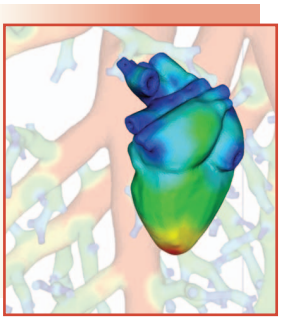

Error Detection and Unit Conversion

Automated Unit Balancing in Modeling Interface Systems

BY HOWARD JAY CHIZECK, ERIK BUTTERWORTH, AND JAMES B. BASSINGTHWAIGHTE

Computer models of physiology are based on physical laws governing chemical reaction kinetics, magnetic fields, hydrodynamics, ion fields, charge transfer, and other phenomena. Equations that instantiate these models represent the relationships among time, mass, distance, force, pressure, chemical concentration, and potential difference. In the Systeme Internationale (SI) system, the seven base units are ampere, candela, kelvin, kilogram, meter, mole, and second. The units may be fundamental, as in SI units, or derived, as with velocity (m/s) or pressure in its varied forms (e.g., 1 mmHg = 1.00000014 torr = 1.33.322 Pa = 1333.22 $g \cdot cm^{-1} \cdot s^{-2}$). Pressure may also be expressed as kilopascals, atmospheres, bars and microbars, pounds per square inch, etc. Models often refer to experimental values expressed in units that are convenient to use during the collection of data and vary in form or in the system of units that is used.

For the security in computation, one should be able to describe data and write equations using the habitually used units for a particular variable and to have the system guarantee correct interpretation and conversion from one form to the form used for the computation. Unfortunately, standard computational packages have historically lacked a means of unit balance checking, even for space missions. A classic example is the Mars Climate Orbiter mission planning [1], where one team used metric units and the other did not, leading to a miscalculation that put the spacecraft in too low an orbit. It crashed.

Most mathematical models published in peer-reviewed journals are not reproducible: they contain the authors' errors of commission and omission, augmented by the errors introduced by editors and typesetters. Therefore, an exactly reproducible model is a rarity. Modeling in cardiac electrophysiology has set a high standard of reproducibility in the works of Hodgkin and Huxley [2], Beeler and Reuter [3], Winslow et al. [4], Michailova and McCulloch [5], all of which can be reproduced, figure by figure, from the articles. The latter two were vetted in advance of publication by reviewers, who iterated with authors to assure that the computer code matched the equations appearing in the article. In all of these studies, the published equations had balanced units, which we define as having the same units on the two sides after simplification.

Digital Object Identifier 10.1109/MEMB.2009.932477

The modeling of biological systems requires validation to assure the reader that the model is a good representation of the real biological or mechanical system, even though it is necessarily a simplification. This means that there is an element of arbitrariness in each of the equations and that there is no easy way to affirm the correctness of the derivations or the assumptions on which they are based. In addition, there is a need to verify the code used to embody the mathematical model. This includes definitions of variables with their units. Articles in the mathematics literature are easier to compare, as the development of a theorem can be reviewed exactly.

Given that every biological or physicochemical model is a hypothesis defining a concept rather than an exact reality, reproducibility is critical to the advancement of science: a hypothesis disproved is a stepping stone toward an improved hypothesis; i.e., toward a better model. The goals of publication in science are manifold, but idealistically, the central one is to make the advancements known exactly so that they can be challenged and built upon.

To this end, we now see the development of model databases and of tools for modeling. What we would expect of databases such as CellML and Biomodels (these are archives of published models using ordinary differential equations in physiology and systems biology), and of our National Simulation Resource (NSR) Physiome site, is that the models should be reproducible and correct. In curating published models to make them available for public dissemination on the Biomodels and CellML databases, the vast majority have needed correction by the curators; the published articles contained errors by the authors, editors, and typesetters. Although errors can obviously be introduced in the process of putting models into a markup language, there is no doubt that the curation teams are raising the standards for model archiving.

The ideas discussed here are not new, having been formally introduced in Buckingham's classic article [6] and captured in Wittgenstein's *Tractatus Logico-Philosophicus* [7], a work begun in 1914. Both were cognizant of earlier uses of dimensionless variables such as the Reynold's number for characterizing fluid flow in tubes [8], as reviewed by Sterrett [9]. In 1941, Gagge et al. [10] introduced a system of units for use in making measurements for thermal exchange in physiological systems. Pappenheimer [11] collated a system for respiratory

 0739-5175/09/$25.00©2009IEEE

Standard computational packages have historically lacked a means of unit balance checking.

physiological terminology, including units, in 1950. A 1986 terminology for transport phenomena in physiological systems [12] extended the nomenclature and emphasized consistent physical and chemical units, allowing for different standards for units and their interconversion.

In 1978, Karr and Loveman [13] were the first to directly address the difficult but highly desired capability to incorporate units into programming languages. Their work is an early example of the type of operations described in this article. Gruber and Olsen [14] proposed an ontology for engineering models that addresses many of the underlying philosophical issues involved in unit balancing. Novak [15] presents elegant methods of automated unit conversion and unit balance checking.

Any mathematical modeling language (MML) can be summarized as a collection of variable declarations and mathematical equations using those variables. Variables should be assigned units as they are declared. Variables with no assigned units are initially inferred to be dimensionless but may later be inferred to have particular dimensions from the context of equations in which they are used. The tasks of interest in automated unit matching involve the following:

1) *Equation balance checking*: Equations containing improper unit arithmetic (e.g., $m/s + m/s^2$) should generate error messages (for manual correction).
2) *Unit scaling factors*: These should be inserted into calculations as needed (e.g., substituting 1 mm/s for 6 cm/min to match the two sides of the equation). This we call *unit conversion*; it is automated.
3) *Unit inference*: Model variables and constants whose units were not specified should be assigned units based on context wherever possible. For example, if A has been assigned the unit cm, then when the expression (A+B) appears, it implies that A and B have compatible units, and thus B is assigned the unit cm. Although this strategy appears to work well, it is not without risk. For example, if B's units have been inferred, then they may be in error. The best practice is therefore to assign units to all of the variables.

