UNITS IN LPL

T. Hürlimann

Working Paper No. 182

(updated April 1992)
(updated January 1993)
(updated July 1993)
(updated August 1995)

April 1991

INSTITUTE FOR AUTOMATION AND OPERATIONS RESEARCH (formerly)

INSTITUTE OF INFORMATICS (actually)

University of Fribourg

CH-1700 Fribourg / Switzerland

email: tony.huerlimann@unifr.ch

phone: ++41 37 298 328 fax: 299 726

Units in LPL

Tony Hürlimann, Dr. lic. rer. pol.

Key-words: Model Building, Modeling Language design, units.

Abstract: A method is proposed to incorporate a system of measurement units into the modeling language LPL. Benefits of keeping track of units in a modeling system can be to trap more errors, to enhance reliability and readability of the model, or to scale the data automatically. A short introduction on manipulation rules with measurement units is given; the full syntax of this LPL extension and several examples are presented; and finally some implementation aspects are rehashed.

Stichworte: Modellierung, Modelliersprache, MIP Programmierung.

Zusammenfassung: Es wird eine Methode vorgestellt, die erlaubt Masseinheiten in the Modelliersprache LPL einzubinden. Die Vorteile sind, dass mehr syntaktische Fehler automatisch entdeckt werden können, dass das Modell u.U. an Lesbarkeit gewinnt, und dass Daten automatisch skaliert werden. Es wird eine kurze Einführung in die Algebra der Einheitenrechnung gegeben; sodann breiten wir die volle Syntax in LPL aus und geben einige Beispiele, schliesslich werden noch einige Implementationsaspekte berührt.

1. INTRODUCTION

"...but at the system level I don't think you'll ever be fully confident that somebody somewhere hasn't punched in feet instead of miles in the computer program." (John Pike).

The crew of the space shuttle Discovery in 1985 fed the number 10023 into the onboard guidance system. The number was correct, but it was supplied in feet to the crew, whereas the system expected a unit in nautical miles. Thus, the space shuttle flew upside-down over Maui.

Most quantities in models are measured in units (dollar, hour, meter etc.). As the introductionary example shows, there may be a big difference between a quantity being given in one or another unit. Experiences in Operations Research teaching in our Institute revealed that one of the most frequent errors of students in modeling was the inconsistency of measurement units.
In physics and other scientific, technical and commercial applications, using units of measure has a long tradition. It increases the reliability and the readability of calculations. Thus, it is not surprising that there have been different proposals to include this concept into the programming languages design [Karr 1978, House 1983, Männer 1986, Dreiheller 1986, Baldwin 1987]. The concept of units seems to fit quite well into the much broader concept of data type used in different, strong-typed programming languages such as PASCAL and ADA. One may think that units can simply be included by a strict form of name equivalence, whereby two types should be considered different, even if they are based on the same basic type. For example the two types TIME and DISTANCE in a PASCAL type declaration

```
TYPE
  TIME = REAL;
  DISTANCE = REAL;
```

should be different types (in ADA, they are indeed different types). Unfortunately, this does not solve our problem. In unit calculations, we need derived units such as speed which is defined as distance per time. Combining different units (types) in the same expression would produce a type error. Hence, the type concept must be somewhat extended to allow full integration of unit calculations.
Our concern here is not programming languages design but modeling language design. As far as I can see, there has been one proposal to incorporate a unit system into an executable modeling language. Bradley/Clemence [1987]

specifies a type calculus for an extended dimensional system for modeling languages. Each model entity as well as input and output are assigned a type that consists of its concepts, quantities and units of measurement.

A modeling language should in some way keep track of physical and other units, e.g. volts, horsepower, hertz, dollar etc. Explicit mention of units can enhance readability of model, can give the compiler additional checking power which may reduce the number of syntax errors, and can let the compiler do the job of automatic unit conversion and scaling.

We do not consider how units are related to the real world. The question of which units are elementary will lead to interminable discussions unless the following resolution is taken: The model language should not commit itself to any particular set of units. The modeler must be free to define his own unit system. It is true, however, that there exist international standards and conventions. In physics, e.g., all quantities seem to be reducible to the following seven basic units:

length	[m]	meter
mass	[g]	gram
time	[s]	second
electric current	[A]	Ampere
thermodynamic temperature	[K]	Kelvin
amount of substance	[mol]	mole
luminous intensity	[cd]	Candela.

