Dimensional Inference Using Symbol Lives

Brian Farrimond
Liverpool Hope University,
University of Cape Town
farrimb@hope.ac.uk

John Collins
Simcon Ltd
University of Cape Town
jcollins@simcon.uk.com

Abstract

Scientific equations embedded in computer programs must obey the rules for dimensional consistency. Many efforts have been made to enforce these rules within computer programs. Some methods require the user to modify the program by adding dimensional information either as extensions to the programming language or else by using existing language facilities. The large amount of effort required to modify large scientific and engineering programs in this way is usually uneconomic and can be prone to error if done manually. Consequently, methods that do not require modifications to the program have been developed. However, these methods are limited in what they can discover and find it hard to tell the user where, precisely, in his program the error has been made.

This paper describes a novel approach which employs the concept of symbol lives to analyse the dimensional implications of the users source code. The method, implemented for Fortran programs in the software engineering toolkit, WinFPT, is fast, systematic and identifies the locations of errors in the user's source code.

Keywords--- Software Engineering, Dimensional Analysis, Static Analysis, Fortran

1. Background

An important class of errors, particularly within scientific and engineering programs, is the incorrect implementation of formulae. Many of these errors are often manually detectable by applying **dimensional analysis** which applies the principle that the physical dimensions of components of a formula must be consistent otherwise the formula must be incorrect.

In physics and engineering, the dimensions of a physical quantity describe how the quantity is related to fundamental or base dimensions. In mechanics these base dimensions are mass, length and time. The International System of Units (SI) [8] adds to these three, temperature, electric current, luminous intensity and amount of substance. All physical quantities have dimensions that are products of these base dimensions i.e. the result of multiplying together powers of these base dimensions. Consider, for example in mechanics that the base dimensions mass, length and time are represented by the symbols M, L and T. Then the physical quantity speed has the dimensions $L.T^{-1}$. Area has the dimensions L^2 and force has the dimensions $M.L.T^{-2}$. The powers may be fractional. In general, if the base dimensions are `A1, A2, ... An` then any physical quantity will have dimensions:

$A1^{a1}.A2^{a2}. \ldots An^{an}$

where `a1, a2, a3 ... an` are real numbers.

The terms "units" and "dimensions" are sometimes used interchangeably but in fact they mean two different things. A dimension is a particular property while a unit is a measure of that property in terms of some arbitrary scale. Thus length is a dimension while a metre is a unit of length defined as the length of the path travelled by light in vacuum during a time interval of 1/299,792,458 of a second. [8]. In Fortran, the term DIMENSION refers specifically to a mathematical property of arrays. In order to avoid confusion over the terminology, we decided to use the word "units" instead of "dimensions" when describing the mechanism employed in WinFPT in this paper. Since the method does not rely on physical units and deals entirely with dimensions (it does not distinguish, for example, between metres and centimetres) we felt able to do this without causing confusion. Consequently, we use the terminology that *Units(A)* represents the dimensions of quantity A. We describe, at the end of Section 8, circumstances in which the method can identify erroneous mixing of units such as metres and feet.

WinFPT [14] is a suite of tools for writing, maintaining and migrating Fortran programs. It has several hundred commands for checking, reporting on and modifying Fortran source code. It is currently being extended to include dimension checking by inference. This paper reports on the progress of this work.

The methods described are applicable to any high level languages that deal with scientific or engineering quantities. Fortran was chosen in the present study because the authors had access to a powerful existing analysis tool in WinFPT for Fortran. Implementation of the methods in other languages would require equivalent analysis tools.

2. Dimensional Constraints

We now identify the dimensional constraints, labeled C1 to C7, that WinFPT aims to apply in its analysis. (*m* and *n* are real numbers.)

C1. X relational operator Y $\Rightarrow$ *Units(X) = Units(Y)*
e.g. *X > Y* $\Rightarrow$ *Units(X) = Units(Y)*

C2. $X+ Y$ or $X - Y \Rightarrow Units(X) = Units(Y)$

A converse of this is:

C2c1. $X^n + X^m$ (where $m \neq n$) $\Rightarrow Units(X) = 1$

where 1 means dimensionless.