Each of these automated unit matching tasks is described later in greater detail.

An automated system for error detection and unit conversion in a MML should meet the following goals.

1) User specification of units should be simple, intuitive, and unambiguous.
2) Variants from the basic SI and centimeter gram second (CGS) units such as commonly used units for physiology and systems biology must be predefined in terms of SI or CGS units.
3) Modelers may define new units or abbreviations if desired.
4) Equations with unbalanced units must be detected and an error message generated.
5) Equations with compatible but dissimilarly scaled units (e.g., cm/min versus mm/s) should be detected and unit conversion factors should be inserted into the computations. Any change in a variable's unit assignment (such as cm to m) should cause computations to rescale correctly.
6) In the near term, the modeling interface system should accommodate existing models that do not use automated unit balance checking; i.e., the unit balance checking should be optional (until modeling standards are adopted generally).

Unit Balance Checking

Unit balance checking is the first of a series of checks for achieving correctness and validity of models. It is the simplest and much easier method that achieves balance of mass, energy, or charge. It precedes the steps of validation of the model against the observed data and is a prerequisite for them; there is no point validating an incorrect model. A checklist for standards can be found at http://www.physiome.org/standards. Failure to balance units means that the equations are erroneous. One source of the error is that units may not have been specified so that they were inferred from context. The context may be ambiguous or an inference may depend on units defined by a previous inference. Since the specification of units is the duty of the original authors, there is no good rationale for the failure to define the units for parameters and variables, nor for physical or chemical constants. Another source of error is the use of conflicting systems of units. Methods to address these issues are described by Chen et al. [16] in writing about rule-based dimensional safety; they defined more than 2,000 rewriting rules in the Maude and BC programming languages to achieve their objective, which included the coding of complex matrix inversions. This is far greater than the number of rules used in the Java simulation (JSim) unit balancing and conversion system (which is 14; see Table 5). This suggests that their work covers a broader range of situations.

In complicated models, balancing units manually in equations with many terms is time consuming. Constructs such as $(m/s + m/s^2)$ are inherently erroneous, and though they will be detected by a unit balance check, they must still be corrected by the programmer. In standard programming languages like FORTRAN, C, and MATLAB, when the programmer uses centimeters in one equation and meters in another, this usually involves the insertion of numeric unit conversion factors into the model code. It is difficult to find even overt errors, because these languages have no provision for unit balance checking. It is also a nuisance to have to convert everything to a common base system of fundamental units, especially when the experimental observations are best remembered in the format in which they have been acquired. For correctness in equations, each group of terms to be summed in a particular equation must

Most mathematical models published in peer-reviewed journals are not reproducible.

have identical units, and units to the left of an equal sign must be the same as those on the right.

The incentives to automate unit balance checking in computer programs include speed and reproducibility. Automated unit balance checking provides a substantial acceleration in model code development. The development of standards for the performance and reproducibility of algorithms for computer modeling and simulation is in its infancy. It is likely that future innovations will continue to improve the speed of model code development.

What we show here is that unit balance checking can be automated. Although our example implementation is under a specific simulation system, JSim (http://www.physiome.org/jsim/), the program to automate the unit balance checking is general and can be applied to other systems in which units can be specified and checked. Although JSim was apparently the first simulation system that did this (in its 1999 release), others have implemented similar improvements to numerical simulation packages. For example, the caretakers of CellML [17], [18] require units on all variables, allow no inference of units, and they are implementing unit balance checking [19] in an ODE-based simulator Physiome CellML Environment (PCEnv; http://www.cellml.org/tools/pcenv/). The methodology we describe here exploits that fact that unit balancing follows rigid logical rules and can thus be automated. Automated unit balance checking not only reduces errors in individual equations, but by alerting the model builder to imbalances it also helps to identify any conceptual errors in their formulation.

Unit Compatibility and Scaling Factors

Two units are said to be compatible if one can be converted to another via a dimensionless scaling factor. For translating millimeters per minute (mm/min) to centimeters per second (cm/s), the scaling factor is 0.1 cm/mm ÷ 60 s/min or simply 0.1/60, since the components cm/mm are both length measures and s/min are both time measures and are therefore compatible. However, moles per gram (mol/g) and moles per liter (mol/L) are not; (mol/g) × (g/mL) would be compatible with mol/L.

The scale factor is used for converting compatible units such as cm/s and mm/min. The dimension vector whose length is equal to the number of basic units used in the particular model is used for determining compatibility, and it may be more of less than the seven fundamental units since some may be derived units like pressure or flow. Two units are compatible if their dimension vectors are identical.

Scaling invariance is the property of a mathematical modeling system, such that numeric calculations remain correct under changes of the scale of the units such as millimeter versus meter. Consider a variable V that is specified in volts when a model is first developed but is changed to millivolts later to conform to usage for a particular application. A scaling invariant modeling system responds by adjusting all calculations using V to be correct under the new scaling. This invariance is also a desirable property for component-based model building, where new models are built from a set of simpler modules previously constructed. If variables in the composite model are represented at different scales (e.g., millivolts, microvolts) in the original component models, then a scaling invariant modeling system reconciles them automatically. This capability thus improves prospects for retaining modularity when developing multiscale models.

