FORTRAN CALCULUS: A new implementation of synthetic calculus

Joseph Thames, Digital Calculus Corporation

FORTRAN CALCULUS is the first of a new series of synthetic calculus implementations, available on personal computers, mainframes, and supercomputers as an extension to FORTRAN environments for systems optimization and higher mathematical simulation.

Synthetic calculus is a hierarchical form of implicit systems mathematics, first introduced in the late sixties via the SLANG language [1,14] at TRW, and later marketed through national time-sharing services via the PROSE language [6,7,12,13,15,16]. Originally designed for systems optimization, its foundation is an object-oriented* environment of analytically-based differential arithmetic which dynamically differentiates algebraic models. This extended arithmetic produces arrays of first and second-order partial derivative values (accurate to computer precision) as a by-product of model execution. It is capable of differentiating numerical integration algorithms, producing partials of boundary conditions with respect to initial conditions or equation parameters. It also performs differential coordinate transformations to mechanise the hierarchic nesting of system-level problems (nonlinear programming, simultaneous algebraic equations, and simultaneous differential equations) to solve higher-order inverse problems such as boundary-value problems, optimal control problems, and inverse problems of partial-differential equations.

FORTRAN CALCULUS (FC) is a PROSE-like extension of FORTRAN 77, providing a unification of well-known mathematical programming algorithms, numerical integration algorithms and simultaneous equation solvers, as well as linear algebra functions employing dynamic storage allocation.

Differential Arithmetic

Figure 1 shows a simple problem posed as a bilevel optimization hierarchy in FC. Also indicated is the arithmetic hierarchy imposed by this operation. The outer context is ordinary arithmetic and the inner context is differential arithmetic producing values of partial derivatives of every dependent variable of the inner context with respect to the independent variables specified in the FIND macro-statement.

Constrained Optimization Example

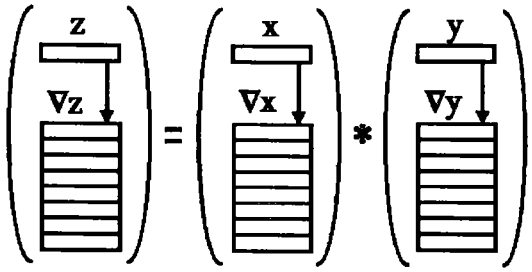
```
GLOBAL ALL
PROGRAM SIMPLE                          <- Ordinary Arithmetic
  X=-1 : Y=0
  FIND X,Y; IN EQS; BY THOR;
~   MATCHING G; HOLDING H;
~   TO MINIMIZE F
END
MODEL EQS                               <- Differential Arithmetic
  G=1-X**2/4-Y**2
  H=EXP(X*Y)-X-2
  F=(X-2)**2+(Y-1)**2
END
```

Figure 1: The basic program hierarchy is formed by the FIND macro-statement in the outer context which iteratively executes the inner context where differential arithmetic differentiates the formulas as they are executing.

This differentiation process, illustrated schematically in Figure 2, is referred to in the mathematical literature as automatic derivative evaluation (ADE), see [3-12]. No formula manipulation occurs. The formulas for differential computation, including the chain-rule, are extensions to the arithmetic of the machine. Note that this is vector arithmetic because partial derivatives are being evaluated. Thus it may be viewed as a generalization of basic arithmetic that computes function shape information (gradient vector and Hessian matrix) at the point where the function is evaluated.

Notice that the differentiation process and the derivative values themselves are hidden from the programmer. Programming has become calculus based, but has also become substantially simpler. The major purpose of the generalized arithmetic is to internally support built-in Newton-based equation solving and non-linear programming algorithms, and Jacobian-based numerical integration algorithms. The programmer applies these generalized methods to mathematical models using the problem-solving macro-statements.

* See Appendix for a discussion in terms of object-oriented programming concepts.



Program Language:	z = x*y
"Machine" Substructure:	∇z = x*∇y + y*∇x

Figure 2: Differential arithmetic evaluates partial derivatives according to fundamental calculus formulas producing gradient vectors (and Hessian matrices) as a by-product of formula evaluation.

Mathematical Modeling Taxonomy

Synthetic calculus renders algorithmic high-level languages into non-algorithmic very-high-level languages by decomposing problem formulations according to intrinsic mathematical cleavage:

- ***Model/algorithm*** - The migration of partial differentiation from model definition to automatic arithmetic permits the separation of models from algorithms, enabling algorithms to become part of the environment rather than the program;
- ***Explicit/implicit*** - Problems are decomposed according to whether the unknowns are dependent variables (explicit) or independent variables (implicit) and implicit solution operators are provided that enable the separated implicit subproblems to be stated explicitly;
- ***Solution criteria*** - Problems are decomposed according to how the solution is achieved: *simulation processes* where predictive projections are computed (e.g. numerical integration), *correlation processes* where roots are found to match equality constraints in zero degrees of freedom or conditions of redundancy, and *optimization processes* where roots are found to achieve maxima or minima, subject to constraints;
- ***Prediction/Control*** - The separation and nesting of prediction processes within control processes is a necessary method of avoiding paradoxical problem statements in mathematical modeling, and decomposes the problem description into a nested hierarchy of separate mathematical problems, each functioning as a complementary part of a dual prediction-control process.

