AT&T Technical Journal
Vol. 64, No. 9, November 1985
Printed in U.S.A.

HEQS—A Hierarchical Equation Solver

By E. DERMAN* and E. G. SHEPPARD†

(Manuscript received April 9, 1984)

HEQS is a set of tools for numerically solving sets of algebraic equations from their description in a text file. It allows users to compactly define (or alter an already defined) set of equations (a *model*), and then analyze and solve it with minimal intervention. HEQS automatically checks the algebraic and logical consistency of the equations, reports errors, and allows users to correct the errors by simply changing the original textual description of the relevant equations. HEQS can deal with unsubscripted or multiply subscripted (array) variables. Because it relieves users of the need to perform repetitive algebraic substitution and evaluation, it is most useful for sets of equations involving from tens to thousands of variables. HEQS commands can be used (1) interactively, to define and solve models, or (2) as a set of *UNIX*™ operating system high-level algebraic tools for building applications that require model solving. This paper describes the motivation and design of HEQS, illustrates its use, and highlights its underlying algorithms.

I. INTRODUCTION

1.1 HEQS: A modeling environment

HEQS (Hierarchical Equation Solver) is a package of C and *UNIX* system shell programs that allows users to interactively define, debug, modify, and obtain the numerical solution to models described by sets of algebraic equations in a text file. The equations it handles can involve unsubscripted or multiply subscripted (array) variables. Used interactively, HEQS programs provide a *modeling environment* for end users to build and solve models. Used from within shell scripts or menus, they form a set of high-level algebraic tools for builders of applications that require model-solving capabilities.

* AT&T Bell Laboratories; now with Goldman Sachs & Co. † AT&T Bell Laboratories; now with Asymetrix Corp.

HEQS frees users from tedious and repetitive checking of the logical consistency of equations, from algebraic substitution, and from numerical solution. For this reason, it is most useful for repetitively solving a set of equations while changing some data values or equations, or for solving large equation sets involving from tens to thousands of variables. Even for just a few equations, however, HEQS makes it easy to enter them as text and directly obtain the solution.

1.2 Nonprocedural model solving

The input or primitive of HEQS is a *model*—an easily edited text description of a set of equations (and associated comments) to be solved. Models are most naturally kept in *UNIX* system files. Their equations can be written in any order the user finds conceptually useful. HEQS itself finds an order, or *hierarchy,* for the numerical solution of the equations, hence its mnemonic name.

To analyze, find the errors in, or solve models, users invoke simple one-word HEQS commands. Despite the logical and algebraic operations HEQS performs on the model, all subsequent HEQS output about model errors or solutions refers to the original user description; this localizes for users the source of the error, and so suggests the necessary correction or modification. This feature, that users need deal only with their model description entered in any order, without specifying a method for analysis or solution, makes the equation solving system *nonprocedural.*

HEQS is therefore useful for mathematically naive users, who cannot themselves solve small (but perhaps complicated) sets of equations, as well as sophisticated users, whom it frees from repeated algebraic and numerical analysis of large models that are frequently changed.

1.3 HEQS under the UNIX *operating system*

HEQS programs are tailored to the *UNIX* system and its shell.* The programs can be used in the two modes:

1. As simple high-level commands to solve models, or

2. Combined with the control structures of the shell, to provide a model-solving language for use by applications system builders that require equation-solving capabilities for their users.

1.4 Main features

HEQS provides

1. A shorthand *language* for compactly describing large sets of algebraic equations involving multiply subscripted variables,

* The commands are all implemented as separate programs that communicate with each other through intermediate files, analogous to the *UNIX* Source Code Control System package.

2. An *interpretive nonlinear solver* with predefined and user-definable functions,

3. Extensive checking for common logical and algebraic errors in the description of models, and for numerical problems that occur during their solution,

4. *What-if analysis* (determining the numerical effect of a change in some model equations),

5. *Goal-seeking* (determining what data values guarantee particular output values),

6. *Sensitivity analysis* (determining the variation in variables of interest when the data changes), and

7. *A model compiler*—that is, an automatic code generator that produces a C program for rapidly solving a particular HEQS model with user-specified data values. This allows applications developers to provide numerical solution of particular models more efficiently than the interpretive HEQS solver does for general models, and furthermore allows the applications program to execute on machines that do not run HEQS.

Graphics, report generation, etc., may be obtained by linking HEQS output to other *UNIX* system tools.

HEQS is more flexible and powerful than standard commercially available "spreadsheet" programs, which usually handle only fixed-length time series of limited algebraic complexity. HEQS accepts large sets of equations for subscripted or unsubscripted real variables involved in both simultaneous and nonlinear relationships, and has more error-detection facilities.

This paper is arranged as follows. Section II contains some varied illustrations of the use of HEQS. Section III describes the current status of HEQS. Section IV briefly describes the motivation for building HEQS, and the design. Section V gives an overview of the implementation, and Section VI discusses error reporting, an important feature for users. Finally, Section VII describes our conclusions. An Appendix on the algorithms and data structures used is included.

II. USING HEQS: EXAMPLES

This section contains three fairly lengthy transcripts of HEQS sessions run under the *UNIX* system. They illustrate, in successively more complex examples, some of the features of HEQS.

In order to annotate the transcripts with explanations, we adopt the following convention. Boldface text below denotes HEQS commands being invoked by the user. A $ sign denotes the computer prompt. Italicized text before or after a HEQS command contains explanatory remarks about the command and the reason for its invocation; these remarks are not part of the transcript; they explain why particular commands are being invoked. Finally, typewriter text denotes the computer's response.

2.1 Defining, correcting, and solving a simple model

This example illustrates the solution of a few equations kept in the file *badmod.* The original model contains some errors, which are successively found and corrected with the HEQS system's help.

```
$ cat badmod        Read the model in the file badmod.

# This file badmod contains a simple model with errors
# written in HEQS equation notation
# This is a comment because the line starts with a # sign
         /* This is also a comment because it started with a
            /* and ends with the following */
# Here are the equations
# Note that some equations are simultaneous and that they
#   are
# written in an arbitary order which is not the order of
#   solution.
B = 2*C - D + 1
D = 4
A = (B+C)//D + sqrt(E)
  # note: typo - two divide signs
C = 3*B - 2*D
  # note: no equation for E

$ cheq<badmod       This command reads the model into HEQS and
                    checks equations for errors.
error in line 3:
A = (B+C)//

CANNOT PROCEED!
CORRECT MODEL  AND REENTER.
```

The model has been corrected—look at it in file bettermod1.

```
$ cat bettermod1

B = 2*C - D + 1
D = 4
A = (B+C)/D + sqrt(E)
  # note: typo fixed
C = 3*B - 2*D

  # note: no equation for E
$ cheq<bettermod1      Read correct model into HEQS.
$ canislv        Check consistency of the model—can it be solved?
```

```
CANNOT SOLVE FOR:
        A
BECAUSE NO DATA OR EQUATIONS AVAILABLE FOR:
        E
```

$ cat bettermod2 *Look at edited and recorrected model.*

```
B = 2*C − D + 1
D = 4
A = (B+C)/D + sqrt(E)
        # typo fixed
C = 3*B − 2*D
E = −1
       # added missing equation
       # but sqrt(negative) is illegal
```

$ cheq <bettermod2 *Read in next corrected version of model.*
$ slv *Solve the model.*

```
D=4
C=3.4
B=3.8
E=−1

An argument to a mathematical function is outside the legal
range in equation
       A=(B+C)/D+sqrt(E)

Probable origin in model is
       line 6: equation: A = (B+C)/D + sqrt(E)
```

$ cat bestmod *Look at final corrected version.*

```
B = 2*C − D +1
D = 4
A = (B+C)/D + sqrt(E)
C = 3*B − 2*D
E = +1
       # made E positive
```

$ cheq <bestmod *Read final version into HEQS.*
$ slv *Solve final corrected version.*

```
D=4
C=3.4
B=3.8
E=1
A=2.8
```

```
$ whatif        Now tinker with model using HEQS command 'whatif'.
          D=5
D=5
C=4.4
B=4.8
E=1
A=2.84

$ sens A D=5    Ask how sensitive A is to D at D=5 using 'sens' com-
                mand.
A +1.0% change in variable ''D'' causes a
         +0.1% change in variable ''A''
```

