UC-350

LBL-26262

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

APPLIED SCIENCE
DIVISION

Presented at the USER-1 Conference,
Ostend, Belgium, September 6–8, 1988

Dynamic Extension of the Simulation Problem Analysis Kernel (SPANK)

E.F. Sowell and W.F. Buhl

July 1988

APPLIED SCIENCE
DIVISION

Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

Proceedings of the USER-1 Conference
Ostend, Belgium September 1988

LBL-26262

Dynamic Extension of the Simulation Problem Analysis Kernel (SPANK)

E. F. Sowell
Department of Computer Science
California State University Fullerton
Fullerton, California

W. F. Buhl
Simulation Research Group
Applied Science Division
Lawrence Berkeley Laboratory
Berkeley, California

July 15, 1988

Abstract

The Simulation Problem Analysis Kernel (SPANK) is an object-oriented simulation environment for general simulation purposes. Among its unique features is use of the directed graph as the primary data structure, rather than the matrix. This allows straightforward use of graph algorithms for matching variables and equations, and reducing the problem graph for efficient numerical solution. The original prototype implementation demonstrated the principles for systems of algebraic equations, allowing simulation of steady-state, nonlinear systems (Sowell 1986). This paper describes how the same principles can be extended to include dynamic objects, allowing simulation of general dynamic systems. The theory is developed and an implementation is described. An example is taken from the field of building energy system simulation.

This work was supported by the Assistant Secretary for Conservation and Renewable Energy, Office of Building and Community Systems, Building Systems Division of the U. S. Department of Energy under Contract No. DE-AC03-76SF00098.

Review of SPANK Methodology

The Simulation Problem Analysis Kernel (SPANK) allows the user to describe problems in terms of user-defined and/or library objects, interconnected as specified with a simple network specification language (Sowell 1986). The objects consist of interface variables and a mathematical model expressing a relationship among these variables. Importantly, the interface variables carry no input/output specification; rather, all viable inverses of the equation are defined, and the SPANK matching algorithm selects an appropriate inverse for each object that will be consistent with the needs of the overall system solution procedure. This matching results in a one-to-one correspondence between equations and problem variables.

The idea of specifying objects without *a priori* identification of the input/output variables is central to the SPANK methodology. For one thing, it allows the above-described matching process, which ensures a formula for every variable and a variable for every formula. Without such a matching, automatic selection of iteration variables and algorithm generation is, in general, not possible. Moreover, omitting input/output designation makes object models more transportable. For example, if a particular physical process involves three variables, x,y, and z, three different models are possible if we insist on *a priori* identification of input/output variables, whereas there is a single model if we do not.

After the matching, the overall solution procedure is determined automatically by analysis of the problem represented as a directed graph. In this graph, the vertices are the objects, which may be viewed either as equations or the associated "output" variables, since there is a one-to-one correspondence. Viewing the vertices as equations, the edges represent the output variables, which become inputs to other vertices. In the analysis of this graph, the presence of possible cycles is detected, and a small cutset is identified. This minimizes the number of variables that need to be iterated in the numerical phase of the solution, and consequently, minimizes the size of the Jacobian. In a second step, the cut variables allow construction of an acyclic directed graph that represents the direct computation of each cut variable given only the estimated values of cut variables and the exogenous variables. Obviously, the estimated values together with the calculated values can be used to calculate new estimates using a variety of nonlinear solution algorithms. In the current implementation, the Newton-Raphson algorithm is employed.

The acyclic directed graph can be viewed as a dataflow model of the nonlinear core of the problem. In the current implementation, this graph is used directly in the numerical phase of the solution, following the dataflow computation model. That is, at any time there may be one or more vertices with all inputs known; these are said to be "fireable". Initially, these will be those vertices with only exogenous and cut variable estimates as inputs. The solution process employs a stack that always contains all fireable vertices. The top of the stack is repeatedly popped and fired. Firing means that the method of the vertex object is executed, thereby determining its output variable. The output variable is put into the data structure, and any adjacent vertices are updated with regard to their fireability. Thus each fired vertex may result in one or more new vertices being placed on the stack. The process is continued until the stack

In the object-oriented language (OOL) terminology, we can think of the selected inverse as the "method" for the object. During the numerical phase, the associated interface variable will be calculated by means of this inverse, and then may appropriately be thought of as an "output" variable.

becomes empty, indicating that all vertices have been fired and all calculated cut variable values are available. We say that the estimates have been "pushed through" the dataflow graph, resulting in calculated values of the same variables. The Jacobian is then evaluated (numerically in the present implementation), and the Newton-Raphson formula is used to calculate new estimates of the cut variables.

Although not done in the current implementation, the acyclic directed graph could alternately be used to construct a symbolic form of the solution algorithm, e.g., a compilable module.

The current SPANK implementation solves only algebraic equations, and therefore solves only static problems. In most dynamic simulations there are many nonlinear algebraic equations in addition to the differential equations, so that solution of systems of nonlinear algebraic equations is the central numerical task in dynamic simulation. In what follows we show that the dynamic part of the simulation also reduces to coupled, nonlinear, algebraic equations, so that the SPANK methodology can be directly applied to dynamic as well as static problems.