All other physical units are derived from these seven basic units. For example, Joule [J] is defined as [$1000m^2gs^{-2}$]. But in other domains these units are different. A modeling language should be independent of any specific application domain. The modeler must even be free to make use of the unit system or not. It should be entirely optional. He or she alone should be responsible for a good (or bad) model formulation. The modeling language designer, on the other hand, must offer powerful checking possibilities. Whether they are used or not to formulate a concrete model should be decided by the modeler. Sometimes it may be a mere matter of taste whether units should be incorporated into a model. In a model, for example, where we know that all time quantities are given in weeks and every money quantity is given in dollars, there is no need to add this information explicitly to the model.

DEFINITIONS

Units are something which is carried along in calculations. They act like numeric identifiers in numerical expressions, obeying commutativity and associativity laws. In the expression

$g = 9.81*(m/sec)/sec = 32*feet/sec^2$

the '9.81' can be regarded as being multiplied by the expression 'm/sec^2'. Thus, when we want to calculate how far an object falls in 3 seconds using $d=1/2gt^2$, we may write

$$d = 1/2 * 9.81 * (m/sec^2) * (3\ sec)^2 = \frac{9.81*m*9*sec^2}{2*sec^2} = 44.145\ m$$

or using feet instead of meters we get

$$d = 1/2 * 32 * (m/sec^2) * (3\ sec)^2 = \frac{32*m*9*sec^2}{2*sec^2} = 144\ feet$$

Ordinary algebra rules can be used to calculate expressions using units. Units are useful because we may define relationships among them; for example we may write expressions like

1 m = 3.261 feet

12 inches = 1 foot

watts = volts amps

Meters can be converted into feet using such expressions and vice versa. Units are also useful, because they disallow operations which do not make sense; for example, '2 m + 3 watts' is an addition which does not make sense, because m and watts are not commensurable. But the calculation '1 m + 1 foot' is something we may accept; it is 4.261 feet or 1.306m, depending on whether we express the result in feet or meters. I do not give a complete overview of unit calculus here, which can be found in any physics course; only some basic concepts on unit calculus are introduced.

Since we are only interested in how units are manipulated, we may abstract those properties of interest to us. We start with a finite set of (user-defined) *elementary units*, which we think of simply as symbols with suggestive names: feet, dollar, week. A *derived unit* A is *commensurable* to another unit B, if and only if A/B is a dimensionless quantity; in this case there must exist a well defined *commensurateness relationship* between the units A and B. A quantity is *dimensionless*, if the division A/B can be reduced to a simple number. A commensurateness relationship consists of the derived unit name, an

assignment symbol, and a *unit expression*:

```
Derived_unit_name = Unit_expression
```

Example

feet	*feet* is an elementary unit
inch = feet/12	*inch* is a derived unit

Inch is commensurable with feet, since $\frac{\text{inch}}{\text{feet}} = \frac{\text{feet/12}}{\text{feet}} = 1/12$ and no unit is left in the last expression. Also *compound unit* definition may be useful, which are derived units, compound of more than one unit. An example is

watts = volts*amps

Note that the units 'watts' and the units 'volts/amps*amps^2' are commensurable using the relationship above since

$$\frac{\text{watts}}{\text{volts/amps*amps}^2} = \frac{\text{volts*amps}}{\text{volts/amps*amps}^2} = 1$$

can be reduced to the number 1.

A *unit expression* has a very limited syntax: Only other unit names - elementary or derived ones - together with numeric literals and the two operators * and / are allowed. Principally, the power operator ^ might be allowed providing the exponent is rational (see syntax rules in [House 1983]). We will disallow the power operator as well.

Units calculations depend on the operators; they must be handled differently. We present now a short summary on rules of expression checking

Assignment: The units of the left hand side must be commensurable to the unit of the right hand side. If they are not commensurable, an error occurs. The left hand side must be multiplied by the corresponding scale factor.

Addition, Subtraction, and relational operators: The units of the operandi must be commensurable. If they are not commensurable, an error occurs. Each side must be scaled before the comparison or operation can take place. For boolean operators, this can give rise to real arithmetic precision errors like

```
IF  ( inch = 1*Feet/12 )  THEN ...
```

Will the test be true or false? Unfortunately, this may depend on the real arithmetic calculation of the specific computer. Unit calculation must be carefully applied in these cases. The resulting unit of all operators is commensurable to one of its operandi.

