2

Conf-901295--1

LBL-29419

Lawrence Berkeley Laboratory

UNIVERSITY OF CALIFORNIA

APPLIED SCIENCE DIVISION

Presented at the 3rd International Conference on System Simulation in Buildings, Liege, Belgium, December 3–5, 1990, and to be published in the Proceedings

The U.S./EKS: Advances in the SPANK-based Energy Kernel System

F. Buhl, E. Erdem, J.-M. Nataf, F.C. Winkelmann, M. Moshier, and E. Sowell

December 1990

APPLIED SCIENCE DIVISION

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

Proceedings of the Third International
Conference on System Simulation in Buildings
Liege, Belgium Dec 3-5, 1990

LBL--29419

DE91 012659

The U.S./EKS:

Advances in the SPANK-based Energy Kernel System

Fred Buhl, Ender Erdem, Jean-Michel Nataf and Frederick C. Winkelmann

Simulation Research Group
Lawrence Berkeley Laboratory
University of California
Berkeley, CA 94720

Michael A. Moshier

Program in Computing
University of California
Los Angeles, CA 90024

Edward F. Sowell

Computer Science Department
California State University at Fullerton
Fullerton, CA 92634

August 30, 1990 — Rev. December 21, 1990

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

The U.S. EKS:

Advances in the SPANK-based Energy Kernel System

Fred Buhl, Ender Erdem, Jean-Michel Nataf and Frederick C. Winkelmann
Simulation Research Group
Lawrence Berkeley Laboratory
University of California
Berkeley, CA 94720

Michael A. Moshier
Program in Computing
University of California at Los Angeles
Los Angeles, CA 90024

Edward F. Sowell
Department of Computer Science
California State University at Fullerton
Fullerton, CA 92634

August 30, 1990 (Rev. December 21, 1990)

Abstract

The Simulation Problem Analysis Kernel (SPANK) was originally described as a prototype Energy Kernel System in a paper presented at the Second International Conference on System Simulation in Buildings in 1986. Since that time, it has undergone several enhancements and has been integrated into a larger software system that may be more properly called a prototype Energy Kernel System for building energy analysis, EKS/US. Among the enhancements is the capability to simulate dynamic problems. Also, symbolic manipulation techniques are now used to generate objects and macro objects from equations expressed as text. Currently underway is the development of a graphical user interface. Newer developments include a reevaluation of the semantics of dynamic problem definition, which will ultimately result in much greater generality in user specification of numerical methods. This paper reports on these developments and indicates directions for future EKS/US development.

1. Introduction

Efforts were launched in 1985 to improve modeling tools for building energy systems [Hirsch 1985, LBL 1985, Clarke 1986]. In this connection, the name "Energy Kernel System" (EKS) was coined to refer to the envisioned software genre in which the basic elements (i.e., components or objects) were to be packaged along with tools required to assemble them into arbitrary building simulation programs. An outgrowth of that effort was the Simulation Problem Analysis Kernel (SPANK), comprising methodology for describing and solving equation systems such as those that arise in simulation problems. Although the EKS then had yet to be fully defined, SPANK was intended to lie along the EKS evolutionary path.

It soon became evident that the fundamental EKS ideas admit to at least two significantly different implementations. One of these, currently under development in the UK [Clarke 1987], defines objects to be modules of substantial size and complexity that may be extracted from existing software. The second possible implementation takes a lower-level view, with individual equations as atomic objects that are interconnected to form larger entities (macro objects), which in turn are interconnected to form simulation problems. SPANK has evolved into an EKS of the second kind, referred to in this paper as EKS/US to distinguish it from its UK counterpart.

Although ambitiously described in earlier papers as a prototype EKS, SPANK is really a software system for description and solution of general differential-algebraic equations, with one of many possible applications being to building energy system simulation. EKS/US is a more complete embodiment of the original EKS/US with SPANK at the nucleus providing the means for object interconnection and problem solution.

Figure 1 represents the overall organization of the EKS/US and shows the relationship between EKS/US and SPANK. The user interacts with the system in four basic ways: defining objects (e.g., component models); defining problems by linking objects together; specifying run-time data (e.g., coefficients, time-varying data); and specifying desired output. The objects are defined in text files, either as mathematical equations or as component models in Neutral Model Format [Sowell 1989]. These files are processed symbolically with programs written in MACSYMA, producing C language functions and objects that are stored in libraries. Problems are defined by interconnecting objects using the graphical user interface, producing a problem specification file in the Network Specification Language (NSL) [Anderson 1986]. The nucleus or kernel is the dynamic SPANK program system. It works from the NSL description, generating internal data structures based on graphs. Matching and reduction algorithms are employed with these graphs to automatically devise an efficient solution algorithm, producing an executable program for a particular problem. This program reads constant and time-varying data from files, producing the problem solution. The output processor reads the result file and generates graphical displays according to interactive user requests.

