, but unsatisfactory, approaches to expressing measurement units in the CCA component model, while in Section 4 we show a detailed view of our measurement unit design. Finally, we finish with conclusions and future work of the project in Section 5.

2. BACKGROUND AND RELATED WORK

Le Systeme International d'Unites (SI) is an international standard that describes dimensions and their measurement units. The SI standard is based on seven mutually independent dimensions: *mass, length, time, electric current, thermodynamic temperature, amount of substance, and luminous intensity*. Each dimension has a corresponding base unit: *kilogram, meter, second, ampere, kelvin, mole and candela*. The SI system also recognizes 22 derived units, which often span several dimensions, such as meter per second squared for acceleration or a *newton* (kilogram times meter per second squared) to measure force. Factors of existing units are used as prefixes in constructing new units, such as grams to represent 10^{-3} kilograms and milligram to equal 10^{-3} grams and 10^{-6} kilograms. Many common internationally used units are outside of the SI, such as the liter, minute, hour, and others.

The large number of measurement units adds a level of complexity to the problem of dimensional analysis as it is difficult to determine compatibility (commensurability) between two given units. Other measurement systems that are commonly used, such as the English system (e.g. miles, pounds and ounces) further increase the number of units that are represented in a software system that performs dimensional analysis. A popular C++ template based library for dimensional checking, called SIUnits [4], uses the SI system's seven base dimensions to classify all possible measurement units. Each measurement unit is encoded by one rational number per dimension. For instance, the acceleration unit *meters per seconds squared*, can be expressed as 1 in the length column and -2 in the time column (see Table 2). In this way, commensurability between two given units can be easily computed by comparing their dimension vectors.

The work of Allen et al. [3] presents an extension to the Java language to support measurement units that provides intuitive ways of expressing operations involving measured quantities. By introducing several new constructs in the Java language (such as meta-classes, instance classes, abelian classes and others) the authors build an extensible method of describing dimensions and converting commensurable quantities that relies heavily on static checking and exhibits low performance overhead. One of the main challenges addressed by this work is to be able to treat measured quantities as both values and types in a programming language in order to maximize static checking, and to be able to allow the user to specify new units. The approach also designates a set of primary units and uses them as an intermediate value in converting between two non-primary units of the same dimension.

Not all previous work on this subject has taken the approach of modifying existing programming languages with measurement unit support. A recent approach to providing dimensional analysis to the BC language by using the Maude rewriting meta-language is presented in [5]. BC is a calculator language common in Unix platforms to which the authors add annotations in the form of comments. These annotations are interpreted by Maude, and used to perform dynamic (run-time) or static (compile-time) analysis. The dynamic analysis uses Maude's rewriting capability, but is ill-suited for production code as it increases runtime by an order of magnitude. The static analysis seems promising in scenarios where the BC program is simple enough for this analysis to converge. In order to reduce the amount of tagging needed, the authors use a locality principle, where Maude assumes that the programmer is doing the right thing, with respect to dimensional safety, at the scale of several lines of code.

UDUnits is a units library that enables unit conversion through an Application Programming Interface (API). Applications using UDUnits have to make explicit use of this API, supplying unit and quantity information in string or binary form. String data has to be converted into binary before conversion between commensurate units can be performed by this system.

The Fortress [2] programming language specification includes support for statically checked dimensional units. At the time of writing, the implementation of the language does not yet contain this feature.

Our work was influenced by the language level approaches mentioned above, especially the classification of units based on the seven base dimensions presented in SIUnits, the object-oriented design philosophy of the Java-based approach by