Multiplication, division: Any units may be multiplied or divided. The resulting unit is the multiplication or division of the units of its operandi.

Exponentiation: The exponent must be rational and its unit must be commensurable to 1. Any other unit calculation is nonsense. The resulting unit is obtained by exponentiating the unit of the basis by the exponent.

Build-in functions: Each function belongs to one of the three following groups: 1. The argument(s) must be dimensionless (sin, log, and all other transcendental functions); 2. The argument(s) may have any units, the function does not care about units (ceil, floor, rnd, rndn, etc.); 3. Functions which change the resulting units (sqrt, sqr).

Index operators: They have their corresponding meaning of their binary operators, e.g. SUM is treated in the same way as the addition operator.

Parameter passing in user defined functions: If the parameters are passed by value, then this can be treated in the same way as assignment: formal and actual parameters must be commensurable and the scaling is executed each time. The same is true for the return value(s). If the parameters are passed by reference, then, of course, formal and actual parameters must also be commensurable, but the conversion is more complicated. The following solution can be found in Karr [1978].

We consider a function F with the formal parameter X with units u. Suppose the i-th call on F is done with the actual parameters Z_i with unit u_i. The conversion factor of u_i to u is c_i. Then the following transformations are made:

- change the declaration of F(X:u) by F(X,C) - both parameters with unit 1.
- replace the i-th call on $F(Z_i:u_i)$ by $F(ZZ_i,c_i)$, where ZZ_i is Z_i but dimensionless.
- replace all occurrences of X within F by X*C.
- replace all assignments X = ... within F by X = (...)/C
- handle all calls within F of a function G(X) - supposing G was declared to have a reference parameter - as follows: replace the call G(X) by $G(X,c*c_0)$ where c_0 is the conversion factor between the unit of the formal parameter of G and u. This works even in recursive function calls, and in particular when G=F.

The transformation given above work in any case. Of course, there are better solutions in specific cases; if we know that Z_i is not used as global with the function call, then the Z may be multiplied by C just after entering F and divided by C just before exiting F. If Z is not modified within F, then C*Z may be copied to a temporary variable and Z itself never be used thereafter.

There are several 'pathological units' which cannot be manipulated in the way we explained above. A linear conversion, e.g., $u_2 = a*u_1 + b$ of two units u_1 and u_2 are not allowed, when $b \neq 0$. The conversion between Celsius and Fahrenheit is of this form. Other examples include measurements with an arbitrary reference point such as AD. The above rules cannot be applied to this 'unit'; for example, we cannot write 1990AD - 1986AD = 4AD, but the subtraction makes sense, since we may interpret it as 4 YEARS. Even a more pathological case is decibel. "The justification for designing constructs especially for them is doubtful". [Karr], because they are rarely used in calculations.

SYNTAX AND APPLICATIONS IN LPL

LPL needs to be extended by a UNIT statement, where the elementary units are declared as unique identifiers, and the derived units are defined through their commensurateness relationship. Furthermore, the declaration of any numeric entity (COEFs, VARs, and EQUATIONs) may be extended by a unit declaration. Numeric literals within expressions must also be extensible by indication of the units. Finally, the input and output statements must be extended by an optional unit declaration.

The entire unit concept in LPL is based on the following syntax elements:
First, any unit used within the model must be defined by the modeler through a (new) UNIT statement which starts with the reserved word UNIT followed by a unit declaration. The complete syntax of the UNIT statement is

```
UnitStatement ::= UNIT unit ';'
unit ::= ElementaryUnit  |  DerivedUnit
ElementaryUnit ::= UnitName
DerivedUnit ::= UnitName '=' UnitExpression
UnitExpression ::= UnitFactor { UnitOperator UnitFactor }
UnitFactor ::= number  |  '(' UnitExpression ')'  |  UnitName
UnitOperator := '*'  |  '/'
UnitName ::= identifier
```

