Simplification of Coding of NRU Loop Experiment Software with Dimensional Generator

R.S. Davis
Chalk River Laboratories
Atomic Energy of Canada Limited

1. INTRODUCTION

1.1 The Task

BURFEL−PC is a program to calculate in detail what happens to various components of CANDU® fuel bundles when they are irradiated in an experimental loop in the NRU Reactor at Chalk River. BURFEL−PC calculates fuel pins' linear powers, fuel pins' burnups, and fast-flux fluences of material samples, all as functions of time and axial position. An auxiliary program, WIMSBURF, accepts information from the lattice-cell program WIMS−AECL, and prepares some of the information BURFEL−PC needs.

Early in the coding of these programs, the author noted that dimensional analysis of the source code could be automated, to prevent a large class of errors in the coding of physical expressions and to somewhat reduce human effort, in the writing of these and subsequent scientific/ engineering programs. The program Dimensional Generator [1, 2] enables such automation by the principles of object-oriented programming, in a manner that is useful even to a programmer with no knowledge of object-orientation and only common knowledge of dimensional analysis.

Although existing code could be retrofitted with the provisions of Dimensional Generator with reasonable effort, Dimensional Generator is primarily an aid in writing new code, or a modular new component of existing code. This paper's topic is the actual experience gained in this, the first use of Dimensional Generator in writing new reactor-physics programs. Dimensional Generator confirmed its theoretically predicted abilities [2]. Indeed, although the programmer made several mistakes in the coding of physical expressions, not one of those mistakes made it into executable code. Every such mistake was detected as a fatal error by the compiler, even though the equivalent statements without the use of Dimensional Generator would have been standard-conforming code. More subjectively, the programmer found that coding was more pleasant, because some menial work was automated, and that productivity (lines of debugged code per day) was higher than expected, partly because of the use of Dimensional Generator.

1.2 Topics of This Paper

The following are specific topics of this paper:

1. There is much creativity in the manner in which Dimensional Generator can be applied to a specific programming task [2]. This paper tells how Dimensional Generator was applied to a reactor-physics task.

® CANDU® is a trademark of Atomic Energy of Canada Limited.

2. In this first practical use, Dimensional Generator itself proved not to need change, but a better user interface was found necessary, essentially because the relevance of Dimensional Generator to reactor physics was initially underestimated. It is briefly described.
3. The use of Dimensional Generator helps make reactor-physics source code somewhat simpler. That is explained here with brief examples from BURFEL−PC and WIMSBURF.
4. Most importantly, with the help of Dimensional Generator, all erroneous physical expressions were automatically detected. The errors are detailed here (in spite of the author's embarrassment) because they show clearly, both in theory and in practice, how Dimensional Generator offers quality enhancement of reactor-physics programming.

2. SUMMARY OF THE FUNCTION OF DIMENSIONAL GENERATOR

2.1 Mode of Operation

Dimensional Generator accepts information characterizing a specific task (as distinct from characterizing a specific compiler, computer, or operating system) through a user-friendly interface. It produces a file of Fortran source code, which the programmer inputs to the compiler among the programmer's own source-code files. The programmer's code may be written in the programmer's choice of style except that, in place of the `Integer`, `Real`, or `Complex` declaration for each physical quantity, the programmer declares the variables to be of types that Dimensional Generator defines according to the programmer's input to Dimensional Generator. A Fortran 95 `Use` directive in each program unit imports those type definitions. (Although a program written with the help of Dimensional Generator does not need to be written in Fortran 95 style, the compiler must have Fortran 95 capability.) These provisions effectively convey dimensional-analysis information to the compiler.

2.2 Definition of Terms of Dimensional Analysis

Since most physicists are familiar with basic dimensional analysis, an equation is necessary only to resolve ambiguities in conventional terminology. If an analyst composes a list of physical units for a particular program, the definition of the i^{th} unit in the list may be written

$$U_i = A_i \prod_k \left(D_k^{p_{i,k}} \right). \tag{1}$$

The quantities D_k constitute a subset of the relevant units, chosen by the analyst to form a "basis". The significance of a basis is that

- none of the quantities D_k can be expressed in this way in terms of the others, but
- all relevant units can be expressed in this way using the chosen basis.