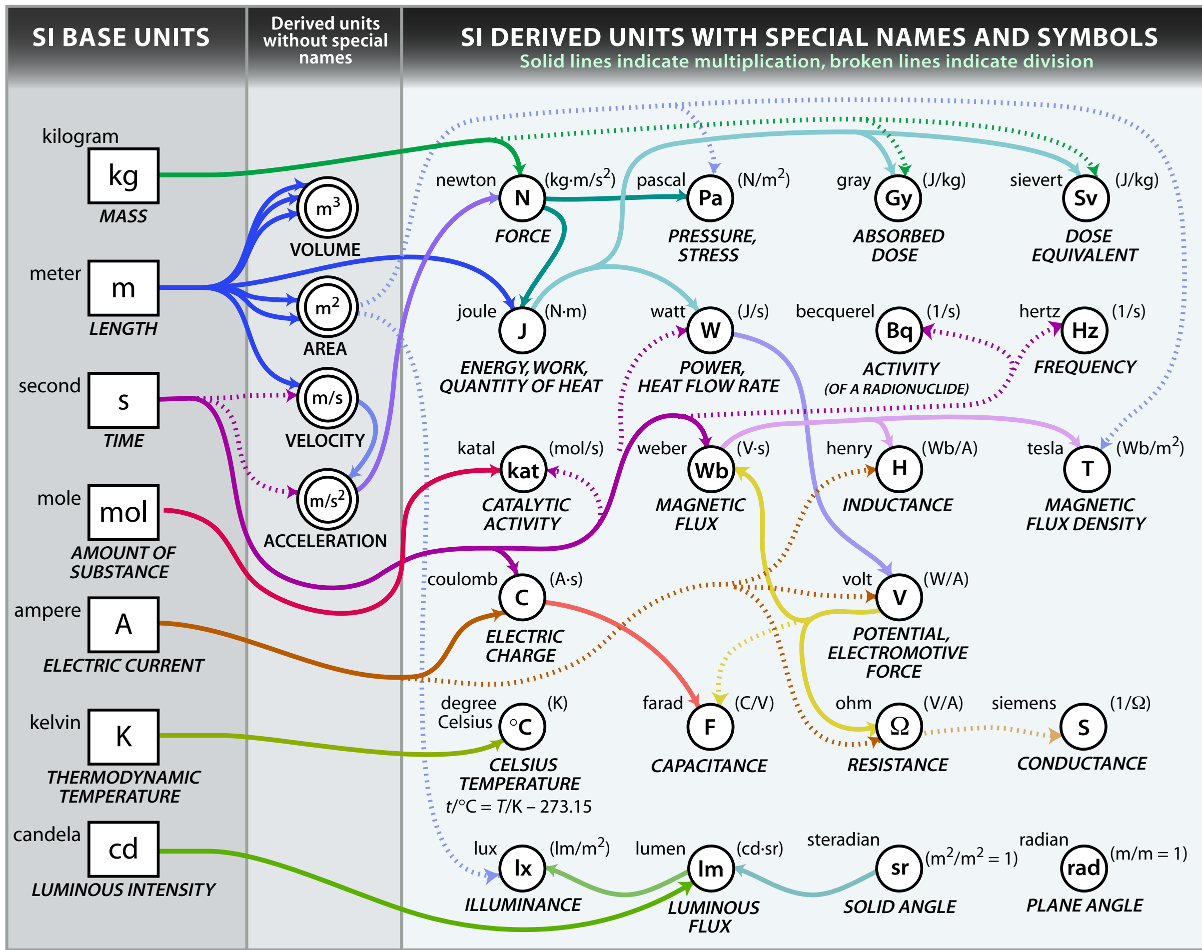

Figure 1: Diagram of relationships among the SI units. (Image: National Institute of Standards and Technology, http://physics.nist.gov/cuu/Units/SIdiagram.html)

Allen et al., and the locality principle of the work using the Maude rewriting system. Compared to programming language level dimensional safety, our system's design is made simpler by not being concerned about arithmetic operations using units, which presents a significant reduction in complexity. However, much of the other issues in representing units and determining commensurability are similar, while other complications arise by the need of component users (or clients) to express their unit requirements in component interaction.

A CCA component consists of one or more ports, and a port is a group of method-call based interfaces. There are two types of ports: **uses** and **provides**. A provides port (or callee) implements its interfaces and waits for other ports to call them. A uses port (or caller) issues method calls that can be fulfilled by a type-compatible provides port on a different component. A CCA port is represented by an interface, which is specified through the Scientific Interface Definition Language (SIDL). A SIDL specification is compiled into glue code that is later compiled into an executable together with the user-provided implementation code. The prevalent way of compiling SIDL is by using the Babel compiler [9]. Babel has the capability of compiling SIDL to bindings for several popular programming languages (C++, Java, Python, Fortran77 and Fortran95), which allows creating applications by combining components written in any of these languages. Babel has a large and growing user community and has become a cornerstone technology of the CCA component model.

3. POSSIBLE APPROACHES

In designing our system to enable the expression of measurement units at the interface level, a concern is to avoid changes to SIDL and to minimize changes to Babel. Incremental improvements to Babel are commonplace and have occurred continuously throughout the existence of the CCA. Before settling on a particular solution, next we explore the design space of adding measurement units to CCA by evaluating three design approaches that fail to meet all of our objectives (presented in Section 1). Our chosen design, presented subsequently, builds on many of the ideas presented in the three possible, but unsatisfactory, directions.