C3. $Units(X^m * Y^n) = Units(X^m) * Units(Y^n)$

C4. Transcendental function of $X \Rightarrow Units(X) = 1$

e.g. $sin(X) \Rightarrow Units(X) = 1$

C5. The return values of transcendental functions are dimensionless.

e.g. $X = sin(Y) \Rightarrow Units(Y) = 1$

C6. Certain Fortran instrinsic functions are polymorphic

e.g.

$X = MAX(Y, Z) \Rightarrow Units(X) = Units(Y) = Units(Z)$

C7. Function definitions can determine the dimensional relationships between actual arguments

e.g. given the following subroutine definition:

```
SUBROUTINE S(A, B)
   A = B*B
END
```

then

```
CALL S(X,Y)
```

implies $Units(X) = Units(Y^2)$

At first glance, it might appear that each side of an assignment statement must be dimensionally consistent. This assumes that the left hand side already has a dimension before the assignment takes place. If the left hand side has been annotated to have a dimension in its declaration then this will indeed be the case. However, if no dimensions have been assigned to variables when declared then all an assignment does is to establish the dimensions of the left hand side. It is not a constraint. Our concept of lives, introduced below, is based on this notion.

Note that none of these constraints make reference to base dimensions. This paper will show how the constraints can be used to check programs without reference to base dimensions and so demonstrate how a powerful set of checks can be carried out by a suitable tool without the need for any modification of the source code under examination.

The rest of this paper is organised as follows: Section 3 describes related work, Section 4 shows how dimensional analysis reveals possible errors, Section 5 introduces the concept of symbol lives, Section 6 shows how a symbol's dimensions are related to its lives, Section 7 illustrates the WinFPT dimensions analysis mechanism by means of an example. Section 8 describes WinFPT output and discusses the opportunities and issues associated with the work done developing WinFPT, Section 9 contains closing remarks.

3. Related work

Many efforts have been made to detect dimensional inconsistency in order to trap errors in computer programs. Attempts normally employ modifications to the programmers source code. Some of these modifications have required language extensions. Examples include: House [5], Dreiheller et al [2] and Van Delft [12].

Other proposed program modifications do not rely on changes to a programming language but instead, use existing language features. Hilfinger [4] defines an Ada package to handle units. Cmelik and Gehani [1] introduce C++ unit aware classes to replace the double and int data types. Overloading of operators is used to detect attempts to violate dimensional consistency. Umrigar [11] uses templates in C++, defining operations on dimensions which enable inconsistencies to be detected at compile time. Petty [9] proposes using a structure to replace the REAL data type in Fortran. The structure stores dimension information alongside the magnitude of the variable value. Operator overloading detects dimensional inconsistencies in the use of these variables at run time. This approach follows that established by Cmelik and Gehani.

In the 1990s, new approaches were inspired by the type inference mechanisms provided in languages such as ML. Type inference enables a compiler to infer types if the programmer leaves them out. Several authors, including Wand and O'Keefe[13] and Kennedy [7], have proposed using this feature to embed unit types in the language type systems. The inference mechanism looks for consistency of use of unit types derived from a user-supplied set of base units. Linear algebra is used to solve the constraint equations generated from dimensional consistency analysis. The ML language has been extended to enable unit type annotations for symbols. Similar work has been done for the gPROMS simulation language by Sandberg et al [10] who use annotations in the form of tagged block comments to avoid the need for language extensions.

In scientific and engineering computing where dimension checking has its most obvious application, it is quite normal to find large bodies of complex legacy code in use. Applying modifications as described in the previous paragraphs is often not economically feasible. The Osprey system developed by Jiang and Su [6] uses type inference to carry out dimensional analysis of C programs without the need for comprehensive code modifications. Osprey uses a pipeline of tools for constraint solving. It can use annotation of unit types in the C code but it is not required. However, Osprey may not discover any unit errors when there are not sufficient annotations in the program.