This use of the word "basis" can be seen to be analogous to its conventional usage in physics, by considering the logarithm of equation (1).

The term "dimension" has several common meanings. In this paper, a dimension is a physical quantity D_k, and each dimensionless quantity $p_{i,\,k}$ is called a "dimension-power". Once a basis has been chosen, each unit is characterized by a unique set of dimension-powers and by its

"amount", A_i, a dimensionless quantity that gives the unit's size. In particular, a dimension-power of zero, corresponding to multiplication by 1, means that the corresponding dimension is not part of the unit.

A conventional basis includes units of general physical phenomena, such as length, mass, time, electric charge, temperature, and the like. However, Dimensional Generator's usefulness is enhanced if the basis also includes characteristic features of the system that is to be analyzed, as is exemplified in section 4 below.

2.3 Output from Dimensional Generator

The source code that Dimensional Generator generates defines "derived data types" to represent relevant types of physical quantities. It contains type-definition statements like the following example from WIMSBURF:

```
! Watt_per_Gram = 1000 Metre ** 2 * Second ** -3
Type:: Watt_per_Gram
    Real (Kind = Selected_Real_Kind (6)):: Number
End type Watt_per_Gram
```

(This and other code snippets below are slightly altered from actual code, to focus on essential features.) This code is based on the programmer's input to Dimensional Generator. The derived type's name, `Watt_per_Gram`, is the programmer's choice for this data type. `Metre` and `Second` are among the names of dimensions listed in the input. The amount, `1000`, and the dimension-powers, `2` and `-3`, were calculated by the analyst to represent that unit, per equation (1), and were among the input to Dimensional Generator. The intrinsic data type and kind, `Real` with at least `6` significant decimal digits, are also analyst's decisions, expressed by input to Dimensional Generator. ("Intrinsic" means "defined as part of the language", as distinct from being defined by source code input to a compiler.) Dimensional Generator generated the above code, and a large amount of associated code discussed below, on the basis of that input.

It can be useful to define more than one data type with the same units. For example, to calculate calendar times, BURFEL−PC uses both 6-significant-figure and 15-significant-figure `Real` times in seconds, and uses both `Integer` and `Real` numbers of years.

Quality-assurance of the large amount of source code that Dimensional Generator generates requires only verification of the generated type statements and comments exemplified above. If they are correct, then the quality of the rest of the generated module is assured by the quality assurance of Dimensional Generator itself. Dimensional Generator conforms to, and has been approved in accordance with, AECL's standards for quality assurance of safety-related computer code, which is based on the software-quality-assurance standard CSA N286.7-99.

2.4 Usage of Output from Dimensional Generator

The main difference that Dimensional Generator makes in coding is that the programmer defines variables with the data types generated as above, instead of with intrinsic data types. The programmer uses these variables in executable statements, in array declarations, and in further derived-type declarations, just as if they were intrinsic-type variables (`Real` etc.).

The main difference that Dimensional Generator makes in compilation is with executable statements. A Fortran 95 compiler does not contain any intrinsic information on how to apply intrinsic arithmetic operators (e.g. `+`, `/`, `**`) or intrinsic functions (e.g. `Sqrt`, `Huge`, `Atan2`) to derived types, such as those that Dimensional Generator defines. However, a Fortran 95 compiler can accept source code that contains additional definitions of intrinsic operators and functions, which tell the compiler how to apply those operators and functions to derived types. In object-oriented terminology, that source code "overloads" the intrinsic operators and functions.

Dimensional Generator thus overloads intrinsic operations on the derived types it defines, but only in ways that make sense according to the physical meanings of the operands. For example, pairs of variables declared to represent the same physical phenomenon, e.g. two energies or two fluxes, may be added with a `+` sign, just as if they were intrinsic variables. The result will be correct even if they are in different units. However, for variables declared to represent different phenomena, such as an energy and a flux, a `+` sign between them is undefined. Similarly, a microscopic cross section can be multiplied by a number density with the `*` sign, and the result can be used only as a macroscopic cross section, with automatic units conversion as appropriate. Similarly, the `=` sign is overloaded to include a units conversion when appropriate, and is undefined if, for example, source code tries to store a flux as an energy.