Each level of the control/prediction hierarchy is a complete system problem. The exchange of functional dependence for prediction occurs visibly through the exchange of variables between the levels. The exchange of differential dependence for control occurs invisibly through underlying differential propagation mechanisms. Different types of differential propagation are invoked depending upon the explicit or implicit structure of the nested problems:

- ***Explicit within Explicit*** - in the degenerate case there is no control/prediction hierarchy, and differential propagation is merely the result of differential arithmetic.
- ***Explicit within implicit*** - the typical case of prediction nested within control where the inner (e.g. numerical integration of differential equations) process is partially differentiated with respect to the independent variables of the outer (e.g. nonlinear optimization) process, producing functional partial derivatives;
- ***Implicit within implicit*** - the case of a dual control/prediction hierarchy where the inner control/prediction (e.g. nonlinear simultaneous equations solving) process produces partial derivatives with respect to the coordinate system of the outer (e.g. nonlinear optimization) process via a differential coordinate transformation; and
- ***Implicit within explicit*** - the case of a control/prediction hierarchy within a prediction where the inner (e.g. nonlinear simultaneous equations solving) process is used to compute implicit values (e.g. ordinary derivatives) that are used in the computations of the outer (e.g. numerical integration of ordinary differential equations) process, as in the case of implicit differential equations.

Macro Statements & Stereotypes

Correlation and optimization processes are invoked using FIND macro-statements, and simulation processes are invoked by INITIATE and INTEGRATE macro-statements. These processes are the *primitive* bilevel hierarchies of synthetic calculus programs. They may be combined in nested multilevel structures to address more complex mathematics. The following discussion treats several problem *stereotypes* which characterize broad classes of higher mathematics.

Implicit Problem Nesting

In correlation and optimization processes, differential arithmetic is invoked by the first phrase of the FIND macro statement:

FIND *parameter vector*; **IN** *model procedure*.

Differential arithmetic is a form of ADE in which the *parameter vector*, defining the differential coordinate system, can be dynamically altered in execution. This enables the nesting of implicit problem hierarchies as illustrated in Figure 3. For example, the model procedure of the FIND operation could contain another FIND statement referencing another parameter vector and another (nested) model procedure (Figure 4). Since each parameter vector represents a coordinate system, a differential coordinate transformation is performed following the execution of the inner FIND statement to evaluate differentials of the outer FIND corresponding to implicitly computed variables of the inner FIND. This transformation mechanism makes synthetic calculus *ubiquitously* hierarchic, because it is a local exchange between the coordinate systems of adjoining levels of a dual control/prediction hierarchy. Any number of nesting levels is therefore feasible.

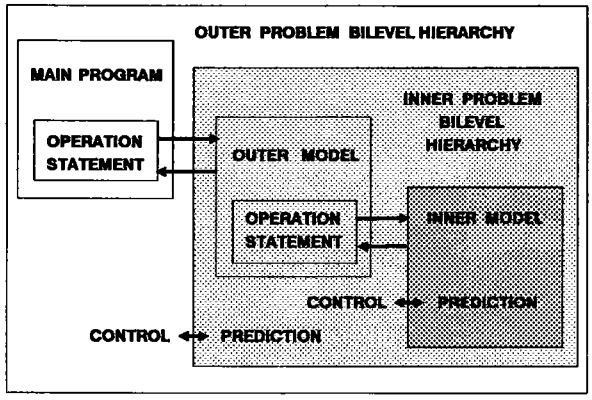


Figure 3: More sophisticated problems are posed as multi-level hierarchies of nested bilevel hierarchies. Internal coordinate transformations and differentiated integration algorithms propagate partial derivatives through nested contexts.

Figure 4 illustrates this structure in the solution of an equality constrained optimization problem where a nested nonlinear equations solver is used to match constraints during each iteration of a nonlinear unconstrained optimization solver . The coordinate transformation computes the partials of F with respect to vector X required in the outer context from the partials of vector G and scalar F with respect to vector Y obtained in the inner iterations. This process involves three differential passes through model TWO following convergence of the inner search, evaluating first partials and second partials with respect to X. See [6].

Integration within Differentiation

In cases where a problem contains differential equations, two additional macro statements are used to invoke simulation:

INITIATE *solver*; **FOR** *model*;
EQUATIONS *derivative-vector/state-vector*;
OF *independent variable*;
STEP *size variable*; **TO** *integration limit*

and

INTEGRATE *model*; **BY** *solver*

which specify and solve an initial value problem.