Note that this syntax allows also dimensionless units as in

```
UNIT
  gram;              "a elementary unit gram"
  Mile; inch; year; "three other elementary units"
  kilo = 1000;       "derived and dimensionless"
  kg = kilo*gram;    "derived and compound, but commensurable to gram"
  SquareMile = Mile*Mile;          "a compound unit"
  acceleration = inch/year/year;  "another one"
```

Other examples can be found in the Model *EnergyLoss* below. In contrast to other identifiers, unit names cannot be redefined within the entire model. Derived units must be declared and assigned at the same place; they cannot be

declared, and assigned later on. This excludes cyclic unit declarations (e.g. meter is defined in feet or vice versa but not both). This excludes also *inconsistent unit* declarations. But this does not exclude redundant unit declarations. A *redundant unit* is a unit which can be derived by another commensurateness relationship. As an example, we have

```
UNIT
  inches;             "an elementary unit"
  feet = 12*inches;   "a derived unit"
  Fuss = 12*inches;   "'Fuss' is the same as feet and redundant"
```

Redundant units are useful as any other 'redundant' identifier: the same entity has different names. There is nothing wrong with this.
Units might be predefined in separate files as any other part of the model, and they can then be included in any model just by adding the include statement. The modeler does not need to define them each time he uses the same units. Commensurable units are linked by a commensurateness relationship in the UNIT statement as explained above.

Secondly, any numeric entity, such as data, variables, or constraints, may be extended with a unit option simply by adding the reserved word UNIT just after the declaration together with a unit name or a unit expression. An numeric entity without this option is dimensionless. The syntax is

```
Options ::= INTEGER | DEFAULT Number | '[' Range ']' | UnitOption
UnitOption ::= UNIT UnitSpec
UnitSpec ::= UnitName | UnitExpression
```

Examples are

```
COEF  weight UNIT kg;                    "unit of weight is kg"
VAR   cars INTEGER [.,100]  UNIT 1000;   "unit of cars is in 1000"
EQUATION r UNIT 12*kg;                   "r is in dozen of kilogram"
```

Third, unit expressions are allowed in four different parts of the LPL program:
1. in the UNIT statement as left hand side of an assignment.

```
UNIT derived_unitname = <UnitExpression>;
```

2. in the COEF, VAR, and EQUATION (MODEL) statement, to declare the identifier of a specific unit. (Note that SETs do not have units.)

```
COEF MyData UNIT  <UnitExpression>;
VAR MyVar   UNIT  <UnitExpression>;
EQUATION MyCons UNIT <UnitExpression>;
```

3. In a regular expression when using a numeric literal. The numeric literal is extended by a bracket unit expression as in

```
....+  600[hour/day] - ....
```

which means that the numeric literal 600 is to be read in hour/day.
4. in the input and output statements (READ and PRINT statement). The unit option must be indicated just before the semicolon. The read or printed data are automatically converted to the specified unit

```
{ .... see model example below .... }
PRINT profit UNIT DailyIncome; Robots UNIT piece/day; HC;
READ HC UNIT hour/day;
```

Normally, inputs and outputs are measured in units declared by the entity. Thus, if HC is declared in hour/week, then *READ HC;* or *PRINT HC;* are automatically supposed to be given in hour/week. But in the input/output statement this measure can be overwritten, providing the units are commensurable, otherwise an error occurs.

We give now an entire model example to illustrate the use of units. To see the contrast, first a LPL formulation without any use of unit is given.

A firm produces i={1..10} different types of robots. Three production steps must be carried out: a) Production of the components, which takes HC_i hours for each robot i, with a total capacity of 3500 hours per week; b) Mounting (capacity=920 hours per day - say a week has 5 days) taking HM_i hours for each robot i and c) Testing (capacity=3000 hours per week) taking HT_i hours for robot i. The selling prices for each robot i is $price_i$. There are already some robots of each type ordered. How many robots of each type can be produced per week, if the firm wants to maximize the selling profit?
The model may be formulated using LPL as

```
(************  LPL formulation without using units  ************)
SET  i = / 1:10 /;   { ten robots types }
VAR  Robots(i);
COEF HC(i) = [ 5 5 4 5 6 5 7 8 4 7 ];
     HM(i) = [ 4 8 5 6 4 8 7 6 5 3 ];
     HT(i) = [ 6 2 4 6 3 4 5 2 5 3 ];
     Ordered(i)  = [ 20 15 7 6 5 8 9 8 7 5 ];
     Price(i)  = [ 300 200 100 50 50 100 200 100 400 200 ];
MODEL
  Components: SUM(i) HC(i)*Robots(i) <=  3500;
  Mounting:   SUM(i) HM(i)*Robots(i) <=  4800;
  Testing:    SUM(i) HT(i)*Robots(i) <=  3000;
  Order(i):   Robots(i)              >=  Ordered(i);
MAXIMIZE profit: SUM(i) Price(i)*Robots(i);
PRINT profit; Robots; HC;
END
```