Guo and McCamant [3] have developed a system for C programs that infers the unit types of variables from their use in the programs. It infers base units and identifies a combination of these for each variable and constant in the program. Their work does not require the user to make any unit type annotations at all to the C code. The resulting analysis can be used by programmers to identify bugs indicated by the unit types not matching up with their

intuition. Their system also provides a framework for systematically associating base units with real world units and automating the process of identifying combinations of these base units for all the program variables. There are four stages in their system:

- **constraint generation** from an analysis of the program source code
- **constraint simplification** by applying meaning-preserving transformations and heuristics
- **constraint solving** using linear algebra techniques outputting a minimal set of inferred base units. Each variable is expressed in terms of these inferred base units.
- **user interface for guided annotations** to allow the user to provide user-defined units for variables under the guidance of the solved constraints.

Their method is limited by a static analysis that is flow-insensitive. It only associates a variable with one unit type whereas it is quite possible that the same variable may be used for several different unit types over the course of a program. In one example program, a single global variable is used repeatedly when reading floating point values of a number of different units causing them all to be considered dimensionless. We explain below how this problem can be overcome using our concept of symbol lives. The analysis proposed by Guo and McCamant is also context-insensitive. A function being called at several different locations may on each occasion be passed arguments whose unit types differ. The method is only able to assign one set of units to the return value of a function. The method avoids resulting inconsistencies by treating all return values as dimensionless. Different unit types fields within a data structure are not handled. All fields are assumed to have the same unit type. The elements of an array are treated as all having the same unit type.

An important issue arising in the various methods applied is the meaningfulness of the generated warning and error messages. For example, in Petty's work, an error is reported but no indication of where in the code that it took place. This does not help the efficiency of debugging. In work that solves constraint equations, the source of the reported error can be very difficult to identify.

This paper describes a technique based on dimensional inference that overcomes problems present in the related work described above. In the next section we shall indicate how dimensional inference can be used

4. Applying dimensional inference

We now consider the implications of applying dimensional constraints to Fortran programs. Only real variables, real constants and functions returning real values are checked for consistency.

In the code fragment:

```
R = B + C
```

where **R**, **B** and **C** are real variables which represent some physical quantities in the program, the dimensions of **R** must be the same as the dimensions of **B** and the dimensions of **C**. Similarly, in the fragment:

```
A = B * C
```

the dimensions of **A** must be the dimensions of **B** multiplied by the dimensions of **C**.

Inferences like these are propagated through a program. The analysis yields a set of dimensions which describe the physical quantities used, and the relationships between the dimensions are checked for consistency. If the fragments above are followed by a fragment:

```
E = A + C
```

then **C** is measured in the dimensions of **C** but **A** is measured in the square of the dimensions of **C**. An error has been detected.

5. Symbol lives

It is not always possible to attach a single inference of dimensions to each variable. Consider the code fragment:

```
! Sort the tables by weight
change_f = .TRUE.
DO WHILE (change_f)
  DO i = 2, n
    IF (weight(i) < weight(i-1)) THEN
      temp = weight( i )            ! (1)
      weight( i ) = weight( i-1 )
      weight( i-1 ) = temp
      temp = height( i )            ! (2)
      height( i ) = height( i-1 )
      height( i-1 ) = temp
      change_f = .TRUE.
    ENDIF
  ENDDO
ENDDO
```

In this fragment, **temp** must first have the dimensions of weight and then the dimensions of height. The implication is that weight and height should have the same dimensions. This problem is overcome by defining “lives” of each variable.

A variable symbol life starts with an assignment (or, in Fortran, with initialisation in a DATA specification) and ends with another assignment or when the variable is no longer in use. In the example above, **temp** has two independent lives, started at (1) and (2) respectively. The different lives can, and in this case do, have different dimensions.

Note that every assignment does not necessarily start a new life. In the fragment:

```
IF (loaded) THEN
   weight = vehicle_wt + payload ! (3)
ELSE
   weight = vehicle_wt           ! (4)
ENDIF
acc = thrust / weight            ! (5)
```

either of the values of weight assigned at (3) and (4) may be used in the expression at (5). In cases like this, the

lives are combined and the dimensions of weight in the two assignments must be the same.

