%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; by way of entry, 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 here was the problem. The mysterious constant of value 1 had a dimension. Although it was an 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 units 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. The idea of 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 structure of a Modern Digital computer.[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 for the programmer to check that they have programmed 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 relect 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 in a conversation with Prof. Eric Roberts (Stanford). His view was that although type checking may eliminate a few errors, these are usually the errors that even a novice programmer would notice straight away. The tkeden tool has very limited type contraints and this is a pattern being followed by many recent

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 [KDBL78] and claims that the earliest mention of the idea was [?]. This work provides dimension as an augmentation to the existing type-systems.

[CRV] and [?] discuss classic software problems in safety critical systems caused by numerical values that represented metric quantities being interpreted as imperial quantities. They claim that a programming language that supported units of measure would have had been able to protect against such errors.

Although they might have use in formal verification of software; I would like to draw attention to a possible use in making source code more accessible. This "increased readability" is also pointed out by [KDBL78].

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 Empirical Modelling 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 is also possible to enhance parts of existing models with the new notation which will assist with any evaluation of modelling with dimension.

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.

4 %physical

Constructing a unit system for eden is fairly straight-forward if the the agent-oriented parser [4] 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 implementation using Pascal and what appears to have been the only example constructed for industrial use.

Previous work in the field has used a vector of integers, reals or bits to represent different exponents. 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.

developments in scripting languages (perl,python etc.). In the literature relating to implementing variables with dimension, attempting to mix both units and types seems to provide unneccesary complexity.

[4]The agent-oriented parser (aop) is a parser built in to the tkeden tool. It is particulary 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.

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 some literature regarding whether dimension checks should be all 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 (for example) 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 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.[5]

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 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).
[6]

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.
[7]

What is needed for the notation's parser is to be able to accept textual representations of these units and their prefixes and to convert them into the internal list implementation. For the initial implementation, the prefixes were not implemented. The discussion of the implementation of these will be returned to later.

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
```

[5]%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 input that results in a value becoming undefined.

[6]add stuff about units vs dimension

[7]Add stuff about prefixes

```
%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 only defined 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.

```
%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.

```
%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 made translation slightly easier because the right hand side of a definition could be translated indpendently of the left hand side.[8]

Also it was found necessary to change the predicate used in the dimension definition for sums and differences to a function. 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 : @;
```

Here the predicate would translated 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 (`@`) 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 already said, querying is denoted with the `?` operator and it has the following syntax.

```
? <NAME> ;
```

The `<NAME>` tag 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> ;
```

[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 the correponding dimension). By making all definitions have a guard, the code of the guard can be generated by the same parser rule that parses the left hand side.

[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`

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_]*
```

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 are 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>]
```

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

```
<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 when such a simple function could be expressed in the notation as `a * a`, but the intention is that it 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 particulary 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"]
```

9 What type of function?

The choice is between something that is procedural and allows the tweaking of individual parts and one that is definitive and can be viewed as a way of generating structures of definitions.

10 The Agent Oriented Parser

The syntax that we use to interact with the Agent Oriented Parser (aop) is not very accessible. It consists of a complicated structure of lists and strings to define each projection.

If we were to have a notation to interact with the aop then the interface could be significantly simplified.

11 Adding new dimensions and units

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. E.g. m.s-1 and W.hour.

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

New unit hour (hr) = 3600 s; New unit Newton (N) = kgms-2;

would allow the use of

```
a = 12 hr;
b = 12 N.hour;
c = 123 Nhr;
```

12 Changing the grammar of a language as it is parsed

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 units will require a change to the parser definitions. For the time being, no extra dimensions will be added, just derived units.

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. If the base units are matched last then this problem will be overcome (providing no-one uses the first characters of `kg` or `cd`, but since these are standard for kilo- and centi- it should not b too much of a problem.

Other than that, the parser rules would need to be intercepted. At the moment, the first unit rule is called by the literal rule. Defining a new unit could add a rule between these two that will match the new unit. However, this will be problematic if it is desired to *forget* or remove that new unit. The only obvious 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 have a structure and the different rules could be placed around using dependency.

For example, a chunk of the current parser is given below.

```
%eden
## Literal - consists of a value and a dimension

PHparse_literal = [ "pivot", " ", ["PHparse_value", "PHparse_unit"],
["fail", "PHparse_edenObservable"],
["action",
  ["later",
"$v=[str($p1),str(PH_determineDimension($p2))];"
   ]]];

...

PHparse_unit = [ "prefix","mol{","PHparse_exponent", ["fail","PHparse_unit2"],
["action",["later","$v=[\"mol_\"]//$p1;"]]];
```

Instead of this we could add an extra rule `PHparse_userUnit`.

```
%eden
## Literal - consists of a value and a dimension

PHparse_literal = [ "pivot", " ", ["PHparse_value", "PHparse_userUnit"],
                        ["fail", "PHparse_edenObservable"],
                        ["action",
                          ["later",
                                "$v=[str($p1),str(PH_determineDimension($p2))];"
                                ]]];

...
```

```
PHparse_unit = [ "prefix","mol{","PHparse_exponent", ["fail","PHparse_unit2"],
                        ["action",["later","$v=[\"mol_\"]//$p1;"]]];

PHparse_userUnit is PHparse_unit;
```

Now the functionality is no different than before, but supposing that we have a list of user-defined units. Then we can re-define the dependency to reflect the need to look at different rules.

