THE STRUCTURE OF SYNTHETIC CALCULUS

A Programming Paradigm of Mathematical Design

by

Barnet Krinsky

Hughes Aircraft Corporation

and

Joseph Thames

The Aerospace Corporation

PROCEEDINGS

OF THE

INTERNATIONAL WORKSHOP

ON

HIGH-LEVEL COMPUTER

ARCHITECTURE

MAY 21 - 25, 1984

LOS ANGELES, CALIFORNIA

SPONSORED BY

THE DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF MARYLAND

COLLEGE PARK, MARYLAND

20742

ABSTRACT

Synthetic calculus is a paradigm of mathematical modeling based upon synthetic differentiation, an expanded arithmetic producing exact partial derivatives -- a motivation for special hardware architecture. Neither finite difference approximations nor symbolic derivation approaches provide the combined accuracy, speed and memory efficiency of exact-dynamic evaluation achieved by synthetic differentiation. But a more subtle advantage has a direct impact on program structuring. This is a mathematical decoupling effect that simplifies modeling and mathematical design. The result, illustrated by examples, is orthogonal cleavage between problem models and numerical algorithms, permitting interchangability of solution tools without reprogramming. It effectively insulates engineering modeling from numerical analysis, promoting a convenient division of labor and an experimental approach to mathematical software design.

THE STRUCTURE OF SYNTHETIC CALCULUS

A Programming Paradigm of Mathematical Design

by

Barnet Krinsky and Joseph Thames

Hughes Aircraft Corporation The Aerospace Corporation

ABSTRACT

Synthetic calculus is a paradigm of mathematical modeling based upon synthetic differentiation, an expanded arithmetic producing exact partial derivatives -- a motivation for special hardware architecture. Neither finite difference approximations nor symbolic derivation approaches provide the combined accuracy, speed and memory efficiency of exact-dynamic evaluation achieved by synthetic differentiation. But a more subtle advantage has a direct impact on program structuring. This is a mathematical decoupling effect that simplifies modeling and mathematical design. The result, illustrated by examples, is orthogonal cleavage between problem models and numerical algorithms, permitting interchangability of solution tools without reprogramming. It effectively insulates engineering modeling from numerical analysis, promoting a convenient division of labor and an experimental approach to mathematical software design.

INTRODUCTION

Synthetic calculus is a metaphoric computer image or "metacomputer". To the computer architect this represents a high-order instruction-set architecture (ISA) and an interpretive engine. To the problem solver it is a problem-describing language and a mathematical engine capable of synthesizing numerical experiments and finding particular solutions to understandable problem statements. What follows is written for the computer architect on behalf of the problem solver. It is a follow-up to Reference 1 which explains the foundation of the mathematical architecture, an "extended value" arithmetic of differentiation.

The purpose of this discussion is to demonstrate the facility of mathematical modeling that can be built upon the foundation of exact differential arithmetic. The modeling paradigm operates above the knowledge level of numerical analysis, typical of scientific programming, at a higher level of problem conceptualization from which the details of mathematical solution are hidden subassemblies. Since the partial derivatives produced by synthetic differentiation are explicitly involved only in the submerged solution processes, only the merest hint of their mechanics is apparent at the programming level.

At the programming level, the modeling paradigm does not involve the methodology of classical mathematical analysis, but subsumes this methodology in the medium of solution in the same fashion that analog-computer programming subsumes electrical subassemblies to model and integrate differential equations. The necessity of understanding the mechanics of the solution process is removed. Instead, one perceives a metaphoric analogy of the mechanism, and uses this analogy as a higher-order programming paradigm, unencumbered by the details of the mechanism. This paradigm has been employed by engineers and scientists of many disciplines to solve mathematical problems which were often too complicated to be programmed with conventional programming tools, or required more programming effort than they could afford to spend.

The architecture focus of this work is the need for general purpose hardware design that provides a high-performance foundation of synthetic differentiation. As an arithmetic foundation of higher mathematics this process is the key to greater abstraction in problem-oriented programming. It produces accurate and efficient mathematical reckoning that is more automatic and reliable because of its freedom from domain-sensitive approximations requiring empirical tuning. It is this mathematical univerality or local exactness of functional reproduction that permits the uncoupling (syntactic cleavage) of models and methods at higher levels of expression.

Synthetic Differentiation - The foundation itself is a generalization of computer arithmetic into a calculus hierarchy in which ordinary arithmetic is regarded as order-zero arithmetic. Order-one arithmetic is the synthetic (interpretive) production of gradient vectors with respect to a specified basis of independent variables. Order-two arithmetic is the production of Hessian (curvature) matrices, and so on. Thus higher-order arithmetic may be regarded as a numerical sampling of the local shape parameters of a function surface at a position defined by the values of the basis vector.

The mechanism of synthetic differentiation, and a summary of its history of evolution is treated in Reference 1. An important factor in the simplification resulting from synthetic calculus is that this arithmetic foundation is invisible to the user.

A Double Interpreted Foundation - Synthetic calculus as a programming paradigm evolved from systems engineering application software developed in the Apollo program. It fulfilled a need for the rapid synthesis of optimization programs to address wide varieties of system models in short time frames consistent with engineering deadlines. In the mid-seventies it was introduced commercially on Control Data, Univac, and IBM mainframes as the application language PROSE (PROblem Solution Engineering - Refs. 1,2).

As problem-solving tools, the PROSE metacomputers were very successful in demonstrating the promise of synthetic calculus. But, with the essential use of assembly language for high-speed execution and the close dependence upon evolving operating systems, their installations proved too costly to maintain. Further work in pure software was discouraged because the most that could be achieved in performance was ultimately limited by the speed of the assembly languages of the mainframes, which are now insulated from the hardware by microprogramming. Microprogrammed "virtual machines" make synthetic calculus subject to double interpretation resulting in a net penalty of two to three orders of magnitude in execution speed.

Since the assembly languages used to interpret synthetic differentiation are now interpreted by microsoftware, this programming paradigm remains in a position far removed from its potential performance levels, even with conventional off-the-shelf electronics. With the promise of VLSI electronics, the gap to be bridged by transforming proven software architectures into hardware is truly revolutionary. The net benefits of low cost, yet highly sophisticated mathematical system development for industrial and military control applications is no longer a question of feasibility. It is now merely a question of compatible computer architecture.