Use 'repslv' command to see how A varies with D by repetitively solving the model for several D values.

```
$ repslv A D=1,2,3
D[1]=1    A[1]=2.2
D[2]=2    A[2]=2.6
D[3]=3    A[3]=2.733333333
$
```

2.2 Solving a projectile problem

This example illustrates the kinematics of a projectile fired vertically upwards from the earth's surface. The model's equations are defined in the file *rocket_eqs*. HEQS is then used interactively to explore what the model predicts. The `goalsk` command, which solves the equations implicitly, is used to find the time at which the projectile reaches the apex of its trajectory.

```
$ cat rocket_eqs
/*  variables   */

# s - distance above earth's surface in feet
# v - final velocity in ft per sec

/*  parameters  */

# u - initial velocity in ft per sec
# t - time of flight in seconds
# a - acceleration (due to gravity) in ft per sec sq

/*  equations  */

s = u*t + 0.5*a*(t**2)
v = u + a*t
```

```
/*  initial parameter values  */
t = 1
a = −32
u = 88
```

`$ cheq <rocket_eqs` *Read model into HEQS.*
`$ slv` *Solve it for initial parameter values.*

```
a=−32
t=1
u=88
s=72
v=56
```

Find value at t=2.

```
$ whatif
      t=2

a=−32
t=2
u=88
s=122
v=24
```

Find value of t at which velocity v is zero—i.e. projectile is at apex.

```
$ goalsk
      t:v=0

a=−32
v=0
u=88
t=2.75
s=121
```

Use 'repslv' command to see how v varies with t
by repetitively solving the model for several t values.

```
$ repslv v  t=0:3:13

  t[1]=0          v[1]=88
  t[2]=0.25       v[2]=80
  t[3]=0.5        v[3]=72
  t[4]=0.75       v[4]=64
  t[5]=1          v[5]=56
  t[6]=1.25       v[6]=48
  t[7]=1.5        v[7]=40
  t[8]=1.75       v[8]=32
```

```
    t[9]=2        v[9]=24
    t[10]=2.25    v[10]=16
    t[11]=2.5     v[11]=8
    t[12]=2.75    v[12]=0
    t[13]=3       v[13]=-8
$
```

2.3 A complicated financial model

Table I illustrates the use of HEQS on a toy financial model kept in the file *bearmod*; it includes many comments, and should be self-explanatory. This model uses features of the HEQS language for describing multidimensional arrays, for writing conditional (IF-ELSE) equations that depend upon Boolean values, special built-in functions like SUMOF that sum elements of arrays, and macro definitions that make the model easily maintainable.

Note that the model is truly nonprocedural: equations are written to describe the financial relationships in an order that makes them easy to understand, but not necessarily easy to solve. HEQS provides the intelligence to determine an order and method for solution.

III. PRESENT CAPABILITIES

HEQS' main virtue is to allow the easy definition, alteration, and solution of models like the ones given above. The characteristic feature of such models is that although they may involve hundreds or thousands of variables, they decompose into interdependent irreducible subsets of simultaneous equations, each subset involving only a few (generally less than ten) variables.*

Analyzing and solving the 112-variable model of Section 2.3 on a Western Electric 3B20 Simplex or Digital Equipment Company VAX 11/780 minicomputer running the *UNIX* System V operating system requires less than ten seconds of user plus system CPU time, as reported by the *UNIX* system command `time`. About six seconds of this time is devoted to reading the model into the system (via the HEQS command `cheq` described in more detail later) and analyzing it to find an order for solution. The numerical solution itself, invoked by the `slv` as illustrated in the examples of Section II, requires the remaining four seconds of user plus system CPU time.

Whenever any equations are altered or added to the model, subsequent analysis, error-checking, and solution still require at most this amount of time. Changing the model means simply reediting the

* Although HEQS imposes no actual limit on the number of variables in a simultaneous subset, it was not designed to efficiently solve irreducible subsets of simultaneous equations involving hundreds of variables. The financial and market models for which it has been typically used so far have all satisfied this criterion.

original model file. Prior to the existence of HEQS, a model of this type would have been solved by writing an ad hoc procedural Fortran or C program of approximately a thousand lines, or by writing special preprocessors to commercially available spreadsheet programs with less power. This would have required at least several hours of work, plus a similar amount of maintenance time for any significant alteration to the model. Altering or expanding a HEQS model requires no new programming; the savings in time and labor gained by using HEQS is therefore obvious.

For larger models with thousands of variables, HEQS' running time increases roughly in proportion to the number of equations in the model, as well as the equations' complexity. For example, a one-thousand-variable model requires 37 seconds of user plus system time to pass through `cheq`, and 51 seconds to be solved via `slv`. Changes to equations or data values in the model can be made interactively and the model again solved in similar amounts of time. The advantage of using HEQS therefore becomes even more apparent.

Although 51 seconds is not a long time to wait for a solution when one is developing or refining a model, it may be too long when that refined model becomes part of an application program. For such cases HEQS provides a command `compmodel`, described in Section V; `compmodel` compiles a completed model into an efficient C program that solves the model for different data values much more quickly, as described in Section 1.4. Compiling the thousand-variable model mentioned above results in a reduction of solution time from 51 seconds to 3 seconds, an appreciable decrease that ensures that the response time of the compiled solving program is more than adequate for applications end users.

Future HEQS enhancements may include integrating these equation-solving capabilities with reports and graphics functions, which users of HEQS must currently obtain by passing their solutions to other *UNIX* system tools.

IV. MOTIVATION AND DESIGN

4.1 History

HEQS arose out of the desire to automate the often tedious development and maintenance of Fortran-based financial modeling programs being written or planned in our organization in 1979/1980. We realized[1] that the programs being written were really solving an algebraic representation (a *model*) of some corporate financial structure under varying end-user assumptions, and we aimed our efforts at automating this process.

The implementation of these programs was the result of many error-susceptible interactions between analyst end users and programmers.

Table I—Three bear model

```
$cat bearmod
#########################################################
#                    THREE BEAR MODEL
#########################################################

       /*
       * This is a financial model for the bear family.
       * Once upon a time there were three bears: poppa, momma and baby bear.
       * Every winter they hibernated, every fall they were unemployed.
       * To survive, they needed 400 pounds of porridge a month.
       * Each month they could use both their savings and earnings
       * to buy porridge.
       * This model lets them determine how their survival each month is
       * affected by prices, salaries, etc.
       */

       #     A definition of the family, months and seasons follows.
       #     The operator << used below, as in t<<1, denotes a shift of
       #     the time series t to the left by one unit; thus, if t is
       #     the series 1 through 12, t<<1 is the series 0 through 11.
       #     Note how the DEFINE statements below allow easy maintenance.
       #     Adding an extra bear to the whole model, or altering the definition
       #     of winter simply requires changing a DEFINE statement to propagate
       #     change through the whole model.
```

```
###########################################################
DEFINE BEARS          poppa momma baby
DEFINE MONTH          1:12
DEFINE PREV(t)        t<<1    /*  this defines the macro function PREV()  */
DEFINE SUMMER         MONTH > 5 && MONTH < 9
DEFINE FALL           MONTH == 9 | | MONTH == 10
DEFINE WINTER         MONTH > 10 | | MONTH < 3
###########################################################

        /*  Note that the equations below have been entered in      */
        /*  the order suitable for thinking about the model,        */
        /*  NOT the order for solving it.                       */

        /*  Note also that although income[poppa,MONTH] and         */
        /*  income[momma,MONTH] for any MONTH in the SUMMER         */
        /*  are involved in simultaneous sets of equations,         */
        /*  the user need not worry or even be aware of this.       */
        /*  HEQS will figure this out and solve it.               */
        /*  Finally, note that when the left-hand side of an        */
        /*  equation involves a subscripted variable like           */
        /*  income[poppa,MONTH], many equations are being           */
        /*  simultaneously defined. This is much easier and         */
        /*  less procedural than writing something like           */
        /*      FOR i in 1 to 12                                */
        /*      DO                                              */
        /*        income[poppa,i]=. . . .                         */
        /* DONE                                                 */
```