3.1 Approach One: Unit Tags

In current component systems, a developer would represent measurement unit quantities as variable types: integers, floats or doubles, providing no visible unit information. Our first approach in enhancing interfaces with mea-

unit	mass	length	time	amount of substance	electric current	thermodynamic temperature	luminous intensity
meter	0	1	0	0	0	0	0
candela	0	0	0	0	0	0	1
$meter/second^2$	0	1	-2	0	0	0	0

Table 1: Encoding units according to their seven SI base dimensions.

surement unit information is to augment interface parameter types with unit tags. The Babel compiler contains a mechanism to add tags within a SIDL interface without modifying its parser by using the *%attrib* keyword. This keyword specifies key-value pairs, that are processed by extensions to the compiler's backend in order to produce appropriate generated code. An example of this approach, using an interface that assigns and queries the earth's circumference in miles is given here:

```
class Earth {
  void setCircumference(in %attrib{unit=Miles}
                        double circumference);
  %attrib{unit=Miles} double getCircumference();
}
```

To enable this design, Babel would internally need to contain the set of all possible units and dimensions, perhaps encoded in an XML file or a similar in-memory or on-disk data structure. Expressing new units and commensurability of units would also need to be expressed within this data structure, and modifying and maintaining a Babel-internal data structure is not an ideal task for an end-user. In addition, this approach falls short of our specified goals in that, while it expresses the provides side of the interface properly, the uses side has no defined way of expressing its measurement units (as stated in objective #2). If, for instance, a user of this interface has the earth's circumference available in meters instead of miles, he or she has no way to specify this when invoking the *setCircumference* method.

3.2 Approach Two: Simple SIDL Classes

To remedy the problem with the previous approach, we need to devise an approach for interface users and providers to separately express measurement unit information. Since SIDL allows user-defined types or classes to be used as arguments to methods, our second approach is to build a SIDL class hierarchy of measurement units. In this approach, class is defined for each measurement unit. Each object of a particular unit class boxes (wraps) a quantity in that specific unit; the unit's quantity can be a class variable for which setter and getter methods are provided.

```
interface Unit {
  void setQuantity(in double q);
  double getQuantity();
}

class Mile implements-all Unit {}
class Second implements-all Unit {}
class Kilometer implements-all Unit {}

class Earth {
  void setCircumference(in Mile m);
  Mile getCircumference();
}
```

The above classes *Mile, Second,* and *Kilometer* are available to be used to specify the measurement unit needs of the component's user, which addresses our previous concern. New units can be declared by defining new SIDL classes for them. If using kilometers to specify the earth's circumference one would write (in pseudo C++):

```
Earth earth = Earth::_create();
KiloMeter circumference = Kilometer::_create();
circumference.setQuantity(40041);
earth.setCircumference(circumference);
```

At first glance, using classes to represent units is an acceptable design. However, a deeper look reveals that it is impossible to call the *setCircumference* method passing an instance of the *Kilometers* class, as attempted in the above snippet of code. This is because the Babel generated code only accepts instances of the Mile class as parameters to the *setCircumference* method. We want to enable passing commensurable units as arguments to methods (per objective #3), where the system will automatically convert the kilometers into miles before passing the value to the implementation. Therefore this approach fails short of our objectives.

3.3 Approach Three: Unit Base Class

An improvement to the second approach, based on the problem identified, is to use the *Unit* base class to generalize interfaces that use measurement units, instead of providing specific units. This would make it possible to pass any unit (all of which would inherit from the *Unit* base class) as a method parameter, enabling the system to determine commensurability and bypass Babel's strict interface type checking. An added complexity of this design is that each method's units would now need to be specified programatically in both the uses and provides sides. Once specified, the system would determine at runtime whether the two units mismatch or are commensurable. In order to determine commensurability between units, we opt to add a *getDimensions* method to the *Unit* base class. By comparing vectors of one rational number per SI dimension [1] (as defined by the SIUnits [4] library) we can determine whether two units are from the same dimension, even for complex derived units. The dimensional vector is encoded within the implementation of each unit class by implementing the *Unit* interface.

```
interface Unit {
  void setQuantity(in double q);
```

[1]For simplicity, in this paper we use integers to represent rational numbers. In practice this can be extended to by using an integer for each of the denominator and numerator.

```
  double getQuantity();

  array<int,1> getDimensions();

  bool isCompatible(in Unit impl);
}

class Mile implements-all Unit {}
class Second implements-all Unit {}

class Earth {
  void setCircumference(in Unit u);
  Unit getCircumference();
}
```