Dynamic Problems

A dynamic system model with N variables x_i can be described in general by m algebraic equations and n=N—m differential equations, i.e.,

$$\begin{aligned} 0 &= f_1(x_1,x_2,\cdots,x_N) \\ 0 &= f_2(x_1,x_2,\cdots,x_N) \\ &\cdots \\ &\cdots \\ 0 &= f_m(x_1,x_2,\cdots,x_N) \end{aligned} \tag{1}$$

$$\begin{aligned} y_1 &= g_1(x_1,x_2,\cdots,x_N) \\ y_2 &= g_2(x_1,x_2,\cdots,x_N) \\ &\cdots \\ &\cdots \\ y_n &= g_n(x_1,x_2,\cdots,x_N) \end{aligned} \tag{2}$$

where it is understood that the y_i are derivatives of a subset of the variables, i.e.,

$$y_i = dx_{J(i)}/dt \ ; \ i = 1..n \tag{3}$$

Here J(i) is the x index of y_i. Thus the x_J are called the dynamic variables, and the y variables are their derivatives. (The x_J variables are sometimes called state variables.)

In the most general case, the algebraic and differential equations are coupled, so that the algebraic set must be solved in order to evaluate the derivative formulas g_i.

The numerical solution of such systems is accomplished by choosing a formula for calculation of the dynamic variables at the next time point, often called an integrating formula. The two major classes of integrating formulas are "open", which are functions only of past values and derivatives, and "closed" formulas which involve values and derivatives at future time points as well. Here we shall focus on the predictor-corrector methods (CONTE 1980) which employ closed formulas because these are the most difficult to implement; modification of the technique to use the simpler open formulas is staightforward.

The fourth-order Milne corrector formula can be written as

$$x_{I(i),k+1} = a_1\, x_{I(i),k-1} + dt(\, a_2\, y_{i,k+1} + a_3\, y_{i,k} + a_4\, y_{i,k-1}\,) \tag{4}$$

where I(i) is the y index of x_i and k indicates the time step index. Thus the (k+1)st value is based on the value at k−1, and on the derivatives at k+1, k, and k−1. Because the (k+1)st derivative is present, and these derivatives involve (k+1)st variable values, these formulas are implicit in the (k+1)st variables.

It will be observed that by introduction of the y variables we have added n new variables to the set of system variables x, so that there are N+n variables in the augmented set. There are also N+n equations, namely, m algebraic (Eq(1)), N−m derivative equations (Eq(2)), and n integration formulas (Eq(4)). If we take the view that the x_i in Eq(1) are the (k+1)st values, these can be solved simultaneously, yielding the (k+1)st values of all variables x_i and all derivatives y_i. Afterwards, time is advanced by dt, and the process is repeated. A suitable predictor formula would be used to calculate the starting x values for the next time step.

The above procedure neglects the start-up process. Typically, a Runge-Kutta technique could be employed to get the predictor-corrector process started. In many problems, however, it is sufficient to assume that the dynamic variables have been at their initial conditions for the required number of prior time steps, and that all prior derivatives are zero. Then no special start-up procedure is required. This is the approach being taken in the prototype dynamic SPANK now under development.

It is clear from the above that the central problem in dynamic simulation is solution of simultaneous algebraic equations. The integrating formulas and derivative formulas simply augment the algebraic set. When the integrating formulas are of the closed type, they are implicit in the simulation variables. This is not much of an added burden, however, since the nonlinear nature of the algebraic set requires iterative solution regardless.

Using SPANK Methodology to Solve Dynamic Problems

An example will show how the SPANK methodology applies to problems as described above.

Figure 1 shows a simplified diagram of a room in which the ceiling height is negligibly small relative to the other room dimensions. The floor is massive, but the ceiling closure has negligible mass. The room air is also considered to be massless. The floor and ceiling are considered to be black radiators, and also convect to the room air. We consider the problem of finding the dynamic variation of floor and ceiling temperature, starting from some initial floor temperature $T_2(0)$ while air temperature is held constant at $T_3 \neq T_2(0)$.