THE CALCULUS OPERATION HIERARCHY

The procedural structure of synthetic calculus programs is not arbitrary, as it is in conventional language programs, but is mathematically prescribed by the calculus operations. Each operation is a control interface between two software procedure levels, an operator level and an operand level (See Figure 1). But unlike a procedure call, the operation does not directly transfer control to the lower level. Instead it first establishes a temporary data structure representing a mathematical control frame, containing: (a) the sampled basis

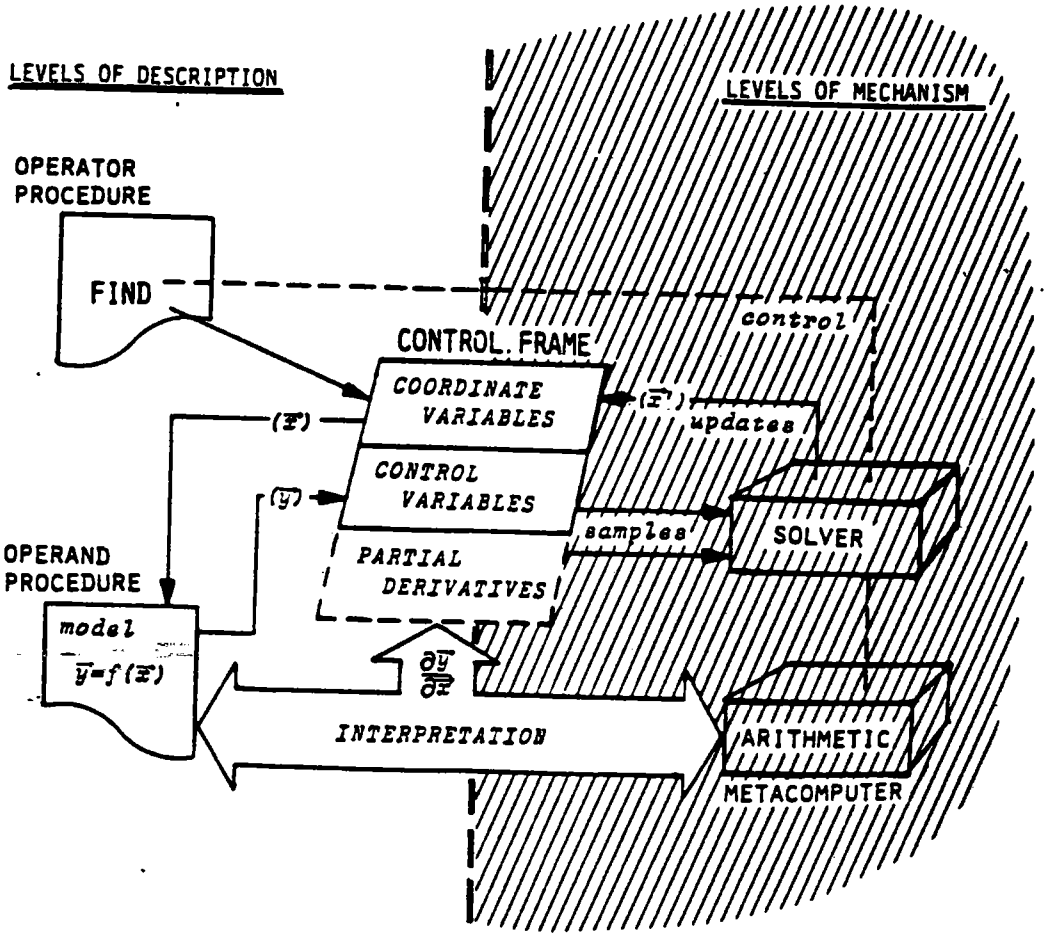


Figure 1. Control Frame Interfaces

vector of a parametric coordinate system and (b) arrays to contain a set of function point values in the coordinate space for use as control constraints. Then it transfers control to a background solution process (solver) which controls the execution of the operand procedure and manipulates the control-frame basis vector.

The control frame is a temporary four-way interface between the two procedure levels, the arithmetic metacomputer, and the background solution process. It is disconnected when the operation is complete. Its structure and content govern the mathematical domain of the operand procedure, prescribing the local basis of the differential arithmetic and the computed control variables measuring its parametric response. The operand procedure, containing the visible (model) formulas of an application problem, is differentially interpreted with respect to the sampled basis vector. The visible formulas perform a predictive computation that the background metacomputer uses to generate values of the control functions and their partial derivatives as numerical samples for induction. The operand procedure does not alter the basis vector sample.

The background solution process employs the sampled function values and their background-generated partial-derivative values to steer the induction toward an apparent solution. This process alters the control frame basis vector in value and sometimes in structure to guide the prediction computations in successive passes of the operand procedure.

Unified Solution Processes

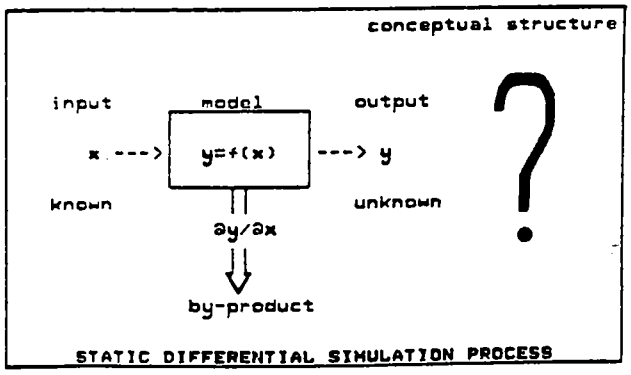
Systems engineering has evolved high-order program structuring concepts which distinguish modeling as orthogonal to numerical analysis and fold these separate technologies into a complementary, modular relationship. The basic concept is that a model is an operand of its solution process, but is separate and distinct. Models are also regarded as mathematical system functions represented by simultaneous equations, i.e., multidimensional formulations of parallel phenomena. Thus the interfaces between a model and its solution process are vectors representing variables of simultaneous equations, e.g. basis and constraint vectors for simultaneous algebraic equations, rate and state vectors for simultaneous differential equations.

A calculus solution process is represented by a tool procedure called a solver. A solver is a numerical procedure (algorithm) which controls the behavior of the the modeled system by varying its inputs. Solvers are classified as representatives of three generic solution processes, each having a characteristic type of control frame interface and mathematical outcome. They are the processes of simulation, correlation, and optimization. Simulation is the deductive (logically forward) process of generating output from a model from a given set of inputs. Correlation is the inductive (logically inverse) process of finding specific values of inputs that cause the outputs to match prespecified values. Optimization is the inductive process of finding specific values of inputs that maximize or minimize one or more outputs. All three processes may have constraints applying to the output which limit the range of variation of the unknowns.