Table I—continued

```
#------------------------------------------------------------------------------------
#                  INCOME STATEMENT
#------------------------------------------------------------------------------------

monthly_income[MONTH]      =    SUMOF income[BEARS,MONTH]

income[poppa,MONTH]        =    IF       (WINTER || FALL)
                                THEN     0
                                ELSE     100 - penalty_2inc * income[momma,MONTH]
                                /*******************************************************
                                poppa's income diminishes when momma earns and
                                he must babysit; penalty_2inc is the coupling
                                factor between the two incomes.
                                *******************************************************/

income[momma,MONTH]        =    IF       (SUMMER)
                                THEN     0.25 * income[poppa,MONTH]
                                ELSE     0
                                /*******************************************************
                                momma works only a quarter of the time poppa
                                does, and only in summer
                                *******************************************************/
```

```
income[baby,MONTH]          =   0

bank_balance[MONTH]         =   bank_balance[PREV(MONTH)]
                                + monthly_income[MONTH] - money_spent[MONTH]

money_spent[MONTH]          =   porridge_needed[MONTH] * price_of_porridge

# -----------------------------------------------------------------
#            FOOD AND SURVIVAL CONSTRAINTS
# -----------------------------------------------------------------

survival[MONTH]             =   IF ( survival[PREV(MONTH)] == 1 &&
                                porridge_avail[MONTH] >= porridge_needed[MONTH] )
                                THEN   1
                                ELSE   0
                                /*****************************************************
                                a value of 1 means they live  ;
                                0 means death by starvation
                                *****************************************************/

porridge_needed[MONTH]      =   IF (WINTER)
                                THEN  0         /*  hibernation  */
                                ELSE  400

porridge_avail[MONTH]       =   (bank_balance[PREV(MONTH)] + monthly_income[MONTH])  /
                                price_of_porridge
```

Table I—continued

```
# ----------------------------------------------------------------
#                  DATA
# ----------------------------------------------------------------

price_of_porridge          =    0.25
penalty_2inc               =    0.1     /*  estimated penalty factor for 2 incomes  */
bank_balance[0]            =    0       /*  initial bank balance  */
survival[0]                =    1       /*  they start out alive  */

$ cheq<bearmod       Initial HEQS command to read the model from its file.
$  canislv       HEQS command to ensure all unknowns can be solved for.
$  slv      HEQS command to numerically solve the model.
   . . . . .
survival[0]=1        Only the answers of interest, not the whole solution, are shown here.
survival[1]=1
survival[2]=1
survival[3]=1
survival[4]=1
survival[5]=1
survival[6]=1
survival[7]=1
survival[8]=1
survival[9]=0
survival[10]=0
survival[11]=0
survival[12]=0
   . . . . .
```

```
$  whatif       HEQS command to speculatively change and solve model.
          price_of_porridge = 0.26
   .....
survival[0]=1
survival[1]=1
survival[2]=1
survival[3]=1
survival[4]=0
survival[5]=0
survival[6]=0
survival[7]=0
survival[8]=0
survival[9]=0
survival[10]=0
survival[11]=0
survival[12]=0
   .....
$
```

System requirements and model equations were elicited from analysts. Maintenance was driven by the frequency of the analysts' needs to change the model to reflect actual and permanent regulatory, legal, accounting, and corporate structure changes. Finally, analysts often wanted to make speculative and temporary changes to models to investigate their assumptions in forecasting.

We decided to create tools that would let analysts do their work without having to rely on programmers. Since their underlying need was to solve an algebraic model, we started to design a system that would solve models from a high-level description of its structure and equations. Such a system would let analysts define, solve, and alter their models as their needs dictated; it would also let programmers who still built applications for analysts do their work more easily and rapidly.

The first system built was EXEC, a 1980/1981 prototype automatic code generator. It produced hard-coded Fortran subroutines to calculate the values of variables in a model by accessing a database of corporate and financial model equations previously set up by an administrator. EXEC was implemented in the *UNIX* system shell language and could solve only nonsimultaneous equations.

It soon became apparent that it was preferable for users to build and solve models interactively, since model refinement is an iterative process requiring frequent debugging. In contrast to EXEC's compilation of models into Fortran subroutines, HEQS was therefore designed as an interpreter that directly solved models. Its specifications and some design features are described below.

4.2 Requirements

To allow nonprogrammers to develop and solve models, we required that HEQS' only primitive should be a text file of equations. However complex the algebraic analysis and solution of the model, HEQS users should only have to deal with a straightforward text description of the model. Users should be free to write equations and comments in any conceptually useful order, in a notation as close to common algebra as possible.

We further required that HEQS provide interactive model definition and solving, with extensive error checking in all phases of modeling. Errors that might prevent the complete solution of a model had to be reported to the user in terms of the responsible equation or problem in the original model. Error correction should require only that the user correct by reediting the original high-level description of his model. We also decided that HEQS should make *no* attempt to remedy user errors in their models by guessing at their intention. Responsibility for the meaning of a model should lie with the modeler only. HEQS should provide power, but not at the expense of safety.

These requirements for interactiveness led us to implement large parts of HEQS as an interpreter. This has the obvious advantage of making it easy to provide good debugging, and the disadvantage of slower execution (i.e., numerical solution) of debugged models. We judged this worthwhile, but in fact later added a `compmodel` command, described in Section V, that allowed the production of efficient C programs to solve particular models.

The objects of interest in financial and other types of modeling are often multidimensional structures, for example, the revenues of a company for each division and each year. Such structures are well represented by subscripted variables (arrays) and so HEQS was required to accept equations for both subscripted variables (sometimes called *tensors* in the discussion below) and unsubscripted (*scalar*) variables, and provide a variety of operators for manipulating them. Since not all functions that users might need could be foreseen, HEQS had to allow users to define their own functions. It needed numerically to solve sets of nonlinear simultaneous equations for these variables. Finally, to cater to the investigatory nature of modeling, it needed to solve models in a whatif, goal-seeking, and sensitivity mode, as defined in Sections 1.4 and V.

Since we expected that HEQS would be used as a high-level applications-building language, we also required it to be compatible with other *UNIX* tools. This led us to implement the whole system as a collection of programs, each performing a simple analytic step in model solving, with each program returning appropriate error codes to indicate successful or unsuccessful task completion. Applications developers could then build larger modeling systems quickly by embedding HEQS commands in shell scripts under the control of the shell.

4.3 Design

4.3.1 Solving sets of equations

Our aim was to provide a modeling environment where users maintain and solve their own models without programmer invervention. We therefore tried to make HEQS mimic human algebraic reasoning, analyzing, and solving systems of equations in a manner similar to the way people would. In this way, when an error occurs, preventing the complete solution of a model, it can be localized for users at a point close to where they would have discovered it themselves.

How would one solve large sets of algebraic equations* of the form

$$lhs\_variable = expression?$$

* From now on, "equation" always means an equation of the form *left-hand-side variable* = *expression*, as distinct from the more general algebraic equation *expression* = *expression*. The "*left-hand-side variable*" above may, however, be an array or tensor implicitly involving several scalar variables.

As an illustration consider the set

$$a = 2b - c, \tag{1a}$$

$$b = 3a + c^d, \tag{1b}$$

$$c = 4d - e, \tag{1c}$$

$$d = 7c + 3, \text{ and} \tag{1d}$$

$$e = 4. \tag{1e}$$

One would carry out the following steps.

1. Check the algebraic syntax of the equations.
2. Analyze the variable dependencies. In this example, we note that

   a depends on *b* and *c*;
   b depends on *a*, *c*, and *d*;
   c depends on *d* and *e*;
   d depends on *c*;
   e is known.

3. Determine those irreducibly simultaneous subsets of variables whose equations must be solved simultaneously. Here,

   a and *b* are irreducibly simultaneous,
   c and *d* are irreducibly simultaneous,
   e is (trivially) irreducibly simultaneous.

4. Find an order for successive numerical solution of the simultaneous subsets which guarantees that, as each subset is solved, all unknowns on its right-hand side have already been solved. Here,

   e is known,
   thus *d* and *c* can be evaluated,
   finally *a* and *b* can be evaluated.

5. Solve the subsets numerically in this order.

These are the steps HEQS should carry out.

HEQS uses a graph-theoretic approach[2] to model analysis and error checking, and a variable substitution algorithm for the numerical solution of equations.[3] For nonlinear equations, the variable substitution is supplemented by an iterative approximation.[4] These together provide a good paradigm for simple human analysis, error detection, and solution.

