%physical

A definitive notation for modelling with physical quantities

Charlie Care

January 2005

1 Introduction

It was in a second-year revision lecture that I came across a formula that did not make sense. I was attempting to understand the subject matter and found myself puzzling over it. I tried putting some numbers in and there I found the fault. The offending equation was not homogeneous.

My state of confusion lasted for a week or two until I found an old exam question that provided the same equation for a slightly different scenario. The difference between the two scenarios came down to one constant in the equation. In the original equation this constant had a value of 1 and in the new equation, a value of 2. Since it was just a multiplication by unity, the constant had been left out of the first equation, but there was the problem. The mysterious constant of value 1 had a dimension. Although it was a numerically unnecessary feature of the first equation, it was a metrologically essential component for the mathematics to make sense. The process of checking the homogeneity of an equation is commonplace in the Physics classroom – students are trained to use the SI unit derivations of the various quantities in an equation and to compare the dimension of the right hand side with the left. If the two dimensions are not equivalent, then the equation is certainly incorrect. If the two dimensions *commensurate*[1] then the equation might be correct. Checking dimensionality can expose errors but is not a proof technique in itself.

Formality in the structure of computing languages is often argued to be necessary if software is ever to be a robust product. In functional languages such as SML, there is great emphasis placed on the importance of *type*. In conventional programming languages, variables are given types such as string, integer or real. Typing variables links in with the representation of quantities on our machines and had great importance when computer memories were small. The distinction between integer and real is however quite unnecessary for the programmer who is uninterested in the internal representation of data.[2]

In a strongly typed language such as SML, there are particularly strict conventions relating to the use of types. For example, in SML there is no direct provision to add an integer to a real; the statement `a = 1.0+2` is meaningless unless the programmer provides a definition of what the `+` operator should do when supplied with a real number and an integer. Restrictions of this kind are claimed to be sensible because type checking provides a mechanism to check that the program represents something meaningful. Getting the types correct is thought to be a good indication of sensible semantics.

I would argue that types are almost unnecessary in a programming language[3] and that the

[1] This term is used in the majority of the referenced literature to indicate the compatibility of two dimensions.

[2] Much thought has gone into the relationship between *type* and *dimension*. Gehani [Geh77] felt that "units of measure reflect the physical characteristics of a quantity while the type indicates how the magnitude of the quantity being measured is to be stored."

[3] I came to this conclusion during a conversation with Prof. Eric Roberts (Stanford University). His view was that although type checking may eliminate a number of errors, these are usually the errors that even a novice programmer would notice immediately. The tkeden tool has very limited type constraints – this is a pattern being followed by modern 'scripting' languages (Perl, Python etc.). In the literature relating to implementing variables with dimension, attempting to mix both units and types seems to cause unnecessary complexity.

programmer should be liberated from having to define internal representations of data. I say almost because it is useful to think of strings as separate from numbers. For the rest of this paper I will be concerned with data that represents numbers. These will be mainly numerals, but I also will use characters as algebraic variables – where the strings simply represent data but are not actually data.

Removing the need for types will enable those who work with computers to be able to think about numbers in the way that is more in tune with the way we think about quantities in everyday situations.

2 Dimensions and programming

The idea of using dimension as a error-checking mechanism for programming has been suggested by a number of people during the last forty years. Most writing on the subject follows the lead of [KL78] and claims that the earliest mention of the idea was by Cheatham.[4] This work provides dimension as an augmentation to the existing type-systems: "Within a restricted application area one can go further and categorise objects not just by their type but also by their meaning" [Gre80]. Early work saw the role of units as "partitioning what would normally be the mode REAL" [KL78].

[CRV03] and [Hal02] discuss classic software problems like the crash of NASA's Mars Climate Orbiter due to numerical values that represented imperial quantities being interpreted as metric quantities. They claim that a programming language that supported units of measure would have had been able to protect against such errors. Although the application of dimension has a use in formal verification of software [CRV03, HM94] possibly more important is the "increased readability" pointed out by [KL78]. Attaching extra information to a programs variables can make the code more accessible to a future reader.