Inferring Units Defined Implicitly in Equations

Ideally, there should be no need for inferring units. The need to do so arises, because when taking ratios or other mathematical combinations of quantities having well-defined units, it is convenient to derive the units for the resulting products. Automatic assignment carries risk; however, if there are multiple assignments of units to be made in a given equation, it is essential to remove the ambiguity that arises from excessive degrees of freedom.

It is best practice to assign units to all variables and parameters, including those of computed quantities of interest. The model archiving markup language CellML [17], [18] requires unit specification for variables. The systems biology markup language (SBML) [20], [21] allows unit specification, and it does permit inference of units from context, but does not require this. It should be noted that the practice of designating variables as dimensionless when they are really dimensioned quantities precludes unit balance checking. The choice here is a practical tradeoff between user convenience, conciseness, and safety.

The JSim Modeling and Analysis Interface System

We describe here a simulation system that accomplishes the aforementioned design goals and tasks. It represents one realization of an automated unit balance system. JSim [22], [23] is a Java-based [24] simulation system for building quantitative numeric models and analyzing them with respect to experimental reference data. JSim was developed primarily for generating model solutions for use in designing experiments and analyzing data in physiological and biochemical studies, but its computational engine is general and equally applicable to solving equations in physics, chemistry, and mechanics. JSim has been under development at the NSR for mass transport and metabolism since 1999. JSim uses a model specification language, MML, which supports ordinary and partial differential equations, implicit equations, integrals, summations, discrete events, and allows calls to external procedures. JSim's compiler translates MML into Java code in which the numeric results are calculated. Within the JSim graphical user interface (GUI), users adjust parameter values, initiate model runs, plot data, and perform behavioral analysis, sensitivity analysis, parameter optimization for curve fitting. Alternatively, one can use JSim's command line interfaces (`jsbatch` and `jsfim`). JSim's capabilities are more advanced than previous NSR software systems such as

Unit balance checking is the first of a series of checks for achieving correctness and validity of models.

SIMCON [25] (for simulation control) and XSIM [26] (for X-terminal operation). JSim source code, binaries (for Windows, Macintosh, and Linux), documentation, and the defining file, `nsrunits` [27] (automatically incorporated into each JSim project file), are available free for noncommercial use at http://physiome.org/.

For purposes of describing JSim's automated unit balance checking, the MML can be summarized as a collection of variable declarations and mathematical equations using those variables. MML allows using physical units in the declaration of the variables, but does not require them. If unit conversion is turned on (this is also optional), then JSim's compiler will perform the following operations while translating MML to executable Java code:

- equations containing improper unit arithmetic will generate error messages (for manual correction)
- unit scaling factors for conversion of compatible units will be inserted into calculations as needed
- model variables and constants whose units were not specified in MML will be assigned units based upon context wherever possible.

MML Unit Definitions and Variable Unit Specification

Before an MML variable's unit can be specified, the units themselves must be defined. Units are either fundamental or derived. Fundamental units are defined first. Derived units are defined in terms of previously defined fundamental and derived units. The following MML example fragment defines three fundamental and six derived units:

```
unit meter = fundamental, kilogram = fundamental,
second = fundamental;
unit m = 1 meter, cm = 1/100 meter;
unit g = 1/1,000 kilogram, kg = kilogram;
unit newton = 1 kg*m/s^2;
unit dyne = 1 g*cm/s^2;
```

Model writers can define units any way they wish, but for model-to-model compatibility, using JSim's common unit definition file (`nsrunit.mod`, [27]) is recommended whenever possible. The file `nsrunit.mod` defines 121 units and 21 decade prefixes (milli, micro, etc.) following the "Terminology for Mass Transport and Exchange" [12] and the CellML specification [17], [18]. It defines seven fundamental units following the SI [meter kilogram second (MKS)] convention: kilogram, meter, second, ampere, kelvin, mole, and candela. The choice to use SI as fundamental instead of CGS is arbitrary, but of no consequence for model writing, since units in both systems are included. Modelers can define new fundamental or derived units in addition to those in `nsrunit.mod` as needed. For example, many English system units (gallons, miles, etc.) are not defined in `nsrunit.mod`, even though it would be easy to do so, since we wish to discourage the use of nonstandard systems.

Once units are defined, units for MML variables are specified by combining units into algebraic expressions using the multiplication (*), division (/), and exponentiation (^) operators. For example,

```
import nsrunit;  // import standard definitions
unit gallon = 3.7854118 L;  // define gallons,
                            needed for this model
math main{              // start of main
                           calculation section
  real F = 5 gallon/min;     // F is flow in
                               gallons/minute
  real g = 9.81 m/s^2;     // g is gravitational
                             acceleration
  ...                     // rest of model omitted
```

Unit Compatibility and Scaling Factors

Each unit in a JSim model is represented internally by a Java data structure consisting of a scale factor and a fundamental dimension vector. These are defined in double precision:

```
double scale;  // scale factor, e.g. 100 for cm
                  when m is fundamental
double[] dim;  // fundamental dimension vector
```

JSim calculates unit scale factors and dimension vectors using the following rules. For fundamental units, the scale factor is set to 1.0 and the dimension vector is set to all zeroes, except for 1.0 in the vector element corresponding to the fundamental unit itself. In `nsrunit.mod` for example, the second fundamental unit is meters, so the dimension vector for meters is [0,1,0,0,0,0,0]. For derived units, the scale factor and dimension vector are calculated using the values of the units from which they were derived. The rules given later are applied in order of standard algebraic precedence (parentheses first, then exponentiation, then multiplication and division, with ambiguities resolved from left to right):

Rules

1) *Multiplying by a constant (e.g., min = 60 s)*: Multiply the original scale factor by the constant and the dimension vector remains unchanged.
2) *Multiplying two units (e.g., cm × g)*: Multiply the two original scale factors and add the original dimension vectors element by element.
3) *Dividing two units (e.g., cm/s)*: Divide the two original scale factors and subtract the original dimension vectors element by element.