To provide a formalism for model analysis and solution, models in HEQS are internally represented as directed graphs[5] and are described in greater detail in the Appendix. The illustrative set of equations

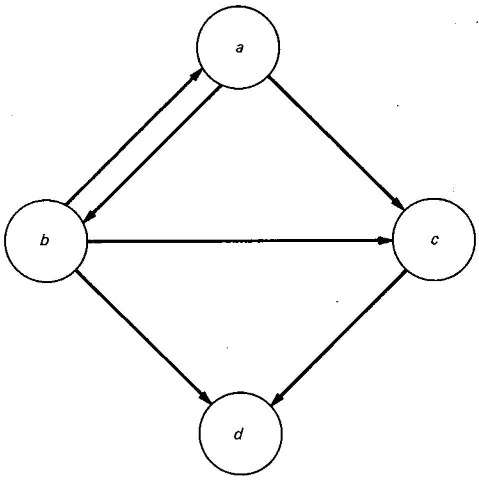

Fig. 1—Directed graph displaying the dependency structure of the model in eq. (2).

$$a = 2b - c, \tag{2a}$$

$$c = d + 2, \tag{2b}$$

$$d = 3, \text{ and} \tag{2c}$$

$$b = 3a + c^d \tag{2d}$$

is represented by the directed graph in Fig. 1. Here, graph vertices correspond to variables, and directed edges or arrows correspond to the equal sign that shows the dependency of left-hand-side variables upon the right-hand-side variables in their defining equations. Strong components (sets of vertices such that any two are connected to each other by paths in either direction) correspond to sets of irreducibly simultaneous variables whose equations can be solved only with simultaneous solution techniques. Given this correspondence between equations and graphs, graph theory provides a rich source of algorithms for analyzing and solving models. Some of these are listed below.

1. Strong component algorithms are useful for isolating simultaneous subsets of variables.

2. Depth-first search from graph roots is useful for determining the order in which these subsets can be solved.

3. Depth-first search from a vertex corresponds to tracing the effect of changing the value of a variable (i.e., the vertex) upon other variables in the model, that is, doing "what-if" analysis.

4. Algorithms for finding paths between vertices in the graph can be used to help solve the implicit equations that occur in goal-seeking. These algorithms are described in more detail in the Appendix. In

brief, graphs provide a formal way of representing human thinking about models.

The graphical representation and strong component algorithms also allow HEQS to subdivide the large model into a sequence of smaller ones, each more easily inspected and solved. This lets HEQS give human-oriented error messages. Helping users debug large models precludes solving a model by inverting a matrix for all the equations, or using any other technique that treats the whole model as one irreducible set of equations. A matrix inversion solver is inadequate in any case because models of interest are generally nonlinear and sparse.

4.3.2 Modular architecture

The basic architecture of the system was chosen to model corresponding steps in the equation-solving process. It is displayed in Fig. 2. Details of the system are contained in Section V. Its front end was a macro preprocessor that parsed and then *scalarized* compactly written scalar or tensor equations into their scalar text equivalents; it then abstracted and stored in a graph (as described above) the dependent (left-hand-side) variable and its independent (right-hand-side) variables for each scalar equation. These scalar equations and their graph were then passed to a module that decomposed them (using strong component algorithms) into the smallest possible sets of simultaneous equations, and determined an order (the *hierarchy* of the mnemonic "HEQS") for their solution. A numerical solver then found the solution to the scalar equations passed to it in the appropriate order. The advantages of this modular approach are listed below.

1. It allows the crucial separation of the "scalarizer," which defines the language for writing equations, from the solver, which finds the solution. This makes possible the independent enhancement or replacement of either, and has already proved extremely useful.

2. Each module can be independently developed, with one module assigned to a programmer.

3. Each module can be independently debugged owing to the weak coupling of all modules, which communicate only by intermediate files. These files are also useful in finding programming errors.

4. Modeling can be conveniently halted when errors occur, because each module performs a limited analytic task.

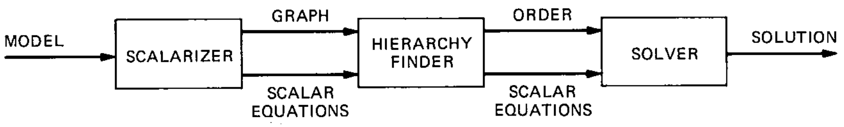

Fig. 2—HEQS functional architecture.

The main disadvantage of this approach is suboptimal time and space efficiency owing to information transfer between modules, but this can easily be dealt with by further development and integration if necessary.

V. HEQS ARCHITECTURE

We now present a brief description of the architecture of HEQS, and the commands available. The commands in HEQS are implemented as separate programs, described below. Their interactions via intermediate files are displayed schematically in Fig. 3. In this figure, rectangles denote HEQS commands, ellipses denote intermediate files, and arrows show the information flow.

`cheq` (CHeck EQuations) is the primary gateway into HEQS' modeling environment. It reads equations and comments from a file or standard input, parses them, reports syntax errors, scalarizes as necessary, and builds a dependency graph for future model analysis. `cheq` has full macro capabilities and understands tensor notation, allowing

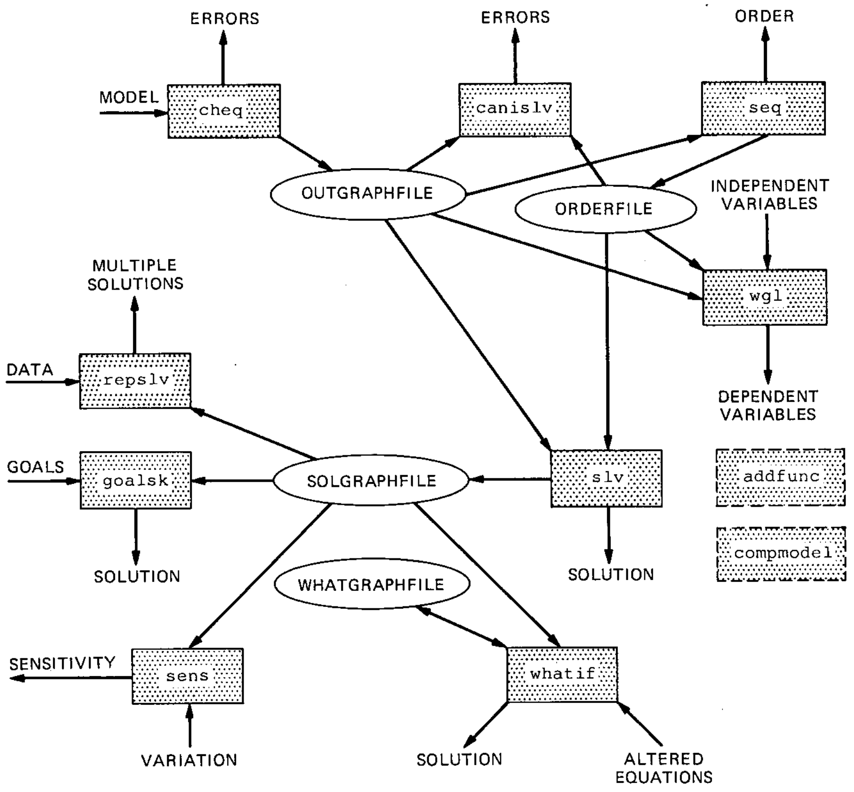

Fig. 3—HEQS architecture.

powerful and compact representation of large equation sets. The equation language it accepts was illustrated in Section II. The dependency graph that contains all model information is stored in an intermediate file (the *outgraphfile*) for use by other programs, whose first action is almost always to reconstruct all relevant information about the model by opening this file.

`seq` (Sequence EQuations) decomposes the model entered via `cheq` into linked simultaneous subsets of equations and then determines an order for solution that allows them to be solved subset by subset. The order is stored in another intermediate file (*orderfile*) for use by `slv` below.

`canislv` (CAN I SoLVe) uses the order for solution in *orderfile* to test whether the model is underdetermined. It reports all variables that do not appear on the left-hand side of some equation in the model (and thus cannot be solved for), as well as variables whose values cannot be found because they depend upon the value of such variables. It is intended to provide model builders with interactive help in completing their model.