Conservation of energy and radiant flux at model vertices yields the system equations, Eq(5)-Eq(7). These are augmented by the Milne fourth-order corrector formula, Eq(8).*

$$\text{E1:}\ \ 0 = -hT_{1,k+1} + hT_{3,k+1} - \sigma T^4_{1,k+1} + \sigma T^4_{2,k+1} \tag{5}$$

$$\text{E2:}\ \ 0 = q^o_{3,k+1} + hT_{1,k+1} + hT_{2,k+1} - 2hT_{3,k+1} \tag{6}$$

$$\text{E3:}\ \ \dot{T}_{2,k+1} = \left[-hT_{2,k+1} + hT_{3,k+1} + \sigma T^4_{1,k+1} - \sigma T^4_{2,k+1} \right] / \alpha \tag{7}$$

$$\text{E4:}\ \ T_{2,k+1} = a_1 T_{2,k-1} + \Delta t \left(a_2 \dot{T}_{2,k+1} + a_3 \dot{T}_{2,k} + a_4 \dot{T}_{2,k-1} \right) \tag{8}$$

where

T_i	=	node temperature (K)
q_i^o	=	source heat at node i (W/m^2)
h	=	convective heat transfer coefficient (W/m^2-K)
α	=	floor thermal capacitance (J/m^2-K)
σ	=	Stefan-Boltzmann constant (5.67x10^{-8} W/m^2–K^4)
a_1 .. a_4	=	integrating formula coefficients
dt	=	time step (sec)

The variables are:

T_1	=	ceiling temperature (K)
T_2	=	floor temperature (K)
$\dot{T}_2$	=	derivative of floor temperature (K/sec)
q_3^o	=	heat added/removed at air node (W)

Figure 2 shows the bipartite graph for matching variables to equations. The upper vertices represent the equations, and the lower represent the variables. An edge is shown from an upper vertex to each lower vertex whose variable is in the equation. In the SPANK implementation, matching is done automatically. Here we do it manually to demonstrate the technique. Following conventional practice, we match T_2 to E_4. (This ensures that the integrating formula is used for integration instead of being solved for the derivative!) Once that decision is made, it is imperative to match E_3 to $\dot{T}_2$. Also, we have no choice but to match E_2 to q_3^o. Finally, by elimination we are left with E_1 matched with T_1. Thus we see that the matching is unique in this example; in general, matchings are not unique, and an arbitrary decision is made by the algorithm. (SPANK could be improved by devising a weighted matching algorithm, perhaps using numerical

* The predictor formula acts only to provide a starting value for the solution at the new time. It is not part of the system of equations to be solved simultaneously.

information from the problem to set the weights.) The final matching is shown by the heavy lines in the diagram. Based on this matching, the formulas for each variable are:

$$T_{1,k+1} = T_{3,k+1} - \left(\sigma/h\right) T_{1,k+1}^4 + \left(\sigma/h\right) T_{2,k+1}^4 \tag{9}$$

$$q_{3,k+1}^{o} = -hT_{1,k+1} - hT_{2,k+1} + 2hT_{3,k+1} \tag{10}$$

$$\dot{T}_{2,k+1} = \left(-hT_{2,k+1} + hT_{3,k+1} + \sigma T_{1,k+1}^4 - \sigma T_{2,k+1}^4 \right) / \alpha \tag{11}$$

$$T_{2,k+1} = a_1 \, T_{2,k-1} + \Delta t \left(a_2 \dot{T}_{2,k+1} + a_3 \dot{T}_{2,k} + a_4 \dot{T}_{2,k-1} \right) \tag{12}$$

Figure 3 shows the directed graph for the problem after the matching. Note that the graph is cyclic, due to the implicit formula for T_1 (due in turn to the 4th-power radiation model), and the integration formula. The SPANK implementation would apply a cutset algorithm to this graph, but due to its simplicity we can see that a minimum cutset (breaking all cycles) is $\{T_1, T_2\}$. Alternately, $\{T_1, \dot{T}_2\}$ could be used. Arbitrarily selecting $\{T_1, T_2\}$ leads to the acyclic directed graph of Fig. 4, where we have added vertices for the cutset variables, T_1^g and T_2^g. This is the "dataflow" graph upon which the numerical solution procedure is based. T_1^g and T_2^g are guessed, whereupon T_1 and $\dot{T}_2$ can be calculated. This allows calculation of T_2, and finally q_3^o. Having calculated values of T_1 and T_2, we can apply the Newton-Raphson formula to get new guess values. Thus the dataflow graph guides the iterative solution process.

Time Loop Algorithm

The time loop algorithm for a program implementing SPANK is shown in Fig. 5. It is seen to be exactly like a normal predictor-corrector implementation. The only differences are that the flow graph is created prior to starting the time loop, and the corrector step is in fact an invocation of the static SPANK solver.

It will be noted that the time loop suggested here has a single call to the solver during the time step. This limits the current dynamic SPANK to predictor-corrector integration methods or others of similar structure. Methods that require multiple evaluation of the derivative functions during a time step, such as Runge-Kutta, are not allowed; future developments may allow removal of this limitation. Moreover, when the basic methodology is predictor-corrector, all of the problems as well as the virtues of that method are retained. For example, the well-known potential instability of the Milne method will still be a consideration if these are the formulas used for the integrator object. SPANK aims to improve the manner in which dynamic simulation problems are specified and converted to a form suitable for solution, and not necessarily the underlying numerical methods.

Extensions to SPANK Syntax

When problems such as those described above are examined, it is seen that the only new object needed is an "integrator" with the corrector formula as its method, and having two interface variables, namely the dynamic variable and its derivative. For user convenience, however, we define a macro integrator object (see SOWELL 1986) that embodies the predictor object as well as the corrector object. The predictor object is not placed in the problem graph. Instead, it is used in the predictor step of the time loop to initialize the dynamic variables prior to iterative solution with the dataflow graph. Figure 6 shows the example problem definition file.