Note that the modeler must be careful to translate all capacity measures to the same unit. So 920 hours per day must be translated manually to 4800 hours per

week. If, by neglect, the manual translation was not carried out, the solution of this model is quite different from the 'true' solution (see below):

```
{ ---------------- faulty solution with 920 as right hand side --------------- }
PROFIT = 49770.0000

ROBOTS(I)
      1       2       3       4       5       6       7       8       9      10
   20.0    15.0     7.0     6.0     5.0     8.0     9.0     8.0    87.8     5.0
```

The same model is now formulated in LPL using units for all entities. Note that no manual translation of units is necessary for any expression.

```
(***********  LPL formulation with units  ***********)
UNIT
  piece;                    "quantity unit (numbers)"
  dollar;                   "money unit"
  d100 = 100*dollar;        "another compatible money unit in $100"
  hour;                     "time unit"
  day=8*hour;               "another time unit"
  week=5*day;               "still another time unit"
  DailyIncome = dollar/day; "a compound unit"

SET  i = / 1:10 /;          "10 different robots to produce"

VAR  Robots(i) UNIT piece/week;  "number of robots per week"

COEF HC(i) UNIT hour/piece = [ 5 5 4 5 6 5 7 8 4 7 ];
     HM(i) UNIT hour/piece = [ 4 8 5 6 4 8 7 6 5 3 ];
     HT(i) UNIT hour/piece = [ 6 2 4 6 3 4 5 2 5 3 ];
     Ordered(i) UNIT piece/week  = [ 20 15 7 6 5 8 9 8 7 5 ];
     Price(i)  UNIT d100/piece = [ 3 2 1 0.5 0.5 1 2 1 4 2 ];

EQUATION
  Components UNIT hour/week;
  Mounting UNIT hour/week;
  Testing UNIT hour/week;
  Order(i) UNIT piece/week;
  profit UNIT dollar/week;

MODEL
  Components: SUM(i) HC(i)*Robots(i) <=  3500[hour/week];
  Mounting:   SUM(i) HM(i)*Robots(i) <=  920[hour/day];
  Testing:    SUM(i) HT(i)*Robots(i) <=  3000[hour/week];
  Order(i):   Robots(i)              >=  Ordered(i);
MAXIMIZE profit: SUM(i) Price(i)*Robots(i);
PRINT profit; profit UNIT DailyIncome; Robots; Robots UNIT piece/day; HC;
END
```

The PRINT statement produces the following output (after the model has been correctly solved)

```
PROFIT UNIT DOLLAR/WEEK = 238332.3529

PROFIT UNIT DailyIncome = 47666.4704

ROBOTS(I) UNIT PIECE/WEEK
   1       2       3       4       5       6       7       8       9       10
20.0   281.0     7.0     6.0     5.0     8.0     9.0     8.0   426.1      5.0

ROBOTS(I) UNIT PIECE/DAY
   1       2       3       4       5       6       7       8       9       10
 4.0    56.2     1.4     1.2     1.0     1.6     1.8     1.6    85.2      1.0

HC(I) UNIT HOUR/PIECE
   1       2       3       4       5       6       7       8       9       10
```

```
 5.0     5.0     4.0     5.0     6.0     5.0     7.0     8.0     4.0     7.0
```