`slv` (SoLVe) applies a numerical solver to the simultaneous subsets of the model in the order determined by `seq`, and reports the solution. The graph corresponding to the solution is left in an intermediate file (*solgraphfile*). If the solution fails for some reason (unknown functions, unspecified variable values, overflows, illegal function arguments, nonconverging iterative solution to a nonlinear equation, etc.), `slv` reports the problem and the simultaneous subset in which it occurs. In this way, it pinpoints the nature and location of the model error, and so suggests (one hopes) how the user can correct it.

`slv` has several additional features. It contains a library of common mathematical functions it can evaluate when they occur in a model's equations. It solves nonlinear equations by iteration, and requires that users provide an estimated initial value for any variable whose value is found in this way. Finally, HEQS has facilities for customizing `slv` by linking user-defined C subroutines to its library (see `addfunc` below).

`Heqs` (Hierarchical EQuation Solver) is a short shell program utilizing `cheq`, `canislv`, and `slv` to parse, check for consistency, and solve equations in a file. It provides a simple one-word command for new users to solve and debug models, and illustrates how *HEQS* commands can be used with the shell to write model-solving scripts.

`Whatif` (WHAT-IF) determines the numerical effects of tinkering with a model by changing its equations. It accepts from the standard input a small set of new equations to replace speculatively some equations in the model, determines what variables of the model need be recalculated, and finds the value of these variables again. It leaves the altered model and its solution in a new intermediate file (`whatgraphfile`). This is used for doing successive `whatif`'s or `goalsk`'s (described below) on models already altered as well as on the original model.

`Goalsk` (GOAL-SeeKing) allows the determination of the data values in the model that guarantee particular output values for the solution. Models normally contain all unknowns of interest on the left-hand side of defining equations, with the understanding that these unknowns are to be determined. `Goalsk` allows users to specifiy (1) a target value for a set of left-hand-side unknowns, and (2) a set of right-hand-side variables for which they would like to know values that guarantee the targets. HEQS then tunes the right-hand-side variables to the appropriate values, and reports them. This is equivalent to finding the numerical solution of implicit equations in the model.

`Sens` (SENSitivity) calculates for users the percentage change in the values of each of a set of model variables that result from a specified percentage change in another specified variable. Its effect is similar to that of several `whatif` commands in succession.

`Repslv` (REPetitive Solution) repetitively solves a model for a range of different input values for data variables. It is analogous to a Monte Carlo solution of a model.

`Wgl` (WigGLe) does variable impact analysis on a model. Given one or more variable names as arguments, it reports the names of variables whose values are affected by changes ("wiggles") in the values of the arguments, owing to the implicit dependence induced by the model's equations. It is intended to aid users interested in analyzing the effects of tinkering with a model.

`Addfunc` (ADD a user-defined FUNCtion) allows users to add their own C subroutines defining their own functions to the numerical solver used by `slv`, `heqs`, `goalsk`, `whatif`, `repslv`, and `sens`. It produces an extended version of the `slv` program for personal use by the user. Since it is a "meta" command that modifies the HEQS `slv` command rather than solving a user's model, it is displayed as a dashed box next to the `slv` command in Fig. 3.

`Compmodel` (COMPile a MODEL) produces a compilable C program that solves a particular model more rapidly than the standard HEQS `slv` command interpretively solves a general model. It too is a meta command.

VI. ERROR REPORTING IN A MODEL ENVIRONMENT

Programs written in compiled procedural languages may contain either compile-time or run-time errors. Models in HEQS may analogously contain compile-time errors of logic or syntax detectable during analysis, or run-time errors corresponding to numerical problems found by the solver as it tries to proceed through the simultaneous subsets of equations. In either case, our main design criterion has been to ensure that HEQS reports solubility problems in terms of the original user equations that seem to cause the error, despite any analysis and transformation the model may have undergone in the system's graph-theory algorithms. HEQS makes no attempt to fix models by using its own "intelligence" in place of the modeler's. We believe that detecting and reporting a large class of errors to users in the language of their original model will adequately enable them to fix things themselves. With this in mind, we list below some of the model errors found by the system and explain where they are reported.

Syntax errors in the model's equations are detected by `cheq` while parsing and scalarizing the equations. Since HEQS scalarizes tensor equations and has full macro capabilities, `cheq` has an option that lets users see the equivalent scalar equations that constitute their model after scalarization and macro expansion. This feature is useful for advanced users making liberal use of `cheq`'s compact notational power. `Cheq` reports all syntax errors in a model; it only builds a dependency graph for error-free models.

Semantic errors (errors that make a model "meaningless") are detected in `cheq`, `canislv`, and `slv`. Over-determined models—models where a variable is used as the left-hand side of more than one equation—are caught and reported in `cheq`. Under-determined models, in which some equations or data values necessary for complete solution of the model are missing, are detected in `canislv` and `slv`. `Canislv` has been found to be especially useful for detecting typographical errors, since mistyped variable names often lead to under-determined models.

Unknown functions are detected in `slv`. `Cheq` assumes all functions in a model are available in the function library in `slv`, so that such errors are caught during numerical solution after model analysis. Functions called with the wrong number of arguments are detected here too.

Mathematical errors involving underflow, overflow, functions called with illegitimate argument ranges, etc. are detected in `slv`. Its error messages report the offending scalar equation and function or operation, the original model equation from which it stems by scalarization, the data values used in the model equation, and the equations in the simultaneous subset to which they belong.

Nonlinear equations are detected by `slv`, which first eliminates any nonlinearities that can be removed by linear solution and substitution. To avoid the problem of finding the "wrong," that is, unwanted solution to a nonlinear equation with multiple solutions, users are then asked to provide an estimate of the answer to be used as a starting point for iterative solution. Problems involving nonconvergence or the absence of a real solution are reported here too.

Typographical errors are found only when they lead to some insolubility like those listed above. Here, HEQS' detailed output in terms of the original user equation is usually adequate for localizing the error.

VII. WHAT WE LEARNED

HEQS has shown that it is feasible to give users an easy way of writing, maintaining, and solving large sets of algebraic equations. Its language, with its subscripted variables and macro facilities, allows powerful, user-definable, easily maintainable and yet *compact* description of algebraic relationships. Its error-checking facilities help even naive users correct their models. Its solver provides nonprocedural solutions to many algebraic problems of interest.

The decision to implement HEQS as an interpreter rather than a compiler seems successful, since our users work in a field where models change quickly, and easy model building and consequent error correction is crucial. The interactive debugging provided by an interpreter is particularly important; it is hard to imagine users with a nonprogramming background having the patience to debug models that are solved by first translating them into a procedural language, then compiling them, and finally running the resultant program.

The graph-theoretic internal representation of models has provided a natural paradigm for equation solving by humans. It offers a standard mathematical way of representing all the analytic steps of modeling. We have found that even mathematically naive users seem to have an intuitive grasp of dependency trees. This suggests that a two-dimensional display of the dependency trees would provide a useful interface to the modeling commands.

The separation of the parser/scalarizer from the solver in the implementation has allowed flexible development. The parser and the solver were modified or rewritten independently at several times during the upgrading of HEQS. In each case, integration into the system as a whole was simple. The possibility of building special-purpose front ends for particular areas of modeling without affecting the overall system is attractive.

The disadvantage of this separation is subtle, and was slow to emerge. Since the solver sees only scalarized equations, it has no understanding of tensors, which are parsed only by the scalarizer. Thus, user-defined precompiled subroutines linked to the solver can only accept scalar arguments. It is not possible for users to define their own subroutines that take tensor arguments, since this would require dynamically modifying the precompiled parser, an unrealistic goal. In practice, however, the addition of full macro capabilities to the HEQS parser allowed users to define macros that take tensor arguments, which eliminated much of the problem.

A final important lesson in implementation was the benefit of HEQS' modular design. HEQS programs were small, each performing one analytic task. They communicated via intermediate text files that held the relevant data structures. The weak coupling implicit in this design allowed different programmers to work on different modules with little conflict and much independence, and inspection of the intermediate text files aided debugging. Now that the system has stabilized, it may be desirable to join the separate programs into one large one, dispose of the need for intermediate files, and thus obtain a more efficient and tested integrated version.

