Automated Unit Balancing in the JSim Modeling System

Erik Butterworth[1], Howard Jay Chizeck[1,2], and James B. Bassingthwaighte[1]
Departments of Bioengineering[1] and Electrical Engineering[2], University of Washington, Seattle WA 98195

Abstract

The combining of simpler models into larger ones which represent multi-scale and other complex systems can be facilitated by using three methods which automate balance checking for the unit conversions that inevitably result when disparate models are combined. These unit balancing processes are: (1) the identification and rejection of model equations containing invalid unit arithmetic; (2) the insertion of scalar multipliers into the equations to convert existing units into fundamental units (in the SI or cgs systems); and (3) the assignment of units to variables and constants whose units are undeclared, based upon context. Used together, these three methods automate the otherwise tedious and error-prone process of manual unit balance checking thus producing more accurate and readable models. The implementation of these approaches, as in the JSim simulation modeling system, are general and could be incorporated into other modeling platforms.

Introduction

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 terms denoting unitary physical properties, such as mass, distance, force, pressure, chemical concentration, and potential difference, along with terms that denote algebraic combinations of unitary properties, such as meters/sec^2 and gram/ml.

The physical properties denoted by the terms of the model equations may be expressed in various units, at various scales: for example, pressure may be expressed as pascals, kilopascals, atmospheres, torr or mmHg, 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 unit system used.

In complicated models, balancing the units in equations with many terms can be tedious and time-consuming. In standard programming languages like FORTRAN, C, and Matlab, one usually has to insert numeric unit conversion factors into the model code. Constructs such as (meters/sec + meters/sec^2) are inherently erroneous and have to be detected by the programmer. This is difficult to do however. Although the units are not identified within the equations, each group of terms to be summed in a particular equation must have *identical units at identical* *scales*, and units to the left of an equal sign must be the same as those on the right. This is a requirement for correctness in the equations. The methodology proposed here exploits that fact that unit balancing follows rigid, logical rules and thus can be automated. Automated unit balancing not only reduces error in individual equations, but by alerting the model builder to imbalances, it can identify conceptual errors in their formulation.

"Unitary scaling invariance" is the property of a mathematical modeling system such that numeric calculations remain correct under changes of unitary scale. 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 unitary scaling invariant modeling system responds by adjusting all calculations using V to be correct under the new scaling. Unitary scaling 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 unitary scaling invariant modeling system reconciles them automatically. This capability thus improves prospects for retaining modularity when developing multiscale models.

JSim

JSim [1] is a Java-based [2] 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 National Simulation Resource for Mass Transport and Metabolism (NSR) since 1999. JSim uses a model specification language, MML (for Mathematical Modeling Language) 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 SIMCON [3], for simulation control, and XSIM [4] for X-terminal operation. JSim source code, binaries (for Windows, Macintosh and Linux) and documentation are available free for non-commercial use at http://physiome.org/.

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

1. Equations containing improper unitary arithmetic (e.g. meters/sec + meters/sec^2) will generate error messages.
2. Unitary scaling factors will be inserted into calculations as needed (e.g. 6 cm/min = 1

mm/sec).

3. Model variables and constants whose units were not specified in MML will be assigned units based upon context, wherever possible. For example, if A has been assigned the units cm then when the expression (A+B) appears, it implies A and B have compatible units, and thus B is assigned units cm.

4. Variables with no assigned units and not declared dimensionless are assigned dimensionless.

Design goals

JSim's automated unitary correction system was developed to meet the following goals:

1. User specification of variable units should be simple, intuitive and unambiguous.
2. Variants from the basic SI and cgs units such as commonly used units for physiology and systems biology must be predefined in terms of the basic units.
3. Modelers may define new units or abbreviations if desired.
4. Unitarily incorrect equations must be detected.
5. Unitary scaling factors must be automatically inserted into computations.
6. A change to a variable's unit assignment will cause computations to rescale correctly (unitary scaling invariance).
7. The system should be accommodate existing models that do not use automated unitary correction technology.

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, gram = fundamental, sec = fundamental;
unit m = 1 meter, cm = 1/100 meter;
unit g = 1 gram,  kg = 1000 gram;
unit newton = 1 kg*m/sec^2;
unit dyne = 1 g*cm/sec^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, [5]) is strongly recommended. The file defines 121 units and 21 decade prefixes (milli, micro, etc.) following "Terminology for mass transport and exchange" [6] and the CellML specification [7]. nsrunit.mod defines 7 fundamental units, following the SI (MKS) convention: kilogram, meter, second, ampere, kelvin, mode, candela. The choice to use MKS 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. However, it is recommended in order to make model code most easily understandable that modelers use the standard terminology in nsrunit.mod wherever possible.