SPANK input for the Example

Figure 6 shows a block diagram of the example problem. There is an object for each of the physical objects in the problem, namely the ceiling, floor and air node. Additionally, there are a predictor object and a corrector object, which together represent the integrating formulas. The interfaces of the objects are interconnected so as to indicate equivalence with the problem variables. The interfaces to be specified as problem constants are also connected and labeled with the constant symbols. This diagram bears a strong resemblance to the physical problem and is used to guide the user in preparation of the SPANK input.

Figure 7 shows the example coded in the Network Specification Language (NSL). The object definitions for the problem are given in Figure 8. In the interest of brevity, we show only the supporting C functions that are selected by the matching, Figure 9. In practice, one would provide all functions referenced in the object definitions.

Referring to Fig. 7, we see that five objects are declared. The ceiling (ceil) is defined as an instance of the "massless" class of objects. From Figs. 7 and 8, it can be seen that such objects enforce equation Eq.(1). Similarly, the floor is declared to be of the "massive" class, enforcing E3, and the air node is of the "air" class, enforcing E2. Note that all inverses of the these equations are included in the object definitions, so that the matching process is free to make a suitable choice for each variable. There is a LINK statement for each problem variable, and an INPUT statement for each problem constant. Both the LINK and INPUT statements have a symbol preceding the parenthetic expression that becomes the symbol for the problem variable or constant. (Values for constants are provided at run time.) In the parentheses is placed a list of all interfaces to which the variable or constant applies. The form of these interface specifications is

object_instance.internal_interface_name.

Thus, ceil.h refers to the h interface variable in the ceiling object instance of the class massless.

The dynamic nature of the problem is indicated by the presence of the predictor and corrector objects, milne_4p and milne_4c, and the "history" object. History is a SPANK predefined class that provides storage for histories of dynamic variables, as needed by integrating formulas, e.g. Eq(5). The predictor and corrector object classes are user-defined, as exemplified in in Figure 8, with the supporting C functions in Figure 9. One limitation in the version currently being implemented is that the predictor and corrector formulas can only employ historical data stored in the predefined history class. In a more ambitious implementation, the user should be allowed to define special history classes to allow a wider range of integrating formulas.

The C functions in Fig. 9 show that all arguments are passed in an array of unions, predefined in SPANK as type VAL. This is in contrast to the static implementation which passes an array of doubles. Unions (called variant records in Pascal) allow the data item to be of various types. This was a necessary enhancement because dynamic SPANK must pass pointers to dynamic variable histories as well as doubles. The VAL union allows doubles (dval) integers (ival) dynamic pointers (dp), and character pointers (cval).

Status and Future Work

The implementation described above is expected to be operational sometime this year. When completed, a series of trial problems from various disciplines will be devised to explore its capabilities and limitations. Based on the outcome of these tests, new features will probably be suggested. Already mentioned is the need for user-defined history objects. We also see the need for integration methods other than predictor-corrector. In a completely flexible system, the user should be allowed to define the overall strategy for time advancement in an object-oriented manner.

Acknowledgements

This work was undertaken as part of the international Energy Kernel System development effort. Contributing to the ideas expressed here were members of the Simulation Research Group at LBL, including Fred Winkelmann and Ender Erdem, and several individuals from other collaborating institutions in the U.S. and Europe.

References

CONTE 1980 **ELEMENTARY NUMERICAL ANALYSIS: AN ALGORITHMIC APPROACH, by S.D. Conte and C. DeBoor, McGraw–Hill, NY, 3rd Edition, 1980.**

SOWELL 1986 PROTOTYPE OBJECT-BASED SYSTEM FOR HVAC SIMULATION, E.F. Sowell, W.F. Buhl, A.E. Erdem, and F.C. Winkelmann, presented at the Second International Conference on System Simulation in Buildings, Liege, Belgium, December 1986.