3 Dimensionality of observables in Empirical Modelling

I think that Empirical Modelling (EM) is an ideal environment to explore the power of programming with dimension. Firstly, the discipline of EM makes great use of observation as a concept; preferring to refer to a model's observables rather than a program's variables. Secondly, tkeden (the EM tool) is particularly well-suited to the construction of domain-specific programming notations. By using the tkeden notation framework, it should be possible to enhance parts of existing models with the new notation (see section 11). Comparing the readability of a model with and without the assistance of dimensional notation will indicate how successful such notation might be.

Observation relates the visible state of a model to experience. By this we mean that observables directly relate to the referent of a model and convey a meaning richer than a value. Dimension units might be found to provide some of that context and help enhance tkeden's support for observable-oriented modelling. Currently, the context of an observable is only given by its name and its importance in the modeller's mind. The philosophy of EM has been compared to James' Radical Empiricism with the dualism of mental model and computer model coming together into one computer-based *construal*.[5] One difficulty with this position is that it relies heavily on the assumption that the meaning that a modeller builds into his or her model can be communicated to others through interaction and experience of the model. Due to the complexity of software, EM artifacts are not always straightforward to understand and therefore meaning attached to the model is never communicated from the one modeller to the next. It is a hope that adding this extra information to the observables will help expose the meaning that a modeller is attaching to an observable in their modelling.

[4]The paper references: Cheatham, T. Handling fractions and n-tuples in algebraic languages. Presented at the 15th ACM Annual Meeting, Aug,1960.

[5]Need a reference for this.

4 %physical

Constructing a unit system for eden is fairly straight-forward if the agent-oriented parser[6] is used. This allows the creation of a new *notation* which can be given any textual structure required. The notation will be called %physical – the name *Physical* was also given to an early prototype implementation of dimension using Pascal [DMM86].

Previous work in the field has used a vector of integers, reals or bits to represent different exponents [Bal87, Cle75, Ham96]. I have followed this implementation strategy, but since no significant distinction is made in eden between the numeric types real and integer, I have used a list of numbers.

This list of seven numbers represents the exponents of the seven SI base dimensions. The current implementation of %physical only supports these seven base dimensions; adding new dimensions will be discussed later.

The previous work has always been done in the compiler, or in a pre-processor to the compiler [DMM86]. This meant that dimension checking was done at compile-time and there have always been discussions about whether dimension information should be available at run time. An important point was raised in the literature regarding whether dimension checks should be done in the compiler which would check for all such errors. It was thought by [] that there was no use in having run-time code – possibly in a safety-critical application – throwing an error because a length was added to a mass.

Because eden is an interpreted language these issues are not hugely relevant. The main motivation for providing unit support in eden is to provide the modeller with a more elaborate toolkit of notations that allow him or her to think about their modelling in a more efficient way. It is hoped that physical units would bring the discussion of observables in the modelling process closer to the referent being modelled.

However, there is also an essence of the program (or model) reliability issue attached to this work. Bringing the modeller closer to the physical structure of observables will not only aid the modelling process, but also assist in making models more accessible and easier to understand. This might help modellers reason about the reliability of their models more effectively.[7]

As referred to above, the implementation uses a list of seven numbers to represent the dimension of an observable. The elements of this *dimension vector* refer to the exponents of the different base dimensions in the following order: length, mass, time, electric current, temperature, amount of substance, luminous intensity. Hence the list `[1,0,-1,0,0,0,0]` represents length•time^{-1} – length per unit time or more simply *speed*.

Each dimension has a base unit, these are (respectively): metre (m), kilogramme (kg), second (s), Ampere (A), Kelvin (K), Mole (mol) and candela (cd).