Solution processes are further subclassified into static and dynamic processes, according to whether the solution is a point or a function. Static simulation constitutes the evaluation of explicit algebraic equations and explicit calculus operations such as derivatives and definite integrals. Dynamic simulation constitutes the solution of initial value problems of differential equations or approximations such as difference equations. Static correlation constitutes the point solution of systems of implicit equations, derivatives, and integrals. It encompasses linear and nonlinear systems, determined, overdetermined (least squares) and underdetermined systems. Dynamic correlation encompasses the area of function matching, including the techniques of dynamic estimation and process (system) identification (Kalman filtering, etc.). Static optimization encompasses the area of linear and nonlinear programming. Dynamic optimization encompasses the area of optimal control and dynamic programming.

The organization of synthetic calculus into the three method classes (solution processes) is best illustrated by a few simple examples of calculus usage. The following examples are used merely to illustrate the syntactic structure of the calculus operations.

Simulation Operations - The simplest of all calculus operations is static differential simulation. This amounts to the execution of a model as a subroutine utilizing differential arithmetic to produce partial derivative values as a by-product of each formula calculation. The inputs are known and the outputs are the unknowns computed from sequences of explicit algebraic equations.



STATIC DIFFERENTIAL SIMULATION PROCESS

Programatically, one merely uses the statement

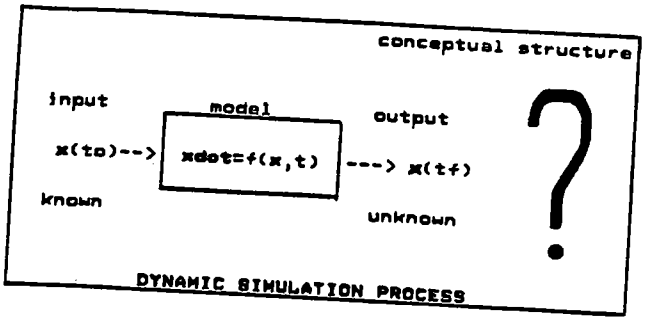
EXECUTE model WITH GRADIENTS ON x

to perform static simulation of a model, differentiating all of its computed variables with respect to the basis vector x. In this case, x is the control frame which specifies the mode of the arithmetic in the operand level of the model

procedure. This type of operation is often used to perform sensitivity analysis of algebraic models.

In dynamic simulation, the equations have the form of ordinary differential equations with initial conditions. Since this is a more complicated process, a solver is involved. Thus a "black-box" must be interfaced with the model to control its execution.

In this case, the model has the following conceptual structure (In all conceptual structures, the variables in bold type denote arrays):



To activate dynamic simulation, two statements are involved:

(1) INITIATE solver FOR model
EQUATIONS xdot/x OF t
STEP dt TO tf

(2) INTLGRATE model

The first statement initializes the numerical integration solver and prescribes the control frame. It identifies the operand (model), the input and output (rate/state) of integration, the independent variable (t), the nominal step size variable (dt) and the upper limit of integration (tf).

The second statement commands the solver to integrate the equations up to the final point. In case of higher order equations, the same two statements are involved as illustrated in the following example:

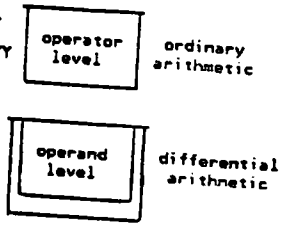
Simulation Example

Integrate from t=0 to t=1
dx/dt = 3x+y
d2y/dt2 = sin(x) + exp(yt)
where x(0)=0, y(0)=1, dy/dt(0)=0.

Program

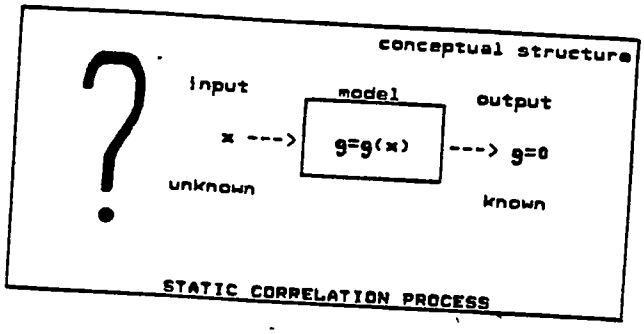
```
PROBLEM .CODES
 T=3, X=0, Y=1, DYDT=0, DELT=.1, TF=1
INITIATE MERCURY FOR .SYSTEM
   EQUATIONS VELX/X,ACCELY/VELY,VELY/Y
   OF T STEP DELT TO TF
INTEGRATE .SYSTEM
END

MODEL .SYSTEM
 VELX=3*X+Y
 ACCELY=.SIN(X)+.EXP(Y*T)
END
```

Arithmetic Contexts



This example is a simultaneous system involving a first order and a second order differential equation. The equations appear in their original form in the model procedure and a PROBLEM procedure is used to control integration via the INITIATE and INTEGRATE statements. In this problem there are three integrations whose respective inputs and outputs are identified by the sequence of arguments in the EQUATIONS phrase of the INITIATE statement. In its normal operation, the integration solver MERCURY will intermittently activate operand arithmetic causing the rate variables to be differentiated with respect to the state variables. These partial derivatives (∂xdot/∂x) are used to optimize the integration step size. This change in operand arithmetic from operator arithmetic is symbolized by the change in structure of the illustrated contexts on the right (above).

Correlation Operations - The process of correlation matches outputs of a simulation model to some known quantity by varying its inputs. Thus correlation is an induction process characterized by inputs that are unknown and outputs that are known. Thus in its conceptual structure we move the question mark to the left side:



The functions of the model would originally have the implicit form, x=f(x). The unknowns x may not be solved for directly, because they appear in non-reducable form on the right hand side. Thus, values of the unknowns must be found which balance both sides of the equations. An equality constraint vector g(x) is defined by placing all of the unknowns on the right. The value of the constraint vector is the deviation from balance computed during each simulation pass of the model. The process of correlation searches for the unknown inputs (x) which match the outputs (g) to zero, thus balancing the equations. This process is invoked by the statement:

FIND x IN model BY solver TO MATCH g

If the dimensions of x and g are the same, there are an equal number of unknowns and equations, the problem is said to be determined. In this case, the correlation process employs methods epitomized by the Newton-Raphson algorithm which utilize the partial derivatives ∂g/∂x to find values of x yielding zeros of the g vector. If the dimension of g is greater than that of x, there are more equations than unknowns, the problem is overdetermined. In this case, the correlation

process employs methods epitomized by the Newton-Gauss algorithm to match the constraints in a least squares sense. If the dimension of x is greater, there are more unknowns than equations, the problem is underdetermined (an optimization problem). In this case, the correlation process yields a "smallest least squares" solution.