The incentives to *automate* unit balance checking in computer programs include speed and reproducibility.

4) *Exponentiation (e.g., cm^3)*: Raise the original unit's scale factor to the exponent and multiply each original dimension vector element by the exponent.

Consider the processing of unit "grav" (representing gravitational acceleration) in the following example:

```
unit kg = fundamental, meter = fundamental,
sec = fundamental;
unit cm = 1/100 m;
unit grav = 980 cm/s^2;
```

Exponentiation (s^2) has the highest precedence in the *g* definition expression. After that, multiplication (980 cm) and division (cm/s^2) have equal precedence and are processed left to right. Calculations for grav proceed as in Table 1.

Units are considered dimensionless if their dimension vector is uniformly zero. Units are said to be compatible if their dimension vectors are identical (within a machine rounding error of 10^{-7}). For example, accelerations in m/min^2 and cm/s^2 would both have a dimension vector of [0,1,2,0,0,0,0], although their scale factors (1/3,600 and 0.01) would differ. However, speed (e.g., cm/s) would have a dimension vector of [0,1,1,0,0,0,0] and thus be incompatible with the accelerations mentioned earlier.

Compatible units are converted by multiplying by the ratio of the scale factors. For example, conversion from m/min^2 to cm/s^2 is accomplished by multiplying by cm/m (=100 = 1/scale factor) and dividing by s^2/min^2 (=3,600), with the result $(cm/s^2) = (m/min^2)/36$.

Unit Conversion Within Equations

MML models declare either unit conversion on or unit conversion off. In the former case, the compiler checks for compatibility of the units in each algebraic operation, rejecting incompatible ones and inserting appropriate or any needed conversion factors into compatible ones. In the latter case, compatibility is not checked and no conversion factors are introduced (i.e., the units are used only for documentary purposes). The choice of unit conversion declaration is important, because correct equation formulation differs in the two cases. For example, if *A* (in meters) and *B* (in centimeters) are equated, the correct MML code is as in Table 2.

The remainder of this description will consider only with the case where unit conversion is on. Unit declarations are optional for MML variables and constants. We will first describe processing when units are declared for all variables and later will deal with missing unit declarations.

JSim's compiler starts by parsing each model equation into a tree based on operator precedence. MML operator precedence (Table 3) is similar to that of many computer languages (C, Java, etc.). Precedence ambiguities are resolved from left to right.

For example, consider the following model,

```
unit conversion on;
import nsrunit;
math example1 {
  real A = 2 m;
  real B = 30 s;
  real C = 1 min;
  real D cm/s;
  D = A/(B + C);
}
```

Following the MML precedence rules, the equation for D is parsed into the tree as in Figure 1.

The tree consists of internal nodes (light blue) that represent operators and leaf nodes (dark olive) that represent the four variables. Internal nodes have two children (left and right) in this example, any positive number in the general case. Internal nodes are examined, depth first, for unit compatibility. Addition and equality nodes require compatibility of their children, but division nodes do not. When compatibility is required,

Table 1. Units in vector form.

Unit	Scale Factor	Dimension Vector	Rationale
cm	0.01	(0,1,0)	Previous definition
s	1	(0,0,1)	Previous definition
s^2	1	(0,0,2)	Exponentiation (rule 4)
980 cm	9.8	(0,1,0)	Constant multiply (rule 1)
980 cm/s^2	9.8	(0,1,−2)	Unit multiply (rule 3)
grav	9.8	(0,1, −2)	New definition

Table 2. Compare unit conversion *on* or *off*.

With Unit Conversion On	With Unit Conversion Off
$A = B$	$A = B/100$

Table 3. Operator precedence.

Operator	Meaning
( )	Parenthetical groupings
=	Equality
^	Exponentiation
*, /	Multiplication, division
+, −	Addition, subtraction

internal nodes are assigned the units of either the leftmost or the rightmost child, and an appropriate conversion scaling factor is inserted into the tree. When compatibility is not required, internal nodes are given a unit appropriate to the operation (here, division). The choice of associativity (leftmost or rightmost child) is arbitrary but results in equivalent calculations, as in Figure 2.

The resulting calculations are shown in Table 4.

MML Operator and Function Unit Conversion Summary

Table 5 summarizes how the JSim compiler handles unit conversion for MML's predefined operators and functions.

The dimensionless requirements for truncation and rounding are motivated by scaling invariance. For example, the following model will give a different value for B if A is declared as 0.5 kg instead of 500 g, which should be equivalent.

```
real A = 500 g;
real B = round(A);
```

The dimensionless argument requirement for transcendental functions is motivated by both scaling invariance for compatible units and by their Taylor series expansions, which are unbalanced if the argument has nontrivial units, e.g.,

```
exp(x) = 1 + x + x^2/2 + x^3/6 + ...
```

Radians are defined as dimensionless in `nsrunit.mod` for modeler convenience. The alternative would be to require MML writers to convert every trigonometric function argument to radians, which we consider verbose and awkward. Steradians are treated similarly. Centigrade and Fahrenheit temperature scale conversions require additive factors that are not handled by the methodology described here. JSim models use the Kelvin scale.

