Design of an Interactive Financial Planning Package

W. H. Wilson, C. F. Burrows and M. M. Sidhom*

This paper describes features of the design of an interactive financial planning system for financial planning professionals. The system consists of a modelling language, with translator and code interpreter, facilities for generating reports, for creating and editing financial models, and file-handling functions.

As the modelling language is designed to enable financial planners to describe models in familiar terms, it is non-procedural, and supports financially oriented built-in functions. File-handling functions allow users to avoid complicated interactions with an operating system. Users can investigate the effects of "what-if"-type changes to a model. Informal descriptions of the syntax adopted for the MODELER package, (developed at the University of Sydney Computing Centre), are included.

Keywords and phrases: financial planning, non-procedural languages

CR categories: 3.52, 4.22

1. LANGUAGE FEATURES NEEDED FOR FINANCIAL PLANNING

We shall first describe our assumptions about the underlying structure of the data in financial planning. We assume that the basic entity in financial planning is a vector representing the values of a variable at successive periods in time. An example would be the tax payable by a company over a number of years. The number of years would be, or could be considered to be, the same for all the variables considered (sales, profit, and so on). Thus the underlying data structure for a financial model is a matrix, thought of as a collection of rows, each representing a variable.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TAX</td>
<td>2250</td>
<td>2475</td>
<td>2700</td>
<td>2925</td>
<td>3150</td>
</tr>
<tr>
<td>SALES</td>
<td>20000</td>
<td>24000</td>
<td>28000</td>
<td>32000</td>
<td>36000</td>
</tr>
<tr>
<td>PROFIT</td>
<td>2750</td>
<td>3025</td>
<td>3300</td>
<td>3575</td>
<td>3850</td>
</tr>
<tr>
<td>MARGIN</td>
<td>5000</td>
<td>5500</td>
<td>6000</td>
<td>6500</td>
<td>7000</td>
</tr>
<tr>
<td>COSTS</td>
<td>15000</td>
<td>18500</td>
<td>22000</td>
<td>25500</td>
<td>27000</td>
</tr>
</tbody>
</table>

Figure 1.

The actual values in Figure 1 exhibit further structure, some of which can be expressed by conventional algebraic equations, as shown below.

\[
\text{MARGIN} = \text{SALES} - \text{COSTS} \\
\text{PROFIT} = \text{MARGIN} - \text{TAX} \\
\text{TAX} = 0.45 \times \text{MARGIN}
\]

Note that these equations express vectorial relationships between the variables. In the form given, each equation serves to define one variable in terms of others. Thus they bear a formal similarity to assignment statements in languages like Pascal and FORTRAN and to the COMPUTE statement in COBOL. Note however, that the operations are, in many cases, vector operations.

The other relationships between the variables are

relationships within rows – for example, the successive values of SALES and of COSTS are in arithmetic progression. In fact, causality being what it is, financial variables typically depend on past values of themselves, and present and past values of other variables, and the dependency could be complex (not, in general, merely an arithmetic or geometric progression). Typically, the “relative” dependencies will not vary from period to period – if X in 1980 depends on Y in 1979, then X in 1981 will depend on Y in 1980. Thus, the modelling language should permit expressions which incorporate relative time lags. Furthermore, a user may want to “work backwards”, that is, specify a target for a future period, and deduce the level of, say, investment needed to obtain this target. Hence, a feature providing for relative time lags seems useful.

Again, some variables may essentially consist of data values -- known liabilities, for example. In other cases, the first few data values of a variable may be known, and subsequent values are predicted from a formula or some sort of trend function. The modelling language should contain features to express these constructs.

To illustrate these points, we might use definitions for variables SALES and COSTS, as follows:

\[
\text{SALES} = 20000, \text{lag} \text{SALES} + 4000 \\
\text{COSTS} = 15000, \text{lag} \text{COSTS} + 3500
\]

These definitions mean: “for the first period, give SALES the value 20000; for subsequent periods, use the previous value of SALES plus 4000”; and similarly for COSTS. Another example:

\[
\text{OUTFLOW} = 2000000, 250000, 0,0,0,0,500000, \text{lag 5 OUTFLOW};
\]

This means, “there will be a major outflow in period 1 running on into period 2, then no significant investment for several periods, then a substantial outflow (presumably equipment overhaul), and so on on a 5-period cycle”.