As well as these standard units, there are standard prefixes to allow for scaling. For example, the length 12mm (twelve millimetres) corresponds to 12×10^{-3}m and the kilometre (km) refers to one thousand metres. The prefixes have not been implemented in the initial implementation due to a number of issues:

1. Consider the quantity `12 mmol`: is this 12 millimoles (12×10^{-3} mol), or is it 12 metre•moles? There is no guarantee that the prefixes have their own name-space.

2. The base unit for mass – kg – has a standard prefix built into it. While ms corresponds to a scaling factor of 10^{-3} of the base unit of time, mg corresponds to 10^{-6} scaling of the base unit for mass.

[6] The agent-oriented parser (aop) is a parser built into the tkeden tool. It is particularly useful for building new notations because the rules that govern parsing are stored as internal eden data structures. The flexibility of being able to dynamically modify such rules supports on-the-fly experimentation with the structure of a programming notation.

[7] %physical could be adapted to support a safety-critical system. In %physical, values that are dimensionally inconsistant are represented as undefined. Such a situation could be protected against if the dependency maintainer was augmented so that the modeller would be asked to confirm any input that results in a value becoming undefined.

3. Some of the standard prefixes are not ascii characters (e.g. μ) so the set of prefixes cannot be completely represented by a single character.

The literature has not gone very far in looking at how these prefixes could be included in such a notation. Männer [Män86] saw units with prefixes as another derived unit.

```
UNIT g;        (* mass    *)
     cm;       (* length  *)
     sec;      (* time    *)
     cents;    (* money   *)
```

[Män86, page 14]

From defined units it is also possible to declare a unit as a derivation.

```
UNIT k  = 1000;                (* scale factor    *)
     kg = k * g;               (* mass            *)
     m  = 100 * cm;            (* length          *)
     N  = kg * m/(sec * sec);  (* force           *)
```

[Män86, page 14]

This is not a very elegant solution because every possible scaled unit would have to be manually defined. A number of other writers criticise Männer for this position but it's not clear how they plan to get around the problem.

5 An example %physical translation

The notation %physical will accept physical quantities as input. However, the parser will need to translate the code into eden variables and dependencies. In this section, we will look at some simple %physical statements and show (as boxed code) what the parser translates these statements into.

```
%physical
A = 5 kg;     ## Example of %physical
B = 10 kg;    ##  assignment
```

```
%eden
PH_val_A is 5;                ## Eden representation of the value of 5 kg
PH_dim_A is [0,1,0,0,0,0,0];  ## Eden representation of the dimension of 5 kg

PH_val_B is 10;               ## Eden representation of the value of 10 kg
PH_dim_B is [0,1,0,0,0,0,0];  ## Eden representation of the dimension of 10 kg
```

The sum of two quantities is defined if and only if they have the same dimension. This is managed in the translated eden code below by adding a guard to the `PH_val_C` formula that checks whether the associated dimension is defined.

```
%physical
C=A+B;
```

```
%eden
PH_val_C is (PH_dim_C != @) ? PH_val_A+PH_val_B : @;
PH_dim_C is (PH_dim_A == PH_dim_B) ? PH_dim_A : @;
```

Difference is defined in the same way.

```
%physical
D=A-B;
```

```
%eden
PH_val_D is (PH_dim_D != @) ? PH_val_A-PH_val_B : @;
PH_dim_D is (PH_dim_A == PH_dim_B) ? PH_dim_A : @;
```

The product of two quantities is defined for all dimensions. The dimension of the result is the sum of dimension vectors for the two quantities. An eden function `PH_vectorAdd` is provided that performs pairwise addition on two lists of equal length.

```
%physical
E=A*B;
```

```
%eden
PH_val_E is PH_val
_A*PH_val_B;
PH_dim_E is PH_vectorAdd(PH_dim_A,PH_dim_B);
```

The quotient of two quantities is defined for all dimensions. The dimension of the result is the difference of the operands dimension vectors. `PH_vectorMinus` performs pairwise subtraction on the different exponents.

```
%physical
E=A/B;
```

```
%eden
PH_val_E is PH_val_A/PH_val_B;
PH_dim_E is PH_vectorMinus(PH_dim_A,PH_dim_B);
```