The basic ideas and theoretical development of SPANK were described by Sowell, Buhl, Erdem, and Winkelmann [1986], with extensions to include differential models reported later by Sowell and Buhl [1988]. Buhl, Sowell, and Nataf [1989] have expanded upon these ideas and analyzed their importance in relation to other simulation methodologies. More recently, the use of symbolic manipulation in connection with SPANK modeling has been demonstrated by Sowell, Nataf and Winkelmann [1990], and Moshier and Sowell [1990] have reported semantic extensions to allow more flexibility in describing dynamic models. Several application examples for SPANK have also been reported [Sowell 1986, Sowell 1988, Sowell 1990, Nataf 1990].

The current paper summarizes recent developments in SPANK and reports new work in progress on a graphical user interface. For the benefit of those unfamiliar with the earlier work, we begin with a brief review of the motivation and basic ideas behind the SPANK kernel.

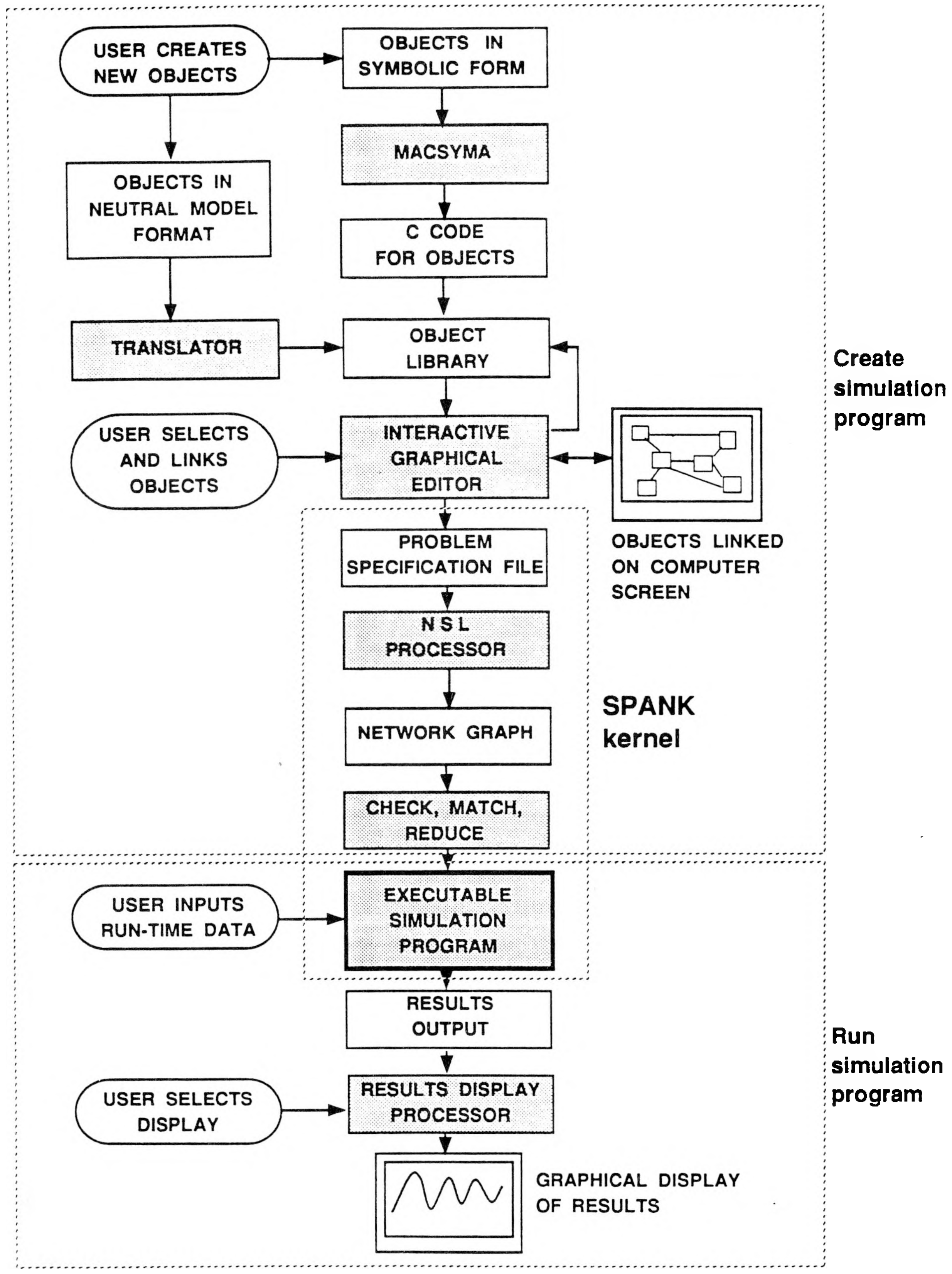

USER CREATES NEW OBJECTS
OBJECTS IN SYMBOLIC FORM
MACSYMA
OBJECTS IN NEUTRAL MODEL FORMAT
C CODE FOR OBJECTS
TRANSLATOR
OBJECT LIBRARY
Create simulation program
USER SELECTS AND LINKS OBJECTS
INTERACTIVE GRAPHICAL EDITOR
OBJECTS LINKED ON COMPUTER SCREEN
PROBLEM SPECIFICATION FILE
N S L PROCESSOR
NETWORK GRAPH
SPANK kernel
CHECK, MATCH, REDUCE
USER INPUTS RUN-TIME DATA
EXECUTABLE SIMULATION PROGRAM
RESULTS OUTPUT
Run simulation program
USER SELECTS DISPLAY
RESULTS DISPLAY PROCESSOR
GRAPHICAL DISPLAY OF RESULTS