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

Take the problem of adding feet to %physical - rename the PH_parse_unit to PHparse_unit1. All things point back to PHparse_unit as the first so this rule can be now linked to the start through dependency.

```
PHparse_unit is PHparse_userUnit;
PHparse_userUnit is PHparse_userUnit1;
PHparse_userUnitN is PHparse_unit1;

PHparse_userUnit1 =  [ "prefix","ft{","PHparse_exponent",
["fail","PHparse_userUnit1a"],
["action",["later","$v=[\"m_\",$p1[1],\"scale_\",0.33]//tail($p1);"]]];
PHparse_userUnit1a =  [ "prefix","ft","PHparse_unit",
["fail","PHparse_userUnitN"],
["action",["later","$v=[\"m_\",1,\"scale_\",0.33]//$p1;"]]];
```

So all that's needed is to automatically generate the above based on a set of user-defined units. One possible representation might be to have a list of lists. For example:

```
PH_userUnits = [["feet","ft","0.33","m"]];

PHparse_unit is (PH_userUnits# > 0) ? PHparse_userUnit : PHparse_unit1;

proc PH_updateUserUnits : PH_userUnits {

if (PH_userUnits# > 0) {


## generate the code for each parser rule


}
```

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

E.g. is 1 ft = 0.33 m and we enter 5 ft. What happens when we improve the conversion factor afterwards? Presumably we would want the system to keep track of what was feet and to improve the quality of the translation. The next question then is are we allowed to change the dimension of 'feet' after it has been defined. It s possible that we might want to consider a situation where feet is a unit of time for example. This will be particularly difficult with the translation system given above which basically uses the SI idenies tro thmake the parser think it parser d the SI units. What would be better would be to sotre feet as a translation function that takes in a value/dimension pair and returns a dimension value pair. In this sense, the ft definition would then be more in tune with the EM philosophy (although I cannot ever see myself needing to model feet as a unit of time it might be useful in some context. Perhaps when modelling human activity (a very EM thing to do) because there we might want to model confusion which this most certainly is.).

13 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 us of dimensional analysis. This is clearly seen in this 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]             */
```

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 11). 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.

14 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;
new unit hours = 3600 s;
mph = 3 milehour-1;
>> mph = 16 ms-1
```

This would need the notation to allow the user to provide some expectation about the units desired. e.g.

```
%physical
speed = 3 milehour-1;
mph = MPH(speed);
```

Although this is actually quite a hard thing to derive since the there are so many possible 'casting' functions possible. The functions could be defined specifically, however they then couldn't be represented in the %physical system. The need to get such output quantities is only an interface issue since internal representation as base units would not make the following equation meaningless.

```
%physical
speed = 3 mile.hour-1; ## speed now has an internal representation in ms-1
speed2 = speed + 15 ms-1;
speed3 = speed + 10 mile.hour-1;
```

The need for a particular unit is however useful for output formats. E.g.

```
%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. These would have the same dimension vector. Perhaps we should be carrying scaling info around as well as dimension info. E.g. The vector could have the following form:

```
x_val = 5;
x_dim = [1,[1,0,0,0,0,0,0]]; ## for x = 5 metres
x_dim = ["k",[1,0,0,0,0,0,0]]; ## for x = 5 km
x_dim = [1600,[1,0,0,0,0,0,0]]; ## for x = 5 miles
```

The benefit with such a representation would be that the scaling factor could be an observable itself. Hence when the mile is defined an internal value could be set and then (by dependency) the scale factor could be refined at a later date.

```
PH_unit_val_mile = 1600;
PH_unit_dim_mile = [1,0,0,0,0,0,0];
...
x_dim = [PH_unit_val_mile,PH_unit_dim_mile];
```

This is similar to the syntax proposed by [] where new units were simply scaling values i.e. X = (5 m)*(x mile.m-1);

The scaling is more associated with the absolute magnitude than with the dimension so perhaps the scaling factor could be attached to the value. Having said that units themselves link to the scaling factors so perhaps its own observable name should be used.

```
PH_val_x; ## value of x
PH_scale_x; ## scaling factor of x
PH_dim_x; ## dimension of x
```

then for addition c = a+b;

```
val_c = scale_a*val_a + scale_b*val_b
scale_c = 1;
dim_c = dim_a = dim+b;
```

Here the scaling factor will be eliminated through equations so it seems slightly redundant. It would be better to stick with the current implementation and to provide better handles for scaling.

Problem: we have a quantity in the system which has a value of x ms-1. We would like an output in the form. Mile-per-hour when we query the observable. We have already defined the mile and the hour. We would like to get this output as (a) a query (human agent) and (b) as an external (eden) observable.

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

a = 5 ms{-1};

?a; ## will give current value in base units

## (a)
?a{m->mile,s->hour} ## could give 5*1600*(3600)^{-1} mile.hour{-1}

## (b)
eden{eden_speed} =  a : m->mile,s->hour;

eden{eden_speed} =  mile.hour{-1};
```

15 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; it is perhaps more that those involved in the research have found that the formality often promoted actually gets in the way of the experimental approach to modelling. Perhaps taking metrology seriously in our programming will improve our ability to add the discipline that those interested in formal methods may feel that 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 such a system would be seen. By adding units to the observables in a model they would be much easier to interact with since the newcomer to a model could get far more meaning from observables than they could from raw numbers.

References

[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/.

[CRV] Feng Chen, Gigore Rosu, and Ran Prasad Venkatensan. Rule-based analysis of dimensional safety. 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.

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