If source code specifies a physically meaningless operation, the compiler finds that one or more operators and/or functions are undefined, it declares a fatal error, and the bug is caught before it enters an executable file. Thus, the programmer is relieved of low-value work on units conversion and, more importantly, a large class of errors becomes impossible.

Dimensional Generator does not define operations between intrinsic types and derived types. For example, if π is used, the programmer must declare it to be unitless.

2.5 The Utility of Dimensional Analysis

Buckingham's Π Theorem [3] shows that each dimension introduces one degree of redundancy into mathematical expressions of physical reality. Buckingham's purpose, in 1914, was to minimize redundancy by the use of dimensionless quantities. However, in the 21st century, better practice is that the compiler works harder to save human effort, and especially to improve the reliability of software. With Dimensional Generator, increasing the number of dimensions increases redundancy, costs extra work for Dimensional Generator and for the compiler, and reduces human effort and error. (The effect of Dimensional Generator on a program's run time is neutral with normal optimization.) This paper explains how a set of dimensions was extended beyond conventional engineering/scientific dimensional analysis, using specifics of particular reactor-physics tasks, and how the added redundancy indeed caught errors.

The use of Dimensional Generator is analogous to the use of automatic spelling and grammar checkers when composing English. They will catch many errors, but note errors that replace a word with another valid word (such as "note" in place of "not" in this sentence). They are also plagued with false positives. The experience described here indicates that Dimensional Generator and a compiler police physical expressions more effectively than automatic checkers police English, because dimensional analysis is simpler and more rigid than English grammar. However, use of Dimensional Generator cannot cause the compiler to catch such errors as

incorrect multiplication by a dimensionless quantity, such as π or a Reynolds number, or any use of a wrong variable with the right dimension-powers.

3. IMPROVEMENT OF THE USER INTERFACE

Dimensional Generator works from a text input file. That file was initially expected to be easy to prepare, according to an estimate that a typical program would use around 12 physical units, but experience has shown that the input file is typically much bigger. BURFEL−PC uses 21 data types, and WIMSBURF uses 60 data types, which made editing of its input file unreasonably laborious. The large number of data types resulted from considerations which are typical of reactor-physics programming:

1. Several non-physical dimensions were introduced as recommended in section 2.5 above and detailed in section 4 below.
2. The main input to WIMSBURF is from a file, produced by the cell-lattice code WIMS−AECL and known as "Tape16", which uses a profusion of units because of legacy issues. For example, it uses at least four units of energy. With Dimensional Generator, every input datum should be read into a variable with the datum's unit.
3. The users require further diversity of units. For example, although Tape16 gives burnups in Megawatt-Days per Tonne, the users require Megawatt-Hours per Kilogram. Some neutron fluxes are required in Neutrons per (Centimetre-Squared)-Second, and others in Neutrons per (Metre-Squared)-Second. Output variables' units too should be represented by their data types.
4. Many, diverse physical phenomena are involved.
5. More than one data type was defined for some units, per section 2.3 above

The new user interface is written as a Microsoft Excel spreadsheet. The input parameters, described in section 2.3 above, go into a table. For the dimension-powers, column headings are also filled-in by the user. When the spreadsheet is filled-in, the programmer clicks a button labeled "Make Fortran", whereupon a macro writes the corresponding Dimensional Generator text-input file, and opens that file with Dimensional Generator, thus producing the corresponding source code.

4. USE OF SPECIALIZED DIMENSIONS

4.1 Dimensions for Relative Fluxes

The file Tape16 includes a quantity called `P`. This quantity gives relative values of fluxes in different regions of the cell. That means that the fluxes are only meaningful in relation to each other. Therefore, a dimension is used in WIMSBURF called "Units of P", and the quantities `P` are in a unit that consists of that dimension. Because no output quantity has a unit with Units of P among its dimensions, Dimensional Generator provides an iron-clad guarantee that the units of `P` correctly cancel out.