VIII. ACKNOWLEDGMENTS

C. J. Van Wyk furnished us with algorithms and programs for the solution of algebraic equations by substitution. T.-W. Pao devised a way to allow users to link their own subroutines to HEQS' solver. Both of them provided useful suggestions and conversion. C. Childs and R. Rodriguez gave helpful support and advice.

REFERENCES

1. E. Derman and Z. M. Ma, unpublished work.
2. E. Derman and C. J. Van Wyk, unpublished work.
3. Christopher J. Van Wyk, *A Language for Typesetting Graphics*, Ph.D. dissertation, Stanford University, 1980.
4. D. W. Marquardt, "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," J. Soc. Ind. Appl. Math., *11*, No. 2 (June 1963), pp. 431–41.
5. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, New York: Addison-Wesley, 1974.

APPENDIX

HEQS Models As Graphs

This Appendix describes the internal graphical representation of a user model, and explains how it is used to obtain the order of computations for model solution, to perform impact analyses, to eliminate extraneous computations from a what-if analysis, and to reorder model computations to perform goal-seeking. In particular, the method used to reorder the computations for goal-seeking is described in some detail. We first give an intuitive description of the HEQS graphical representation of models, and we later present a rigorous definition.

A HEQS model is a sequence of equations, each of whose left-hand sides is a variable to be computed. Furthermore, a model is considered complete only if every model variable is constrained to appear once

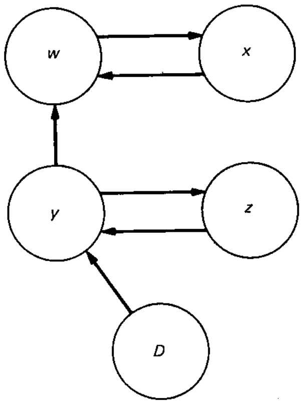

Fig. 4—Graphical representation of the model in eq. (3).

and only once on the left-hand side of a model equation. Thus, the list of equations

$$w = 2{*}x + y - 10, \quad (3a)$$

$$x = w{*}{*}0.5, \quad (3b)$$

$$y = z{*}D - 3, \quad (3c)$$

$$z = y - 5, \text{ and} \quad (3d)$$

$$D = 20 \quad (3e)$$

(where ** denotes exponentiation) forms a legitimate HEQS model. Internally, HEQS represents each model as a directed graph whose nodes correspond to model variables and whose arcs indicate computational dependencies. The graphical representation of the model given above is shown in Fig. 4.

It is important to understand that each node in the graph "points" to the model equation that is used to derive a value for the variable represented by the node. Given the constraints on a HEQS model, that is, each variable occurs exactly once on the left-hand side of an equation, this node to equation matching is implicit within the model equations themselves. Later, in conjunction with the discussion of goal-seeking, we describe a method for performing this matching process when no constraints are placed on the form of equations, so that both sides of an equation may be arbitrary expressions.

The value of a variable corresponding to a node can only be determined when all the right-hand-side variables in its equation are known. For the graph, this implies that the computational ordering of a model (the order in which it can be solved) must satisfy the constraint

The value of a model variable is determined only after all variables connected to it by in-arcs have been evaluated.

This statement is, of course, not quite complete, as can be seen in the model of eq. (3). Here, the variable w cannot be evaluated until the variable x has been solved and vice versa. In other words, the variables w and x form a set that must be computed simultaneously, that is, produce a simultaneous set of equations. In fact, the nodes corresponding to w and x form a strong component of the graph representing the model (a strong component of a directed graph is a set of nodes such that there is a path from any node in the set to any other). It should be clear that any simultaneous set of equations in a model must correspond to a strong component in the graph representing that model. Thus, to obtain the computational ordering of a model, we first collapse the graph of a model into strong components. For our example model, we obtain the directed graph of strong components depicted in Fig. 5. The computational ordering of a model can then be fully described by the statement

The value of the variables in an individual strong component (simultaneous equation set) cannot be determined until all strong components connected to it by in-arcs have been evaluated.

Thus, we must select the set of strong components in an order satisfying this statement and solve them in that order. (A depth-first search of the collapsed graph provides one suitable order.)

Using this graphical representation to determine the computational

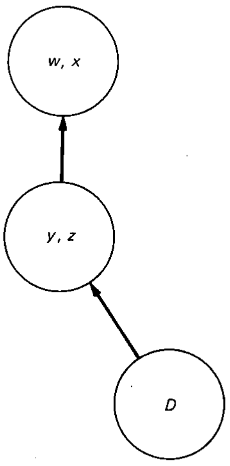

Fig. 5—Collapsed graph corresponding to the model in eq. (3).

ordering for obtaining a model solution has several advantages over other methods.

1. Some seemingly nonlinear computations can be easily solved through substitution (i.e., without using nonlinear solution methods). In the example, the equations $y = z*D - 3$ and $z = y - 5$ form a linear (not a nonlinear) simultaneous set since the variable D is evaluated prior to the evaluation of y and z (i.e., the strong component containing D is solved prior to the strong component containing y and z).

2. This method isolates computational errors (division by zero, overflows, underflows, etc.) to the equations in which they occur. For example, the equation $x = y/(a + b)$ could result in an error if $a = -b$, and not otherwise. Detecting these types of errors helps localize logical errors within a model.

3. The time required to solve a linear simultaneous set of equations with a linear substitution method (the method used in HEQS) varies in proportion to the number of atomic arithmetic operations (addition, subtraction, etc.) in the set of equations. This is a clear improvement over treating a model as a large matrix in which the number of operations increases with the square of the number of model variables. (Of course, matrix methods have many other disadvantages. For example, they are implicitly linear.)

4. When a truly nonlinear simultaneous set of equations does occur (as with w and x in the above example), the smaller the set, the faster the convergence of any iterative solution method. Furthermore, solving two nonlinear simultaneous sets independently will, in general, be faster than solving the two sets as a single, larger simultaneous set. (We use Marquardt's iterative method of solution for nonlinear simultaneous sets[5]).

5. The extraction and ordering of strong components is fast since it is linear in the number of nodes in the graph.

The HEQS graphical paradigm of a model is useful in several other respects. For example, a common question to ask about a model is, "If I change the value of this variable, what other variables in the model are changed (impacted)?" This type of question is referred to as impact analysis. Our graphical representation provides a straightforward and fast method for performing such an analysis. The inverse of an impact analysis (what variables produce changes in a particular variable) is also quite easy and fast. Furthermore, both these capabilities allow the elimination of extraneous computations in a so-called what-if analysis. An example of the type of question posed in such an analysis would be

What is the effect on corporate profits if the price of product A rises by 10 percent?

The variables which must be recomputed to solve this what-if question are exactly those that lie on a path from the "price" variable to the "profit" variable. All other computations may be excluded.

Another common capability required in a modeling system such as HEQS is goal-seeking, which is, in an intuitive sense, the inverse of the what-if capability. For example, a typical question posed would be

What must the price of product A bé to increase corporate profits by 10 percent?

Given our model in eq. (3), this is similar to asking, "What value must the variable D be given so that the constraint $w = 15$ is satisified?" That is to say, we wish to find a solution to the set of equations

$$w = 15, \tag{4a}$$

$$w = 2*x + y - 10, \tag{4b}$$

$$x = w**0.5, \tag{4c}$$

$$y = z*D - 3, \text{ and} \tag{4d}$$

$$z = y - 5. \tag{4e}$$

(Note that the equation $D = 20$ has been removed from the model, and the equation $w = 15$ has been added. These model perturbations are characteristic of the goal-seeking problem.) In this goal-seeking analysis, we refer to the w as the goal-seeking variable, and to D as the target variable. Our problem, then, is to determine an order of computation for the new goal-seeking model. Ideally, we would like to obtain this computational ordering by a sequence of simple transformations applied to the graph representing the original model. We present such a method whose most complex step is discovering a path from one node to another.

The previous method for ordering computations (forming a graphical representation using the implicit matching of nodes to equations) in a HEQS model fails for this goal-seeking problem for an obvious reason: the new model violates the constraint that each model variable occur exactly once on the left-hand side of an equation (w is on the left-hand side of two equations while D is not on the left of any equation). What we require is an algorithm for obtaining the matching between the model variables represented by nodes in the graph, and the new set of model equations. Before giving a precise description of our goal-seeking algorithm as applied to the general case, we first illustrate the steps involved by reordering the computations in the goal-seeking problem given here. Later, we give a rigorous description of this process.