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 liter;  // 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 = 9.8 m/sec^2;         // g is gravitational acceleration
  ...                           // rest of model omitted
```

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 to centimeters per second the scaling factor is 0.1 cm/mm divided by 60 sec/min or simply 0.1/60 since the components cm/mm are both length measures and sec/min are both time measures and therefore are compatible. However, moles per gram and moles per liter are not. Moles/gram times grams/ml would be compatible with moles/liter. 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
```

The scale factor is used for converting compatible units such as cm/sec and mm/min. The dimension vector, whose length is equal to the number of fundamental units in the model (e.g. 7 for a model using only the nsrunit.mod definitions), is used for determining compatibility. Two units are compatible if their dimension vectors are identical.

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 2nd 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 below are applied in order of standard algebraic precedence (parentheses first, then exponentiation, then multiplication & division, with ambiguities resolved from left to right):

RULES:

A) Multiplying by a constant (e.g. min = 60 sec): Multiply the original scale factor by the constant, the dimension vector is unchanged.
B) Multiplying two units (e.g. cm*gram): Multiply the two original scale factors, add the original dimension vectors element by element.
C) Dividing two units (e.g. cm/sec): Divide the two original scale factors, subtract the original dimension vectors element by element.
D) Exponentiation (e.g. cm^3): Raise the original unit's scale factor to the exponent, multiply each original dimension vector element by the exponent.

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

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

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

Unit	Scale factor	Dimension vector	Rationale
cm	.01	[0,1,0]	previous definition
sec	1	[0,0,1]	previous definition
sec^2	1	[0,0,2]	exponentiation (rule D)
980 cm	9.8	[0,1,0]	constant multiply (rule A)
980 cm/sec^2	9.8	[0,1,-2]	unit multiply (rule C)
grav	9.8	[0,1,-2]	new definition

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 1e-7). For example, accelerations in m/min^2 and cm/sec^2 would both have a dimension vector of [0,1,2,0,0,0,0], although their scale factors (1/3600 and .01) would differ. However, speed (e.g. cm/sec) would have a dimension vector of [0,1,1,0,0,0,0], and thus be incompatible with the accelerations above.

Compatible units are converted by multiplying by the ratio of the scale factors. For example, conversion from m/min^2 to cm/sec^2 is accomplished by multiplying by cm/m (=100 = 1/scale factor) and dividing by sec^2/min^2 (=3600), with the result (cm/sec^2)= (m/min^2)/36.

Basic unitary correction of equations

MML models declare either "unit conversion on" or "unit conversion off". In the former case, the compiler checks for unit compatibility in each algebraic operation, rejecting incompatible ones and inserting appropriate any needed conversion factors into compatible ones. In the latter case, compatibility is not checked and no conversion factors are introduced (i.e. units are 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 follows:

with unit conversion on	**with unit conversion off**
A = B	A = B/100

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 consider how to deal with missing unit declarations.

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

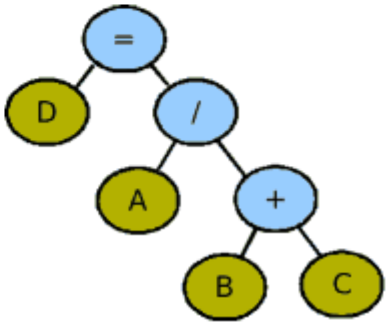
Operator	**Meaning**
()	parenthetical groupings
=	equality
^	exponentiation
* , /	multiplication, division
+ , -	addition, subtraction

For example, consider the following model:

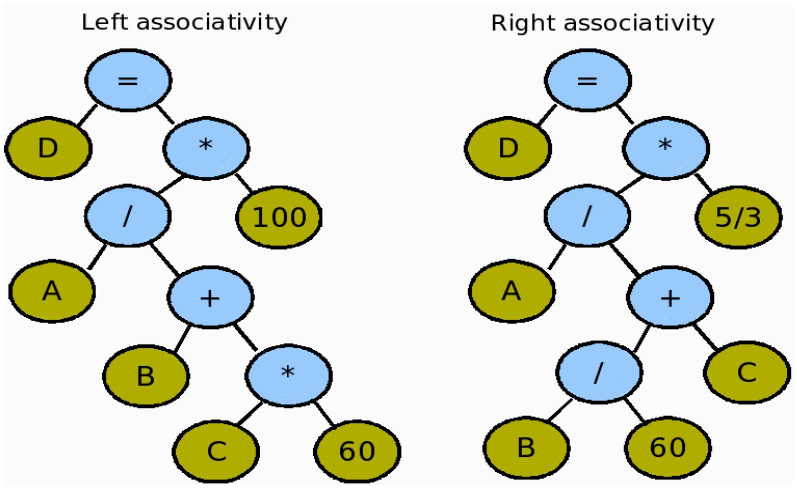
```
unit conversion on;
import nsrunit;
math example1 {
  real A = 2 meter;
  real B = 30 sec;
  real C = 1 min;
  real D cm/sec;
  D = A / ( B + C );
```

}

Following the MML precedence rules, the equation for D is parsed into the following tree:



The tree consists of internal nodes (light blue) which represent operators and leaf nodes (dark olive) which 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 unitary compatibility of their children, but division nodes do not. When compatibility is required, internal nodes are assigned the units of either the leftmost or rightmost child, and an appropriate unitary scaling factor is inserted into the tree. Where compatibility is not required, internal nodes are given a unit appropriate with the operation (here, division). The choice of associativity (leftmost or rightmost child) is arbitrary, but results in equivalent calculations, as shown below:



Resulting calculations are as follows:

Left associativity	Right associativity
D=(A/(B+C*60))*100=20/9	D=(A/(B/60+C))*(5/3)=20/9

MML operator and function unitary conversion summary

The table below summarizes how the JSim compiler handles unitary conversion for MML's predefined operators and functions.

Operators	Unit of result	Argument requirements
add(+), subtract(-), equals(=), comparison(<, <=, > & >=), remainder(rem(x,y)), arctangent(atan(x,y))	leftmost child's unit	unitary compatibility
absolute value: abs(x)	same as argument	none
Multiply (*), divide (/)	see unit multiply/divide rules A, B and C above	none
Derivative (:)	same as divide	none
Power (^)	see unit exponentiation rule D above	base is unrestricted, exponent must be dimensionless
square root: sqrt(x)	see unit exponentiation rule D 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

Table notes:

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

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

The dimensionless argument requirement for transcendental functions is motivated by both unitary scaling invariance and by their Taylor expansions, which are unbalanced if the argument has non-trivial 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.

Handling undefined units

Using unit declarations for MML variables and constants is not absolutely required, and missing units will be assigned by the compiler based upon equation context. In the following model, C is automatically assigned units m/sec (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 on the Input list, resulting in the calculation of C=10 m/sec:

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

Such assignment carries risk, exemplified by the use of an unassigned 1 in the equation for C, and it is better to assign units to all variables and parameters, as is required in the model archiving markup language, CellML [7]. (The Systems Biology Markup Language SBML [8] allows but does not require unit specification for variables.) The choice is a practical tradeoff between user convenience, conciseness and unitary scale invariance. In JSim's MML, the above formulation is acceptable shorthand. The completely specified equivalent model below makes clear the writer's intention, using the unit *sec* within a parenthesis along with the 1:

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

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 unitary 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 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.

Results and Discussion

Programming convenience: Modelers generally find the system easy to use. They do not need the technical understanding of the internal calculations described above to balance units properly. The system helps them find conceptual errors in their equations and allows them to inter-mix 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 incompatibly redefine units in nsrunits.mod, even if the common definitions file is not explicitly imported. This bug is not profound, and will be corrected in a future JSim version.

Conversion on or off?: Enabling automatic unit conversion is optional ("unit conversion on" or "unit conversion off"), but virtually all modelers choose to enable it. The small amount of additional work required to add unit declaration to one's MML is rewarded by the error detection functionality described below.

Error detection and correction: Automated unitary correction helps find equation typos such as missing terms or parentheses. This is especially important in a large model such as the cardiovascular/respiratory model VS001 of Neal and Bassingthwaighte, a subset of which is published [9]. (The whole model is available for free download [10]) 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.