4.2 Other Fluxes with Different Normalizations

Other fluxes in Tape16 also call for distinctive dimensions, because they are normalized in different ways. Most of the fluxes in Tape16 are normalized to one neutron absorption per second. However, for burnup calculations, fluxes are normalized to specified power levels. The ratio of the two fluxes, i.e. the conversion factor between their units, is different in each time step. Consequently, two additional dimensions are called Reaction-Neutron and Power-Neutron. The corresponding units of flux are the Reaction-Neutron per (Centimetre-Squared)-Second, and the Power-Neutron per (Centimetre-Squared)-Second. (The above-mentioned fluxes in Power-Neutrons per (Metre-Squared)-Second involve additional units, but not additional dimensions.)

4.3 Fission Cross-Sections

To represent fission cross sections correctly, dimensions also used are Power-Fission, i.e. fission caused by a power-neutron, and Reaction-Fission. The derivation of a power therefore involves a quantity with units of energy per Power-Fission. The absence of a quantity with units of energy per Reaction-Fission ensures that inappropriate fluxes can never be used to calculate power.

4.4 Fuel Pins

Experience shows that it is easy to confuse properties of a whole fuel bundle with properties of a fuel pin. Therefore, a dimension of Pin is used to define units of per-Pin quantities.

5. CONSEQUENCES OF USING DIMENSIONAL GENERATOR

5.1 Automation and Clarity

Variables of the types defined by Dimensional Generator fulfill the fundamental goal of object-oriented programming; they act just like the physical quantities they represent, rather than as numbers. This is illustrated by the following statement from BURFEL−PC:

```
SegmentTimeStep % Burnup = SegmentTimeStep % Previous % Burnup + (  &
    SegmentTimeStep % LinearPower (FromFission) *                   &
    SegmentTimeStep % TimeStep % BurnTime /                         &
    IHEMass (SegmentTimeStep)                                       &
)                                                                 ! &
```

Readers unfamiliar with pointers may understand this statement by interpreting the `%` sign as a possessive sign. Thus, this statement may be literally translated into near-English as the sentence "(The segment's burnup) equals (the segment's previous burnup) plus ((the segment's linear power from fission) times (the segment's time step's burn time) divided by (the initial heavy element mass of the segment))." This method of naming variables is a separate matter from the use of Dimensional Generator, which is equally applicable to variables with short, single names and with variable subscripts.

When input-data sources and users' requirements involve non-standard units, one good practice is to convert input quantities to standardized units immediately after input, and to convert output quantities to users' desired units just before output. However, this example illustrates another

good alternative. Object-orientation implies that internal representations of data objects, and procedures of operation on them, are "hidden" from code that uses the objects (although not hidden from the programmer, who can read their definitions). Variables defined with the help of Dimensional Generator are physical quantities, not just numbers. That means their units are properties of their internal representations, and therefore are properly hidden . Thus, it makes no difference if the program uses the input and output data's units. This practice simplifies input and output, and may also simplify inspection of intermediate results for debugging and/or verification. In the present example, to conform with input data and users' wishes,

- the burnup is of type `MegawattHour_per_Kilogram`,
- the linear power is of type `Kilowatt_per_Metre`,
- the burn time is of type `Day`, and
- the function `IHEMass` (for "initial heavy element mass") returns values of type `Gram_per_Centimetre` when its argument is a fuel-pin segment, in accordance with a convention that characteristics of a segment are per unit length when applicable.

A unit of burnup is a squared speed. Dimensional Generator stipulates the data types of the results of the `*` and `/` operators with the types of operands here, through properly hidden provisions, so that they ultimately yield a squared speed, whose units are properly hidden. Dimensional Generator defines the `+` sign in this statement, because the units on both sides of it are squared speeds. Since these squared speeds may be in different units, the `+` sign may include a hidden units conversion. The `=` sign also may include a hidden units conversion, depending on the units of the result of the `+` operation, which are properly hidden.

The practical consequence is that Dimensional Generator makes the statement clearer, because a visible units-conversion factor is unnecessary and correct units conversion is assured, even though the units of the input and output data are used. Many opportunities for the programmer to err are thus eliminated. For example, if the argument of `IHEMass` were a discrete object, such as a fuel bundle, then `IHEMass` would return a result of type `Kilogram`. Then the `*` and `/` operators would not yield a unit of speed squared, none of the `+` signs defined by Dimensional Generator would suit the combination of types of operands, the compiler would declare a fatal error, and the error would never make it into executable code.