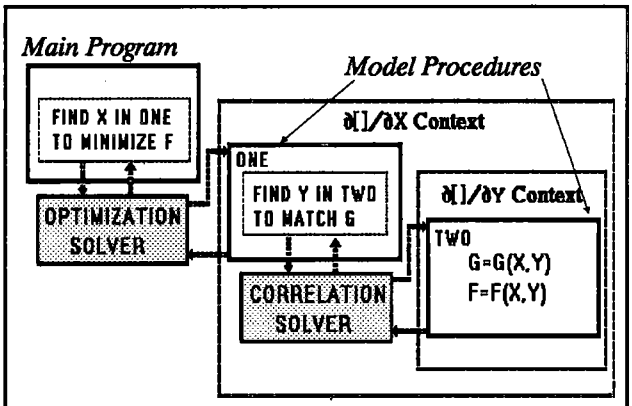


Figure 4: Dynamic redefinition of differential coordinates enables the hierarchic nesting of iterative contexts. Inner partials are transformed into outer partials via a differential coordinate transformation invoked by the hidden solvers.

In hierarchic problems where the inner problem is an initial value problem, differential arithmetic is used to differentiate the integration process as it is executing. This process will differentiate functions that have no analytic form and therefore could not be differentiated by symbol manipulation methods (e.g. derivatives of boundary conditions with respect to initial conditions, Figure 5).

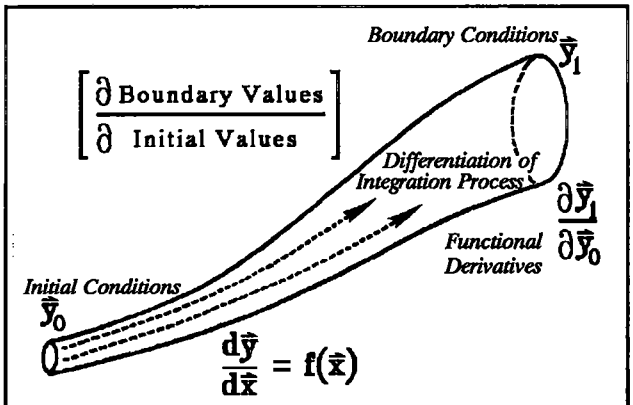


Figure 5: Differentiation of numerical integration computes derivatives that cannot be produced by symbolic differentiation

Figure 6 illustrates a four level hierarchy involving implicit problem nesting and integration within differentiation. This is an optimal design and control problem originally programmed as shown on the right and published in [2] as an example of the technique of quasilinearization. This problem aptly illustrates the benefit of the hierarchic mathematics of synthetic calculus. Particularly important is that the hierarchic formulation, being hardly more than a mathematical statement of the problem, is non-algorithmic, and is therefore readily comprehensible. The models used to describe the problem unit functions are expressed as "open-loop' predictive procedures containing only explicit non-iterative formulas. Novice programmers usually have no difficulty learning to program sophisticated applications when formulations are this simple. Difficulty of understanding and obscurity of syntax develops when some sort of solution process is blended with the problem formulation, as is the case of the FORTRAN program shown on the right. Then the program loses its identity as a problem statement and becomes as algorithm that must be studied to be understood.

The hierarchic structure of this program is illustrated schematically in Figure 7 showing the nested partial derivative contexts and the hidden solvers. Each of the three nested processes is a complete problem in a mathematical sense.

HIERARCHIC

```
GLOBAL ALL
PROBLEM OPTDES
~   FIND A; IN TWOPT; BY HERA;
    TO MINIMIZE OBJ
END

MODEL TWOPT ! Two point boundary value problem
  Y0=.7  ! Guess for initial condition
  FIND Y0; IN TRAJ; BY AJAX; TO MATCH Y1
  OBJ=Z/2+A**2/2
END

MODEL TRAJ     ! Initial value problem
  Y=Y0 : X=1 : Z=0 : T=0 ! Initial conditions
  DT=0.25 : TF=1
  INITIATE ISIS; FOR DIFF;
~    EQUATIONS ZDOT/Z, XDOT/X, YDOT/Y;
~    OF T; STEP DT; TO TF
  INTEGRATE DIFF; BY ISIS
  Y1=Y ! Boundary condition
  TERMINATE DIFF
END

MODEL DIFF
  ZDOT=X**2+Y**2 ! Objective differential equation
  XDOT=-A*X+Y    ! State differential equation
  YDOT=X+A*Y     ! Control differential equation
END
```

Figure 6: The optimal design and control problem is posed in FORTRAN CALCULUS (above) as an initial-value problem nested within a correlation problem solving a two-point boundary value problem, nested within an optimization problem solving for the design parameter to optimize a functional. The leverage of hierarchic math is illustrated by the two programs. The FORTRAN CALCULUS one is simple, non algorithmic and therefore comprehensible. The original FORTRAN version is complex, algorithmic and somewhat incomprehensible.