6 Implementation issues

In the final implementation, it was more straightforward to put the dimension guards introduced in the previous section on all values rather than just for formulae involving addition and subtraction. This modification made translation slightly easier because the right hand side of a definition could be translated independently of the left hand side.[8]

Also it was found necessary to re-write the predicate used in the definition of a sums (or differences) dimension to a function names `PH_dimensionComp`. This was because not all statements consisted of one operator and two operands. For example, consider how the following might be translated:

```
%physical
A = b+c+d;
```

```
%eden
PH_val_A is (PH_dim_A != @) ? b + c + d : @;
PH_dim_A is (PH_dim_b == PH_dim_c == PH_dim_d) ? PH_dim_b : @;
```

[8] If guards were added by the rules matching the `+` and `-` operators then these rules would have to know what the left-hand side of the definition was (because the guard refers to a dimension corresponding to the left-hand side). By giving all definitions a guard, the code of the guard can be generated by the same parser rule that parses the left hand side.

Here the predicate would translate to a three-term equality test which is not compatible with the eden syntax.[9] The solution was to replace the test with `PH_dimensionComp`. This function takes two dimensions and returns undefined (denoted by the symbol `@`) if they do not match. If however they do match then the function returns the dimension. This provides a nested-comparison solution to the above problem and also allows for sub-expressions in parentheses to be added to %physical notation.

```
%physical
A = b+c+d;
```

```
%eden
PH_val_A is (PH_dim_A != @) ? 1 + 2 + 3 : @;
PH_dim_A is  PH_dimensionComp(PH_dim_b,PH_dimensionComp(PH_dim_c,PH_dim_d));
```

7 Description of the syntax of %physical

There are two types of statement in the %physical notation. The first is *querying* the value of %physical observables and the second is *definition*. Querying in the %physical notation uses the same syntax as in eden and scout. An observable name is prefixed with the `?` symbol and information about that observable is printed to the terminal. Other than this construction, every other statement is a form of definition and uses the `=` symbol.

7.1 Querying

The provision for being able to query observables relates closer to the experimental nature of EM. It allows a human agent, to observe the state of the model and to follow dependencies. As described above, querying is denoted with the `?` operator and it has the following syntax.

```
? <NAME> ;
```

The syntax of `<NAME>` will be defined in the next section.

7.2 Definition

A definition in %physical creates translated eden dependencies. The syntax of a definition (or redefinition) has the following form:

```
<NAME> = <EXPRESSION> | <FUNCTION> | <FUNCTION_DEFINITION> ;
```

Names A %physical name follows standard variable name conventions. Any combination of alpha-numeric characters (upper and lower cases) and the underscore (_) is allowed, providing that the first character of a name is not a numeric digit or an underscore.

The %physical name-space uses a subset of the eden name-space. A %physical name maps to two eden observables, one of these represents the *value* of the %physical name and the other the *dimension*.[10] For the %physical name `myName`, the corresponding eden observables are `PH_val_myName` for the value and `PH_dim_myName` for the dimension.

As a regular expression, the syntax for **name** is:

```
<NAME> = [a-zA-Z][a-zA-Z0-9_]*
```

[9] It's not even enough to add parentheses. E.g. `(PH_dim_b == (PH_dim_c == PH_dim_d))`, because the predicate `(PH_dim_c == PH_dim_d)` just returns true or false and so cannot be matched with `PH_dim_b`.

[10] This was seen in the example translations given in section 5.