The following example illustrates the structure of a determined system of implicit, nonlinear equations. Again, the equations are written in a MODEL procedure with the constraint variables appearing on the left. The PROBLEM procedure controlling the correlation process contains an ALLOT statement to dynamically create the arrays X and G and a vector data statement to define an initial guess for the unknowns. Finally, a FIND statement is used to solve the problem.

Correlation Example

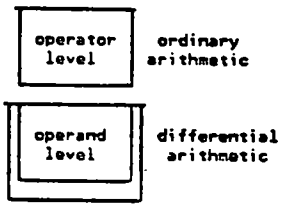
Solve the implicit equations:

$$g1(x1,x2,x3) = 12.5 - 3(x1)(x2) - x3 = 0$$
$$g2(x1,x2,x3) = 3.317 - \sin(x1) - \exp(x3) = 0$$
$$g3(x1,x2,x3) = 1.609 - (x2)\ln(x3) = 0$$

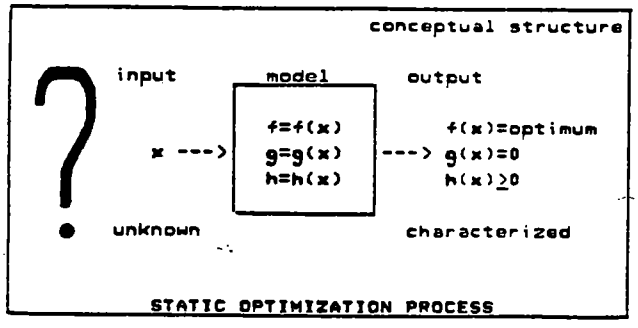
Program

```
PROBLEM .SIMUL
  ALLOT X(3),G(3)
  X=.DATA(2,2,2)
  FIND X IN .EQS BY AJAX TO MATCH G
END

MODEL .EQS
  G(1)=12.5-3*X(1)*X(2)-x(3)
  G(2)=3.317-.SIN(X(1))-.EXP(X(3))
  G(3)=1.609-X(2)*.LOG(X(3))
END
```

Arithmetic Contexts



Optimization Operations - The optimization solvers are designed to solve static optimization problems of all kinds, including linear and mixed integer programming and nonlinear programming. The conceptual structure of a static optimization process is illustrated as follows



Again, the process is inductive, because the inputs are the unknowns. The output is not known, as in correlation, but is characterized by the fact that one of the outputs is to be optimized and constraints (if any) are to be satisfied. In this case, two additional phrases are added to FIND statement to specify constraints:

```
FIND x IN model BY solver
   MATCHING g HOLDING h TO MINIMIZE f
```

where MATCHING identifies variables to be matched to zero, and HOLDING identifies variables to be held greater than or equal to zero.

Optimization Example

Find x and y to minimize

$$f(x,y) = (x-2)(x-2) + (y-1)(y-1)$$

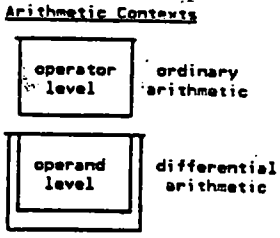
Subject to:

$$g(x,y)=1-(x)x/4-(y)y = 0$$
$$h(x,y)=\exp(xy)-x-2 \geq 0$$

Program

```
PROBLEM .SIMPLE
  X=-1, Y=0
  FIND X,Y IN .CONF BY JUPITER
    MATCHING G HOLDING H TO MINIMIZE F
END
MODEL .CONF
  G=1-X**2/4-Y**2
  H=.EXP(X*Y)-X-2
  F=(X-2)**2+(Y-1)**2
END
```

Arithmetic Contexts



Nonlinear programming algorithms perform many execution passes on the operand models, and may alternate in both the level of differential arithmetic and the basis of differentiation. Because of this requirement and even more dynamic requirements involved in structured calculus (see next page), compiler approaches to the generation of differential functions, have proved too limiting. The fully synthetic approach of dynamic (interpretive) arithmetic, having no such limits, allows the same model code to be employed as an operand of multiple solution processes in the same program.

Another example of optimization is one of linear, mixed-integer programming for which additional phrases are added to the FIND statement to identify discrete variables. Partial differentiation is employed to generate the initial LP matrix for linear programming, enabling model structures to have the same form as those of nonlinear programming, except that the formulas are linear.

Optimization Example

Using the following table for feed blend analysis:

Ingredient	Protein	Fat	Fibre	Cost/lb.
Screenings	0.130	0.030	0.070	$0.022
Barley	0.115	0.020	0.060	0.025
Corn	0.086	0.038	0.025	0.011
Salt	-	-	-	0.011

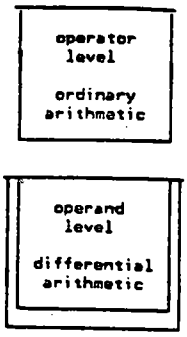
find the least expensive feed blend which satisfies the following:

(1) Total weight of blend = 2000 lb.
(2) Total protein ≥ 200 lb.
(3) Total fat ≥ 54 lb.
(4) Total fibre ≤ 90 lb.
(5) Amount of corn ≥ 400 lb. and ≤ 1000 lb.
(6) Weight of salt = 5 lb.
(7) Barley and corn purchased in 50 lb. lots