Another example is the following LPL model which computes the energy distribution of monoenergetic particles after they have passed a thin foil. [Männer 1986]

```
PROGRAM EnergyLoss;
UNIT
  g;          "grams"             "basic units"
  cm;         "centimeters"
  sec;        "seconds"
  C;          "coulombs"

  k       = 1000;                 "dimensionless scale factor"
  M       = k*k;                  "still dimensionless"

  kg      = k*g;                  "kilograms"
  m       = 100*cm;               "meters"
  mu      = m/M;                  "microns"
  N       = kg*m/(sec*sec);       "newtons (force)"
  erg     = g*cm*cm/(sec*sec);    "ergs (work)"
  eV      = 1.602e-12*erg;        "electron volts"
  MeV     = M*eV;                 "mega electron volts"
  g_cm3   = g/(cm*cm*cm);         "volume mass density"
  C2_Nm2  = C*C/(N*m*m);          "dielectricity"

COEF
  { parameters }
  FinalEnergy, MeanFinalEnergy UNIT MeV  [0,1e3];
  ChargeParticle, ChargeTarget  [0,105];
  AtNrTarget  [0,260];
  DensityTarget UNIT g_cm3  [0,10];
  ThicknessTarget UNIT mu  [0,1e6];

  { result }
  EnergyDistribution;

  { constants }
  Pi                        = 3.14159;
  Epsilon0  UNIT C2_Nm2     = 8.85419e-19;
  e UNIT C                  = 1.6021e-19;
  AMU UNIT g                = 1.660431e-24;
  k1                        = 1.33;

  { intermediate results }
  Alpha, I   UNIT MeV;
  n1  UNIT 1/(cm*cm*cm);

  { calculations }
  I     = 11.5 [eV] * ChargeTarget;
  n1    = DensityTarget/(AtNrTarget*AMU);
  Alpha = sqrt(1/(4*Pi*Epsilon0^2) * n1 * ChargeParticle^2 * e^4 *
          ChargeTarget * ThicknessTarget * (1+k*I/2*MeanFinalEnergy) *
          log(4*MeanFinalEnergy/I)));
  EnergyDistribution = 1[MeV] / (Alpha * sqrt(Pi)) *
                       EXP(-((FinalEnergy-MeanFinalEnergy)/Alpha)^2;
END
```

Two errors have been detected in Männer's formulation of this problem using the LPL compiler. In his PASCAL source code presentation, he wrote the last two formula as

```
  Alpha = sqrt(1/(4*Pi*Epsilon0^2) * ChargeParticle^2 * e^4 *
          ChargeTarget * ThicknessTarget * (1+k*I/2*MeanFinalEnergy) *
```

```
        log(4*MeanFinalEnergy/I)));
EnergyDistribution = 1 / (Alpha * sqrt(Pi)) *
                     EXP(-((FinalEnergy-MeanFinalEnergy)/Alpha)^2;
```

In the first line of this piece of code a factor 'n1' is missing and in the fourth line [MeV] is missing. Two errors which can easily be detected using units within the model!

Another example comes from dynamic systems [Mesterton 1989, p 52]. Let's consider the pollution of a lake over time. Let V be the volume of the lake, x(t) the concentration of pollutants at time t, r the rate at which the water flows out the lake, and P quantity of new pollutants entering the lake per a time unit. Then we have:

Rate of change of pollutants = pollutants inflow – pollutants outflow

$$\frac{dV \cdot x(t)}{dt} = P - r \cdot x(t) \qquad (*)$$

Let's see whether this formula is correct from the dimensional point of view. On the left hand side we have the dimension $[V]\cdot[x]/[t] = m^3 \cdot \frac{g}{m^3} / s = g/s$.

On the right hand side, we have: $[P]-[r]\cdot[x] = \frac{g}{s} - \frac{m^3}{s} \cdot \frac{g}{m^3} = \frac{g}{s} - \frac{g}{s} = g/s$.

Now the solution of (*) is:

$$x(t) = \frac{P}{r} + \left(x(0) - \frac{P}{r} \right) \cdot e^{-rt/V} \qquad (**)$$

Again we can check the validity from the dimensional point of view. The dimension of the left hand side is: $[x] = \frac{g}{m^3}$; the right side also gives

$$\frac{[P]}{[r]} - ([x] - \frac{[P]}{[r]}) \cdot e^{-[r]\cdot[t]/[V]} = \frac{g/s}{m^3/s} - (\frac{g}{m^3} - \frac{g/s}{m^3/s}) \cdot e^{m^3/s \cdot s/m^3} = \frac{g}{m^3} \cdot e^1 = \frac{g}{m^3}$$