Expressions In the simplest case an expression can just be another %physical name or a literal value. Examples of these are the following %physical definitions.

```
%physical
a = b;       ## Dependency between two %physical observables
b = 12 m;    ## Defining b to be the literal quantity, 12 metres.
```

However, expressions can also be the formula constructed with the operators +,-,*,/. Parentheses can also be used to represent sub-expressions.

```
<EXPRESSION> = <NAME> | <LITERAL> | <FORMULA> | ( <EXPRESSION> )
<FORMULA>    = <EXPRESSION> + <EXPRESSION>
             | <EXPRESSION> - <EXPRESSION>
             | <EXPRESSION> * <EXPRESSION>
             | <EXPRESSION> / <EXPRESSION>
```

For example, other valid %physical definitions are:

```
%physical
a = b*c;
b = (b+c)*(e/(f-j));
```

Literals In %physical, there are three forms of literals. The first two are numeric quantities and either have a dimension or are dimensionless. The third allows a value to be linked to an eden observable. For example:

```
a = 30 m;      ## 30 metres, a length
b = 23 kg;     ## 23 kilogrammes, a mass
c = 12 ms{-1}; ## 12 metres-per-second, a speed
d = 11;        ## 11, a dimensionless quantity
e = eden{a};   ## The value of the eden observable "a"
```

Dimensions are given with their base units. These are the seven SI units. Each unit can be given an optional exponent in curly-braces. If no exponent is given then it is taken as 1. If there is no occurrence of a particular unit in a literal, then the exponent is taken as 0. The syntax of a literal is:

```
<LITERAL>   = <REAL> <UNIT> | eden{<EDEN_NAME>}
<REAL>      = [0-9]+(.[0-9])*
<EDEN_NAME> = [a-zA-Z_][a-zA-Z0-9_]*

<UNIT>      = [<M>][<KG>][<S>][<A>][<K>][<MOL>][<CD>]
<M>         = m   [<EXPONENT>]
<KG>        = kg  [<EXPONENT>]
<S>         = s   [<EXPONENT>]
<A>         = A   [<EXPONENT>]
<K>         = K   [<EXPONENT>]
<MOL>       = mol [<EXPONENT>]
<CD>        = cd  [<EXPONENT>]
<EXPONENT>  = { <NUMBER> }
<NUMBER>    = [0-9]+
```

Function Just as in eden, functions can be on the right-hand side of a dependency. Only functions defined in the %physical notation can be called within %physical. Functions are used in the standard way:

```
%physical
a = 30 m;
b = square(a);
```

The syntax is:

```
<FUNCTION>  = <NAME> ( <NAME_LIST> )
<NAME_LIST> = <NAME> [, <NAME_LIST>]
```

Function definition Functions can be defined in %physical. Inside a function, the programmer can draw on the power of the eden language. The framework provided for defining functions should allow powerful constructions to be added to the notation.

Inside a function a number of predefined features are available. Firstly, if a parameter `a` has been defined then it is possible to refer to that parameters value and dimension using `a_val` and `a_dim` respectively. Secondly, a number of special functions are available to be used. These provide for multiplication and division of dimension.

For example, to produce a 'square' function it will be necessary to take one parameter and to square both its value and dimension. This is done with the following definition. Note the semi-colon after the last curly-brace, the whole block is one physical statement and therefore must end with a semi-colon.

```
%physical
square = function (a) -> b {
## This is eden code below
b_val = a_val*a_val;               ## Scalar multiplication

b_dim = multDim(a_dim,a_dim);      ## Pre-defined function to allow
                                   ##   dimensions to be multiplied
                                   ##   in this 'eden' context
};
```

It must be pointed out that since the body of the function is eden code, the `*` operator in the last example is the eden multiply operator and not the %physical one. It therefore can only multiply scalar values and the dimensions have to be managed in a different line of code. This may appear a little cumbersome; such a simple function could be expressed in the notation as `a * a`. However, the intention is that this construction should provide for the situation where the user would like to write functions that do not obey the standard dimensional algebra.

For example, it might be desirable to write a function that adds 15 to the value of an observable. If the dimension of the observable is not known then it is difficult to write down such a generic formula. However, the function construction provides enough freedom from the constraints of the notation to do exactly what's needed.

```
%physical
add15 = function (a) -> b {
        b_val = a_val+15;
        b_dim = a_dim;
};
```

The syntax is:

```
<FUNCTION_DEFINITION>  = function ( <NAME_LIST> ) -> <NAME> { <EDEN_CODE> }
```

8 Linking with eden

In the previous section the `eden{}` expression was documented. This construction provides a way of linking a value with an external eden observable.