NON-HIERARCHIC

```
      PROGRAM OPTDES(INPUT,OUTPUT)
C     DESIGN PROBLEM
      COMMON N1,KMAX,NP,NMAX,HGRID,CX,T,X,
     ~            Y,U,V,H,PREV,H,P,A,B,C,NEQ
      DIMENSION T(250),W(4,500),H(4,4,500)
     ~          P(4,500),A(2,2),B(2),C(4),PREV(4)
    1 CALL INPUT
    2 CALL START
    3 DO 19,K=1,KMAX
      DO 5 I=1,250
    5 T(I)=0.
      T(2)=0.
      T(3)=HGRID
      T(4)=1.
      T(9)=1.
      T(14)=1.
      N=0
      PRINT 111,(T(L),L=4,431
      NEC=20
      CALL INT(T,NEQ,N1,0.,0,.0.,0.,0.,0.)
      N=0
    8 DO 11 M=1,NP
      N=N+1
      X=W(1,N)
      Y=W(2,N)
      U=W(3,N)
      V=W(4,N)
      CALL INTM
      L=3
      DO 9 I=1,4
      DO 9 J=1,4
      L=L+1
    9 H(I,J,N)=T(L)
      DO 10 J=1,4
      L=L+1
   10 P(J,N)=T(L)
   11 CONTINUE
    6 PRINT 111,N,(T(L),L=4,43)
      PRINT 111,N((H(I,J,N),J=1,4),I=1,4),
     ~   (P(J,N),J=1,4)
  111 FORMAT(1H019X10,4E20.6/(30X4E20.6))
      IF(N-NMAX)8,7,7
    7 PRINT 40,K
   40 FORMAT(1H0/65X9HITERATION,I3)
      N=NMAX
      A(1,1)=H(2,2,N)
      A(2,1)=H(2,3,N)
      A(1,2)=H(3,2,N)+H(4,2,N)
      A(2,2)=H(3,3,N)+H(4,3,N)
      B(1)=-P(2,N)-CX*H(1,2,N)
      B(2)=P(3,N)-CX*H(1,3,N)
      DET=A(1,1)*A(2,2)-A(2,1)*A(1,2)
      G=ABSF(DET)
      IF(G-.000001)200,200,12
  200 PRINT 201,DET((A(I,J),J=1,2),B(I),I=1,2)
  201 FORMAT(1H020X11HDETERMINANT =
     ~ ,E16.6/(40X3E20.6))
      CALL EXIT
   12 C(1)=CX
      C(2)=(B(1)*A(2,2)-B(2)*A(1,2))/DET
      C(3)=(B(2)*A(1,1)-B(1)*A(2,1)/DET
      C(4)=C(3)
      GRID=0.
      PRINT 45,GRID,(C(I),I=1,4)
   45 FORMAT(1H026X4HGRID,14X1HX,19X1HY,
     ~ 18X2HMU,19X1HA/20F10.4,4E20.6)
      N=0
   13 DO 14 M=1,NP
      N=N+1
      W(I,N)=P(I,N)
      DO 14 J=1,4
   14 W(I,N)=W(I,N)+C(J)*H(J,I,N)
      FN=N
      GRID=FN*HGRID
      PRINT 50,GRID,(W(I,N),I=1,4)
      IF(N-NMAX)13,15,15
   15 DO 16 I=1,4
      G=ABSF(C(I)-PREV(I))
      IF(G-.000001)16,16,17
   16 CONTINUE
      GOTO 1
   17 DO 18 I=1,4
   18 PREV(I)=C(I)
   19 CONTINUE
      GOTO 1
   50 FORMAT(20F10.4,4E20.6)
      END

      SUBROUTINE INPUT
      COMMON N1,KMAX,NP,NMAX,HGRID,CX,T,X,
     ~            Y,U,V,H,PREV,H,P,A,B,C,NEQ
      DIMENSION T(250),W(4,500),H(4,4,500)
     ~          P(4,500),A(2,2),B(2),C(4),PREV(4)
      READ 110,N1,KMAX,NP,NMAX,HGRID,CX
  110 FORMAT(4I4,2E8.2)
      PRINT 10,N1,KMAX,NP,NMAX,HGRID,CX
   10 FORMAT(1H1120X14HDESIGN PROBLEM//
     ~ 38X2HN1,6X4HKMAX,8X2HNP,6X4HNMAX,
     ~ 13X5HHGRID,16X4HX(10)/30X4I10,2E20.6)
      RETURN
      END

      SUBROUTINE START
      COMMON N1,KMAX,NP,NMAX,HGRID,CX,T,X,
     ~            Y,U,V,H,PREV,H,P,A,B,C,NEQ
      DIMENSION T(250),W(4,500),H(4,4,500)
     ~          P(4,500),A(2,2),B(2),C(4),PREV(4)
      DO 1 N=1,NMAX
      W(1,N)=CX
      W(2,N)=0
      W(3,N)=0
    1 W(4,N)=0
      DO 2 I=1,4
    2 PREV(I)=W(I,1)
      K=0
      PRINT 40,K
      GRID=0.
      PRINT 45,GRID(W(I,1),I=1,4)
   40 FORMAT(1H0/65X9HITERATION,I3)
   45 FORMAT(1H026X4HGRID,14X1HX,19X1HY,
     ~ 18H2HMU,19X1HA/20XF10.4,4E20.6
     ~ /1H019X22HFOR ALL VALUES OF TIME)
      RETURN
      END
```