Figure 1: Configuration of the U.S. Energy Kernel System. Shaded boxes are programs; unshaded boxes are files. Ovals show user actions.

2. Review of SPANK Principles

Simulation programs differ with regard to how problems are specified and with regard to how they are solved. SPANK differs from most currently available simulation programs in both respects. In this discussion we will attempt to distinguish between these differences.

2.1 Problem Specification

With respect to problem specification, SPANK is closest in kin to existing modular programs used in simulating building service systems, e.g., TRNSYS and HVACSIM+ [TRNSYS 1983, HVACSIM+ 1985]. That is, SPANK allows the user to interconnect component models in a flexible manner so that systems of arbitrary configuration may be defined. However, it goes beyond the existing modular programs in several ways. First, the atomic element in SPANK is an object representing a single equation, whereas other simulators use the subroutine, normally composed of several or many equations, as the smallest element available to the user. Larger SPANK elements, called macro objects, are definable by the user in terms of equation objects. One benefit of the SPANK approach is flexibility, because the user can define new macro objects as the need arises. Another benefit is code reuse, because the same equation object can be used in many macro objects. While it is true that TRNSYS or HVACSIM+ users (who also happen to be FORTRAN programmers) can define new component models, sometimes using all or part of existing ones, SPANK aims for a seamless simulation environment in which the means of object, macro object, and problem definition are identical, and code (i.e., existing objects and macro objects) can be reused without modification.

Another important aspect of SPANK problem specification is that objects and macro objects are defined as mathematical models only, rather than as algorithms. This means that component models do not have *a priori* specification of input or output variables, so that they can be interconnected arbitrarily. In contrast, most widely used modular simulators employ algorithmic component models with prescribed input/output relationships. Such models are inherently less flexible, limiting the class of problems that can be defined without modification of the component models. These arguments were originally put forth by Elmqvist [1978] and recently summarized by Sahlin [1988] and Mattsson [1989].

In SPANK, components are interconnected merely by identifying object interface variables with problem variables. Once all objects are thus interconnected, certain of them are specified by the user to be problem inputs, thereby defining a specific problem. The only requirement is that the problem so specified be well-posed, i.e., have a solution that is uniquely determined from the specified inputs. Although proof of well-posedness in the general nonlinear differential-algebraic system remains an unsolved problem in mathematics, for simulations of most physical systems it is sufficient to be sure the number of problem variables (interface variables minus input variables) is equal to the number of objects (i.e., equations), and that a complete matching is possible between problem variables and objects. These requirements are checked by the SPANK parser.

Summarizing, the important observations regarding SPANK problem specification are: (a) there is a single implementation of a component model rather than one for each possible set of input variables; (b) the user need not be concerned about which are inputs and which are outputs when defining either component models or problems; (c) inverting a problem, i.e., changing which variables are inputs and which are output, can be done without revising component models or interconnections; and (d) the user does not have to devise solution sequences, i.e., algorithms, when defining either component models or problems.

2.2 Solution Methodology

The SPANK solution methodology is also unique. Because the problem is specified entirely in terms of individual equations, graph algorithms can be employed to find a solution sequence. This is a two-step process. The objective of the first step is to select an appropriate equation to calculate each problem variable. To accomplish this, SPANK represents the equation objects and the problem variables as the two disjoint sets of a bipartite graph [Aho 1983]. The variables appearing in each equation object are represented as edges in the graph. When viewed in this way, the selection of an equation for each problem variable is analagous to finding a complete matching in the bipartite graph. There are several well known algorithms for finding such matchings [Johnson 1988]. Currently, SPANK employs the Dinic algorithm [Even 1979], although others would work as well. Upon completion of the first step, there is a one-to-one relationship between equation objects and problem variables. Also, the matching identifies the particular inverse of each equation that gives a formula for the selected variable.

The objective of the second step is to determine a sequence in which the formulas could be evaluated to determine a solution to the posed problem. This would be straightforward if the problem was known to be acyclic, i.e., solvable without iteration; one would simply sort the formulas to ensure that all right-hand-side variables in each are determined by prior formulas (or by problem input). However, in most cases this will not be possible due to situations like $y=f_1(x)$ when $x=f_2(y)$, i.e., cyclic problems. An iterative solution sequence must then be found. Because most simulation problems are of this nature, it is important that the iterative calculations be carried out as efficiently as possible. Since most iterative schemes, e.g., Newton-Raphson, involve solving a linear equation set with a size equal to the number of iteration variables, one way of improving efficiency is to reduce the number of iteration variables. Therefore an important part of determining the solution sequence in SPANK is finding a small number of iteration variables. This is in contrast to conventional simulation programs that typically treat every problem variable as an iteration variable.