As well as this feature, two other eden procedures are also provided to assist with linking the two notations. The first `PHYSICAL_VAL` allows an eden observable to be dependant on a %physical observable, and the second, `physical` provides a mechanism to make some %physical definitions.

Neither of these procedures do anything particularly technical, the first liberates the user from having to remember how a %physical observable is represented in eden, and the second is just running an execute command. E.g.

```
%physical
a = 12 m;
%eden
PHYSICAL_VAL("myObservable","a");
## The current value of myObservable is now [12,"m"]

PHYSICAL("a = 12 kgm{2};");
## The current value of myObservable is now [12,"m{2}kg"]
```

An improved syntax for linking eden observables to %physical is discussed in section 12.

9 Adding new dimensions and units

It is desirable that users of %physical should be able to add their own units to the system. Such a feature would enable automatic conversion between units. Adding extra dimensions is more difficult but could be implemented by adding an extra number to the tail of each dimension vector.

Adding units causes dynamic changes to the parser definitions – this has been implemented with dependency (see section 10). Providing that the units added do not duplicate the existing units this should not be a problem. The existing parser needed `mol` to be searched before `m` because of the common prefix between them. As long as the base units are matched last then there should be no common-prefix problems. This assumes that the first characters of `kg` or `cd` are not used as user-defined units; but since these symbols are standard representations for kilo- and centi- it is probably a fair assumption.

Parsing might be difficult with many user defined units. For example it would be wrong to allow the definition of hour and hours, because there would be no way of distinguishing between hours and hour-seconds. One technique could be to add an optional period between each unit.

```
%physical
a = 12 m.s-1;
b = 1 W.hour;
```

9.1 Methods of defining new units

Currently, the only way to add a user-defined unit to the notation is by adding an entry to a list named `user-defined`. This list has an element corresponding to each user-defined unit. Provision will have to be added to the notation to allow new units to be defined from within the notation. One such syntax could be:

```
New unit N = kgms{-2} ## the Newton
```

Another idea might be to support full names as well as symbols.

```
New unit hour (hr) = 3600 s;     ## Add hour unit - can use hour or hr
New unit Newton (N) = kgms{-2};  ## Add Newton - can use Newton or N
a = 12 hr;
b = 12 N.hour;
c = 123 Nhr;
```

9.2 Issues relating to user-defined units

To be in harmony with the EM paradigm, it should be possible to re-define new units during run-time. For example, initially we may want to declare some imperial units to work in our model.

```
new unit ft = 0.3 m;
...
a = 10 ft; ## internally 3 metres
```

But later we may want to improve our conversion factor:

```
new unit ft = 0.3048 m;
...
b = 10 ft; ## internally 3.048 metres
```

When we re-defined the foot we changed the parser so that input data could be converted into a metric internal representation. Because we are doing this conversion in the parsing the result of the above script is that a does not equal b (even though they have the same definition). A priority for a future implementation of %physical would be to represent 10 ft as some kind of dependency that is updated when the conversion factor changes.

The same problem would arise if I were to decide that I wanted feet to be a unit of time. Would it be desirable for this to be allowed to propagate through the system?

10 Changing the grammar of a language as it is parsed

At the moment, the first unit rule is called by the literal rule. When a new unit is defined, a rule between these two that will match the new unit has to be created. However, this will be problematic if it is desired to *forget* or remove that new unit. One solution is to have some kind of function that generates the whole parser based on a set of simple definitions that when changed cause the whole structure to be rebuilt. Alternatively, the parser could make use of the dependency maintainer and have a structure where different rules are added using dependency.