Thus the values of OUTFLOW in the first several periods would be:

\[
2000000, 250000, 0,0,0,0,500000, 0,0,0,0,500000, 0,0,0,0,0,\ldots
\]

"Copyright © 1982, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted; provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society."

*The first author is with the Department of Computer Science, University of Queensland. The last two authors are with the University Computing Centre, University of Sydney. Manuscript received 13 August 1981, and revised 4 November 1981.

Another group of desirable facilities in most computer languages is that connected with structured programming: the user should have the facilities for a top-down approach to model construction, that is, facilities to break down a model into simpler, smaller submodels. This facility is provided in MODELER by means of user-written "routines". A "routine" consists of a set of variable definitions together with a header which specifies that some of the names referred to in the variable definitions are parameters of the routine. The routine is invoked by including in the main model a statement of the form

use name (actual-parameter-list)

where "name" identifies the routine file and the actual parameters may be variable names or algebraic expressions. Finally, there need to be some house-keeping features. For example, it is necessary to know what the column labels are to be, and hence how many periods the model is to span. Also, it is necessary to know, at least for some purposes, what time interval a period represents.

2. NON-PROCEDURAL NATURE OF THE LANGUAGE

It is desirable that the user should not have to worry about the details of solving the equations which define the model. Thus, for example, the order in which the equations are written down should be immaterial. This contrasts with the situation with assignment statements in FORTRAN and similar languages, where the order of the statements is crucial if one statement affects the values of variables referred to in other statements, as is often the case. The user of an interactive financial planning language is unlikely either to be a programmer, or to have the time and interest to learn the awkward aspects of programming. Thus the user is providing, not a recipe, but a set of ingredients and a description of their interactions. It should be the responsibility of the modelling system to process the ingredients in such a way as to obtain the desired results.

To do this, the system must keep track of the interdependencies between the variables. Let the symbols $A \rightarrow B$ mean "A depends on B". It is necessary to represent dependencies between columns, hence there must be, conceptually at least, a node for every row-column pair, in the matrix of Figure 1. Representing just the two years 1978 and 1979, we would get the graph shown in Figure 2.

![Figure 2](image)

To be in a position to solve a particular location in the matrix, one needs to know that all the descendants of that location's node in the dependency graph have been solved for. Thus the dependency graph appears to be a good tool for determining the sequence of code execution when solving the model.

Let us consider the potentiality for loops in the dependency network. These arise in real-world situations when the inter-relationships between variables determine a system of simultaneous equations. A natural method for solving a model, of the sort discussed in this paper, incorporates a recursive algorithm, which first solves all the variables on which a variable depends, and then executes the code for that variable. It is necessary to detect dependency loops in order for such an algorithm to perform reliably. Preferably, any systems of simultaneous equations found should be solved, (though this may be a group of problems in itself, if the equations are non-linear), and the solution of the system, if unique, should be incorporated into the solution of the model. It is not entirely clear what to do if the solution is not unique: this could either represent a genuine degree of freedom in the model, (to put it otherwise, an under-specified variable), or it could mean an error in the model's specification. For example, the same equation, (suitably permuted), might have been used to define two related variables.

3. DESIGN OF THE INTERACTION WITH THE OPERATING SYSTEM

Whether they know it or not, the users of an interactive financial modelling system will interact with the host operating system in numerous ways. Some of these would normally be visible to or controllable by the user in a typical interactive job control language: for example, file functions, interrupts, and error handling. These topics will be discussed in turn as they affect the design of an interactive modelling package.

File Functions

The basic operations which a user will want to perform on files will include (a) creation or specification of the model file to be worked on, (that is, the file which will contain the text of the model); (b) entering text, modifying it, and subsequently translating this text; (c) some form of backup facility; and (d) destruction of the file when it no longer serves a useful purpose. Of these, the operations in category (b) will be discussed separately, mostly in the section on editing.