Inferring Undefined Units

Using unit declarations for MML variables and constants is not absolutely required, and missing units will be assigned by the compiler based on the context provided by the equations. In the following model, C is automatically assigned units m/s (to match the right-hand side of the equation) and the constant 1 will be assumed to be in seconds (to match B) and listed with values in the input list, resulting in the calculation of $C = 10$ m/s:

```
real A = 60 m;
real B = 5 s;
real C = A/(B + 1);
```

In JSim's MML, the aforementioned formulation is acceptable shorthand. The completely specified equivalent model below makes clear the writer's intention, using the unit *s*

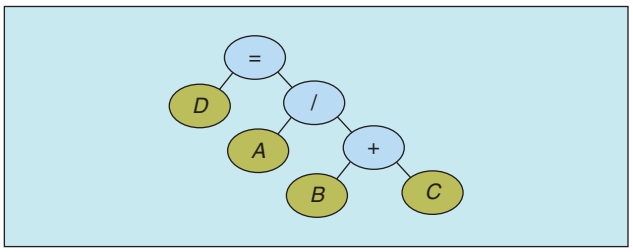

Fig. 1. Operator tree.

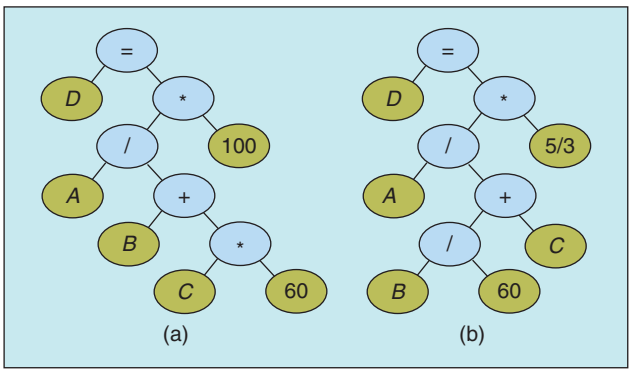

Fig. 2. Associativity. (a) Left associativity. (b) Right associativity.

Table 4. Equivalent associativities.

Left Associativity	Right Associativity
$D = (A/(B + C^{*}60)) * 100 = 20/9$	$D = (A/(B/60 + C))^{*}(5/3) = 20/9$

Table 5. Compiler sequencing of unit conversion.

Operators	Unit of Result	Argument Requirements
Add(+), subtract(−), equals(=), comparison(<, <=, > & >=), remainder(rem(*x*, *y*)), arctangent(atan(*x*, *y*))	Leftmost child's unit	Unit compatibility
Absolute value: abs(*x*)	Same as argument	None
Multiply (*), divide (/)	See unit multiply/divide rules 1, 2 and 3 above	None
Derivative (:)	Same as divide	None
Power (^)	See unit exponentiation rule 4 above	Base is unrestricted, exponent must be dimensionless
Square root: sqrt(*x*)	See unit exponentiation rule 4 above with exponent = 1/2	None
Transcendental functions: exp(*x*), log(*x*), sin(*x*), sinh(*x*), arcsin(*x*), etc.	Dimensionless	Argument must be dimensionless
Truncation: floor(*x*), ceil(*x*). Rounding: round(*x*)	Dimensionless	Argument must be dimensionless

within parentheses along with 1 as follows:

```
real A = 60 m;
real B = 5 s;
real C m/s;
C = A/(B + (1 s));
```

JSim's automated unit assignment algorithm proceeds as follows. A parse tree is generated for each model equation and searched for internal nodes that require unit compatibility (e.g., addition, subtraction, equality). If one of the node's children has no unit assigned, it is assigned the unit of the other child. If, after processing all the nodes in this way, some nodes are still missing unit assignments, one arbitrarily chosen variable or constant is assigned the dimensionless unit, and the entire process is repeated. Eventually, all nodes are assigned units. However, it is possible that variable units assigned in one equation may cause other equations to become unitarily unbalanced. If so, the compiler aborts with a diagnostic error message.

Modelers have generally found the JSim system easy to use. They do not need the technical understanding of the internal calculations described earlier to balance units properly. The system helps them find conceptual errors in their equations and allows them to intermix variables without adding conversion factors. The absence of conversion factors makes their code more readable.

Modelers prefer not having to explicitly assign units to every constant in a model, especially when it is easily understood from the variable name or from the context of the equation. The JSim compiler therefore assigns an appropriate unit, choosing the fundamental unit for it from the context of the equation in which it is used and using unit conversion. If a user prefers to think of the particular parameter in his experimental units, then he can add the definition of this unit to MML variable declaration and define it relative to units defined in `nsrunits.mod`. JSim's compiler currently rejects models that redefine the units that are not compatible with `nsrunits.mod` even if the `nsrunits` file is not explicitly imported.

However, as implied by the decision to use conversion on or off (as in Table 2), a problem can be anticipated. If a programmer introduces a dimensional conversion factor (e.g., s/min) without assigning it units (e.g., just using the dimensionless number 60), this would cause the algorithm to err, as the routine would introduce the scaling conversion factor 60 again by the rules. This does not occur with CellML-curated programs, because they have a policy that such scalars should be properly dimensioned (e.g., as s/min).

Utility for Model Reviewing

JSim's unit balance checking is a useful tool for preliminary checking of models that are downloaded from model databases, whose formats support the assignment of units. The CellML library is large and is in an accelerating stage of curation. Units are used in the CellML files and are inferred in many SBML files.

Many submitted articles contain errors in unit balance that are easily made evident by the reviewer programming them in JSim. Authors can then be guided to fix the problems, which frequently lead to modification of the illustrations. Finding all such errors without using a unit balance checking system is difficult in general but is a problem readily avoided by using automated checking. One might expect that, when a succession of inferred unit assignments has been made, there might be difficulty in discovering the primary error. However, since the inferences are commonly limited to one equation at a time, long series of inferences are seemingly uncommon.