This leads to an extra dimensional constraint in addition to those listed in Section 2:

> *C8. If more than one assignment can start a life (for example if they are embedded in if statements) then their dimensions must be the same.*

i.e. $X = A; X = B \Rightarrow Units(A) = Units(B)$

In general:

- Two lives are considered to be equivalent if any code path exists where either could be used as an input to the same expression.
- The meaning and results of a program would not change if every separate life of each variable were declared independently and given a different name.

In the analysis of dimensions, the separate lives of the variables are identified and dimensions are associated with the lives and not with the variables.

We give two definitions:

usage - the employment of a symbol on the right hand side of an assignment or else as an input argument to a sub-program call or as part of a condition (e.g. in an **`if`** statement).

assignment - the employment of a symbol on the left hand side of an assignment or as an output argument to a sub-program call.

Each symbol is deemed to have one or more lives. A life is started when the symbol is defined or assigned a value. Subsequent stages of a life consist of uses made of the symbol. Each stage is called a **moment**. A life is terminated by the end of the program source code or by an assignment being made to the symbol. For example, in the following code fragments, the symbol X has three lives.

```
INTEGER*4  X        ┐  First life of X
                    │
X = 1               │  Second life of X
                    │
Z = 3*X             │
                    │
Y = 2+X             ┘
                    ┐
X = 8               │  Third life of X
                    │
P = X - 6           │
                    │
Q = 30 / X          ┘
```

The first life of **`X`** begins with the declaration, the second life begins with the assignment **`X = 1`** and the third life begins with the assignment **`X = 8`**. The first life has one moment, the second has three moments and so does the third.

During semantic analysis within WinFPT, lives are determined for each symbol in turn for each usage statement in turn by following this algorithm:

```
For each symbol in the symbol table
    For each usage statement using the symbol
        Trace back to all reachable last assignments
        of the symbol previous to the usage
```

For the example above, for the **`X`** usage statement **`Y = 2 + X`**, the last previous assignment is **`X = 1`**. Thus the life of **`X`** which includes **`Y = 2 + X`** consists of the three statements:

```
X = 1
Z = 3*X
Y = 2+X
```

5.1 Cases

We now describe some particular cases:

1. Branches: If the path of control determines that two or more assignment statements could have started a life that includes a given usage statement then those assignment statements must be considered to have started the same life.

2. Structured data types: Combinations of symbols such as A.X which identifies a field inside a structure or record are treated like symbols.

3. *Constants:* Real constants are handled in the same way as symbol lives and are also associated with dimensions.

4. Arrays: Arrays present two problems in the analysis of units and dimensions. Firstly, it is sometimes difficult to identify the separate lives of an array because some, but not all elements of the array may be assigned in a code fragment. Secondly, the units and dimensions of different array elements may be different. A single array could store, for example, the gender, age, weight and height of a set of medical patients.

For these reasons, the initial implementation of dimensional analysis has been restricted to scalar objects. Algorithms have been designed to handle arrays but were not implemented in the present work.

In the next section we examine the connection between a variable's dimensions and its lives.

6. Symbol lives and dimensions

During a life, the dimensions of a variable are fixed. This rule applies even when a re-assignment may happen during the course of the life dependent on a condition. e.g.

```
X = A
IF (P .LT. 5)
   X = B
ENDIF
PRINT *, X
```

This is a single life for **`X`** which starts with **`X = A`**. The re-assignment which takes place only if the condition is satisfied will be used in exactly the same way after the **`ENDIF`** and so we must presume that the original

assignment and the re-assignment share the same dimensions. Altering the second assignment of **X** as given below results in a potential inconsistency since the final line prints out either something with the dimension A or the dimension A*A.

```
X = A
IF (P .LT. 5)
   X = A*A
ENDIF
PRINT *, X
```