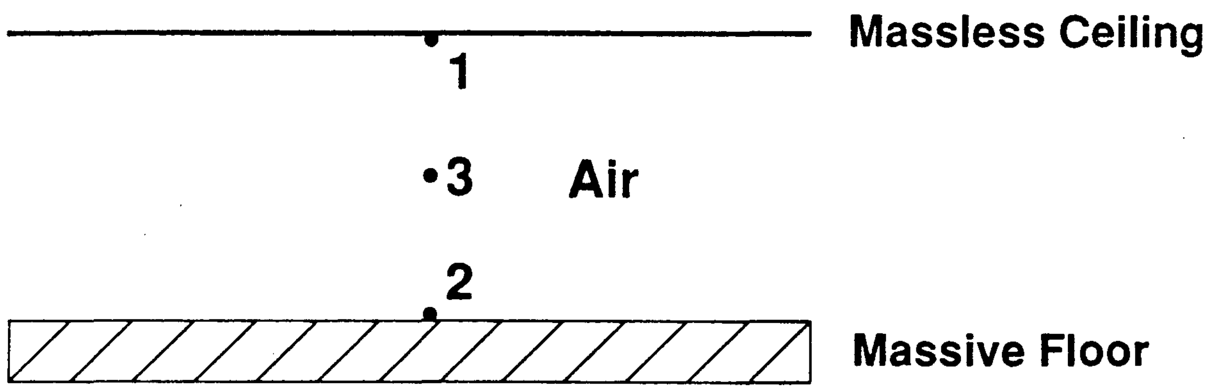

Figure 1: Physical System

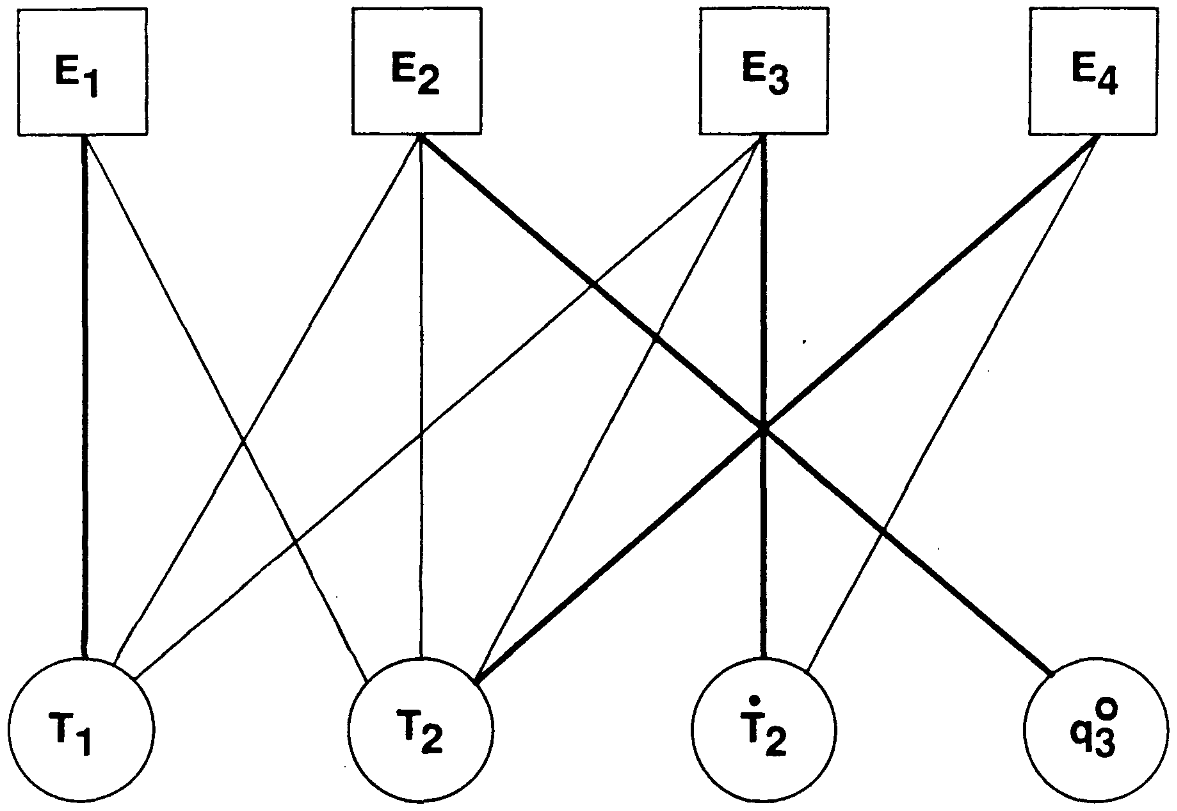

Figure 2: Matching Graph

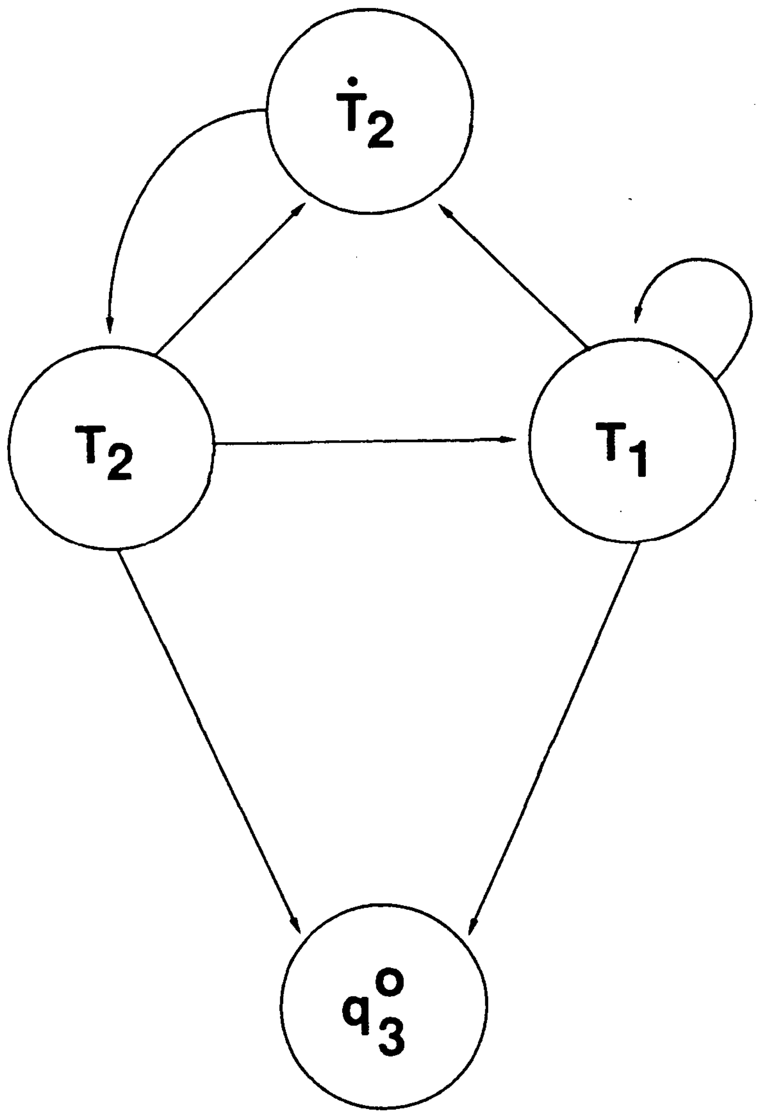

Figure 3: Directed Graph of Matched Equations

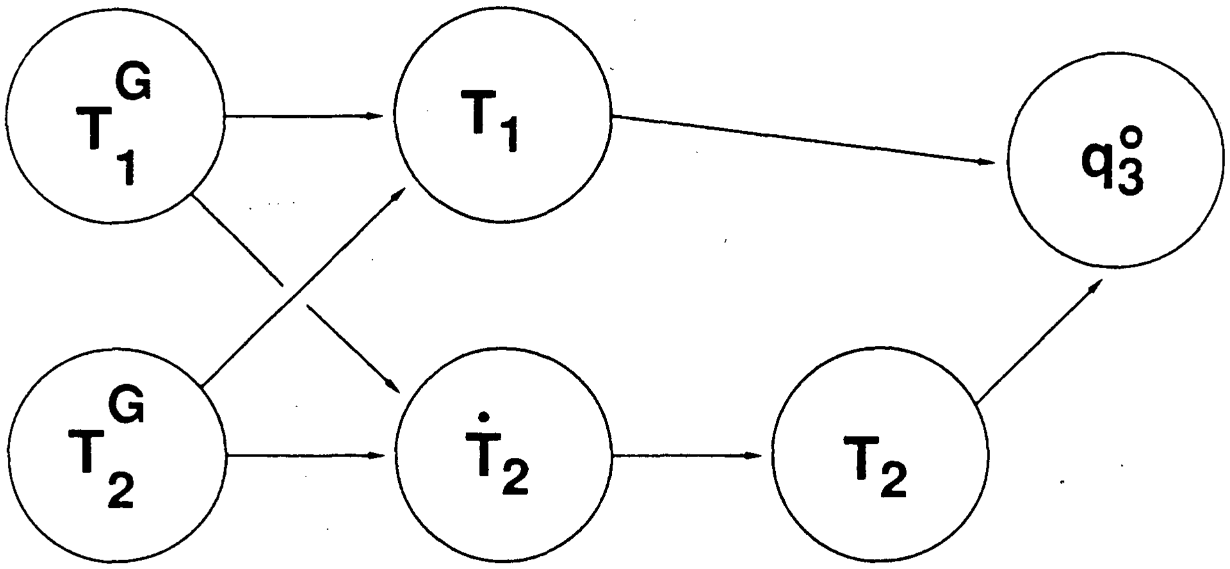

Figure 4: Dataflow Graph

```
Initialize all arrays
Set up a flow_graph for the problem
Set starting guess and limits for break variables
Initialize and write initial-state reports
/* Loop over the time steps: */
    while (t ≤ tlimit) {
        Predictor Step
        Increment time and related things
        Corrector  Step (invoke SPANK solver)
        Update the history of dynamic variables
        Write time-step reports
    } /* End of time loop  */
```