(a) Creation or specification of which file will be worked on involves supplying the operating system with a name, and, probably, an indication of whether the file is a new one. The format for the name may well be operating system dependent, as will the exact significance of the "new or old" information, which in fact may also be relevant to the editing and backup strategies. It is convenient to have the file name the same as the model name. The syntax adopted for MODELER is:

create, filename

for a new model-file, or

get, filename

for an old model-file held on a secondary memory device. The actual keywords are, of course, of no importance apart from their mnemonic value, and the comma separator is optional. As "create" initiates the automatic generation of line numbers, it allows extra parameters — see the section on editing and text entry.

(c) A backup facility is necessary to enable recovery
in case of a disastrous modification to a model-file. The control can either be automatic, as in some text editors, which automatically rename a file rather than replace it with a modified version, or it can be performed manually by the user. In MODELER, perhaps because of the nature of the operating system under which it has been developed (namely CDC NOS 1.4), manual version control is used: one types

replace, filename [ = newname ]

in order to make a changed version of a model-file permanent, with an optional new name, or

save, filename [ = newname ]

to make a new model-file permanent.

(d) Destruction of a model-file is a straightforward concept; the MODELER syntax is

purge, filename

Ideally, an interactive package should be proof against all user errors short of physical destruction of the hardware. At the very least, the package should intercept arithmetic errors and terminal interrupts, and take appropriate action. Normally the appropriate action would be to issue a message, return to the package's directive level and issue a prompt for a new directive.

4. PARTIAL RECOMPIRATION AND THE WHAT-IF FACILITY

Often, a financial planner will want to investigate several or many scenarios within the framework of a single basic model. One way to cope with this is for the user to first translate, solve, and print the basic model, and then to conduct a dialogue along the following lines:

"what if INFLATION = 1.10?"
- - - - - (results printed) - - - - -
"what if INFLATION = 1.12?"
- - - - - (results printed) - - - - -

And so on, as required. (More typically, several parameters would have to be adjusted before each set of new results.) One way to implement this is to change the affected parts of the basic model each time, and then repeat the process of translating, solving and printing. However, in many cases, only a few variables will have their values affected by the change, so that much unnecessary translating and solving is done. For this reason, rather than advising the user to edit, translate, and print the model each time, or providing a directive whose effect is to edit, translate, solve, and print, it seems sensible to implement a partial re-compilation facility. With this facility, only the variables whose definitions are changed have their definitions retranslated. In addition, appropriate adjustments are made to the dependency network described in section 2. In order to minimise the work involved in re-solving, it is helpful to be able to trace the dependency links backwards from the changed variables, marking all and only the variables encountered, as unsolved. That is, one marks as unsolved precisely those variables which depend on the changed variables, either directly or at several removes.

It is likely that this facility would also be used to "tune" a model—the user might vary parameters until near-optimum values appear in those variables which the user regards as "key" variables. On the other hand, the user may want the original model back, or may want to keep an intermediate version. Thus, a directive is needed to provide the user with a means of designating which modifications are to be retained, if any.

The syntax used to describe these facilities in the MODELER package is indicated by the generalised sequence of directives shown below

get, modelfile
process
whatif
say variable1 = expression1; repeat as many times as desired. Each set of changes is assigned a "level number"
say variable2 = expression2;
solve key-variable-list
keep list-of-level-numbers
end

Convenience forms of the "keep" directive are provided: "keep all" and "keep none" describe the two extremes of the "keep" directive.

It is often convenient to express the modifications to a variable in terms of the previous values of that variable. For example, one might want to explore the consequences of SALES being 5 percent higher than the basic model. The syntax used for this construction is exemplified by the following:

say SALES = 1.05 * SALES;

This variable definition would not be legal in the basic model, as it would lead to a loop in the dependency network.

5. TEXT ENTRY AND EDITING FACILITIES, AND MODEL-FILE STRUCTURE

Perhaps the most important consideration in choosing the format for model-files is ease of manipulation by the user, as the target users may not have had experience with text editors. Thus the line-numbered file, as used by BASIC systems, suggests itself, as it provides a very easily learnt way of changing the model—one simply types in the line number and the correct text of the line affected. This scheme has disadvantages, particularly if one wants to correct a systematic error throughout a model. So, for users who are familiar with the system and ready to learn more powerful techniques, commands to locate or change strings can be provided.