For example, consider the following fragment of the %physical parser:

```
%eden
PHparse_literal = [ "pivot", " ", ["PHparse_value", "PHparse_unit1"],
["fail", "PHparse_edenObservable"],
["action",
  ["later",
"$v=[str($p1),str(PH_determineDimension($p2))];"
   ]]];
...
```

This is a rule that *pivots* on the white space between the value and unit of a literal. The value is then matched by `PHparse_value` and the unit by `PHparse_unit1` shown in bold. We can use dependency to create a dynamic grammar. For example, the fragment below has had `PHparse_unit1` replaced with a different observable that links to the old rule by dependency:

```
%eden
PHparse_literal = [ "pivot", " ", ["PHparse_value", "PHparse_unit"],
["fail", "PHparse_edenObservable"],
["action",
  ["later",
"$v=[str($p1),str(PH_determineDimension($p2))];"
   ]]];
...
PHparse_unit is PHparse_unit1;
```

The new definitions have not changed the parser; however, we can now introduce more dynamic behaviour. If we have a list called `user-defined` which has an element for each user-defined unit then we can change the parser based on the existence of extra rules.

```
PHparse_unit is (user-defined == []) ? PHparse_unit1 : PHparse_newUnit1;
```

As long as any new rules are all linked to `PHparse_newUnit1` and eventually *fail* back into the rules for the original units (`PHparse_unit1`) then the parser will continue to work.

Once we have a dynamic grammar, do we want changes to be made through dependency against previous values?

11 Applying %physical to the cruise contol model

The VCCS is a classic EM model by Ian Bridge [Bri91] that models a car going over a hill and a cruise control attempting to maintain a constant velocity. The model has a number of physical quantities and it is hoped that %physical could help provide an account of the relationships between different observables.

There is an LSD account of this model that makes use of units. Bridge's implementation refers to this account in the comments and structure of the model. His comments provide evidence of a modelling that makes rich use of dimensional analysis. This is clearly seen in the following fragment of eden definitions.

```
  pi     = 3.14159;
  mass   = 2500.0;  /* total mass of car & contents [kg]     */
  windK  = 5.0;     /* wind resistance factor [N m^2 s^2]    */
  rollK  = 50.0;    /* rolling resistance factor [N m^-1 s]  */
  gravK  = 9.81;    /* acceleration due to gravity [m s^-2]  */
  brakK  = 1500.0;  /* braking (viscous) constant [N m^-1 s] */
  forcK  = 40.0;    /* torque to force conversion [m^-1]     */
  sticK  = 100.0;   /* static friction force [N]             */
                                                  [Bri91, vehicle_dynamics.e]
```

The eden code above could be re-written as:

```
%physical
new unit N = kgms-2 ## Force = mass x acceleration
                            ## acceleration = distance / time*time
mass  = 2500.0 kg;
windK = 5.0 Nm2s2;
...
```

In the current implementation of %physical, re-definition of the Newton (line 1 above) would not propagate through the system (see section 9). Also the `new unit` construction has not been added to the parser.

Elsewhere in the model some quantities are given in miles per hour and %physical could be used to handle to equivilence between different units of measure. The conversion between units is currently handled by eden functions:

```
%eden
func mph_to_mps {
  /* convert miles/hour to metres/sec */
  para mph;
  return 0.448 * mph;
}

func mps_to_mph {
  /* convert metres/sec to miles/hour */
  para mps;
  return 2.232 * mps;
}
```

There should be provision for this kind of task in the %physical notation. This would require some form of 'casting' of units into the desired format.

12 Managing different output units

Support is already in %physical to allow definitions to be made in different units. However, following the implementation of [] all quantities are represented internally in terms of base units.

```
%physical
new unit miles = 1600 m;   ## Define miles
new unit hours = 3600 s;   ## Define hours
mph = 3 milehour{-1};      ## %physical stores the value: mph = 16 ms-1
```

What is needed is to be able to 'cast' into a particular unit for output formats. For example, in the VCCS speeds have to be converted to miles per hour so that the speedometer can be defined in terms of miles per hour. We would want to do something of the following kind.

```
%eden
/* Some code defining a speedometer in terms of an observable speedVal */
%eden
PHYSICAL_VAL("speedVal","mph(speed)")
```

Here mph could be defined as a physical function:

```
mph = function (a) -> b {
    b_dim = a_dim;
    b_val = scaling_factor * a_val;
}
```

The difficulty is that this would scale the value but still give the unit "m" because we are not distinguishing between m and length. This is not ideal.