If a variable has a more than one life then it indicates that the programmer may be using the same variable for multiple reasons. We cannot assume that the programmer intends that in each life the variable has the same dimensions. Consequently, our dimension checking will only assume a variable has a particular dimension for the course of a life and that its other lives can have different dimensions. This has an important consequence for checking assignment statements. Since an assignment statement starts a new life for the variable on the left hand side, we cannot declare that if the dimensions of the left hand side variable left over from the previous life are different from those on the right hand side then we have an inconsistency. What we can in fact say is that the variable on the left hand side is beginning a new life here with dimensions given by those on the right hand side. For example:

```
X = A*A              ! Start of X life
B = 2*X
X = A*A*A            ! Start of next X life
C = 3*X
```

In this code, **X** has two lives. In the first life, it has dimensions given by A^2 while in the second life it has dimensions given by A^3. Because of our understanding of lives, we do not regard this situation as being inconsistent.

7. Dimension checking by inference in WinFPT

During static analysis, WinFPT constructs an internal representation of the user's source code. The representation includes sequences of statements and their component tokens such as symbol tokens and operator tokens. The symbol lives are identified by using the algorithm given in Section 5 to process the internal representation, resulting in symbol tokens being replaced by life tokens. In a later pass through the internal representation, units are recorded for each life in a data structure illustrated below in Figure 1.

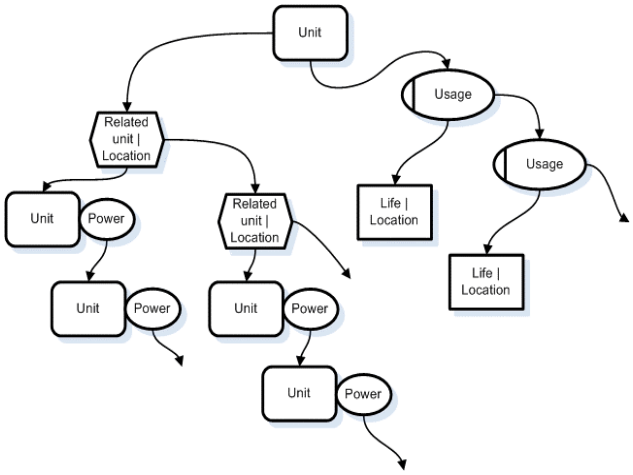

Figure 1: Units data structure in WinFPT

As stated earlier, the name "units" is used in preference to "dimension" to avoid confusion with the properties of array variables. Figure 1 shows that, for each unit found, WinFPT maintains:

- a list of usage records that identify every location in the program where a symbol life has this unit
- a list of related unit records that identify every location in the program where the unit has been found to be identical to a combination of one or more powers of other units through the application of the constraints given in Section 2.

Thus we are able to record every kind of unit used, where it was used and all the ways that it relates to other units and the places in the user's code where this was discovered. WinFPT implements the first six dimensional constraints through a modification of its expression analysis routines. It implements the seventh by carrying out the units pass through the internal representation as a bottom up walk through the user's program's call structure. WinFPT knows at every call site the relationships between the arguments and the return value because it has already analysed the lower levels.

Dimensional checking consists of confirming the consistency between the sets of related units and reporting where inconsistency is found. We illustrate the process with an example but first we introduce a notation to describe the WinFPT units data structure.

We define

Units(X, n)^m

as units of life n of symbol X raised to the power m. In the case of a symbol with only one life, this reduces to

Units(X)^m

The usage records in the data structure enable us to find the locations in the user's code related to a particular unit. We represent this as:

Units(X, n)^m->usages->Life nn of Y, location in code

We extend our notation to describe the related units of a unit as follows. Suppose we have found that units of X are the same as units of $Y^a.Z^b$ and also P^c .