Program

```
PROBLEM .BLEND
  SALT=5,BARLEY=500,CORN=900,SCREENS=595
  LIMITS=.DATA(50,50,0)
  FIND BARLEY,CORN,SCREENS IN .MIX BY HERCULES
    WITH DISCRETE LIMITS
    HOLDING PROTEIN,FAT,FIBRE,CORNLOW,CORNHIGH
    MATCHING WEIGHT TO MINIMIZE COST
END
MODEL .MIX
 WEIGHT=BARLEY+CORN+SCREENS+SALT-2000
 PROTEIN=.115*BARLEY+.086*CORN+.13*SCREENS-200
 FAT=.02*BARLEY+.038*CORN+.03*SCREENS-54
 FIBRE=90-.06*BARLEY-.025*CORN-.07*SCREENS
 CORNLOW=CORN-400
 CORNHIGH=1000-CORN
 COST=.025*BARLEY+.28*CORN+.022*SCREENS+.011*SALT
END
```

Arithmetic Contexts



Qualities of Problem Description

Mathematical Self Organization - Synthetic calculus hierarchies are mathematically self organized in that the operators and operands naturally separate into roles that are inherent in the structure of the problems themselves, i.e., the hierarchy is intrinsic to the problem. The essential cleavage of hierarchic levels results from the necessary change of context between a predictive operand procedure and its higher-level control procedure. In describing the predictive model, we use deterministic formulas whose output is prescribed once the input is given, and this input is held constant during the computation pass of the predictive formulas.

But in order to control the prediction, causing it to balance some desired condition, we must vary the input of the predictive procedure as a parameter in multiple passes, each one producing a sample to be used by the controlling solver. This dual requirement of holding the input constant for prediction and varying it for control would cause a paradox in any single level of description. Thus there is no alternative to the use of two hierarchic levels to describe the process.

Predictive Modeling - The example problems presented above demonstrate the relative simplicity of programming when models are expressed as "open-loop" predictive simulation procedures containing only explicit non-iterative formulas. Novice programmers usually have no difficulty learning programming when formulations are this simple. Difficulty of understanding and obscurity of syntax develops when some form of solution process is blended with the formulation. Then the program loses its identity as a problem statement and starts becoming an algorithm, usually having some sort of strategy that must be studied to be understood. It is at this point that programs lose correspondence with the problems that they describe.

The Transparency of Levels - In software parlance, the word "transparency" is commonly used to indicate the presence of invisible mechanisms which alter the semantics of program descriptions in different contexts, but do not require the programmer to account for this in programming. An example is virtual memory.

As a transparent semantic process of a programming language, synthetic differentiation provides the vital link between "simulation-level" modeling and the far more difficult solution processes of correlation and optimization. Moreover, it serves as the means to uncouple modeling from numerical analysis. Since models and solution methods are transparent to each other and interchangable without reprogramming, numerical methods can be learned via experimental usage as plug-in tools. The interfaces between any model and any solution method in a process class are standardized to promote this.

Because the synthetic differentiation process is an exact method, its accuracy is independent of the nonlinearity of the formulation. Consequently, the user does not have to be sensitive to the customary difficulties of numerical approximation, but can use engineering judgment and experimentation with alternate methods to rapidly overcome solution difficulties. The sophisticated aspects of numerical implementation are thereby submerged below the level of the user's awareness to the same degree as the elemental operations of a computer system.

A further advantage of model transparency is that the same model can be used for simulation, correlation, or optimization without significant reformulation and reprogramming, except for the addition of objective formulas (to be matched or optimized). This enables a correlation or optimization model to be first executed by simulation for model validation purposes and to assess the sensitivity of the model to the implicit unknowns of the subsequent correlation or optimization process.

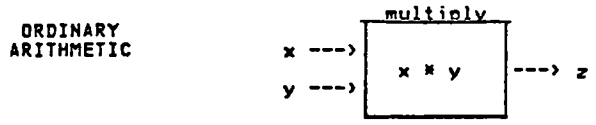
Structured Calculus Programming

Even though the operation statements of synthetic calculus are obviously unified into a consistent syntax, they might well constitute a family of disjointed problem-oriented languages that happen to be similar: a simulation language, a correlation language, and an optimization language. As such, these structures would not constitute a basis for general-purpose programming. The fact that they are true primitives of one language means that they can be combined, in the same manner as the algebraic primitives of lower-level languages, into structured procedures to address far more sophisticated problems than those that they solve directly. This capability is also the direct result of the foundation of synthetic differentiation.

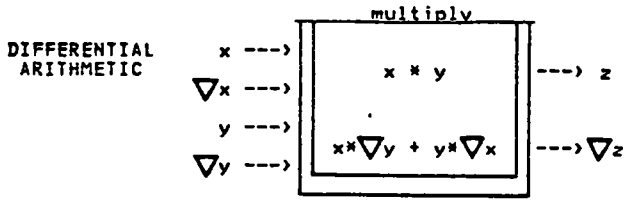
Arithmetic Propagation of Derivatives - The function arithmetic of a digital computer is a process of number propagation between the arithmetic units of the central processor and the storage registers in the computer memory. A mathematical function may be represented by a single arithmetic formula, or a system of formulas in which the output of one formula becomes the input of the next, and so on until the result of the function is computed as the output of the final formula in the system. Each formula in the system is a number processor, made up

of more primitive number processors (hardware instructions), and the overall system of formulas also constitutes a number processor.

We readily accept the fact that numbers flow through and are transformed by a number processor. However, we are used to thinking of numbers only as a single attribute of a function, its pointwise value. This attribute does not completely describe the pointwise behavior of the function because it tells us nothing about how the function is changing at that point. Such information is provided by other attributes of the function, its partial derivatives. Since for every arithmetic transformation (addition, subtraction, multiplication, etc.) there is a known analytical transformation of derivatives, we can generalize the concept of arithmetic to include the derivative transformations as well as the function transformations. Thus, instead of viewing the multiplication black-box of a digital computer as:

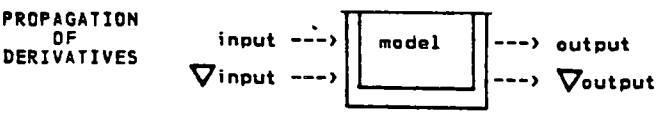


we may generalize it so that it becomes:



Therefore, for every sequence of arithmetic transformations, we obtain a better estimate of the pointwise behavior of the function, i.e., we estimate its shape as well as its size. With successively higher derivative transformations, we obtain successively better estimates of its pointwise behavior, i.e., more information about how its local shape is changing.