In MODELER, the text entry facility is invoked by typing a directive of the form

create, modelname [, start, increment]

where "start" and "increment" refer to the line numbers on the file—if omitted, 10 is used as the default for both. The system responds to this directive by typing the first line number, and waiting for the user to enter a line of text. Subsequently, further line numbers are typed out until the user enters an empty line. The user can now change or insert a line by typing its line number and the corrected
text, or by using other editing directives, and can resume entering text by typing

numbers [ , start, increment]

This time the defaults are the last line number issued by the package, and the previous value of the increment. Other editing directives include

\[
c x/old-string/new-string/ [n]
\]

which changes up to \( n \) occurrences of “old-string” to “new-string” on line \( x \), and

\[
a x/old-string/new-string/ [n]
\]

which changes all occurrences of “old-string” to “new-string” on the next \( n \) lines which contain “old-string”, starting from line \( x \). There is also a locate-string directive

\[
l/\text{string}/[n]
\]

which finds up to \( n \) lines containing “\text{string}” and types them out.

The final editing facility provided in MODELER is

resequence, start, increment

which re-numbers the file with evenly stepped line numbers: this is useful if there is no more room for interpolated lines in a much edited file.

6. PREDEFINED FUNCTIONS

One of the features which distinguishes a special purpose package from a general purpose programming language is the range of special purpose functions and/or subroutines built into the package. A financial modelling package will, of course, need special provisions for tax and loan calculations, along with functions for various economic parameters such as the internal rate of return, and net present value. There also need to be some more common routines such as logarithm and the exponential function, and various numerical routines for curve fitting and extrapolation. Finally, there need to be routines to provide quasi-random numbers drawn from several relevant probability distributions, so that planners can incorporate the effects of uncertainty in their models.

In MODELER, all the built-in routines have function-like calls, and many of the more complicated functions operate on whole rows of the model, so that invocations like

\[
\text{RATE OF RETURN} = \text{irr(OUTFLOW,INFLOW)};
\]

are typical — here “irr” is the name of the predefined function, and INFLOW and OUTFLOW are other variables in the model.

A complete list of the predefined functions currently available in MODELER is given in Figure 3.

7. COLUMN DEFINITION STATEMENTS

All the constructs discussed so far process their data on a row by row (equivalently, variable by variable) basis. While the bulk of the desired calculations are likely to fit neatly into this scheme, there is another common pattern

<table>
<thead>
<tr>
<th>MODELER Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>npv</td>
<td>net present value</td>
</tr>
<tr>
<td>npvdl</td>
<td>net present value, varying lifetime of model</td>
</tr>
<tr>
<td>ntv</td>
<td>net terminal value</td>
</tr>
<tr>
<td>beratio</td>
<td>benefit/cost ratio</td>
</tr>
<tr>
<td>irr</td>
<td>internal rate of return</td>
</tr>
<tr>
<td>modified irr</td>
<td>Irr with multiple cash flow reversals</td>
</tr>
<tr>
<td>loan pymt</td>
<td>payment on amortised and unamortised loans</td>
</tr>
<tr>
<td>intr pymt</td>
<td>interest on amortised and unamortised loans</td>
</tr>
<tr>
<td>straightline deprec</td>
<td>sum of years digits method depreciation</td>
</tr>
<tr>
<td>sumofyears deprec</td>
<td>declining balance method depreciation</td>
</tr>
<tr>
<td>decq balance deprec</td>
<td>sum of its arguments</td>
</tr>
<tr>
<td>moving average trend</td>
<td>moving average with specified weights</td>
</tr>
<tr>
<td>polynomial fit</td>
<td>simple linear regression</td>
</tr>
<tr>
<td>interpolate</td>
<td>extrapolate data using fitted polynomial linear interpolation given data points</td>
</tr>
<tr>
<td>step function with specified jumps</td>
<td>step function with specified jumps</td>
</tr>
<tr>
<td>accum</td>
<td>accumulate sum across a row</td>
</tr>
<tr>
<td>sum</td>
<td>sum of its arguments</td>
</tr>
<tr>
<td>ln</td>
<td>natural logarithm</td>
</tr>
<tr>
<td>exp</td>
<td>exponential function</td>
</tr>
<tr>
<td>floor</td>
<td>integer part of a number</td>
</tr>
<tr>
<td>ceiling</td>
<td>integer above a number</td>
</tr>
<tr>
<td>round</td>
<td>nearest whole number</td>
</tr>
<tr>
<td>maximum</td>
<td>largest of its arguments</td>
</tr>
<tr>
<td>minimum</td>
<td>smallest of its arguments</td>
</tr>
<tr>
<td>xtoy</td>
<td>( x ) to the power ( y )</td>
</tr>
<tr>
<td>abs</td>
<td>absolute value or magnitude</td>
</tr>
<tr>
<td>sin</td>
<td>sine</td>
</tr>
<tr>
<td>cos</td>
<td>cosine</td>
</tr>
<tr>
<td>arctan</td>
<td>arctangent</td>
</tr>
<tr>
<td>sqrt</td>
<td>square root</td>
</tr>
</tbody>
</table>