Finding the SPANK solution sequence begins with the construction of another graph representing the problem. This is a directed graph in which each equation object is a vertex, with edges representing dependencies of an equation on problem variables. In other words, the in-edges of a vertex represent the variables upon which the equation for that vertex depends. Because of the matching, every vertex also represents a problem variable, so every in-edge is an out-edge of another vertex (or an input variable). A graph constructed in this manner is sometimes called a data flow graph. Data flow graphs can be either acyclic or cyclic. In the first case there is an order in which every vertex can be visited without encountering a previously visited vertex. Obviously, problems that can be solved without iteration have data flow graphs without cycles, while cyclic data flow graphs indicate the need for iteration.

Finding a small set of problem variables to serve as iteration variables is equivalent to finding a small set of vertices, called a *cut set*, that break all cycles in the data flow graph. While finding the minimum cut set in the general directed graph is known to be impossible in polynomial time [Karp 1972], there are many well known algorithms for finding small cut set in such graphs [Levy 1986]. SPANK employs an algorithm developed by Levy and Low [Levy 1988]*.

Once the cut set is known, the data flow graph is modified by introduction of a new, auxiliary vertex for each cut set member. These new vertices act as source vertices for the outgoing edges of the cut set variables, thus breaking all cycles and creating a directed acyclic graph. The system of nonlinear equations is then solved with the Newton-Raphson method, using the acyclic data flow graph to guide the evaluation of functions and derivatives. Specifically, a starting guess is made for each cut set variable and assigned to the corresponding, newly introduced cut set node. The graph is then traversed in "topological order", i.e., visiting only vertices whose incoming edges emanate from already-visited vertices. When a vertex is visited, the corresponding formula is evaluated. This process leads finally to calculated values for the cut set variables. The differences between calculated and assumed values of the cut set variables are treated as function values, upon which the Newton-Raphson method operates. Currently, SPANK calculates derivatives numerically, again using the acyclic data flow graph.

Observe that the dimensionality of the simultaneous set is the dimension of the cut set, as opposed to the dimensionality of the original problem. This means a smaller linear set needs to be solved to get the next estimate of the iteration variables. Typically, HVAC systems show very large reductions. For example a five-zone variable air volume system with a simple algebraic zone model has a cut set of size three, giving a reduction of about 30:1; this is very significant since solution time is proportional to the cube of the size of the linear set. Moreover, it often develops that the cut set size grows slower than the number of zones so that larger problems have even larger reduction ratios. For example, with simple algebraic zone models the cut set size is independent of number of zones, so a 50-zone model would exhibit a reduction of 300:1.

The preceding paragraphs describe the essential ideas employed in SPANK for solving algebraic equations. As shown below, the same techniques apply directly to solution of differential-algebraic systems. Many other extensions and refinements are possible, some of which are described later in the paper.

3. Dynamic SPANK

3.1 Basic Ideas

SPANK was originally developed to solve simulation problems which could be described by a set of nonlinear algebraic equations. SPANK has recently been extended to allow the solution of dynamic problems — problems describable as a mixed set of algebraic and first order ordinary differential equations (ODE's). SPANK's new capability of solving differential-algebraic systems was designed to exploit the existing algebraic solver and to be flexible and general in terms of problem definition and choice of integration methods.

* The Levy and Low work was in connection with the ENET program, the direct predecessor of SPANK. See Sowell [1983].

A dynamic problem with N variables can be described by m algebraic and n differential equations ($N=n+m$):

$$\begin{aligned}
0 &= f_1(t, x_1, x_2, \dots, x_N) \\
0 &= f_2(t, x_1, x_2, \dots, x_N) \\
&\vdots \\
0 &= f_m(t, x_1, x_2, \dots, x_N) \\
\dot{x}_{m+1} &= g_1(t, x_1, x_2, \dots, x_N) \\
\dot{x}_{m+2} &= g_2(t, x_1, x_2, \dots, x_N) \\
&\vdots \\
\dot{x}_N &= g_n(t, x_1, x_2, .., x_N)
\end{aligned} \tag{1}$$

Since there are $N+n$ variables (N problem variables and n derivatives), another n equations are needed to form a well-posed problem. These are given by the integrating formulas for the dynamic variables $x_{m+1}, \dots, x_N$.