This will be expressed as:

Units(X) ->related-units->Units(Y)^a . Units(Z)^b
->related-units->Units(P)^c

We shall now describe the dimensional checking of the following Fortran code fragment:

```
READ*, L, W, H     ! Length, Width,
                   ! Height of box
A = 2*(L*W + L*H + W*H) ! Surface area
V = L*W*H          ! Volume
D = SQRT(L^2 + W^2 + H^2)! Diagonal
E = L + V          ! An error situation
```

READ*, L, W, H

This statement starts the lives of L, W and H. Unit records are created for them which we represent as

Units(L)
Units(W)
Units(H)

assuming, for the sake of simplicity of the notation, that the symbols in the fragment only have one life.

A = 2*(L*W + L*H + W*H)

Processing the right hand side of this statement results in:

Units(I1)-> related-units-> Units(L)^2
Units(I2)-> equivalent to-> Units(I1)
Units(I3)-> equivalent to-> Units(I1)
Units(L) -> related units -> Units(I1)^(1/2)
Units(W)-> equivalent to -> Units(H)
Units(H)-> equivalent to ->Units(L)

where **I1**, **I2** and **I3** are intermediate symbols representing the quantities **L*W**, **L*H**, **W*H** respectively. The data structures show the results of WinFPT's analysis which has determined that **I1** has the units of $\mathbf{L^2}$, that **I2** and **I3** have the same units as **I1**, that **L** has units which are the square root of those of **I1** and that the units of **W** are the same as those of **H** and that the units of **H** are the same as those of **L**.

The left hand side is **A** so we know that the units of this life of **A** are those of the right hand side. WinFPT records this life of **A** as being a usage of Units of **I1**. This results in:

Units(I1)-> related-units -> Units(L)^2
->usage -> Life 1 of A, location in code
Units(I2)-> equivalent to -> Units(I1)
Units(I3)-> equivalent to -> Units(I1)
Units(L)-> related units -> Units(I1(1/2)
Units(W)-> equivalent to -> Units(H)
Units(H)-> equivalent to -> Units(L)

where changes from the previous data structure state are shown in bold.

WinFPT has now identified a real symbol for **I1** so we replace **I1** with that symbol:

Units(A)*-> related-units -> Units(L)^2*
->usage -> Life 1 of A, location in code
*Units(I2)-> equivalent to -> **Units(A)***
*Units(I3)-> equivalent to -> **Units(A)***
*Units(L)-> related units -> **Units(A)^(1/2)***
Units(W)-> equivalent to -> Units(H)
Units(H)-> equivalent to -> Units(L)

WinFPT has completed the processing of the statement. It has recorded that **A** has the units of the square of **L**, and that **L**, **W**, **H** are equivalent to each other.

Comparing sets of units for equivalence is at the heart of the process. WinFPT compares two sets of units denoted *f(Units)* and *g(Units)* by solving the equation:

$$f(Units).g(Units)^{(-1)} = 1 \qquad \text{Equation 1}$$

Three outcomes are possible:

case 1) all factors cancel so we are left with *1 = 1*. This tells us that everything is consistent about the two sets of units.

case 2) one factor remains on the left hand side. This tells us that either the factor is dimensionless or we have an error. If we have presumed that all real variables have dimensions then we have found an error and so we report the error and how it was caused.

case 3) two or more factors remain on the left hand side. This gives us combinations of relationships that must hold between the units for consistency.

For example, during the analysis of the statement that assigns a value to A, WinFPT checks *Units(L)^1.Units(W)^1* against *Units(L)^1.Units(H)^1*.

Applying equation 1 we get:

Units(L)^1.Units(W)^1.(Units(L)^1.Units(H)^1)^(-1) = 1

This gives:

Units(L)^1.Units(W)^1.Units(L)^(-1).Units(H)^(-1) = 1

Cancelling factors, this reduces to:

Units(W)^1.Units(H)^(-1) = 1

This is an example of case 3. We can deduce that

Units(W) = Units(H)

V = L*W*H.