Figure 3.

that needs to be provided for: that is, calculations across a row, on a column by column (or period by period) basis.

A typical example is the calculation of a row total. Rather than provide specialised functions for all likely such applications, it seems to be fairly easy to incorporate a general parser for “column expressions” into the translator. Since it appears unlikely that complicated dependencies between column definitions would occur in practice, the code generated from the column definitions can be executed sequentially in order of definition, after all the row code has been interpreted.

The MODELER version of this construct is exemplified by

\[
\begin{align*}
10 \text{ titles} & = Q1, Q2, Q3, Q4, \text{ YEAR TOTAL} : \\
20 \text{ SALES} & = 1200, 1400, 1600, 1800; \\
30 \text{ column 5} & = \text{col 1 + col 2 + col 3 + col 4};
\end{align*}
\]

8. REPORT SPECIFICATION

While a default report format is satisfactory much of the time for reasonably small models, with large models it is necessary to have a method of limiting the amount of information typed out. A reasonable way to do this is to permit the user to specify a list of variables whose values are to be typed out. A few other items of information can be passed to the report generator to vary the format of the report: perhaps page width and number of decimal places, for example. However, eventually a limit is reached — only so much information can be typed in each time a report is wanted, and detailed specifications, if required, need to be held on a report specification file.
Each line in a report specification file may refer to a variable of the model, and can contain either a template for the report line to be generated, or a pointer to a template shared by several lines.

So far, in MODELER only a default report with three switches to set is implemented. An example of report generation follows:

```
set, dp = 2, fw = 10, pw = 78
print [], list of key variables]
```

This would provide a report where the numbers were printed in fields 10 columns wide, with two decimal places, and any information that runs past a multiple of 78 columns is "wrapped around". That is, the extra information is printed with its own column and row titles after all the earlier columns for each variable have been dealt with. This is demonstrated in the sample terminal session in Section 11, which uses the default values for the dp, fw and pw switches.

9. OTHER FORMS OF OUTPUT

A useful facility in most systems dealing with numerical data is the ability to produce graphs on a terminal. Particular users may want to be able to view the results of solving a model in other ways. These are likely to be so numerous and specialised that it would be impractical to cater for them directly. Perhaps the best compromise is to provide a way of saving the solution of a model on a file for processing by a user-written application, to take care of specialised requirements.

The MODELER package allows the user to produce graphs on a terminal by means of the

```
plot, <list of variables>
```

command. This results in the variables specified being plotted against time (i.e. model periods). The data are scaled to fit (by default) onto a 80-by-24 terminal screen. Other configurations can be selected by typing

```
set, ph = page-height, pw = page-width
```

before the "plot" command.

As described in section 10, the

```
combine, SOLN
```

command saves the solution of the current model on file SOLN, for further processing.

10. AMALGAMATION AND CONSOLIDATION OF MODELS

In some situations, a bottom-up approach to model construction is appropriate. For example, independent sub-divisions of a company may be modelled, and then the sub-divisions put together, along with variable definitions which describe the relationship of the company to its sub-divisions. In MODELER terminology, if we call the models for the sub-divisions PART1, PART2, PART3, and the model segment describing the relationship of parts to whole GLUE, then

```
amalgamate, WHOLE = PART1, PART2, PART3, GLUE
```

builds a new model WHOLE with all the information in it. Experience shows that users find other ways of using this facility. For example, it has been used for incorporating data files into models when the amount of data becomes large.