The implementation of the *Unit* class' *isCompatible* method above will contain invocations to the *getDimensions* for each unit and would perform a dimensional vector comparison to determine commensurability. In this case, where even the provider is forced to programatically specify its units, the implementation of the *setCircumference* method would contain the following (in pseudo C++):

```
void
Earth::setCircumference(const Unit& u) {

  if ( ! u.isCompatible(Unit::Mile::_create()) ) {
    //ERROR: incompatible units
    throw new IncompatibilityException();
  }

  //the rest of this method follows...
}
```

The uses side for this approach is similar to the previous one. Although this approach satisfies nearly all of our objectives, it puts an undue burden on the interface implementer to ensure dimensional safety. In addition, the readability of an interface is reduced, as one needs to read the implementation and not just the SIDL definition to determine the correct measurement unit.

4. SYSTEM DESIGN

The progression of the approaches discussed up to this point culminate in the design that we chose and describe in this section. The system design we present, as is the case with the approaches discussed before it, would operate across the number of programming languages Babel supports. In order to enable measurement units to be conveniently expressed in component interfaces, we opt to enhance the Babel SIDL compiler to allow certain dynamic type checking decisions for user-defined (class) types to be performed by a user-supplied method. This is a principled addition to Babel, loosening the type checking constraint in order to properly compare and determine compatibility between measurement unit classes. As the compiler modification we introduce is not ad-hoc, it can possibly be used for other purposes outside of expressing measurement units. To realize this, we modify Babel by adding a *DynamicTypeChecked* interface to the compiler's standard runtime and modifying the code generation of classes that implement this interface. When encountering classes that extend the *DynamicTypeChecked* interface as parameters to methods, the Babel generated stubs are automatically generalized to accept any class instance (using Babel's *BaseClass* metaclass) and to invoke the *isTypeCompatible* method to determine compatibility at runtime. This method may also perform a type conversion, where the type of the class tested for type compatibility can be changed; in which case *isTypeCompatible* method's *newClass* out parameter returns a non null value. Figure 2 displays a flowchart of the dynamic type checking our system introduces in Babel.

```
package sidl {
  interface DynamicTypeChecked {
    bool isTypeCompatible(in BaseClass class,
                          out BaseClass newClass);
  }
}
```

Using the *DynamicTypeChecked* interface defined in Babel's runtime, and the associated changes in code generation we can define a *Unit* abstract base class, inherited by all actual measurement unit classes (e.g. *Meter*, *Second* etc.). The *Unit* class can now be written so that it evaluates the dimensional compatibility between measurement units and enables passing instances of class *Meter* to a method accepting instances *Mile*, while performing an automatic conversion in the process. Implementing new units, *Ampere* for instance, requires defining a new class and implementing the *getDimensions* method. As previously mentioned, this method describes each unit's dimension using the SI system's seven base quantities.

```
abstract class Unit implements
                    sidl.DynamicTypeChecked {

  bool isTypeCompatible(in BaseClass class,
                        out BaseClass newClass);
  void setQuantity(in double q);
  double getQuantity();

  abstract array<int,1> getDimensions();
}

class Meter extends Unit {}
class Second extends Unit {}
```

Since the *Unit* class is the one implementing the *DynamicTypeChecked* interface, the *isTypeCompatible* method can be implemented once for all measurement units. This enables using and defining measurement units in our system to be decoupled from the internals of the Babel compiler. We give a general idea (in pseudo C++) of the implementation of this method below:

```
bool
Unit::isTypeCompatible (
  /* in */::sidl::BaseClass& klass,
  /* out */::sidl::BaseClass& new_klass )
{
  //get klass' type name
  sidl::ClassInfo klsinfo = klass.getClassInfo();
  std::string klsName = klsinfo.getName();

  //extract dimensional arrays
  dims::Unit unit = babel_cast<dims::Unit>(klass);
```