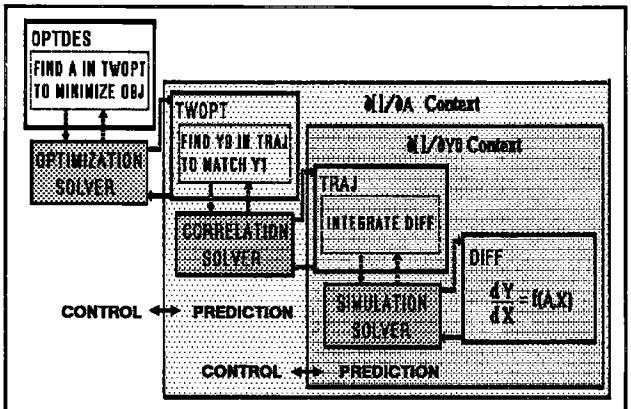


Figure 7: Calculus of variations hierarchy combining three calculus processes. The two inner processes solve a two-point boundary-value problem, and the outer process optimizes a parameter of its differential equations

The hierarchic subdivision of the models, shown structurally in Figure 8, is a natural decomposition into alternating levels of prediction and control because of what might be called "paradox avoidance". In describing a predictive model, deterministic formulas are used whose output is prescribed once the input is given, and this input is held constant during the computation pass of the predictive formulas (Y0 is constant in the initial-value problem and A is constant in the boundary value problem). But in order to control the prediction, the input of the predictive formulas must be varied as parameters in multiple passes, each pass being used as a sample by the control method in iteratively seeking a stationary point (root or extremum). This dual requirement of holding the input constant for prediction and varying it for control would cause a paradox in a single level of description, leaving no alternative to the use of separate hierarchic levels to describe the prediction/control process.

```
GLOBAL ALL
PROBLEM OPTDES
~  FIND A; IN TWOPT; BY HERA;
   TO MINIMIZE OBJ
END

   Differential Coordinate Tranformation

   MODEL TWOPT ! Two point boundary value problem
     Y0=.7 ! Guess for initial condition
     FIND Y0; IN TRAJ; BY AJAX; TO MATCH Y1
     OBJ=Z/2+A**2/2
   END

   MODEL TRAJ    ! Initial value problem
     Y=Y0 : X=1 : Z=0 : T=0 ! Initial conditions
     DT=0.25 : TF=1
     INITIATE ISIS; FOR DIFF;
~      EQUATIONS ZDOT/Z, XDOT/X, YDOT/Y;
~      OF T; STEP DT; TO TF
     INTEGRATE DIFF; BY ISIS
     Y1=Y ! Boundary condition
     TERMINATE DIFF
   END

   MODEL DIFF
     ZDOT=.5*Y**2+Y**2 ! Objective differential equation
     XDOT=-A*X+Y    ! State differential equation
     YDOT=-X+A*Y    ! Control differential equation
   END
```

Figure 8: FORTRAN CALCULUS example of an optimal design and control problem showing domains of underlying differential operations.

Differentiation within Integration

Figure 9 illustrates two kinds of integration processes in which differential arithmetic performs subprocesses for integration stepping. Figure 10 is a schematic illustration of the solution of implicit differential equations. This involves nesting a correlation problem within a simulation problem. Newton methods are used to solve for the implicit derivative variables (rates) of integration by finding the zeros of derivative constraints, and then these rates are integrated by ordinary numerical integration methods.