A difficulty is that though a published model may be technically correct, it will not automatically pass the checking for unit balance if it contains transcendental functions of variables with units, because JSim requires that these be dimensionless. An error is detected, and each one of these occurrences must currently be corrected by making the argument of the function formally dimensionless. For example, for sin (V) with V in mV, one should write "$\sin(V/(1\ \mathrm{mV}))$.

The expected usage is to normalize a transcendental argument via a reference value. Sinusoidal functions of time are ordinarily no problem, since they are generally written as

```
sin(2*PI*f*t),
```

where f is the frequency with units of reciprocal time and t is the time. The problem comes with such expressions as

```
exp((V-V0)/10 − 1),
```

where V and V_0 are in millivolts, and so this requires rewriting

```
exp((V-V0)/(10 mV)-1)
```

to render the argument dimensionless. Such coding requires no change if the programmer should change the units of V to volts rather than millivolts so that scaling invariance holds.

Example Application

In the development and analysis of the large cardiovascular or respiratory model VS001 of Neal and Bassingthwaighte, a subset of which is published [28], we have found that automated unit balance checking helps to find typographical errors in equations such as missing terms or parentheses. VS001 is a closed-loop cardiopulmonary model composed of a four-chamber varying-elastance heart, a pericardium, a systemic circulation, a pulmonary circulation, airway mechanics, baroreceptors, gas exchange, blood gas handling, coronary circulation, peripheral chemoreceptors, and selectable physiological changes. The VS001 model code contains 846 equations relating 718 terms expressed in 64 different units. This model is available for free download [29].

Consider this example taken from the gas exchange and intracellular buffering part of the VS001 model: the expression in bold should be enclosed in parentheses but is not, thus evoking an error message identifying an imbalance in units and giving the line number in which the error is found:

```
PBC_pc:t = (Fpc/Vpc)*(PBC_sc-PBC_pc)
  + kp5*CtCO2_ao*(Cheme-PBC_pc)*
    [[SHbO2_ao/(1+H_ao/K3bgh)
  + (1-SHbO2_ao)/(1+H_ao/K2bgh) ]]
  − kp5*PBC_pc*H_ao*(SHbO2_ao/K6bgh
  + (1-SHbO2_ao)/K5bgh);
```

Omitting the double brackets (which stand for simple parentheses around the terms in bold font) results in an equation that is algebraically seemingly acceptable, but the unit balance fails.

```
kp5*CtCO2_ao*(Cheme-PBC_pc)*SHbO2_ao/
(1+H_ao/K3bgh) has units moles/(m^3*s);
```

while

```
(1-SHbO2_ao)/(1+H_ao/K2bgh)
```

is dimensionless, so with the double brackets missing, lines 2 and 3 would result in a unit balance error. The error message identifies the first term of the third line as having different units than the other parts of the equation. Automated unit balance checking thus pinpoints a problem that would be difficult to find by analysis of the model output. The error message also identifies the nature of the units for the separate parts of the equation, but does not go so far as to recommend where the parentheses should be placed.

JSim unit balance error messages arising from complicated algebraic expressions are sometimes difficult to interpret. This is because unit balance is checked in terms of fundamental units (e.g., $kg \cdot m^{-1} \cdot s^{-2}$) while modelers usually think in terms of derived units (e.g., pascals). To address this, JSim error messages provide both fundamental and derived unit representations of the offending expressions. However, derived unit representations (unlike fundamental unit representations) are not unique, and typographical errors such as missing parentheses can result in error messages reporting rather complex unit problems that are not easy to decipher. The current version of JSim (1.6.85) performs minimal simplification of derived unit expressions.

Discussion

Unit balancing in programming is being increasingly recognized as important. For example, Sun's report Java specification request (JSR) 275 [30] provides a specification, currently in development, for handling physical units in the Java language that may be incorporated in a future version of the Java language. If so, it will be a valuable addition to the Java language for reliably engineering large systems that deal with quantities in a variety of physical units. In contrast with JSim, the use of JSR 275 requires significant programming expertise, is rather verbose, provides no facilities for automated unit assignment, and requires complete rewriting of existing computational code.

Unit checking systems could be further expanded. Incorporating offsets combined with scaling would allow the conversion of temperatures from Centigrade or Fahrenheit to Kelvin. Likewise, pH, pCa, and such logarithmic units could be usefully added for modeling in biology.

The next stage, currently being vigorously pursued by several groups, is to curate the models to try to accurately reproduce the results shown in the original publications. The CellML curation team had, as of August 2008, worked on about 365 models from the literature, taking them through a series of stages of curation and accumulating 883 model versions in all [17], [18]. The Biomodels curation team (Biomodels 2008) in Cambridge checked more than 172 models from the SBML site [20], [21]; in many, there is a dependence on inferring units. On the Physiome site [29] (www.physiome.org/Models), there are approximately 250 models that are archived and downloadable (as are those on the CellML, SBML, Biomodels, and JWR [31] model sites). Almost all of the JSim versions can be run over the Web, with graphics display, parameter adjustment, sensitivity analysis, and even parameter optimization to fit experimental data. Thus, they are demonstrably operational. The Physiome models do allow inferred units and are reproducible, and all have unit balance in every equation, since that is a built-in component of the compilation of the model. Unit checking is impossible for models in which all physical variables have been labeled dimensionless.