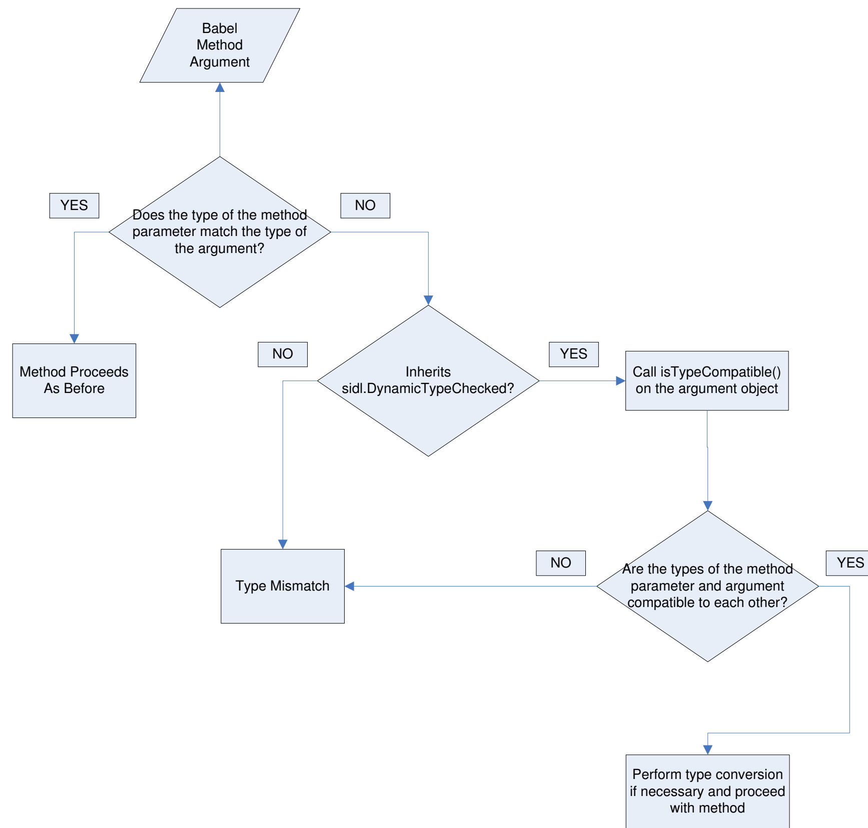

Figure 2: A flowchart describing the process of dynamically checking and converting babel method arguments. This is the general approach that we devise and apply to unit checking.

```
  sidl::array<int> klsdims = unit.getDimensions();
  sidl::array<int> d_dims = this->getDimensions();

  //compare dimensional arrays
  bool isDimsSame = true;
  for(int i=0; i<7; i++) {
    if(d_dims.get(i) != klsdims.get(i)) {
      isDimsSame = false;
    }
  }

  if(isDimsSame) {
    //dimensions are the same, so can convert if
    //these are different commensurable units
    //...

    return true;
  }
  else {
    //units aren't commensurable

    return false;
  }
}
```

Using the *Unit* class' *isTypeCompatible* method, we are able to provide a general implementation of unit conversion and commensurability. The user of an interface providing measurement units can now define his or her own measurement units and use them to invoke an application interface's methods. We give an example of performing these tasks using our system through the running example that uses the earth's circumference.

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

After compiling the above SIDL using the modified Babel compiler, we can write the following code (in C++) that uses the *Earth* interface. Commensurable units are automatically converted: kilometers to meters and meters to miles.

```
Earth earth = Earth::_create();
KiloMeter kmCirc = Kilometer::_create();
kmCirc.setQuantity(40041);
earth.setCircumference(kmCirc);
Mile mileCirc = earth.getCircumference();
```

The implementation (server-side) code remains largely unchanged by our measurement unit system.

```
void
Earth::setCircumference(
  /* in */Meter& circumference )
{
```

```
  double c = circumference.getQuantity();
  this->d_circ = c;
}
```

One aspect of this system's design we have neglected to describe is the process through which we convert between two units of the same dimension. We describe our solution to this problem in the next section.

4.1 Unit Conversion

Converting for one commensurable unit to another is one of the important features of our approach. Selecting a primary unit for each dimension in order to reduce the amount of conversions that need to be encoded is a reasonable solution [3]. To enable this solution to unit conversion in the system we designed, we two methods *inPrimaryUnit* and *setInPrimaryUnit* to the *Unit* baseclass.

```
double inPrimaryUnit();
void setInPrimaryUnit(in double d);
```

The *inPrimaryUnit* method returns the quantity of a unit as a value in its primary unit, while the *setInPrimaryUnit* sets the quantity from a parameter whose value is in the primary unit. Both methods perform a conversion between the represented unit and the primary unit. In the primary unit object, we implement these two methods by performing no conversion to the quantity.

Determining the primary unit for each dimension is done by the implementer of the unit library, and many of them are pre-determined by the SI system. While using a primary unit simplifies conversion, it raises a new concern. Numerical errors due to overflow may occur as a side-effect to the unit conversion steps. We present no systematic solution (if one is possible) to them apart from informing the user of their possibility. If there is loss in precision, the user may choose to perform the unit conversion directly, without the involvement of a primary unit.