An example, where the unit system in LPL cannot be used, is in temperature conversion between Celsius, Fahrenheit, and Kelvin as in the following code. The translation must be done by expressions.

```
{ faulty model : 'pathological units' are not allowed }
UNIT
  Kelvin;
  Celsius=Kelvin-273.2;          { faulty definition }
  Fahrenheit=9/5*Kelvin-460;     { faulty definition }

COEF temp UNIT Celsius := 100;
PRINT temp UNIT Fahrenheit;      { would be nice ! }
END
```

IMPLEMENTATION

It turns out that the implementation of units into the modeling language is quite easy. Say the number of elementary units is n and the number of derived units

is m. Then we reserve space for the unit symbol table as follows:

- declare a matrix M of integer values with m rows and n columns
- declare a vector V of m reals
- initialize both data structures to zeroes
- for every commensurateness relationship (derived unit) fill a row with the corresponding exponents of the elementary units. Enter the conversion factor in the corresponding position of V.

Example

The first model above declares the following units

```
UNIT
  piece;                    "quantity unit (numbers)"
  dollar;                   "money unit"
  d100 = 100*dollar;        "another compatible money unit in $100"
  hour;                     "time unit"
  day=8*hour;               "another time unit"
  week=5*day;               "still another time unit"
  DailyIncome = dollar/day; "a compound unit"
```

The corresponding unit symbol table looks like this

M[i,j]	piece	dollar	hour	V[i]
d100	0	1	0	100
day	0	0	1	5
week	0	0	1	40
DailyIncome	0	1	-1	1/8

To test if two derived units are commensurable is now very easy: all entries in the corresponding row must be equal and the conversion factor is the multiplication of the two corresponding V entries. Thus, the all manipulations with units in expressions can be reduced to completely mechanical procedures via linear algebra. Of course, the physical symbol table may be stored by a sparse matrix data structure to save space. This is also advantageous, because the maximal dimension of the table is not known to the compiler in advance.

CONCLUSION

In this paper, an extension of LPL has been proposed that allows one to incorporate units into the modeling language. Some basic ideas behind the unit calculus was discussed, a full syntax for LPL has been given. Two model examples illustrate all important points concerning units; and an implementation was suggested.

The extensions lead to a model where more errors can be detected by the compiler. This is the most important issue. A convenient effect is the automatic

conversion of commensurable units; but, as I mentioned, this automatic conversion can lead to errors which may be difficult to detect because of real arithmetic round-offs. The advantages, however, are overwhelming, and unit should be used as much as possible in model creation. It enhances the readability extends and the documentation of a model.

The LPL compiler has been implemented with TURBO PASCAL from Borland Inc. under MS/DOS. A version in ANSI C is also implemented. The PASCAL version is actually available at the Institute for Automation and Operations Research, University Fribourg, CH-1700 Fribourg, Switzerland [email: tony.huerlimann@unifr.ch, fax: ++41 37 298 328].

Acknowledgements: This paper was mostly influenced and stimulated by the profound paper of Karr 1978.

REFERENCES

BALDWIN G., Implementation of Physical Units, Sigplan Notices, Vol.22, No.8, August 87, p.45-50

BRADLEY G.H., CLEMENCE R.D., A Type Calculus for Executable Modeling Languages, IMA Journal of Mathematics in Management, 1(1987) p.177-191.

DREIHELLER A., MOERSCHBACHER M., MOHR B., Programming Pascal with Physical Units, Sigplan Notices, Vol.21, No.12, December 1986, p.114-123.

HOUSE R.T., A Proposal for an Extended Form of Type Checking of Expressions, The Computer Journal, Vol.26, No.4, 1983, p.366-374.

HÜRLIMANN T. [1990], Reference Manual for the LPL Modeling Language, Version 3.5, Institute for Automation and Operations Research, Working Paper No. 175, June, Fribourg.

KARR M, LOVEMAN D.B., Incorporation of Units into Programming Languages, Comm. of the ACM, May 1978, Vol.21, No.5, pp.385-391.

MANKIN R., (Letter), Sigplan Notices, Vol.22, No.3, March 1987, p.13.

MÄNNER R., Strong Typing and Physical Units, Sigplan Notices, Vol.21, No.3, March 1986, p.11-20.

MESTERTON-GIBBONS M., [1989], A Concrete Approach to Mathematical Modelling, Addison-Wesley Publ., Redwood City.