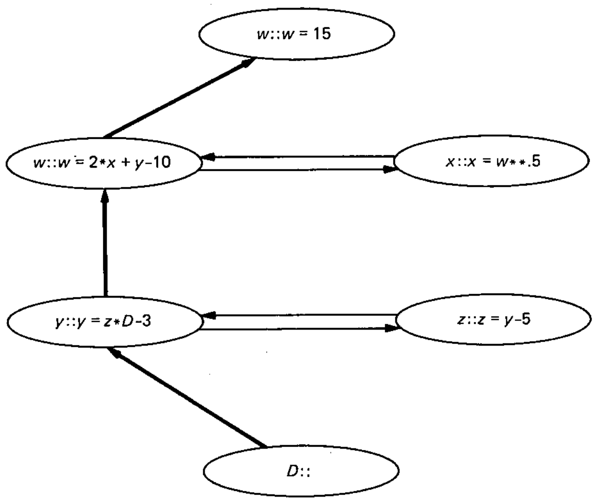

Fig. 6—Graph for goal-seeking in model of eq. (4).

To determine a computational order for goal-seeking using graphs, the first step is to remove the equation corresponding to the target variable D in the graph of Fig. 3, and then add a new node that describes the new goal-seeking constraint. The resulting graph is depicted in Fig. 6. Note that in this figure, the equations associated with each node have been placed in the graph nodes, separated from the variable corresponding to the node by two colons. Also note that only the equation $D = 20$ has been removed from the graph, not the node containing the variable D.

We now proceed to find a path from the node corresponding to the target variable D to the newly added node that contains the goal-seeking constraint equation. The arcs in one such path are shown in bold in Fig. 6. Having found such a path, we then transform the graph by moving the equation in each node in the path from that node to its predecessor node in the path (the result of this operation is depicted in Fig. 7). This places in each variable node an equation that can be used to derive the value of that variable in the goal-seeking problem. We also retain a one to one correspondence between variables and equations. After having moved the equations "backwards" through the path, we redirect each arc that points to a node in the path so that it is directed to the predecessor of that node in the path instead (the graph arcs affected are marked in bold in Fig. 7). This makes the dependencies between the nodes and their new associated equations consistent. The result of this arc redirection process is shown in Fig. 8.

Finally, we remove the new graph node and all its connected arcs. The resulting graph has all the properties we require for ordering

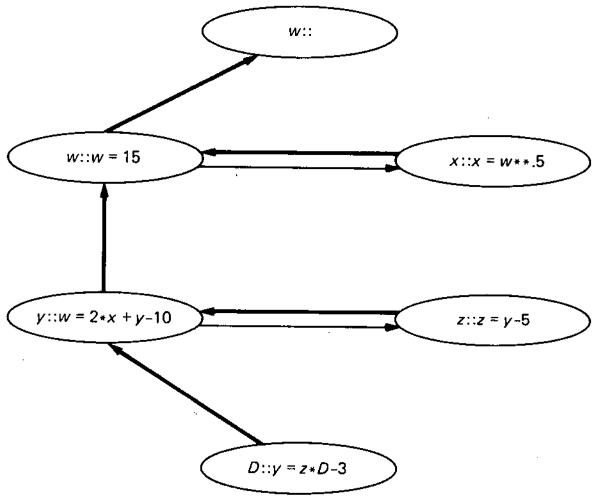

Fig. 7—Transformed graph for goal-seeking in model of eq. (4).

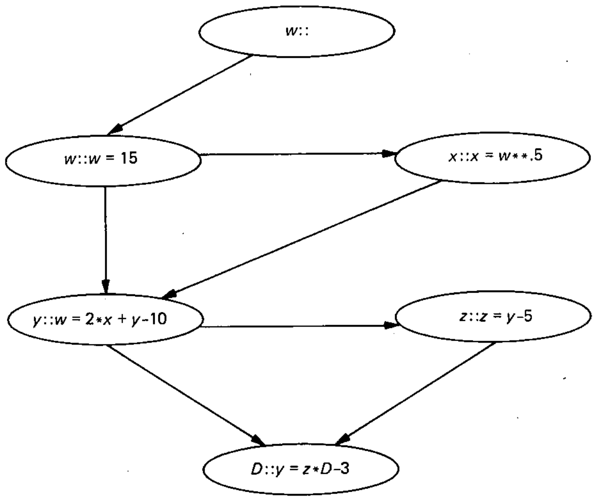

Fig. 8—Final graph suitable for determining the computational ordering of model in eq. (4).

computations in the goal-seeking problem; its strong component collapse and depth-first search yields an appropriate computational order.

We now give a more rigorous definition of model graphs and of the goal-seeking algorithm.

We can conceive of a model **M** as being an ordered triple (**V**, **E**, ∇), where **V** is the set of model variables, **E** is the set of model equations, and ∇:**E** $\rightarrow 2^{\mathbf{V}}$ is a function (from the set of equations to the power set of **V**) defined so that for all $e \in \mathbf{E}$, $\nabla(e)$ is the set of variables mentioned in the equation e. Given a model $\mathbf{M} = (\mathbf{V}, \mathbf{E}, \nabla)$, we can develop a graph representing $\mathbf{M}$ as $\mathbf{G} = (\mathbf{N}, \mathbf{A})$, an ordered pair of a node set and an arc set. The individual nodes in $\mathbf{N}$ are ordered pairs (v, e) taken from the product $\mathbf{V} \times \mathbf{E}$ that satisfy the requirement $v \in \nabla(e)$. Given two nodes $n, m \in \mathbf{N}$, where $n = (v_n, e_n)$ and $m = (v_m, e_m)$, then $(n, m) \in \mathbf{A}$ if $v_m \in \nabla(e_n)$. (We understand $(n, m) \in \mathbf{A}$ to mean that the node n "depends" on the node m. Such an arc is drawn from the node m to the node n.) Finally, we say that a graph $\mathbf{G} = (\mathbf{N}, \mathbf{A})$ is a *solvable* representation of a model $\mathbf{M} = (\mathbf{V}, \mathbf{E}, \nabla)$ if each variable in $\mathbf{V}$ and each equation in $\mathbf{E}$ occur exactly once in a node in $\mathbf{N}$. Such solvable graphical representations are amenable to HEQS algorithms.

We now give a detailed description of the goal-seeking algorithm used in HEQS. Assuming that we have a solvable graphical representation $\mathbf{G} = (\mathbf{N}, \mathbf{A})$ of a model $\mathbf{M} = (\mathbf{V}, \mathbf{E}, \nabla)$, the goal-seeking problem is given as

> *Vary the model variable* **TV** *so that the variable* **GV** *satisfies the constraint* **GE**,

where **GE** is an equation in the normal HEQS form and the variable **GV** is the left-hand side of this equation. Clearly, the variable **TV** is the target variable, the variable **GV** is the goal-seeking variable, and the equation **GE** is the goal-seeking constraint equation. If we let **TE** be the model equation that is paired with **TV** in the set of graph nodes $\mathbf{N}$, what we require to solve this problem is a solvable graphical representation of a new model $\mathbf{M}' = (\mathbf{V}, \mathbf{E}', \nabla')$, where $\mathbf{E}' \equiv (\mathbf{E} - \{\mathbf{TE}\}) \cup \{\mathbf{GE}\}$, and ∇' is extended from ∇ in the natural manner to include **GE** in its domain. The goal-seeking algorithm is then a sequence of transformations applied to the graph $\mathbf{G}$.

1. Form a new graph $\mathbf{H} = (\mathbf{M}, \mathbf{B})$ so that $\mathbf{M} \equiv \mathbf{N} \cup \{(\mathbf{GV}, \mathbf{GE})\}$, and $\mathbf{B} \equiv \mathbf{A} \cup \Delta\mathbf{A}$, where we define $\Delta\mathbf{A}$ as a set of arcs so that $((\mathbf{GV}, \mathbf{GE}),(v, e)) \in \Delta\mathbf{A}$ for all $v \in \nabla'(\mathbf{GE})$. In other words, add to the graph $\mathbf{G}$ the node $(\mathbf{GV}, \mathbf{GE})$ and all necessary in-arcs pointing to this new node. Note that $\mathbf{H}$ is not a solvable model representation since **GV** occurs in two nodes in $\mathbf{B}$.