$$\begin{aligned}
x_{m+1,\, j+1} &= I(x_{m+1,\, j}\ x_{m+1,\, j-1}, \dots, \dot{x}_{m+1,\, j+1}, \dot{x}_{m+1,\, j}, \dots) \\
x_{m+2,\, j+1,} &= I(x_{m+2,\, j}, x_{m+2,\, j-1}, \dots, \dot{x}_{m+2,\, j+1}, \dot{x}_{m+2,\, j}, \dots) \\
&\vdots \\
x_{N,\, j+1} &= I(x_{N,\, j}, x_{N,\, j-1}, \dots, \dot{x}_{N,\, j+1}, \dot{x}_{N,\, j}, \dots)
\end{aligned} \tag{2}$$

Here j labels the jth time step and I is the integrating formula. Open integrating formulas (explicit methods) involve only past values of a variable and its derivative; closed formulas (implicit methods) also involve the present ($j+1$) value of the derivative. Open formulas are decoupled from the rest of the problem and thus can be solved individually. Closed formulas are coupled to the other problem equations through $\dot{x}_{i,\, j+1}$ and must be solved simultaneously with all or part of the complete equation set. Runge-Kutta integration schemes use integration formulas that require evaluation of the derivatives (the g_i in (1)) at several points within the integration step, but past values of the variables and derivatives are not required. Runge-Kutta methods are sometimes called single-step methods in contrast to the multistep methods which use past values of problem variables and derivatives. Note that if a predictor-corrector method is used, the equations in (2) comprise the correctors. Predictors are always explicit formulas involving only prior values and derivatives and therefore are not involved in the simultaneous solution; they are evaluated by a strictly sequential process that yields starting values for the (possibly iterative) simultaneous solution of (1) and (2).

Literature on solving ODE's focuses on individual equations, with much attention devoted to the efficiency, stability, and accuracy of integration formulas and step-size algorithms. The integration of a set of ODE's is usually regarded as a straightforward extension of the methods for solving single ODE's. The additional complications introduced by a mixed differential-algebraic equation set are rarely discussed.

Runge-Kutta methods, for instance, are often favored for their flexibility, simplicity, and efficiency. Because values at prior times are not used, a Runge-Kutta algorithm can easily be started or restarted, and step size can be easily varied. The efficiency of Runge-Kutta schemes depends on the ability to obtain a value for a derivative function g_i at each subinterval point without iteration. While this can be done for a single ODE or for a purely differential equation set, in a general differential-algebraic system the g_i will not be independent of the algebraic equations and evaluation of the g_i at each subinterval point will require a simultaneous solution of all or part of the equation set. Thus the Runge-Kutta schemes lose part of their simplicity and efficiency when extended to general differential-algebraic problems. Implicit multistep methods, on the other hand, which may be less efficient and less simple for single ODE's or for systems of purely differential equations, generalize easily and naturally to differential-algebraic systems.

Dynamic SPANK allows a differential-algebraic problem to be defined in a more general way than (1), namely:

$$
\begin{aligned}
0 &= f_1(x_1, ..., x_N, \dot{x}_{m+1}, ..., \dot{x}_N) \\
0 &= f_2(x_1, ..., x_N, \dot{x}_{m+1}, ..., \dot{x}_N) \qquad (3) \\
&\vdots \\
0 &= f_N(x_1, ..., x_N, \dot{x}_{m+1}, ..., \dot{x}_N)
\end{aligned}
$$

The integrating formulas are the same as (2). This more general problem statement is a natural extension of the statement of the algebraic problem — dynamic variables and their derivatives are not singled out for special treatment. Here there may be no explicit g_i or a g_i may be a function of other derivatives, potentially making direct solution for derivatives impossible. From (2) and (3) it is evident that, for closed integrating formulas, a dynamic problem involving N variables and n derivatives reduces to the problem of solving a system of $N+n$ algebraic equations at each time step. Creating dynamic SPANK then simply involves choosing an integrating formula, devising a scheme for storing, accessing and updating the past values of the dynamic variables and their derivatives, implementing a time step algorithm, and invoking the old, algebraic SPANK on the full ((2)+(3)) equation set at each time step.

3.2 Current Implementation

For the initial implementation of dynamic SPANK it was decided to concentrate on seamlessly merging the integration process with the algebraic solver and making the integration method available to the user by treating the integrating formulas as SPANK objects. Inclusion of variable time step algorithms and the capability to switch integration methods or orders within a calculation was postponed to a future version of the program. Treating these capabilities in an object-oriented way such that the time step and method switching algorithms are choosable and alterable by the user will require extensions to the SPANK formalism and syntax. Definition of such extensions is near completion and is discussed in Sec. 6, **Semantic Extensions.**

For the above reasons, a very simple step size algorithm has been implemented in the current dynamic SPANK — a user-input fixed time step. For similar reasons, integration schemes requiring subinterval derivative evaluations (Runge-Kutta methods) were disallowed. Although a Runge-Kutta integrator object could be written for the current SPANK, it would not fit naturally into the existing SPANK formalism. Such an object would need to invoke another object (the derivative formula), and this ability would need to be hardwired into the integrator object. Thus the capability to include Runge-Kutta methods was also deferred to a future version of SPANK with a more general syntax. (See Sec. 6, **Semantic Extensions.**)

Aside from Runge-Kutta methods, the user has considerable flexibility in choosing or writing an integration method. The methods are embodied in integrating formulas, which are user accessible objects just like the normal problem equations. Any multistep formula can be used, and separate predictor and corrector objects are allowed.