After processing this statement, the data structure becomes:

Units(A) -> related-units-> Units(L)^2
-> related-units-> Units(I4)^1 . Units(L)^(-1)
->usage -> Life 1 of A, location in code
Units(I2)-> equivalent to -> Units(A)
Units(I3)-> equivalent to -> Units(A)
Units(L) -> related units -> Units(A)^(1/2)
-> related units-> Units(I4)^1 . Units(A)^(-1)
Units(W)-> equivalent to -> Units(H)
Units(H)-> equivalent to -> Units(L)
Units(I4)-> related units -> Units(A)^1 . Units(L)^1

where **I4** is the intermediate symbol for **L*W*H**.

The assignment of **V** identifies the units of **V** to be the same as the intermediate **I4**, and **I4** is therefore renamed in the table to show the association with a variable life:

Units(A)*-> related-units-> Units(L)^2*
*-> related-units-> **Units(V)^1** . Units(L)^(-1)*
->usage -> Life 1 of A, location in code
Units(I2)-> equivalent to -> Units(A)
Units(I3)-> equivalent to -> Units(A)
Units(L) -> related units -> Units(A)^(1/2)

*-> related units-> **Units(V)^1** . Units(A)^(-1)*
Units(W)-> equivalent to -> Units(H)
Units(H)-> equivalent to -> Units(L)
Units(V) -> related units -> Units(A)^1 . Units(L)^1
->usage -> Life 1 of V, location in code

D = SQRT(L^2 + W^2 + H^2)

After processing this statement, the data structure becomes:

Units(A) -> related-units -> Units(L)^2
-> related-units -> Units(V)^1 . Units(L)^(-1)
->usage -> Life 1 of A, location in code
Units(I2)-> equivalent to -> Units(A)
Units(I3)-> equivalent to -> Units(A)
Units(L) -> related units -> Units(A)^(1/2)
-> related units -> Units(V)^1 . Units(A)^(-1)
->usage -> Life 1 of D, location in code
Units(W)-> equivalent to -> Units(H)
Units(H)-> equivalent to -> Units(L)
Units(V) -> related units -> Units(A)^1 . Units(L)^1
->usage -> Life 1 of V, location in code

E = L + V

The units of the right-hand-side are evaluated first. The + operator implies that the units of **L** must be the same as the units of **V**. Applying equation 1 to Units(L) and Units(V) we get:

Units(L).(Units(V))^(-1) = 1

WinFPT checks through all the sets of units related to *Units(V)* for consistency. Thus *Units(V)* is replaced by the related units *Units(A)^1 . Units(L)^1*. The equation becomes:

Units(L).(Units(A)^1 . Units(L)^1)^(-1) = 1

This reduces to:

Units(A)^ (-1) = 1

This is an example of case 2. Hence, either **A** is dimensionless or else an error has been detected. This statement is reported as an error.

8. WinFPT Dimensional analysis output

We give below the output produced by WinFPT when carrying out dimensional analysis on the code fragment. (Note that the alternative Fortran notation of ****** for raised to the power is employed by WinFPT.)

```
-----------------------------------
Line    28
File: e:\projects\fpt\fpttest\box.f90
      E=L+V
         ^
     Inconsistency detected in units
     and dimensions
-----------------------------------

***********************************

Units Identified
================
Units(L)
   ==    Units(V)    ** 1
      *  Units(A)    ** -1

   ==    Units(A)    ** (1/2)
   Variables or variable lives
    D    H    L    W

Units(A)
   ==    Units(V)    ** 1
      *  Units(L)    ** -1

   ==    Units(L)    ** 2
   Variables or variable lives
    A

Units(V)
   ==    Units(A)    ** 1
      *  Units(L)    ** 1
   Variables or variable lives
    V

Units(E)
   Variables or variable lives
    E

*********************************

Units of Symbols
================
A   Units(A)
D   Units(L)
E   Units(E)
H   Units(L)
L   Units(L)
V   Units(V)
W   Units(L)
*********************************
```

WinFPT has identified the error in the assignment **E = L + V**. Through the system of recording unit usage locations, WinFPT is able to report the source code line and the operator within the line that gives rise to the error.