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 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 parentheses results in an equation that is algebraically seemingly acceptable, but unitary balance fails:

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

`(1-SHbO2_ao)/(1+H_ao/K2bgh)` is dimensionless, so adding the two quantities (as in lines 2 & 3 above) results in a unitary 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 unitary correction 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/m/sec^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 units representations) are not unique, and any single chosen derived unit representation may not correspond to modeler intuition. The current version of JSim (1.6.80) performs only minimal simplification of derived unit expressions. More sophisticated simplification improvements in future versions would improve error message readability. However, modeler typing mistakes (see above) can easily result in algebraic expressions with non-intuitive derived unit representation. In the future, a syntax-highlighting, unit-aware editor, allowing users to interactively query the units for code fragments, may be the best solution for editing complex models.

Utility for model reviewing: JSim's unit balance checking is a useful tool for preliminary checking of models that are downloaded from model databases who's formats support unitary assignment. The CellML library is larger but is at an earlier stage of curation. Units are now used in the CellML files. Of the 312 available for download (as of April 2007), 60 pass the JSim unit balance checks. However, we have not checked the resulting numerical solutions against the original papers.

Many submitted papers contain unitary balance errors that are easily made evident by the reviewer programming them in JSim. Authors can then be guided to fix the problems, which frequently leads to modifying the illustrations. Finding all such errors without using a unit balance checking system is difficult, a problem readily avoided when using JSim.

It is common knowledge that many published models are incorrect and cannot be reproduced. Some merely lack unitarily balanced equations or have incomplete unit assignments. The number of these models that are actually physiologically inaccurate due to unitary issues is difficult to judge, but determining the answer requires a detailed and knowledgeable examination of each of them.

A difficulty with some published models is that though they made be technically correct, they will not be automatically verified for unitary balance, if they contain 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 (e.g. for sin(V) with V in mV, one writes “sin(V/(1 mV)) before the full model can be compiled.

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

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

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

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

where V and V0 are in millivolts, so this requires rewriting

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

to render the argument dimensionless. An option to allow a relaxation of this requirement will be incorporated in a future JSim release.

Modular Modeling in Multiscale Systems: A particular virtue of using unit-balanced equations is that when two or more smaller JSim unit correct models are combined into a larger model, there are no complications matching units between the components, since conversion factors are automatically inserted. The automated coupling of a set of modules into a composite model requires also 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 automated combining of two modules has been achieved, and so is feasible.

JSR 275 [11] is 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's current facilities, 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.

Conclusions:

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 which embrace more phenomena and are tested against larger numbers of data sets. Guaranteeing unitary balance and scale invariance in the reference literature and in the models which are the future subsidiary modules is therefore a worthy goal. JSim's unit balance checking is one tool for this job, and other simulation systems will undoubtedly be modified to include this powerful feature. The algorithms described herein are adaptable to other quantitative modeling systems so long as model variable can be assigned physical units.

References:

1. Raymond GM, Butterworth E, and Bassingthwaighte JB. JSim: Free software package for teaching physiological modeling in research. Exper Biol 2003 280.5, p102, 2003., and www.physiome.org

2. Gosling J and McGilton H, "The Java language environment a white paper," May 1996. http://java.sun.com/docs/white/langenv.

3. Knopp TJ, Anderson, DU, and Bassingthwaighte JB. "SIMCON--Simulation control to optimize man-machine interaction," *Simulation* 14: 81-86, 1970.

4. King RB , Butterworth EA, Weissman LJ, and Bassingthwaighte JB., "A graphical user interface for computer simulation. *FASEB J:* 9: A14, 1995.

5. NSR Units: http;//physiome.org/jsim/docs/MML_Units_NSR.html

6. Bassingthwaighte JB et al, "Terminology for mass transport and exchange. Am. J. Physiol. Heart Circ. Physiol. 250: H539-H545, 1986.

7. Cuellar A et al, "An Overview of CellML 1.1, a Biological Model Description Language," *Simulation* 79: 740-747, 2003. DOI: 10.1177/0037549703040939.

8. Hucka ML, Finney A, Sauro HM, Bolouri H, Doyle JC, and Kitano H. The system biology markup language (SBML) a medium for representation and exchange of biochemical network models. Bioinformatics 19(4): 524-531, 2003. and http://sbml.org/index.psp

9. Neal ML and Bassingthwaighte JB. Subject-specific model estimation of cardiac output and blood volume during hemorrhage. *Cardiovasc Eng* 7: 97-120, 2007.

10. www.physiome.org/model/doku.php?id=Integrative_Physiology:Highly-integrated_human_with_interventions:model_detail&s=vs001)

11. Java units specification: https://jsr-275.dev.java.net/