One way of solving the problem would be to provide a means for defining a dependency between a dimensionless eden observable and a %physical observable. The proposed syntax takes the following form:

```
%physical
new unit mile = 1600 m;             ## Definition of mile
new unit hour = 3600 s;             ## Definition of hour

a = 5 ms{-1};                       ## a is 5 metres per second

?a;                                 ## Will output current value of a in base
units

eden{myVal} = a;                    ## A new component of %physical that is
equivalent to making
                                    ##   the eden definition
PHYSICAL_VAL("myVal","a");

?a : m->mile,s->hour                ## Will output current value of a in mph

eden{myval} =  a : m->mile,s->hour; ## Will make the value of myVal dependent
                                    ##   on the value of a (in mph)
```

13 Reflection

Working with EM has emphasised my need to link computing with the empirical sciences. In EM there has been a tendency to withdraw from the formal camps of computer science. However, my personal view is that EM is not simply an 'informal' computer science. Perhaps it is more that those involved in the research have found that the formality promoted actually gets in the way of the experimental approach to modelling. Taking metrology seriously in our programming will improve our ability to add the discipline that those interested in formal methods may feel our modelling lacks.

Replacing types with units will provide a discipline of attaching semantics to our calculations. In the example of the non-homogeneous equation, that equation was lacking a formal semantics and was effectively informality under a formal mask.

In EM, many of our models represent state as experienced in the physical world. This is where the power of %physical can be explored. Adding units to the observables can help the newcomer to a model get far more meaning from observables than they could from raw numbers.

References

[Bal87] Geoff Baldwin. Implementation of physical units. *SIGPLAN Not.*, 22(8), 1987.

[Bri91] Ian Bridge. *Vehicle Cruise Control Simulation.* Empricial Modelling, Dept. Computer Science, University of Warwick, 1991. Available at: http://empublic.dcs.warwick.ac.uk/projects/cruisecontrolBridge1991/.

[Cle75] J. C. Cleaveland. Meaning and syntactic redundancy. In Stephan A. Schuman, editor, *New Directions in Algorithmic Languages*, pages 115–124. Institut de Recherche D'Informatique et D'Automatic, 1975.

[CRV03] Feng Chen, Gigore Rosu, and Ran Prasad Venkatensan. Rule-based analysis of dimensional safety. In *Rewriting Techniques and Applications*, Valencia, Spain, 2003. http://fsl.cs.uiuc.edu/ grosu/.

[DMM86] A Dreiheller, B Mohr, and M Moerschbacher. Programming pascal with physical units. *SIGPLAN Not.*, 21(12):114–123, 1986.

[Geh77] Narain Gehani. Units of measure as a data attribute. *Computer Languages*, pages 93–111, 1977.

[Gre80] T. R. Green. Programming as a cognitve activity. In *Human Interaction With Computers*, pages 271–320. Academic Press, London, 1980.

[Hal02] B.D. Hall. Software support for physical quantities. Technical report, Measurement Standards Laboratory of New Zealand, Industrial Research Ltd., 2002.

[Ham96] Bruce Hamilton. A compact representation of units. Technical report, Measurement Systems Department, Hewlett-Packard Laboratories, 1996.

[HM94] Ian J. Hayes and Brendan P. Mahony. Using units of measurement in formal specifications. Technical report, Software Verification Research Centre, University of Queensland, Brisbane, Australia, 1994.

[KL78] Michael Karr and David B. Loveman. Incorporation of units into programming languages. *Communications of the ACM*, 21(5):385–391, 1978.

[Män86] R. Männer. Strong typing and physical units. *SIGPLAN Not.*, 21(3):11–20, 1986.