5.2 What Dimensional Generator (Most Importantly) Did Not Do

More important than elegance is that Dimensional Generator makes a large class of errors impossible. Fundamentally, this is because Dimensional Generator does not define operators that do not have physical meaning. As a result, throughout the entire development process, neither BURFEL−PC nor WIMSBURF ever put out an incorrect result. During early coding, some outputs were not what users wanted, because of misunderstood requirements specifications, but all outputs were what they claimed to be. To explain this achievement, all the physics-related coding mistakes that were made are described here.

1. In the following statement in WIMSBURF, `CouponTimeStep` is a time step in the irradiation of a material-test sample. `CouponTimeStep % FastFluxPerLinearPower` is the ratio of its fast flux to its bundle's linear power, to enable subsequent calculation of its fast fluence in an actual experiment. To calculate this quantity, the following statement was written:

```
CouponTimeStep % FastFluxPerLinearPower = (                          &
    CouponTimeStep % MaterialTimeStep % FluxCondensation (Fast) /    &
    CouponTimeStep % MaterialTimeStep % TimeStep % LinearPower       &
)                                                                  ! &
```

Although the expression is indeed the ratio of a fast flux to a linear power, as the variable names indicate, the compiler declared a fatal error for this statement. The reason is that the flux in this statement is normalized to absorption rate, not to power level, per section 4.2 above. Consequently, the expressions on the two sides of the = sign were in units with different dimension-powers, Dimensional Generator considered their units to pertain to different physical phenomena, and it did not overload the = sign for that combination of operands. Then the compiler found the = sign to be undefined, and the statement never made it into executable code.

The compiler accepted the statement, and the subsequent quality-assurance verification proved it to be correct, when the programmer introduced the following factor to the numerator:

```
CouponTimeStep % MaterialTimeStep % TimeStep %
PowerNeutron_ReactionNeutron
```

2. In the following statement in BURFEL−PC, `RegionTimeStep` is a time step in the irradiation of a region within a fuel bundle. The statement was written to calculate the lifetime cumulative linear energy from a region within a fuel bundle at the start of the time step:

```
RegionTimeStep % LinearEnergy = (                              &
    RegionTimeStep % Burnup /                                  &
    RegionTimeStep % FirstStep % LinearMass (HeavyElement)     &
)                                                            ! &
```

The reader would probably never make such an obvious mistake as this use of / when * would be appropriate, but it happened. Burnup divided by linear mass yields a unit that has no potential use in this task. Consequently, the input to Dimensional Generator did not call for such a unit, Dimensional Generator did not overload the / for that combination of operands, the compiler found that the / was undefined, it declared a fatal error, and the above code never entered any executable file. Even if the / had been defined, Dimensional Generator would not have defined the = sign, because the dimensional analysis would have shown that the units on the two sides were for different physical phenomena.

3. In the following statement in WIMSBURF, `SegmentTimeStep` is a time step in the irradiation of a fuel-pin segment. The statement was written to calculate `SegmentTimeStep % PowerGenerationFactor`, the ratio of linear power in a fuel pin to thermal flux at the cell boundary:

```
SegmentTimeStep % PowerGenerationFactor = (                            &
    Sum (SegmentTimeStep % MaterialTimeStep % LinearPower) / (         &
        SegmentTimeStep % TimeStep % EdgeFluxCondensation (Thermal) * &
        SegmentTimeStep % TimeStep % PowerNeutron_ReactionNeutron      &
    )                                                                  &
)                                                                  ! &
```

This time, the programmer remembered to include the conversion factor between fluxes normalized to reaction rate and fluxes normalized to power. However, the left-hand side represents linear power per fuel pin, but the `Sum` over linear powers in this expression is the total for all the materials in that pin type; and therefore is the total for all fuel pins of that pin type. The expression had to be divided by the number of fuel pins of that type, a quantity of type `PinInteger`, i.e. an integer with the unit 1 Pin. Because of the use of the dimension Pin, the failure to divide by a number with those units yielded an expression with a different set of dimension-powers than the left-hand side's. The `=` sign was consequently undefined, the compiler declared a fatal error, and the above error never made it into executable code.