In dynamic SPANK the past values of dynamic variables and their derivatives needed by the integrating formulas are called "histories". In order to include histories in the SPANK formalism in a natural way, a separate object class (and data structure) for histories was created, as well as the capability to pass history data from one object to another as if it were the value of a problem variable.

In keeping with the decision to make histories "objects" in the eyes of the user, we also chose to make them objects internally. That is, these special objects are stored in the same data structure, i.e., a data flow graph, as normal equation objects. Thus there is now a history class of nodes whose function is to obtain the appropriate history data structure and provide it to the appropriate integrator objects. History nodes are created when a problem variable is denoted as a "history" in the problem definition file.

Actually, histories are not the only "special" nodes in the data flow graph. Even in the original algebraic SPANK, for example, there are several special classes of node that have no in edges. One class comprises input nodes. These nodes obtain values for variables the user has designated as "input" and pass them to the equation objects. These values come from program data structures external to the data flow graph, and are obtained either by querying the user at program initiation in the case of fixed values, or by reading a file in the case of time varying inputs. Another special node class comprises the cut set "guess" nodes. These are the nodes duplicating the nodes in the cut set which are used to pass initial or Newton-Raphson guess values to the rest of the flow graph.

Integrator objects form two more special classes of node. Corrector objects are treated like normal equation objects, but predictor objects need special treatment. They must not be "fired" (executed) when the full data flow graph is executed. Rather, a subset of the data flow graph (all input and history nodes, followed by the predictor nodes) is executed to fire the predictor nodes. The output from the predictor nodes provide initial guess values for the corrector objects in the cut set.

Currently, corrector objects (yielding values for the dependent variables of the differential equations) are always forced to be in the cut set. Fundamentally, this is not always necessary, since (a) the corrector might be an explicit formula, or (b) the derivative could serve just as well to cut the inevitable cycle even with an implicit formula. However, we decided that explicit correctors are rarely used, and initial guess values to start the iteration are problematical for derivatives because most predictors are formulas for the dependent variable, not the derivative. In this manner we explain our decision to force corrector objects into the cut set, but we also recognize the problem it creates, i.e., unnecessarily large cut set size when the user wishes to use explicit integration. The issue will be reopened for future versions.

3.3 Procedure

The general procedure followed by dynamic SPANK is then:

(1) Set up and fill the problem data structure using the input from the user's problem description file.

(2) Perform matching of equations and variables.

(3) Perform reduction to obtain a cut set.

(4) Define a flow graph for the problem.

(5) Obtain an execution sequence for the predictor subset of the flow graph and for the full flow graph.

(6) Set starting guess values for the cut set variables.

(7) Initialize the dynamic variable histories.

(8) Solve the flow graph at the initial time.

(9) Loop over the time steps:

```
while (t <= tlimit ) {
    (a) Execute the predictor subgraph
    (b) Increment time and obtain new values for time varying inputs
    (c) Set the cut set variable guess values using predictor results (if variable is dynamic and there is a predictor for it) or use last step values.
    (d) Invoke the SPANK algebraic solver
    (e) Update dynamic variable histories
} /* End of time loop */
```

Note that the integration of dynamic variables, aside from the optional predictor step, is fully incorporated into the algebraic solver. Integration of a dynamic variable is no different from solving an algebraic equation for a steady-state variable.

The present dynamic SPANK is already a useful real world tool. As part of EKS/US, it has successfully solved a variety of dynamic problems using several different integration methods (see Sec. 5, **Applications**). In the future we plan to increase its sophistication and efficiency.

4. Symbolic manipulation in EKS/US

Symbolic manipulation in mathematical computation refers to automatic derivation of a formula or sequence of formulas that solve a problem. Thus the "answer" is a formula or a procedure that can be used to calculate a numerical answer. This is accomplished by manipulation of the symbols by special software, much as one would do when manually deriving a formula using the rules of algebra. (Hence, the terms *symbolic manipulation* or *computer algebra* are often used.) With the more familiar alternative the calculation is entirely numeric, and the answer is one or more numbers. In all but the most trivial simulation problems, it is unlikely that a totally symbolic solution will be practical. However, it is now recognized that there is also a role for symbolic computation, even though numerical analysis will likely continue to be the keystone of continuous system simulation.

The SPANK methodology offers several opportunities for symbolic manipulation. Most importantly, inverse formulas needed by the solution process must be derived from the object equations. This is a laborious task if done by hand, but one that is readily automated with available symbolic manipulation software. Also, macro objects representing models of physical components can be manipulated symbolically to get the requisite atomic equation objects. This is especially important for component models that are most easily represented by repeated instances of the same equation, e.g., finite-difference models. Such models are tedious to write manually, but are easy to express symbolically, and symbolic manipulation software can be used to generate the models in the required form. Finally, SPANK objects ultimately must be expressed in a compilable language (now C). This step can also be done readily with symbolic manipulation software, producing text files in the format required by the compiler or other software.