2. Find a path $\mathbf{P} = (p_1, p_2, \ldots, p_{|\mathbf{P}|})$ in $\mathbf{H}$ so that $p_1 = (\mathbf{GV}, \mathbf{GE})$, $p_{|\mathbf{P}|} = (\mathbf{TV}, \mathbf{TE})$, and $(p_i, p_{i+1}) \in \mathbf{B}$ for all $0 < i < |\mathbf{P}|$. Assume $p_i \equiv (v_i, e_i)$.

3. Form a graph $\mathbf{I} = (\mathbf{L}, \mathbf{B})$ from $\mathbf{H}$, where

$$\mathbf{L} \equiv (\mathbf{M} - \bigcup_{p \in \mathbf{P}} \{p\}) \cup \{(\mathbf{GV}, \mathbf{GE}), (v_2, e_1), \ldots, (v_{|\mathbf{P}|}, e_{|\mathbf{P}|-1})\}.$$

Note that we have shifted the appropriate equations through the path nodes.

4. Form another graph $\mathbf{J} = (\mathbf{L}, \mathbf{C})$ from $\mathbf{I}$, where $\mathbf{C} \equiv (\mathbf{B} - \Delta\mathbf{B}) \cup \Delta\mathbf{B}'$, given that $\Delta\mathbf{B} \equiv \{(n, m) \in \mathbf{B}\colon n \in \mathbf{P}\}$, and $\Delta\mathbf{B}' \equiv \{(n, m)\colon n =$

$p_i \in P$ and $(p_{i-1}, m) \in \Delta B\}$. Here we have redirected all arcs pointing to nodes in the path **P**. We refer to steps 3 and 4 collectively as *path inversion.*

5. Finally, we remove the excess node (**GV**, **GE**) and all out-arcs pointing away from this node from the graph **J** to obtain the necessary goal-seeking graph $\mathbf{G}' = (\mathbf{N}', \mathbf{A}')$. Note that all in-arcs to the node (**GV**, **GE**) in the graph **J** were redirected in step 3. It should be clear that the graph $\mathbf{G}'$ resulting from this sequence of transformations is a solvable representation of the goal-seeking model $\mathbf{M}'$.

This algorithm will only fail if there is no path as described in step 2. This is in accord with our expectations. Namely, this indicates that the goal-seeking variable is independent of the target variable in the goal-seeking model.

Having developed an algorithm for ordering the computations in a goal-seeking problem, we now note some additional properties of this algorithm. First, we can use the same algorithm to solve simultaneous goal-seeking problems that have the form

> *Find values for the set of model variables* $\mathbf{TV} = \{\mathbf{TV}_1, \mathbf{TV}_2, \ldots, \mathbf{TV}_n\}$ *so that the model variables* $\mathbf{GV} = \{\mathbf{GV}_1, \mathbf{GV}_2, \ldots, \mathbf{GV}_n\}$ *satisfy the constraint equations* $\mathbf{GE} = \{\mathbf{GE}_1, \mathbf{GE}_2, \ldots, \mathbf{GE}_n\}$.

We do this by applying our algorithm repeatedly. In other words, add new nodes for all the variables in **GE**. Then, find a path from $\mathbf{TV}_1$ to one of these new nodes. Invert this path and remove the goal-seeking node on the end of this path. Then, find a path from $\mathbf{TV}_2$ to one of the remaining new nodes, invert it, and remove this goal-seeking node. Repeat this process until both sets of variables have been exhausted. Clearly, this process will fail in several circumstances: if one or more of the goal-seeking variables are independent of all of the target variables, for example. Furthermore, the algorithm will fail for models such as

$$y = 2*x + 5, \tag{5a}$$

$$z = 3*x - 2, \tag{5b}$$

$$x = a - b, \tag{5c}$$

$$a = 2, \text{ and} \tag{5d}$$

$$b = 3, \tag{5e}$$

and the multiple goal-seeking problem is stated as

> *Vary the variables a and b so that the constraints $y = 7$ and $z = 10$ are satisfied.*

The algorithm fails since there are no two vertex disjoint paths from variables a and b to variables y and z. In other words, y and z cannot be given values independently only by varying a and b.

Another interesting point about this algorithm is that it can be used to create a solvable graphical representation of any model (assuming that the model has a solution), even when the equations of the model are not in the standard HEQS form (every variable is the left-hand side of exactly one equation). For example, the model

$$w + 10 = 2*x + y, \tag{6a}$$

$$0 = x - w**0.5, \tag{6b}$$

$$z*D = y + 3, \tag{6c}$$

$$5 = y - z, \text{ and} \tag{6d}$$

$$D = 20 \tag{6e}$$

is closely related to our first example model. In fact, it is exactly the same model, with the same solution, but the equations do not conform to the standard HEQS form. To get a solvable graph for this model, $\mathbf{M} = (\mathbf{V}, \mathbf{E}, \nabla)$, we use a variation of the goal-seeking algorithm. In the variation, we will admit nodes in the graph that possess a model variable, but do not have an associated equation. We call such nodes *improper* (*proper* nodes are those which have both a variable and an equation). All improper nodes will be written as (v, ω), where we use the character ω to indicate the presence of a null equation (or null pointer in computer science terms).

The algorithm proceeds in steps. First, select an equation $e \in \mathbf{E}$, and pick an arbitrary element $v \in \nabla(e)$ to be the temporary left-hand side of this equation. Generate the graph node (v, e). For all other variables in $\nabla(e)$, we create improper nodes for these variables, and make arcs pointing from these nodes to the single proper node (v, e). At each succeeding step, we select an unexamined equation from the model. For each such equation e, one of three cases will apply:

1. None of the variables in $\nabla(e)$ occur in the graph in any node (neither proper nor improper).
2. At least one of the variables in $\nabla(e)$ occurs in an improper node in the graph. Assume that (v, ω) is such a node.
3. All of the variables in $\nabla(e)$ occur in proper nodes.

In the first case, we proceed as in the first step of this process. Select one of the variables in $\nabla(e)$ as the temporary left-hand side of the equation e, generate a proper node for this variable, make improper nodes for the rest of the variables in $\nabla(e)$, and add the appropriate in-arcs to the proper node.

In the second case, place e in the improper node (v, ω) to form a proper node (use e to replace the null equation). Then attach the necessary in-arcs to this node (adding improper nodes for those variables in $\nabla(e)$ that have not yet been placed in the graph).

In the last case, we temporarily add a node (v, e) for some arbitrary $v \in \nabla(e)$ and any necessary in-arcs to this node. We then find a path from some improper node to this temporary node, invert this path, and finally remove the temporary node and any associated out-arcs.

At each step in the process described above, each variable that has been added to the graph occurs in exactly one node. When all equations have been examined, the process ends, and we are left with a solvable graphical representation of the model.

Of course, if the model equations are underdetermined, some improper nodes will exist in the final graph, and the model cannot be solved. If, at any point in the algorithm, case 3, occurs, and no path can be developed, the equations are over-determined, and the model again cannot be solved.

At some future point, the form restrictions placed on HEQS equations will be relaxed so that arbitrary expressions may occur on both the left- and right-hand side of any equation, and this algorithm will be used to derive a solvable representation of the model.

AUTHORS

Emanuel Derman, B.Sc. (Applied Mathematics), University of Cape Town, 1965; M.S. (Physics), Columbia University, 1968; Ph.D. (Theoretical Particle Physics), Columbia University, 1973; 1973–1979: Postdoctoral research in structure of elementary particles at University of Pennsylvania, Oxford University (England) and The Rockefeller University, 1973–1979; Assistant Professor, Dept. of Physics, University of Colorado, 1979–1980; AT&T Bell Laboratories, 1980–1985; Goldman Sachs & Co. Mr. Derman's primary fields of interest are languages and artificial intelligence.

Edward G. Sheppard, B.S. (Mathematics), Emory University, 1980; M.S. (Comp. Sci.), Emory University, 1980. 1980–1983: AT&T Bell Laboratories, 1980–1983; Bell Communications Research, Inc., 1984–1985; Asymetrix Corp., 1985—. Mr. Sheppard's primary field of interest at AT&T Bell Laboratories was application software design and implementation.