A particular virtue of using unit-balanced equations is that when two or more correct models are combined into a larger model, there are no complications matching units between the components, since conversion factors are automatically inserted to bring all into compatibility with the fundamental units. The automated coupling of a set of modules into a composite model also requires that a common ontology be used, that distinct regions be identified, and then that the equations containing variables from more than one module be combined properly. In this situation, the component unit definitions directly facilitate the process. A preliminary success in automating the combination of a small set of modules has been achieved.

A future strategy is to allow stages in a hierarchical review of code for unit balance and automatic unit conversion. To foster rapid evolution of the model code, stage 0 would be to allow compilation without checking for unit balance; this should be allowed even if units have been provided on all variables. Stage 1 would be to detect imbalances and report the list of them and to allow for manual correction before attempting to recompile. Stage 2 would be to detect and report equations where unit conversion factors would be applied, so that the programmer can decide whether or not to assign units differently. Stage 3 would be to choose automated unit conversion.

Multiscale models are becoming even more abundant as it becomes necessary to link cellular level models to whole organ and whole body models. The cardiac models of Hunter and Noble, the Auckland/Oxford collaboration [32], bring the ion channel cellular depolarization models together with the whole organ finite element cardiac contraction models. The cell models of metabolic reaction sequences in purine nucleosides are brought together into a whole organ model for regional blood flows and capillary-tissue exchanges [33]. These two particular cases are primitive examples (even if they do now represent tens to hundreds of thousands of ordinary differential equations) of much larger models to be composed of modules from subcellular domains to whole-body controllers.

Building upon the work of others is fundamental to progress in science. Computer modeling and archiving is marvelously efficient for preserving precise descriptions of a current scientific working hypothesis, subjecting it to repeated evaluation tests and identifying its shortcomings and alleviating them. For quantitative integrative multiscale models of complex systems, models so preserved are modules to be incorporated into more all-encompassing models that embrace more phenomena and are tested against larger numbers of data sets. Guaranteeing balanced units and scale invariance in the reference literature and in models which will become subsidiary modules of a larger model is therefore a worthy goal.

Acknowledgment

This research has been supported by NIH grants BE1973 (JSim) and HL88516 (tutorial design) and NSF grant 0506477 (multiscale modeling).

Howard Jay Chizeck received his B.S. (1974) and M.S. (1976) degrees from Case Western Reserve University and Sc.D. degree (1982) in electrical engineering and computer science from Massachusetts Institute of Technology (MIT). From 1981 to 1998, he was at Case Western Reserve University in Cleveland, serving as chair of the Department of Systems, Control and Industrial Engineering from 1995 to 1998.

He was the chair of the Electrical Engineering Department at the University of Washington (UW) from 1998 to 2003. He was elected a Fellow of the IEEE in 1999 for contributions to the use of control system theory in biomedical engineering. He is a professor of electrical engineering and an adjunct professor of bioengineering at the UW in Seattle. His research interests involve control engineering theory and the application of control engineering to biomedical and biologically inspired engineered systems.

Erik Butterworth received his B.A. degree in mathematics from the University of Chicago in 1977. Between 1977 and 1987, he worked as a computer programmer/analyst for several small commercial software firms. Since 1988, he has worked as a software engineer on various research projects at the UW. Between 1988 and 1993, he developed a real-time data acquisition for the analysis of estuarine sediment transport in the Department of Geophysics. Between 1988 and 2002, he developed I4, a system for the display and analysis of cardiac positron emission tomography (PET) images in the Department of Cardiology. Since 1993, he has worked on physiological simulation systems (XSIM from 1993 to 1999, JSim since 1999) at the National Simulation Resource Facility in Circulatory Mass Transport and Exchange in the Department of Bioengineering. His research interests include simulation systems and medical imaging.

James B. Bassingthwaighte received his B.A. (1951) and M.D. degrees (1955) from the University of Toronto and Ph.D. degree (1964) from the University of Minnesota. He trained in physiology, biochemistry, and medicine (University of Toronto); cardiology (Mayo Graduate School of Medicine); and physiology (University of Minnesota). He was on the faculty at Mayo (1964–1975), heading the Biophysics section of the Department of Physiology (1972–1975). He chaired the Department of Bioengineering at the UW (1975–1980) and established in 1979 a National Simulation Resource Facility in Circulatory Mass Transport and Exchange. In 1997, he initiated the Human Physiome Projects. Being a professor of bioengineering, biomathematics, and radiology at the UW, he conducts research on quantitative and integrative approaches to cardiovascular, respiratory, and cellular physiology. He has 260 peer-reviewed publications and two books. He served as president of the Biomedical Engineering Society and the Microcirculatory Society, chaired the Cardiovascular Section of the American Physiological society (APS), and was the editor of the *Annals of Biomedical Engineering*. He has received honors from Biomedical Engineering Society (BMES), APS, the Netherlands Biophysical Society, the Cardiovascular Systems Dynamics Society, the Microcirculatory Society, and McGill University. He is a member of the U.S. National Academy of Engineering.

Address for Correspondence: Howard Jay Chizeck, Department of Bioengineering, University of Washington, Seattle, WA 98195, USA.