"Consolidation" is another specialised function, where several subdivision models incorporate identical variable names, and it is desired to automatically add these values of these together to arrive at company totals. In MODELER, the "combine" directive provides this feature:

```
combine, FULLSOLN = SOLN1 + SOLN2 + SOLN3
```

sums the solutions of the current model and the solutions held on files SOLN1, SOLN2, SOLN3, and saves the combined solution on file FULLSOLN. In particular

```
combine, SOLN
```

saves the solution of the current model on file SOLN.

11. SAMPLE TERMINAL SESSION

Figure 4 shows a simple MODELER session, included to clarify some of the concepts, not to demonstrate all the facilities implemented.

```
SYDNEY UNIVERSITY INTERACTIVE PLANNING SYSTEM
V 1.6 80/10/20. 17.09
ENTER DIRECTIVES:
? get, example
EXAMPLE CONTAINS 10 LINES
? list
  10 TITLES = 1980 .. 1984;
  20 !
  30 MODELLING THE FIVE YEARS FROM 1980 TO 1984
  40 ;
  50 SALES = VOLUME * SELLING PRICE;
  60 VOLUME = INITIAL VOLUME, LAG VOLUME * 1.09;
  70 SELLING PRICE = IF VOLUME < 3000
  80 THEN 2.30
  90 ELSE 2.05;
  100 INITIAL VOLUME = 2500,0;
? set list off
? process
PROCESSING EXAMPLE 80/10/20. 17.10.20.
NO ERRORS DETECTED
? solve
SOLVING EXAMPLE 80/10/20. 17.10.24.
NO ERRORS DETECTED
? print
SALES    5750.00  6267.50  6831.58  6637.02
VOLUME   2500.00  2725.00  2970.25  3237.57
SELLING PRICE  2.30  2.30  2.30  2.05
INITIAL VOLUME  2500.00  0.00  0.00  0.00
  1984
SALES    7234.36
VOLUME   3528.95
SELLING PRICE  2.05
INITIAL VOLUME  0.00
? end
Figure 4.
```

11. ACKNOWLEDGEMENTS AND REMARKS ABOUT IMPLEMENTATION

The design presented above has been influenced by what we perceive as the strengths and weaknesses of several past and present financial planning packages.

The authors would like to thank the referees for suggestions which improved the presentation and content of this paper.

The MODELER package is now largely implemented and is available on the CYBER/NOS system at the University Computing Centre, University of Sydney, and on the DEC/TOPS-10 system at the Prentice Computing Centre, University of Queensland. In both cases, about 7500 lines of Pascal and about 200 lines of assembly language are used in the implementations.

BIOGRAPHICAL NOTES

William H. Wilson received a BSc with first class honours and a MSc in Pure Mathematics from the Australian National University in 1970 and 1972, a PhD in Pure Mathematics from the University of Sydney in 1977, and a DipCompSci from the University of Queensland in 1979. He taught Mathematics at the University of Queensland before becoming a Programmer at the Sydney University Computing Centre in 1979. Since January 1981 he has been a lecturer in Computer Science at the University of Queensland. His research interests include software engineering and design and implementation of non-procedural programming languages.

Christopher F. Burrows was awarded a BSc (Hons) degree in Physics by Nottingham University in 1974. His first computing work was with G. Maunsell Pttrs (Consulting Engineers) in London. Since 1977 he has worked as an applications programmer at the Computing Centre at the University of Sydney, where he is currently completing a DipCompSci. His interests are in programming languages, operating systems, and microprocessors.

Michael M.B. Sidhom graduated from the School of Mechanical Engineering, University of Alexandria, in 1959. He was one of the first recruits in a group commissioned to establish an Egyptian jet engine production complex, where he became interested in OR production and job-scheduling related problems. In 1973, he joined the University Computing Centre, University of Sydney, as a systems analyst, and was promoted in 1976 to Applications Manager and to Manager in 1980. His interests are in applications of operations research.