4.1 Symbolic Manipulation Software: MACSYMA

There are several widely available symbolic manipulation packages [REDUCE 1987, MACSYMA 1983, MAPLE 1985]. The EKS/US symbolic manipulation software is currently written in the MACSYMA command language. MACSYMA was selected primarily because a public domain version is available. Also, it is probably the best-known package, has good documentation, and runs on a variety of computers. The essential requirement for the SPANK application is the ability to solve symbolically for inverses of equations, together with general list processing capabilities needed for construction of the SPANK files. Other MACSYMA capabilities, such as derivation of symbolic derivative or integration formulas, are not currently used in SPANK.

A modest understanding of MACSYMA is prerequisite to understanding the SPANK/MACSYMA interface. A concise introduction using examples from applied mathematics is provided by Rand [1984]. Here we provide an even more concise introduction with emphasis on the aspects that are especially important in the SPANK interface.

MACSYMA depends heavily on functions. Many fundamental functions are provided with MACSYMA, and users may write their own functions as well. Since function arguments can be of any type, they can be symbols or strings representing, for example, equations. Thus, using the MACSYMA "solve" function, we can write:

```
    solve(equation,variable);
```

Here the argument list has two elements, an equation and a variable that appears in the equation. "solve" performs symbolic operations on the first argument to generate an expression that is a formula for the second argument in terms of other variables that may be in the equation. This is an important MACSYMA function used in the SPANK interface; it is used to generate the inverses of object equations. For example, if we consider an object representing the Stefan-Boltzmann law of radiation, $e = \sigma T^4$, then the inverses, i.e., the formulas for temperature, can be obtained with the MACSYMA command:

```
solve(e=sigma*t^4,t);
```

This command could be issued interactively within the MACSYMA system or from within a program written in the MACSYMA command language. In either case MACSYMA will return the solution list for the variable T, which in this example will be:

```
[-(e/sigma)^(1/4),
  (e/sigma)^(1/4),
 -%I*(e/sigma)^(1/4),
  %I*(e/sigma)^(1/4 )
 ]
```

where %I is the imaginary number i. Knowledge of the physics of the problem must be used to select which of the four, mathematically correct, inverses is appropriate. The MACSYMA command language allows selection rules to be programmed, so this step can also be automated.

The above list of symbolic solutions contains the one we want, along with two complex solutions and another that suggests a negative absolute temperature. The MACSYMA command language can be used to "filter" this list and give the single solution that makes physical sense. A complete description of the details of this operation would require more explanation of the MACSYMA command language than we can present in this paper. Nonetheless, the flavor of the method can be seen from the following code fragment which omits details:

```
/*Condition on the solution t (absolute temperature >0)*/

conditions:[t>0,e>0,sigma>0];

/*Put conditions in current data base*/

for condition in conditions do ( if condition#'true then assume(condition));
/*Solution filter*/

 for solution in solutions do (
   /*Keep real solutions*/
   if ( (member(%i,listofvars(solution))='false
   or realpart(solution)=solution)
   /*Keep solutions within range*/
   and is(ev( subst(solution,t,t>0) ) )#'false)
   then ( goodsolutions:endcons(solution, goodsolutions))
  );

print("Final Solution is ",goodsolutions);
```

In this code fragment, we assume the list of symbolic solutions found by solve is in the MACSYMA variable called "solutions", and the result is placed in "goodsolution". The temporary variable "solution" holds one member of solutions at a time as it is tested against the list of "conditions" that are defined before the loop begins. We omit the rather intricate MACSYMA code that formats the goodsolution to SPANK code.

The MACSYMA solve function is powerful, but not limitless. It is able to solve polynomials up to the fourth order, and can handle equations requiring inversion of exponential, logarithmic or circular functions. In common use, as in SPANK, these forms, together with the standard operations (+,-,*,/), account for most of what is needed, so the function meets the need.

MACSYMA can also solve for systems of equations, but this capability has practical limits. For one thing, symbolic solution of systems of equations is computationally intensive and can take inordinate amounts of computer time. Also, the solution is less reliable than when inverting single equations. Indeed, if the system is nonlinear, MACSYMA usually encounters severe problems, and often fails to find a solution at all. Although we have not yet found beneficial use for this feature in the SPANK/MACSYMA interface, it is being considered for certain advanced capabilities, such as merging of components.

An additional feature of MACSYMA, which is quite useful, is its ability to evaluate and simplify expressions. For example, the function RATSIMP(A) simplifies a polynomial A and returns a ratio of two polynomials. The user can control the way the evaluation and simplification is to be performed through the use of switches, common environment variables, or optional arguments to functions.