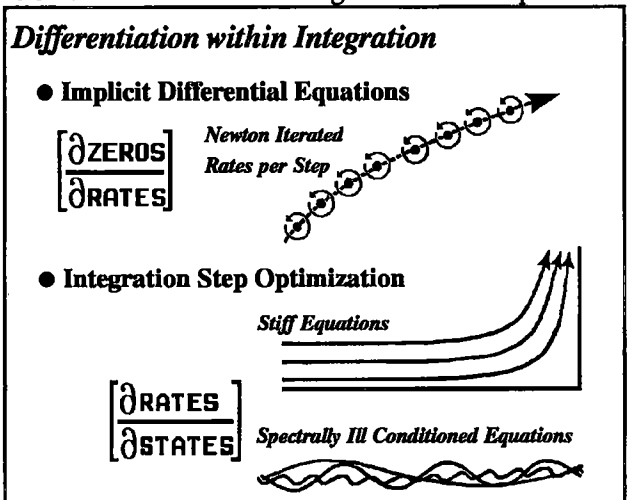


Figure 9: Differentiation within numerical integration is used to solve for implicit rates to be used during integration and to apply Jacobian-based methods to the optimization of integration step size.

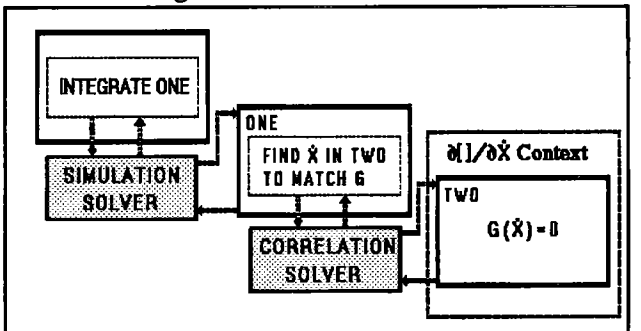


Figure 10: Hierarchic structure of problems involving implicit differential equations, illustrating differentiation within integration.

Figure 11 is a hierarchic FC example of this kind, solving a pair of implicit differential equations characterizing a bipropellant rocket with a longitudinal instability. In this example, the mechanism for sending parameters to solvers is a "CONTROLLER" referenced in the FIND statement. In this CONTROLLER the parameter SUMMARY is set to zero to suppress the normal AJAX summary report generation.

Implicit Differential Equations:

$$\dot{x}+3.2(1+e^{-t/10}\sin\pi t)\sqrt{1-(\dot{x}+\dot{y})(1.15+57.5/(20000+x+y)} = 0$$

$$\dot{y}+1.6(1+e^{-t/10}\sin\pi t)\sqrt{1-(\dot{x}+\dot{y})(1.15+36.2/(20000+x+y)} = 0$$

```
GLOBAL ALL
  PROBLEM IMPDES
    X=14000 : Y=7000     ! Initial Conditions
    XDOT=-50 : YDOT=-25  ! Initial Rate Guesses
    T=0 : DT=.5 : TP=2 : TF=TP
    INITIATE JANUS; FOR IDEQ EQUATIONS;
~     XDOT/X, YDOT/Y; OF T; STEP DT; TO TF
    PRINT *,'    TIME        XDOT        X'
~         ,'    YDOT        Y'
    DO WHILE (TF.LE.50)
      INTEGRATE IDEQ; BY JANUS
      PRINT '(5E15.6)',T,XDOT,X,YDOTT,Y
      TF=TF+TP
    END DO
  END

  MODEL IDEQ  !  Implicit Differential Equations
    FIND XDOT,YDOT; IN IRATE;
~     BY AJAX(ACON); TO MATCH GX,GY
  END

  CONTROLLER ACON(AJAX)
    SUMMARY=0
  END

  MODEL IRATE  !  Implicit Rate Equations
    GX=XDOT+3.2*SQRT(1-(XDOT+YDOT)*
~       (1.15+57.5/(20000+X+Y)))*(1+.1*EXP(-T/10)*SIN(1.5708*T))
    GY=YDOT+1.6*SQRT(1-(XDOT+YDOT)*
~       (1.15+36.2/(20000+X+Y)))*(1+.1*EXP(-T/10)*SIN(1.5708*T))
  END
```

Figure 11: Hierarchic program to solve a pair of implicit differential equations, structured as a correlation problem inside a simulation problem

CASE & AI Program Synthesis

The mathematical modeling taxonomy of synthetic calculus provides a handful of very general problem stereotypes that can be mechanized via CASE and AI techniques to produce mathematical software from non-mathematical specifications or from mathematical protomodels. Such stereotypes have long existed in MIS applications, and have led to the evolution of a class of program generation languages known as 4GLs, enabling a much higher order of software development in commercial fields than in the more highly diversified scientific fields. The synthetic calculus languages can similarly become the foundation for a broad class of scientific program generators, including knowledge-based modeling languages, rule-based expert systems, and extensive engineering modeling libraries.

Transformational Programming

The most promising avenue of development will likely follow the concepts of *transformational programming*, see [17]. Transformational programming is a method of incremental program construction by successive modifications using transformation rules. The basic unit of knowledge in the transformational approach is a transformation rule, which is a partial mapping from programs to programs. Transformational rules are either procedures or production rules that transform a pattern-matching part of a program into another pattern such as a synthetic calculus problem stereotype.