Since any model is merely a system of formulas and any formula is a system of primitive arithmetic operations; by generalizing the arithmetic in the above fashion, we automatically propagate the pointwise derivative values, along with the pointwise function values through the entire model. Therefore, we may view the model as a bigger black-box that has the same interfaces as the smaller ones:



The propagation of derivatives through any arithmetic process is the direct result of the generalization of arithmetic. Since any numerical algorithm is an arithmetic process, it follows that the derivatives of the output of any algorithm may be obtained from the derivatives of its input, as an automatic by-product of the algorithm's execution. Therefore, any algorithm may be nested within a higher level solution process by merely writing the algorithm in a model procedure that is the operand of the higher level process.

Nested Deduction - If the nested algorithm belongs to a simulation process which does not require partial derivatives of its output with respect to its input, the nested solution process is entirely equivalent to a model procedure as illustrated previously. Since static simulation processes such as matrix inversion, eigenvalue decomposition, evaluation of definite integrals and various transformations (e.g. Laplace and Fourier) are merely algebraic processes which do not require derivatives for solution, they may be nested within correlation or optimization processes by simply treating them as operands of the outer processes.

Likewise, dynamic simulation processes such as the solution of systems of differential equations usually involve only function arithmetic, albeit in a recursive manner. Consequently, they also may be nested within correlation and optimization processes in the same manner. This has major ramifications in the solution of boundary value problems and parameter estimation, because the partial derivatives required to match boundary conditions or solve for unknown parameters of differential equations are the partials of the output of integration (integrated state variables) with respect to its input (initial conditions or equation parameters).

Nested Induction - If the outer process is one of simulation, which does not require partial derivatives, the nesting of correlation and optimization processes presents no problem. They may be simply included as part of the simulation model. A primary application of this is in the solution of implicit differential equations.

However, if the outer process is itself inductive, the partial derivatives required for the outer process are dependent upon those required by the nested process. In principle this would require the nesting of differential arithmetic. While this presents no conceptual difficulty, the memory required to store nested derivative values during the propagation process is an order of magnitude greater than that required for the nested process by itself, since this amounts to the evaluation of higher order derivatives. The number of steps in the interpretation process would be multiplied to the same degree.

Fortunately, this is not necessary, because the outer process is constant (stationary) at the point that the inner process converges. The partial derivatives of the outer process may be obtained from the results of the nested process by a coordinate transformation based upon the implicit function theorem. To achieve an understanding of this transformation, consider the following nested process.

An induction process exists, having the unknown coordinate variables

x.1, x.2, ..., x.m.

The solution of this process depends upon the solution of a nested induction process whose unknown coordinate variables are

y.1, y.2, ..., y.n.

Assume, for example, that the nested process constitutes the solution of the implicit equations,

$$G(X,Y) = 0 \qquad (1)$$

in the following problem structure:

```
PROBLEM .OUTER
  ALLOT X(M),Y(N),G(N)
    :
  FIND X IN .INNER BY HERA TO MINIMIZE F
END
        MODEL .INNER
           :
          FIND Y IN .EQUATS BY AJAX TO MATCH G
        END
                MODEL .EQUATS
                  G(1)=g1(X,Y)
                    :
                  G(N)=gn(X,Y)
                  F=f(X,Y)
                END
```

The vector Y may be found by an ordinary correlation process in procedure .INNER, while X is held constant. However, Y, so defined, is a function of X, because all computations depend upon the value of X. If equation (1) is differentiated with respect to X, the following is obtained by the chain rule of partial differentiation:

$$0 = dg.i/dx.j = \partial g.i/\partial x.j + \sum_s (\partial g.i/\partial y.s)(\partial y.s/\partial x.j) \qquad (2)$$

Equation (2) represents the total derivative of ith element of the vector G with respect to the jth element of the vector X and is zero because G is the constant (stationary) zero at convergence of the .INNER nest. From this equation therefore,

$$\partial Y/\partial X = (\partial G/\partial Y)^{-1}(-\partial G/\partial X) \qquad (3)$$

The second derivatives of Y with respect to X may be obtained by further differentiation of equation (3) with respect to X

$$\begin{aligned} 0 = d2g.i/\partial x.j\ \partial x.k = {} & \partial 2g.i/\partial x.j\ \partial x.k \\ & + \sum_s (\partial 2g.i/\partial x.j\ \partial y.s)(\partial y.s/\partial x.k) \\ & + \sum_s (\partial 2g.i/\partial y.s\ \partial x.k)(\partial y.s/\partial x.j) \\ & + \sum_s \sum_t (\partial 2g.i/\partial y.s\ \partial y.t)(\partial y.s/\partial x.j)(\partial y.t/\partial x.k) \\ & + \sum_s (\partial g.i/\partial y.s)(\partial 2y.s/\partial x.j\ \partial x.k) \qquad (4) \end{aligned}$$

hence,

$$\begin{aligned} \partial 2y.i/\partial x.j\ \partial x.k = {} & (\partial G/\partial Y)^{-1}\Big\{\partial 2g.i/\partial x.i\ \partial x.j \\ & + 2\sum_s (\partial 2g.i/\partial x.j\ \partial y.s)(\partial y.s/\partial x.k) \\ & + \sum_s \sum_t (\partial 2g.i/\partial y.s\ \partial y.t)(\partial y.s/\partial x.j)(\partial y.t/\partial x.k)\Big\} \\ & (i=1,\ldots,n) \qquad (5) \end{aligned}$$

Derivative propagation via this transformation consists of four phases of differentiation of the .EQUATS model by the .INNER FIND statement, each having different control frames. The result produced is the transformed first and second partials with respect to X, required by the solver HERA in the outer process.

Phase 1: Compute G, $\partial G/\partial Y$

In this phase the nested implicit equations are iteratively solved to compute Y(X). The differential arithmetic is first order, based upon Y. The matrix

$$(\partial G/\partial Y)^{-1}$$

is saved after computation.

Phase 2: Compute G, $\partial G/\partial X$

This phase employs first-order differential arithmetic based on X. At the end of this phase, equations (3) are executed to compute $(\partial Y/\partial X)$. At this point, the partials $(\partial Y/\partial X)$ are stored, making Y an ordinary dependent variable of the outer process with first partials, but with zero second partials.

Phase 3: Compute G, $\partial G/\partial X$, $\partial 2G/\partial X2$
given Y, $\partial Y/\partial X$

This phase employs second-order differential arithmetic based on X. At the end of this phase, all but the very last term of equation (4) is computed and stored under the label $\partial 2g.i/\partial x.j\ \partial x.k$. The partials $\partial 2Y/\partial X\partial X$ are used as zero, hence the bracketed term on the right hand side of equation (5) is automatically computed. The quantities $\partial 2Y/\partial X\partial X$ are now computed using equation (5), employing the

$$(\partial G/\partial Y)^{-1}$$

matrix that was saved at the end of phase 1.