5.3 Reservations about Dimensional Generator

5.3.1 Situations in which Dimensional Generator's Provisions Must be Bypassed

WIMSBURF creates, and BURFEL−PC creates and uses, polynomials to represent properties as functions of burnup, and as functions of axial position. The coefficients of a polynomial belong in an array. However, unless the independent variable is unitless (as it is not in the present instances), the coefficients of a polynomial have different units. All the elements of a Fortran 95 array must be of the same data type. Consequently, the data type cannot characterize the units in an array of polynomial coefficients.

Polynomial fits are performed by an IMSL Library routine, `RCurv`. A related problem is that this library routine was not written for arguments with types defined by Dimensional Generator.

For these two reasons, the routines for fitting and evaluating polynomials operate on variables of the form `X % Number`, i.e. on the contained, intrinsic variables. Simple interface routines to `RCurv` are used for each combination of units, and provide some of the error-prevention that Dimensional Generator is intended to offer.

There are other situations in which the provisions of Dimensional Generator must be bypassed to conform to the limitations of Fortran 95, such as the use of integers with units as subscripts and as `Do`-loop indices. There too, direct access to the contained, intrinsic number solves the problem, at the expense of the simplicity and reliability of object-orientation. This is the cost of violating the "hiding" principle of object orientation, described in section 5.1 above.

5.3.2 Compilation Time

Because WIMSBURF uses 60 units, and there is a combinatorial explosion of meaningful operations among them, the source code for the module generated by Dimensional Generator amounts to 2.4 megabytes, and the resulting `.mod` file to almost that size. The compiler takes several seconds to input that `.mod` file in preparation for the compilation of each module, thus increasing total compilation time for a whole program from seconds to minutes. (This problem does not affect run time.) This problem is particularly noticeable, because the use of Dimensional Generator causes more errors to be detected during compilation (with corresponding reduction of run-time debugging), and thus necessitates more compilations. It reflects the principle in section 2.5 above, that the compiler works harder to reduce human effort and error.

5.3.3 The Role of Relational Data Structures

Not all the good experience writing BURFEL−PC and WIMSBURF is ascribable to the use of Dimensional Generator. Relational databases were adopted for better user-interfacing, and this decision resulted in additional, unexpected simplifications. These must be the subject of a future paper, but one such advantage is already apparent in the snippets of source code above. Work with databases inspired the use of relational data structures within the programs, which led to the use of the concatenated pointer references in the expressions. (Database users will perceive their similarity to database queries.) They give on-the-spot, detailed descriptions of operands, as is illustrated by the "near-English" translation in section 5.1 above. The conventional alternative, multiple, variable subscripts, shows less directly what each variable is and why it has been used.

There is an analogy with `GoTo` statements. They require intense scrutiny to reveal the flow of a program. When structured programming was developed, `GoTo` statements became widely deprecated. Similarly, relational data structures offer the option of greatly simplifying (but not entirely eliminating) the ways in which variable subscripts are used.

6. CONCLUSION

The ultimate results of the use of Dimensional Generator are that

1. Even though the programmer introduced human error, at no time during the development of these reactor-physics programs did any erroneous physical expressions, other than misunderstandings of the requirements specifications, get into executable code.
2. When changes were necessary, they were made quickly and reliably. In many cases, a change started simply with a change in the units of a quantity, and the compiler then pointed out all the points in source code where consequent changes were necessary.
3. The programmer's productivity, measured in lines of fully debugged code per day, was considerably improved over the author's previous experience.

These benefits are not entirely creditable to Dimensional Generator, because many resulted also from the use of relational data structures. Also, the provisions enabled by Dimensional Generator sometimes must be bypassed. However, in balance, its use in the reactor-physics software discussed here resulted in higher-quality code at lower cost.

7. REFERENCES

1. R.S. Davis, *Dimensional Generator*, http://www.magma.ca/~davises/Software/Dimensional_Generator/index.htm .

2. R.S. Davis, *Automatic Assistance in Source Code Preparation and Quality Assurance by Dimensional Analysis*, 25th Annual CNS Conference, 2004.

3. E. Buckingham, *On Physically Similar Systems; Illustrations of the Use of Dimensional Equations*, Phys. Rev. **4**, 345-376 (1914).