Much experience in the application of PROSE to protomodels developed by non-programmers coincided with the transformational approach. The assistance to the non-programmers by PROSE users, knowledgeable in the synthetic calculus stereotypes, resulted in very rapid prototyping of problem-solving programs by merely casting the protomodel equations into one of the problem stereotypes. Immediate insight, gained from trial runs, into the mathematical behavior of the models, would result in enlightened changes to rapidly evolve successful programs. The interchangability of the various solvers within a stereotype permitted extensive investigation of a variety of algorithms in a very short time, greatly accelerating insight gain and model validation.

Little if any of the knowledge applied by the PROSE users in casting the protomodels into stereotyped programs was of the intuitive kind. It was almost exclusively the recognition of stereotypical mathematical patterns in the protomodel formulas - an intelligence process that could be easily automated via production rules and inference algorithms in commonly available AI tools such as CLIPS [18] and Icon [19].

Migration, Promotion & Reusability

The large inventories of engineering application software in FORTRAN, BASIC, and other scientific languages constitutes a fertile forest of raw material for building new engineering optimization applications. The extraction of engineering simulation models and recasting them into optimization problem stereotypes is a low technology process. If performed by professional programmers coincident with the migration from large computers to workstations, this could easily lead to the by-production of automated transforma-

tional programming systems and the evolution of standardized engineering model bases.

Since the engineering must be extricated from numerical solution algorithms in this process, it is logical to recast it into library modules organized as engineering modeling utilities. Simulation models, the purest form of engineering specifications in existing software, can be automatically rendered into optimization models via differential arithmetic. Consequently, the extracted model bases will be cleansed of the implementation dependences that have prevented the reusability of software in the past. Although object-oriented languages and ADA were developed with the intention of localizing changes and preserving reusability, the lack of differential arithmetic in these languages has left vital and pervasive model-dependent mathematical approximations, i.e. partial derivatives, as programming work and impediments to reusability.

The use of the synthetic calculus languages in the coming migrations will at once achieve several of the major goals of software engineering and application engineering, namely:

- A simplified process of migration that preserves and enhances past software investments;
- The mathematical promotion of application engineering from trial-&-error oriented simulation methods to closed form optimization methods; and
- The integration of reusable knowledge bases of standardized engineering models as the basis for new applications as well as a resynthesis of existing ones.

The net result of the migration process will be a renaissance of scientific computing using integrated knowledge bases as a standard foundation for a new generation of quantitative engineering software.

Language Implementations

The synthetic calculus environment is a mathematical software system which functions as a procedural language extension, rendering the procedural language into a higher-order calculus-based language. It consists of a precompiler and a large run-time library of higher mathematical solvers and utilities. The precompiler translates the higher-order language program into its procedural language counterpart, with calls the library procedures. For differentiation, the real-mode arithmetic is translated into "overloaded" form for access to differential arithmetic. This occurs in MODEL procedures, and their function counterparts, FMODELs. Their real variable assignment statements are translated into subroutine calls, in which the right-hand-side arithmetic operator and intrinsic function calls are replaced by calls to differential arithmetic operator functions.

FORTRAN CALCULUS is the first of the new synthetic calculus environments. It is currently available as an extension to FORTRAN for 32-bit and 64-bit machines. BASIC CALCULUS and PASCAL CALCULUS environments are currently in development.

Appendix: Object Oriented Concepts

The concepts of object oriented programming are useful in clarifying the mathematical dynamics of synthetic calculus. In synthetic calculus the fundamental *class* is the *differential coordinate system*. Objects within this class are *points* in the coordinate space. A coordinate system class is created at the onset of differentiation by a coordinate system *constructor* which:

(1) defines the class structure according to the number of coordinate variables, consisting of the variables, and according to the order of differentiation, their associated gradient vectors and Hessian triangular matrices (if second order, etc.);

(2) stores the *coordinate origin object* of this structure - the set of values of the independent variables that become the coordinates, their gradient values, (unity of each coordinate with respect to itself and zero otherwise), and zeros in the triangular Hessian matrix (if present); and

(3) *overloads* the arithmetic operators and the associated transcendental functions to include differential arithmetic so that each subsequent variable (point) object *inherits* the coordinate system and is differentiated with respect to it. The class of the point objects subsequently computed is therefore a *derived class* which inherits the coordinate system, and contains derivative values with respect to the coordinate origin object.

The termination of differentiation amounts to the destruction of the coordinate system class, and the *disinheritance* of all of its objects, transforming them into ordinary variable objects. This is performed by a coordinate destructor, which also *dis-overloads* the arithmetic operators and transcendental functions, so the ordinary computation resumes. Sometimes, however, this destruction is a temporary step required to change to a different coordinate system, and the *activation object* of the coordinate class (the aggregate of its variable objects) is saved before its active coordinate class is destroyed and a new active coordinate class is

created (or reinstated). This process of changing differential coordinate systems, saving and restoring activation objects is usually associated with hierarchic nesting of inverse mathematical problems and the transformation of partial derivatives from one coordinate system to another.