Phase 4: Compute G, $\partial G/\partial X$, $\partial 2G/\partial X\partial X$,
given Y, $\partial Y/\partial X$, $\partial 2Y/\partial X\partial X$

This phase employs second-order differential arithmetic based on X. At the beginning of this phase, the partials $\partial 2Y/\partial X\partial X$ are loaded into their storage locations so that their influence is registered in the synthetic differentiation process to evaluate $\partial 2G/\partial X\partial X$. At this point the .INNER FIND is complete, and control returns to the solver of the .OUTER FIND (HERA) which determines a new set of values of the basis vector X for the next optimization iteration.

In SLANG (Refs. 3-5) and PROSE, this derivative propagation procedure is automatically invoked whenever a first order correlation or

optimization process is nested within a second order optimization process. If the outer process is first order, three phases are required, phase 3 being a repeat of phase 2 after ∂Y/∂X has been loaded into storage.

Calculus of Variations - The nesting of static correlation and dynamic simulation processes within static optimization processes may be used to formulate function optimization problems. Consider the "optimal design and control" problem (Ref. 6)

$$J(y,a) = \frac{1}{2}\int_0^1 (x^2+y^2)\,dt + a^2/2 \qquad (6)$$

where the state variable y is governed by the equations:

$$dx/dt = -ax + y, \qquad x(0)=1 \qquad (7)$$

$$dy/dt = x + ay, \qquad y(1)=0 \qquad (8)$$

By introducing the "objective state variable" z, we may convert the integral to an ODE:

$$dz/dt = (x)x + (y)y \qquad (9)$$

so that the objective function becomes:

$$J(y,a)=(z(1)-z(0))/2 + (a)a/2 \qquad (10)$$

This problem is solved by the nested combination of the operation statements for optimization, correlation, and simulation, respectively. The program has the triple nested structure illustrated below.

```
                                        Optimal Design & Control Problem
PROBLEM .OPTDES
  INTERACT
  FIND A IN .GETJ BY HERA TO MINIMIZE J
END
        ┌──────────────────────────────────────────────────────────────┐
        │                                  Boundary Value Problem      │
        │MODEL .GETJ                                                   │
        │  FIND Y0 IN .GETY BY AJAX TO MATCH Y1                        │
        │  J=Z/2 +A**2/2                                               │
        │END                                                           │
        │      ┌──────────────────────────────────────────────────┐    │
        │      │                          Initial Value Problem   │    │
        │      │MODEL .GETY                                       │    │
        │      │  Y=Y0, X=1, Z=0, T=0, T1=1                       │    │
        │      │  INITIATE ISIS FOR .DIFF EQUATIONS               │    │
        │      │  XDOT/X,YDOT/Y,ZDOT/Z OF T STEP DT TO T1         │    │
        │      │  INTEGRATE .DIFF                                 │    │
        │      │  Y1=Y                                            │    │
        │      │END                                               │    │
        │      │                    ┌──────────────────┐          │    │
        │      │                    │MODEL .DIFF       │          │    │
        │      │                    │  XDOT=-A*X+Y     │          │    │
        │      │                    │  YDOT=X+A*Y      │          │    │
        │      │                    │  ZDOT=X**2+Y**2  │          │    │
        │      │                    │END               │          │    │
        │      │                    └──────────────────┘          │    │
        │      └──────────────────────────────────────────────────┘    │
        └──────────────────────────────────────────────────────────────┘
```

The innermost nest is a dynamic simulation process to solve an initial value problem, integrating the differential equations from the initial conditions specified on the first line of the procedure .GETY to the terminal point specified by T1=1.

This problem is nested within a correlation process which solves for the initial condition Y0 which causes the output of integration to match the boundary condition Y1 to zero, thereby forcing the output of integration to converge to zero at T=T1. The solution of the boundary value problem is prescribed by the statement

FIND Y0 IN .GETY BY AJAX TO MATCH Y1

in the procedure .GETJ. The subsequent statement computes the objective variable J that is minimized by the solver HERA in the outer optimization process. This variable represents an integral function, since it is a function of Z, the output state variable of the differential equation for ZDOT.

Derivative propagation plays a vital role in this program in that it automatically generates the derivative values required by the correlation and optimization solvers. The gradient of Y1 with respect to Y0 is propagated through the simulation process to be used by the solver AJAX. The first and second derivatives of J with respect to A are propagated from the nested derivatives, via the transformation discussed above, to be employed by the optimization solver HERA.

CALCULUS COMPUTER ARCHITECTURE

The primary motivation for the design of the Aerospace Research Computer (ARC, Ref. 7) was to provide a high-performance environment for the evolution of synthetic calculus and to experiment with semantic intensive algorithmic processes which require hardware speed enhancement for imbedded real-time applications.

Calculus Semantic Enhancement

The ARC was designed to enable the use of bilevel software technology for software algorithmic processes which require hierarchic (nested) algorithms that are used with very high frequency in mathematical computation. The objective of the bilevel (metacomputer) concept is to isolate candidate algorithms for enhancement by improving firmware and hardware design. The following computation processes are exemplary candidates:

(1) Synthetic Differentiation

(2) Synthetic Differential Integration

(3) Synthetic Integral Differentiation

(4) Synthetic Differential Integral Differentiation, etc.

These processes are important for successively more complex parameter induction processes having to do with, for example, boundary-value problems for differential equations, optimal control, and parameter identification for partial differential equations.

Synthetic Differentiation - This arithmetic process (Wengert's method) is the fundamental process of synthetic calculus, described in Reference 1.

Synthetic Differential Integration - This method of integrating differential equation systems is actually a Jacobian-based integration step optimization process. It uses synthetic differentiation to produce the Jacobian matrix of the state derivatives with respect to the state at each integration step to determine the optimum step size for maintaining stability and accuracy. The most famous method in this class is Gear's method (Ref. 8). This methodology has proven to be quite effective in the solution of large systems of stiff (e.g. singular perturbation) and spectrally ill-conditioned dynamic systems, which defy solution by more primitive integration techniques except via the use of infinitesimal step sizes.