4.2 Other Challenges and Limitations

Our system introduces extra overhead in method invocations that use our measurement unit representation. Though relatively minimal compared to approaches providing units in programming languages, this overhead can grow when a method is invoked very frequently in a particular application. We present no solution to this problem in this paper, though we believe that the overhead of our design will be negligible in most uses.

Some units are difficult to express in terms of the seven SI base quantities. An example of such a unit is *radians*, which are used to measure a plane angle and are dimensionless. In such cases, we usually do not need commensurability, as these units' dimensions are unique. To define a uncommensurable unit, we use zeros across all of the seven SI base quantities. On the other hand, if a user should want to express a class of units that needs commensurability and cannot be easily expressed by the SI system, we propose extending our design or directly modifying its implementation to account for this.

5. CONCLUSIONS AND FUTURE WORK

In this paper we discussed our design of measurement units expressed in component interfaces. We argue that expressing units at the software architecture layer offers the right trade-off between discovering dimensional mismatch and application performance. Our design consists of loosening dynamic type checking requirements in order to automatically determine unit commensurability and provide automatic unit conversions in a flexible manner. We implement our design in the CCA component model's BABEL compiler in a way in which minimal effort is required to add new units, and even smaller changes are needed to use units in application code.

The future work of this project involves determining whether our design works well in practice, once it is used to program a scientific application. We would like to gage user experience as well as any potential impact on application performance. In addition, we would like to experiment with approaches to system transparency and accountability in unit conversion.

6. REFERENCES

[1] Allan, B. A., Armstrong, R., Bernholdt, D. E., Bertrand, F., Chiu, K., Dahlgren, T. L., Damevski, K., Elwasif, W. R., Epperly, T. G. W., Govindaraju, M., Katz, D. S., Kohl, J. A., Krishnan, M., Kumfert, G., Larson, J. W., Lefantzi, S., Lewis, M. J., Malony, A. D., McInnes, L. C., Nieplocha, J., Norris, B., Parker, S. G., Ray, J., Shende, S., Windus, T. L., and Zhou, S. A component architecture for high-performance scientific computing. *Intl. J. High-Perf. Computing Appl. 20*, 2 (May 2006), 163–202.

[2] Allen, E., Chase, D., Hallett, J., Luchangco, V., andSukyoung Ryu, J.-W. M., Jr., G. L. S., and Tobin-Hochstadt, S. The Fortress language specification, version 1.0 beta, 2008.

[3] Allen, E., Chase, D., Luchangco, V., Maessen, J.-W., and Steele, Jr., G. L. Object-oriented units of measurement. In *Proceedings of the 19th Conference on Object-Oriented Programming, Systems, Languages, and Applications* (October 2004).

[4] Brown, W. E. Applied template metaprogramming in siunits: the library of unit-based computation. In *Proceedings of the Second Workshop on C++ Template Programming* (October 2001).

[5] Chen, F., Rosu, G., and Venkatesan, R. P. Rule-based analysis of dimensional safety. In *Proceeding so the 14th International Conference on Rewriting Techniques and Applications RTA-03* (Valencia, Spain, June 2003).

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

[7] Hilfinger, P. N. An ada package for dimensional analysis. *ACM Transactions on Programming Languages and Systems 10*, 2 (1988), 189–203.

[8] Isbell, D., and Savage, D. Mars climate orbiter failure board release report, numerous nasa actions underway in response. NASA Press Release, 1999.

[9] Kohn, S., Kumfert, G., Painter, J., and Ribbens, C. Divorcing language dependencies from a scientific software library. In *Proceedings of the 10th SIAM Conference on Parallel Processing* (Portsmouth, VA, 2001).

[10] McInnes, L. C., Allan, B. A., Armstrong, R., Benson, S. J., Bernholdt, D. E., Dahlgren, T. L., Diachin, L. F., Krishnan, M., Kohl, J. A., Larson, J. W., Lefantzi, S., Nieplocha, J., Norris, B., Parker, S. G., Ray, J., and Zhou, S. Parallel PDE-based simulations using the Common Component Architecture. In *Numerical Solution of PDEs on Parallel Computers*, A. M. Bruaset and A. Tveito, Eds., vol. 51 of *Lecture Notes in Computational Science and Engineering (LNCSE)*. Springer-Verlag, 2006.