Figure 5: Algorithm for Time Loop

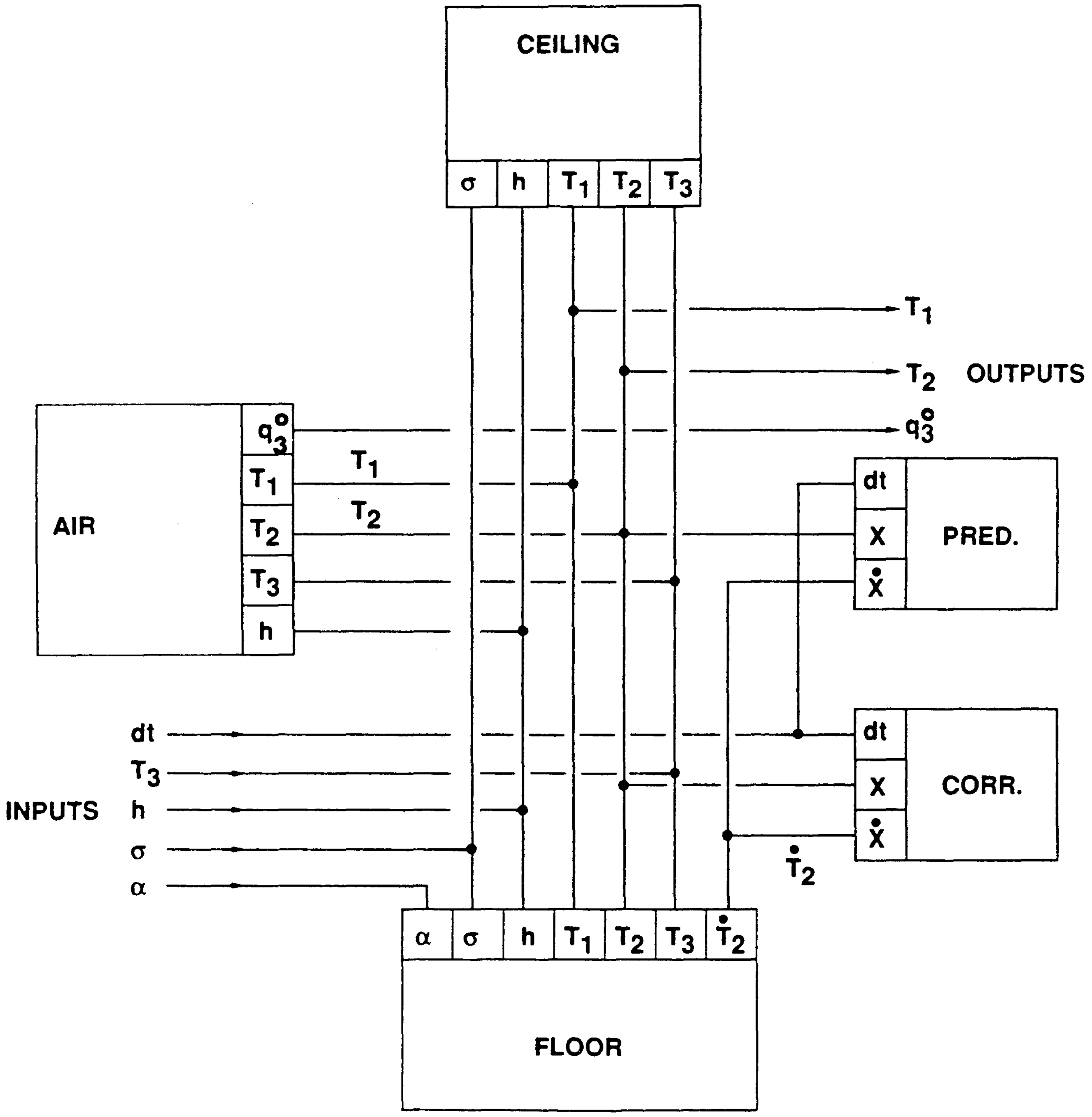

Figure 6: Block Diagram

```
/*              Dynamic Test Problem 2
 *         Two Surface Room with Mass in Floor
 *              two_surf_rm.ps
 */
declare  massless ceil;          /* Massless ceiling */
declare  massive   floor;         /* Massive floor slab */
declare  air  air;              /* Massless air */
declare  milne_4_p p;             /* Milne 4th order predictor*/
declare  milne_4_c c;            /* Milne 4th order corrector */

/* Constant inputs */

input h(ceil.h,floor.h,air.h)[WMK]   /* Film coefficient */
input T3(ceil.T3,floor.T3,air.T3)[K] /* Fixed air temperature*/
input alpha(floor.alpha)[generic]    /* Slab heat capacity   */
input dt(c.dt,p.dt)[TIME]           /* Time step */
input sigma(ceil.sigma,floor.sigma)[generic] /* Stefan-Boltzmann  */

/*Links*/

link T1(ceil.T1,floor.T1,air.T1)[K]        /* Ceiling temperature */
link T2(ceil.T2,floor.T2,air.T2,c.x,p.x)[K]  /* Floor temperature */
link T2_dot(floor.T2_dot,c.xdot)[generic]      /* Derivative of T2 */
link qo3(air.qo3)[W]                      /* Air heat rate */

/* History Object */

history T2_hist(c.x_hist,p.x_hist)         /*  History  of T2 */

/* End of Problem */
```