Synthetic Integral Differentiation - This method of differentiating integration processes enables an algorithmic process to produce local Frechet derivatives of integral process output with respect to integral process input. Thus it constitutes a generalization of synthetic differentiation, enabling integral processes (solution of differential equations) to be differentiated in function space. The most natural application of this process is in the solution of boundary-value problems and process-identification problems. It enables the differentiation of boundary values with respect to initial values or parameters of the differential equations. The resulting partial derivatives are used to match the boundary conditions or data fitting criteria by differential correction methods.

Synthetic Differential Integral Differentiation - This algorithmic process is the next level in the hierarchy of the above processes, in which one differentiation process basis (coordinate system) is dependent upon another. This process requires the nesting of differential contexts in the manner illustrated in the optimal design and control problem where, in addition, the numerical integration process involves Jacobian-based step optimization. While the above processes are within the capability of the PROSE metacomputers, this process is beyond it. It has heretofore been considered too semantic-intensive for pure software interpretation. Its primary need is for boundary-value problems and process-identification problems of stiff or spectrally ill-conditioned systems.

Metacomputer Host Architecture

Since the late sixties most computer architecture designs have employed metacomputers ("virtual machines") to emulate software. These metacomputers are microprogrammed interpreters. From a functional point of view they are analogous to the metacomputers of synthetic calculus. They have provided computer manufacturers the flexibility to evolve hardware design without impacting software, yet maintaining software dependence on the emulated assembly languages. The essential role of programmed interpretation is equivalent. The practical difference is in the level of interpretation. Current machine technology focuses on a much lower order of interpretation than synthetic calculus.

Higher-Order Micro Software - In order to shift the focus of software development to very-high-level language, hardware microinstructions need to be raised to a higher level than current assembly languages. This new microprogramming level can be used to write high-speed emulators for the very-high-level languages of synthetic calculus.

Raising the instruction level of the micro software increases speed by making single micro instructions do the work of several assembler instructions, and taking advantage of instruction pipelining to achieve maximum execution throughput. This increase in micro software speeds can allow the development of very-high-speed programmed interpreters and algorithmic tool systems.

Higher-Order Emulation - A second factor which influences speed is the level of the language that is interpreted. Assembly-level languages have a high percentage of non-functional instructions acting as overhead. Interpretation speed is retarded by the necessity of interpreting these extra instructions, along with the functional instructions. Higher-level languages, being closer to the ideal representation of the problem have fewer non-functional attributes. In addition, higher-level languages, by grouping operations into higher wholes, e.g., vectors, offer greater opportunity for parallelism in execution.

The use of higher-level micro instructions to interpret the very-high-level macro instructions of synthetic calculus promises a 200 to 1000 fold increase in the execution speeds of synthetic calculus metaprograms, rendering this advanced technology fast enough for real-time induction applications of large-scale nonlinear filtering, optimal control and nonlinear optimization.

CONCLUSION

Synthetic calculus, because of its hierarchic operation structure, enforces the correspondence of programs to problem statements. It achieves this by submerging virtually all of the higher mathematics of problem solving into the invisible realm of the metacomputer, leaving only the problem-oriented syntax of the engineering formulation to be defined by the user. The more difficult tasks of program design and development, including the design of numerical solution approaches, or the recasting of models to fit available methods with often necessary simplifying approximations, is all but eliminated. Eliminated also is the usually prohibitive amount of time and effort involved in this process and the attendant errors in formulation and programming that must be ferreted out by debugging.

The structure of synthetic calculus exemplifies an important property of hierarchy theory (Ref. 9) which explains the means of mathematical

simplification. This property, called "optimum loss of detail" is manifested in alternate levels of description and levels of mechanism* in which control passes from a level of description (operator level) through a level of mechanism (solver) to a subordinate level of description (operand level) implemented via a foundation level of mechanism (arithmetic level). Only the levels of description are visible in calculus programming.

The programming paradigm is effective because the levels of mechanism can be understood metaphorically, without direct knowledge of the detail of the mechanism, by reference to their behavior. The behavior is manifested only in the experimental use of the mechanism, not in its internal structure, and this experimentation is available to the synthetic calculus programmer, even if the internal structure is not.

As a programming paradigm for synthesizing mathematical experiments, synthetic calculus is a departure from traditional analytical mathematics which has given rise to symbolic software tools such as MACSYMA (Ref. 10,11). While symbolic tools are used to manipulate mathematical structure, synthetic calculus is primarily used to produce mathematical behavior. In this sense, it is more aligned with the purpose of analog computing or with the experimentation of mathematical physics.

REFERENCES

1. Thames, J.M., "The Evolution of Synthetic Calculus, A Mathematical Technology for Advanced Architecture", Proceedings of the International Workshop on High-Level Language Computer Architecture, Fort Laudardale Fla., Nov. 30 - Dec. 3, 1982.

2. Thames, J.M., "Computing in Calculus", Research/Development 26,5 (May 1975) pp. 24-30

3. Adamson, D.S., "Introduction to SLANG", TRW Doc. 99900-672-r0-00, 1968

4. Thames, J.M., "SLANG, A Problem-Solving Language for Continuous-Model Simulation and Optimization", Proceedings, ACM 24th National Conf. (December 1969) pp. 23-41

5. Adamson, D.S., and Winant, C.W. "A SLANG Simulation of an Initially Strong Shock Wave Downstream of an Infinite Area Change", Proc. Conf. on Applic. of Continuous-System Simulation Languages (June 1969) pp. 231-240

6. Bellman, R.E., and Kalaba, R.E. Quasilinearization and Nonlinear Boundary Value Problems, American Elsevier Publishing Co. New York, 1965

7. Speckhard, A.E., Wood, T.C., and Thames, J.M. , "The Aerospace Research Computer", Proceedings of the Workshop on High-Level Computer Architecture, Los Angeles, CA, May 21-25, 1984 (This volume.)

8. Gear, C.W., "Algorithm 407, DIFSUB for Solution of Ordinary Differential Equations", Communications of the ACM 14 (1971)

9. Pattee, H.H., Hierarchy Theory - The Challenge of Complex Systems, Geo. Braziller Pub., New York, 1973.

10. MACSYMA Reference Manual, MIT Laboratory for Computer Science, 12/77.

11. Pavelle, R., Rothstein, M., and Fitch, J. "Computer Algebra", Scientific American, Vol. 245, No. 6. (Dec 1981), p 136.

* Reference 9 uses "levels of structure"