References

[1] Mars orbiter [Online]. Available: http://www.spaceref.com/news/viewpr.html?pid=2937
[2] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *J. Physiol.*, vol. 117, pp. 500–544, 1952.
[3] G. W. Beeler, Jr. and H. Reuter, "Reconstruction of the action potential of ventricular myocardial fibres," *J. Physiol.*, vol. 268, pp. 177–210, 1977.
[4] R. L. Winslow, J. Rice, S. Jafri, E. Marbán, and B. O'Rourke, "Mechanisms of altered excitation–contraction coupling in canine tachycardia-induced heart failure. II. Model studies," *Circ. Res.*, vol. 84, pp. 571–586, 1999.
[5] A. Michailova and A. McCulloch, "Model study of ATP and ADP buffering, transport of Ca(2+) and Mg(2+), and regulation of ion pumps in ventricular myocyte," *Biophys. J.*, vol. 81, pp. 614–629, 2001.
[6] E. Buckingham, "On physically similar systems: Illustration of the use of dimensional equations," *Phys. Rev.*, vol. 4, pp. 345–376, 1914.
[7] L. Wittgenstein, *Tractatus Logico-Philosophicus* (Transl.: in German by C. K. Ogden, with an introduction by B. Russell). London: Routledge and Kegan Paul, 1922.
[8] O. Reynolds, "An experimental investigation of the circumstances which determine whether the motion of water in parallel channels shall be direct or sinuous and of the law of resistance in parallel channels," *Proc. R. Soc. Lond.*, vol. 35, pp. 84–99, 1883.
[9] S. G. Sterrett. (2002). Physical pictures: Engineering models circa 1914 and in Wittgenstein's Tractatus [Online]. Available: http://philsci-archivepittedu/documents/disk0/00/00/06/61/PITT-PHIL-SCI00000661-00/Sterrett-UNC-CH-PPTalk2pdf
[10] A. P. Gagge, A. C. Burton, and H. C. Bazett, "A practical system of units for the description of heat exchange of man with his environment," *Science*, vol. 94, pp. 428–430, 1941.
[11] J. R. Pappenheimer, "Standardization of definitions and symbols in respiratory physiology," *Fed. Proc.*, vol. 9, pp. 602–605, 1950.
[12] J. B. Bassingthwaighte, et al., "Terminology for mass transport and exchange," *Amer. J. Physiol. Heart Circ. Physiol.*, vol. 250, pp. H539–H545, 1986.
[13] M. Karr and D. B. Loveman, "Incorporation of units into programming languages," *Commun. ACM*, vol. 21, pp. 385–391, 1978.
[14] T. Gruber and G. R. Olsen, "An ontology for engineering mathematics," in *Proc. 4th Int. Conf. Principles of Knowledge Representation and Reasoning*, J. Doyle, P. Torasso, and E. Sandewall, Eds. Bonn: Morgan Kauffman, 1994, p. 18.
[15] G. S. Novak, "Conversion of units of measurement," *IEEE Trans. Software Eng.*, vol. 21, pp. 651–661, 1995.
[16] F. Chen, G. Rosu, and R. P. Venkatesan, "Rule-based analysis of dimensional safety (RTA 2003)," *Lect. Notes Comput. Sci.*, vol. 2706, pp. 197–207, 2003.
[17] CellML [Online]. Available: http://www.cellml.org/workshop/workshop2008/index_html
[18] A. Cuellar, et al., "An overview of CellML 1.1, a biological model description language," *Simulation*, vol. 79, pp. 740–747, 2003.
[19] J. Cooper and S. McKeever, "A model-driven approach to the automatic conversion of physical units," *Softw. Pract. Exper.*, vol. 38, pp. 337–359, 2008.
[20] Sytems biology markup language (SBML) [Online]. Available: http://sbml.org/Main_Page
[21] M. L. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, and H. Kitano, "The system biology markup language (SBML): A medium for representation and exchange of biochemical network models," *Bioinformatics*, vol. 19, no. 4, pp. 524–531, 2003.
[22] G. M. Raymond, E. Butterworth, and J. B. Bassingthwaighte, "JSim: Free software package for teaching physiological modeling in research," *Exper. Biol.*, vol. 280, no. 5, p. 102, 2003.
[23] www.physiome.org
[24] J. Gosling and H. McGilton. (1996, May). The Java language environment: A white paper [Online]. Available: http://java.sun.com/docs/white/langenv
[25] T. J. Knopp, D. U. Anderson, and J. B. Bassingthwaighte, "SIMCON—Simulation control to optimize man-machine interaction," *Simulation*, vol. 14, pp. 81–86, 1970.
[26] R. B. King, E. A. Butterworth, L. J. Weissman, and J. B. Bassingthwaighte, "A graphical user interface for computer simulation," *FASEB J.*, vol. 9, no. A14, 1995.
[27] NSR units [Online]. Available: http://physiome.org/jsim/docs/MML_Units_NSR.html
[28] M. L. Neal and J. B. Bassingthwaighte, "Subject-specific model estimation of cardiac output and blood volume during hemorrhage," *Cardiovasc. Eng.*, vol. 7, pp. 97–120, 2007.
[29] www.physiome.org/model/doku.php?id=Integrative_Physiology:Highly-integrated_human_with_interventions:model_detail&s=vs001)
[30] Java units specification [Online]. Available:https://jsr-275.dev.java.net/
[31] JWRS online = model database [Online]. Available: http://jjj.biochem.sun.ac.za/database/index.html
[32] N. P. Smith, D. P. Nickerson, E. J. Crampin, and P. J. Hunter, "Multiscale computational modelling of the heart," *Acta Numer.*, vol. 13, pp. 371–431, 2004.
[33] J. B. Bassingthwaighte, G. R. Raymond, J. D. Ploger, L. M. Schwartz, and T. R. Bukowski, "GENTEX, a general multiscale model for *in vivo* tissue exchanges and intraorgan metabolism," *Philos. Trans. R. Soc. A Math. Phys. Eng. Sci.*, vol. 364, no. 1843, pp. 1423–1442, 2006.
[34] European Molecular Biology Laboratory, European Bioinformatics Institute [Online]. Available: http://www.ebi.ac.uk/biomodels-main/