Figure 7: Problem Specification File

```
/*                    Massless Surface
 *                      massless.obj
 */
 define massless(T1,T2,T3,sigma,h)
 double T1[K],T2[K],T3[K],sigma[generic],h[WMK];
 {
      T1 = ceil_fun_T1(T1,T2,T3,sigma,h);
      T2 = ceil_fun_T2(T1,T2,T3,sigma,h);
      T3 = ceil_fun_T3(T1,T2,sigma,h);
 }

/*                    Massive Surface
 *                      massive.obj
 */
 define massive(T1,T2,T2_dot,T3,sigma,h,alpha)
 double T1[K],T2[K],T2_dot[generic],T3[K],sigma[generic],h[WMK],alpha[generic];
 {
      T1 = floor_fun_T1(T1,T2,T2_dot,T3,sigma,h,alpha);
      T2 = floor_fun_T2(T1,T2,T2_dot,T3,sigma,h,alpha);
      T2_dot = floor_fun_T2_dot(T1,T2,T3,sigma,h,alpha);
      T3 = floor_fun_T3(T1,T2,T2_dot,sigma,h,alpha);
 }

/*                    Air Node
 *                      air.obj
 */
 define air(T1,T2,T3,qo3,h)
 double T1[K],T2[K],T3[K],qo3[W],h[WMK];
 {
      T1 = air_fun_T1(T2,T3,qo3,h);
      T2 = air_fun_T1(T1,T3,qo3,h);
      T3 = air_fun_T3(T1,T2,qo3,h);
      qo3 = air_fun_qo3(T1,T2,T3,h);
 }

/*                 Milne 4th Order Predictor
 *                      milne_4_p.obj
 */
define_pre
milne_4_p ( x,dt,x_hist)
double x,dt[TIME];
DYNAMIC_PTR x_hist;
{
      x = milne_4_p(x_hist,dt);
}

/*                 Milne 4th Order Corrector
 *                      milne_4_c.obj
 */

define milne_4_c(x,xdot,dt,x_hist)
double x,xdot,dt[TIME];
DYNAMIC_PTR x_hist;
{
      x = milne_4_c(x_hist,xdot,dt);
}
```

Figure 8: Object Definitions

```
/*                      Air Q Function
 *                      air_fun_qo3.c
 */
#include "val.h"
#define T1 arg[0].dval
#define T2 arg[1].dval
#define T3 arg[2].dval
#define h arg[3].dval
#define qo3 result.dval
 VAL
 air_fun_qo3(arg)
 VAL arg[];
 {
      VAL result;
      qo3= h * (2.0*T3 -T1 - T2);
      return(result);
 }
/*                      Ceiling T1 Function
 *                      ceil_fun_T1.c
 */
#include "val.h"
#define T1 arg[0].dval
#define T2 arg[1].dval
#define T3 arg[2].dval
#define sigma arg[3].dval
#define h arg[4].dval
#define T1r result.dval
 VAL
 ceil_fun_T1(arg)
 VAL arg[];
 {
      VAL result;
      T1r = T3 - sigma*(T1*T1*T1*T1 - T2*T2*T2*T2)/h;
      return(result);
 }
/*                      Floor T2_dot Function
 *                      floor_fun_T2_dot.c
 */
#include "val.h"
#define T1 arg[0].dval
#define T2 arg[1].dval
#define T3 arg[2].dval
#define sigma arg[3].dval
#define h arg[4].dval
#define alpha arg[5].dval
#define T2_dot result.dval
 VAL
 floor_fun_T2_dot(arg)
 VAL arg[];
{
      VAL result;
      T2_dot = (h*(T3-T2) + sigma*(T1*T1*T1*T1-T2*T2*T2*T2))/alpha;
      return(result);
 }
/*                      Milne 4th Order Predictor Function
 *                           milne_4_p.c
 */
#include "val.h"
#define x_p (arg[0].dp)
#define dt arg[1].dval
#define tp11 result.dval
#define a1  1
#define a2  2.666667
#define a3  -1.333333
#define a4  2.666667
VAL
milne_4_p(arg)
VAL arg[];
{
    VAL result;
    tp11 =
 a1*x_p->tm3 + dt*(a2*x_p->dot + a3*x_p->dotm1 + a4*x_p->dotm2);
 x_p->tp1 = tp11 ;
 return(result);
}
/*                      Milne 4th Order Corrector
 *                           milne_4_c.c
 */
#include "val.h"
#define x_p (arg[0].dp)
#define xdot arg[1].dval
#define dt arg[2].dval
#define tp11 result.dval
#define a1  1
#define a2  .333333
#define a3  1.333333
#define a4  .333333
 VAL
 milne_4_c(arg)
 VAL arg[];
{
   VAL result;
   x_p->dotp1 = xdot ;
   tp11 =
    a1*x_p->tm1 + dt*(a2*x_p->dotp1 + a3*x_p->dot + a4*x_p->dotm1);
   x_p->tp1 = tp11;
   return(result);
 }
```

Figure 9: Function Definitions (Partial List)