In addition to reporting errors, WinFPT lists the non-equivalent units that appear in the program in the final lists of related units. These are the de facto fundamental units used by the programmer. Their relationships with each other are listed systematically. The user is able to scan these lists to confirm that WinFPT is telling the user that the units are related in the expected way. Meaningful symbol names make a considerable difference to the ease with which this can be carried out!

WinFPT then lists the symbols giving the units of each in terms of the de facto fundamental units.

As mentioned earlier, there are situations in which the method can detect erroneous use of units in the sense of trying to mix feet and metres. Consider the following code fragment:

```
FTPM = 3.2808     ! feet per metre
:
```

```
R_OUTER = 4.1      ! feet
R_INNER = 1.2      ! metres
L = R_OUTER - R_INNER*FTPM
A = PI*(R_OUTER**2 - R_INNER**2)
```

The programmer has created a conversion factor variable but he has forgotten to apply it in the formula calculating **A**. WinFPT would report an error since the assignment to **L** would result in
Units(R_OUTER)->equivalent to->
*Units(R_INNER)*Units(FTPM)*
whereas the assignment to **A** would result in
Units(R_OUTER)->equivalent to->Units(R_INNER)
and WinFPT would see this.

9. Remarks

Previous methods aimed at checking dimensional consistency of computer programs have been described and issues arising from them have been discussed. A novel approach based on the concept of symbol lives has been introduced. This approach, allied with the internal representation of a user's Fortran program, enables WinFPT to carry out fast and systematic checking without the user having to modify source code.

It will be noted that WinFPT carries out a great deal of checking of the unit records as it works its way through the user's code. This is achieved very efficiently and comprehensively by maintaining a set of cross reference tables that enable unit usages etc to be located rapidly within the user's source code. Timing experiments on large bodies of source code indicate that WinFPT can analyse many thousands of lines of code per second.

WinFPT has been used to analyse a number of large Fortran programs. In one large helicopter simulation it reported 90 dimensional errors.

Work is in progress of extending the facility to arrays and sub-programs.

10. References

[1] R.F. Cmelik, N.H. Gehani, "Dimensional Analysis with C++". *IEEE Software*, May 1988, pp 21-27.

[2] A. Dreiheller, M. Moerschbacher, and B. Mohr, "Physcal - programming Pascal with physical units." *Sigplan Notices 21, 12* (December 1986), pp 114-123.

[3] P. Guo, and S. McCamant, "Annotation-less Unit Type Inference for C" *MIT 6.883 - Program Analysis*, Fall 2005.

[4] P.N. Hilfinger "An Ada Package for Dimensional Analysis", *ACM Transactions on Programming Languages and Systems Vol 10, No. 2* April 1988.pp 189-203.

[5] R.T. House, "A proposal for an extended form of type checking of expressions." *Computer Journal 26, 4* (Nov 1983), pp 366-374.

[6] L. Jiang, Z. Su, "Osprey:a practical type system for validating dimensional unit correctness of C programs." *Proceedings of the 28th International Conference on Software Engineering* Shanghai, China, 2006.

[7] A. Kennedy, "Dimension Types", *ESOP*, 1994, pp 348-362

[8] National Institute of Standards and Technology web page: http://physics.nist.gov/cuu/Units/ February 2007

[9] G. W. Petty, "Automated computation and consistency checking of physical dimensions and units in scientific programs." *Software Practice & Experience* 2000; 00:1-7.

[10] M. Sandberg, D. Persson, B. Lisper, "Automatic Dimensional Consistency Checking for Simulation Specifications", *SIMS 2003*, Västerås, Editor(s):Erik Dahlqvist, September, 2003, p 6,

[11] Z. D Umrigar. "Fully static dimensional analysis with C++". *Sigplan Notices 29, 9* (September 1994), pp 135-139.

[12] A. Van Delft, "A Java extension with support for dimensions", *Software Practice & Experience 29, 7* (June 1999), pp 605-616.

[13] M. Wand, P. O'Keefe, "Automatic dimensional inference" *Computational Logic - Essays in Honor of Alan Robinson*, 1991 pp 479-483

[14] WinFPT home web page 2007: http://www.simcon.uk.com