Since synthetic calculus is a technology of solving inverse mathematical problems, its primary process is finding the coordinates (i.e., the values of the independent variables) that represents the solution of an inverse problem (creating an activation object of the coordinate system class in which certain of its objects satisfy certain solution criteria, i.e. they meet optimality conditions or satisfy constraints or both).

Synthetic calculus introduces the *solver* class, analogous to the Smalltalk *method* class as a higher-order operator class (control) and the *model* class as its higher-order operand class (prediction). The model is the problem definition, and the solver is its method of solution. The importance of these classes is that they are orthogonal to each other, and therefore that objects within them are mutually interchangeable, hence reusable. This othogonality exists by virtue of the coordinate system class which extracts the information needed by the solver class (partial derivative values from the model class) by virtue of operator and function overloading, so that the process of producing the information (differential arithmetic) being a private part of the machine class is hidden from both the model class and the solver class but is available to both via public interface procedures of the coordinate system class.

For simple problems in synthetic calculus, only models are differentiated, and the differential coordinate system is created by the solver before it calls the model, so that the model inherits the coordinate system, without being aware of it, i.e. its operators are overloaded, but this is hidden from the syntax of the model. This has the effect of making the engineering of the model orthogonal to the mathematics of solution. But the model can also create a differential coordinate system or can call a solver that does, giving rise to hierarchic problems. If the model which calls a solver has already inherited a differential coordinate system; then if a new coordinate system is created by the called solver, it must first save the activation object of the original coordinate system, and later perform a coordinate transformation when it completes its task to transform its results from its created coordinate system to the original coordinate system. Otherwise the solver would also inherit the original coordinate system and be differentiated with respect to it. The latter often occurs in the solution of boundary values problems, and the former also occurs in a higher nest in the solution of optimal design and control problems.

REFERENCES

1. 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.

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

3. Iri, M.: "Simultaneous Computation of Functions, Partial Derivatives and Estimates of Rounding Errors - Complexity and Practicality", Japan Journal of Applied Mathematics 1 (1984), 223-252

4. Kagiwada, H., Kalaba, R., Rosakhoo, N., and Springarn, K.: Numerical Derivatives and Nonlinear Analysis, Plenum Press, New York, 1986

5. Kedem, G.: "Automatic Differentiation of Computer Programs", ACM TOMS, June 1980

6. Krinsky, B. and Thames, J.M.: "The Structure of Synthetic Calculus", Proceedings of the Int'l Workshop on High-Level Computer Architecture, Los Angeles, 1984

7. Pfeiffer, F.: "Automatic Differentiation in PROSE", SIGNUM Newsletter Nov. 22, 1987

8. Rall, L.B.: "The Arithmetic of Differentiation", Mathematics Magazine, Vol. 59, No. 5, December 1986

9. Rall, L.B.: "Differentiation in PASCAL-SC": Type Gradient, ACM TOMS, June 1984

10. Rall, L.B.: "Differentiation and the Generation of Taylor Coefficients in PASCAL-SC". In a New Approach to Scientific Computation, U.W. Kulisch and W.L. Miranker, Eds., Academic Press, New York, 1983

11. Rall, L.B.: Automatic Differentiation: Techniques and Applications. Lecture Notes in Computer Science, Vol. 120,k, Springer-Verlag, New York, 1981

12. Thames, J.M.: "The Evolution of Synthetic Calculus: A Mathematical Technology for Advanced Architecture", Proceedings of the Int'l Workshop on High-Level Language computer Architecture Fort Lauderdale, Florida, 1982

13. Thames, J.M.: "Computing in Calculus", Research/Development 26,5 (May 1975)

14. Thames, J.M.: "SLANG, A Problem-Solving Language for Continuous-Model Simulation and Optimization", Proceedings, ACM 24th National Conf. (Dec. 1969)

15. PROSE - A General Purpose Higher Level Language, Calculus Applications Guide, Control Data Corp. Cybernet Services, Pub. No. 84000170 Rev. A (Jan. 1977)

16. PROSE - A General Purpose Higher Level Language, Calculus Reference Manual, Control Data Corp. Cybernet Services, Pub. No. 84003200 Rev. B (Jan 1977)

17. Barr, A., Cohen, P.R., and Feigenbaum, E.A., The Handbook of Artificial Intelligence, Volume IV, Addison Wesley, 1989.

18. CLIPS Expert System Tool, NASA Mission Planning & Analysis Division, Johnson Spaceflight Center, COSMIC Progrom MSC-21208

19. Griswold, R.E., and Griswold, M.T., The Implementation of the Icon Programming Language, Princeton University Press, 1986.