MACSYMA can also check whether a proposition can be derived from a set of equations or other propositions, using its "assume" facility. This feature is used in the SPANK/MACSYMA interface to solve for piecewise-defined functions, where the variables to be solved for have a limited validity range.

While MACSYMA serves well in EKS/US, it is not ideally suited for the purpose. Interestingly, the most significant disadvantage is not its weaknesses, but its power; it is really more than is needed for the job. Because of its power it is large (roughly twice as large as SPANK in terms of disk space). Ultimately, we will incorporate a subset of MACSYMA functionality in a C or C++ program to support EKS/US.

4.2 The SPANK/MACSYMA Interface

The SPANK/MACSYMA interface is a collection of programs written in the MACSYMA language. The basic module of this package is about 1500 lines of MACSYMA code. This module allows the user to generate required C functions, objects, and macros in the SPANK format by entering the equations in natural form along with intended object names [Sowell 1990]. A second module (which invokes the basic module) allows generation of a complete simulation file and all associated objects and functions. This module is about 200 lines of MACSYMA code. Additional modules include one for generation of macros that are composed of many instances of the same elementary object (500 lines), and one for merging of equations to eliminate selected intermediate variables (500 lines). Thus, the entire package is not a large program.

So far we have mentioned the central issues in the interface, namely solving equations using the "solve" MACSYMA function, dealing with list of variables to solve for using the list handling utilities, and checking whether they are within range using the relational data utilities. The programs also include code devoted to more mundane issues, such as formatting the solutions into SPANK or C syntax. It is notable that MACSYMA has a built-in translator for arithmetic expressions in MACSYMA to

FORTRAN, but not to C. Therefore a MACSYMA to C expression translator was devised using substitution rules. For example, x^y in MACSYMA becomes pow(x,y) in C, and %PI in MACSYMA becomes M_PI in C. Another problem was the limited formatted output capability of MACSYMA. In order to get text files in the format needed by SPANK and the C compiler it was necessary to develop special file writing functions using Lisp. Another issue that complicates the interface code is bookkeeping. In the case of macro objects and global simulation generation, we have to keep track of what variables are common among different equations to ensure proper linking. Last, string handling routines are used for SPANK file generation and name generation. The syntax of the MACSYMA language is fairly natural and the function names are usually self explanatory (although long). All of these secondary issues constitute about 50% of the code in the interface.

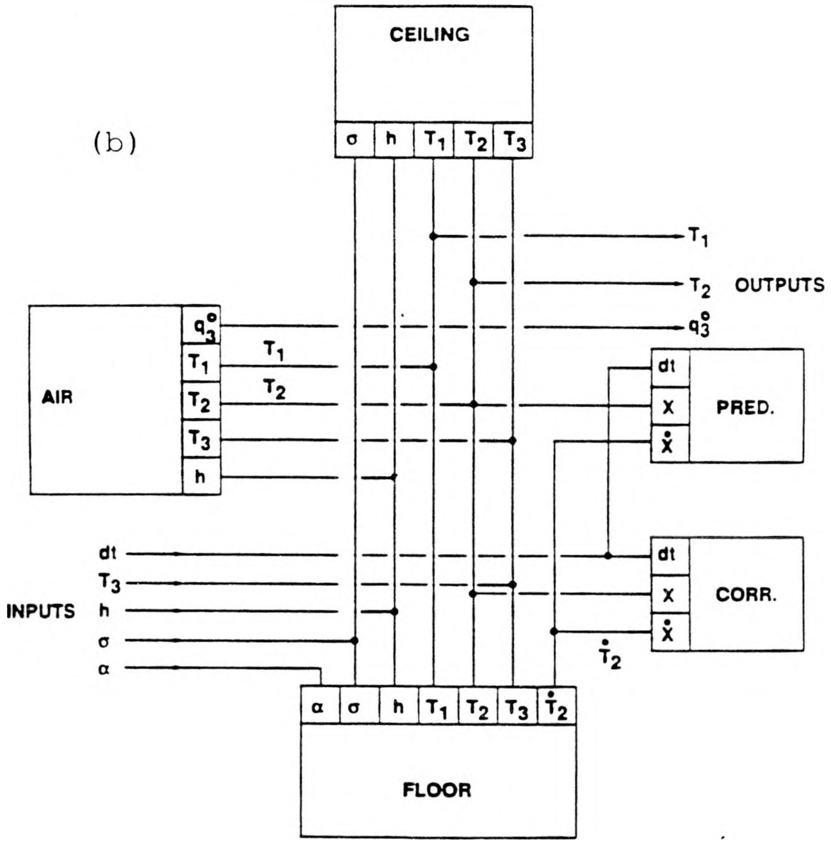
The derivation of equations and generation of files is performed in a reasonable time (from seconds to minutes, depending on system size). Some care must be taken to ensure that MACSYMA is not launched into feasible but extremely time consuming tasks. A typical example is the symbolic resolution of fourth-order polynomial equations. MACSYMA will do it, but will take an inordinate amount of time, ask for much additional information, like the sign of some complicated discriminant, and generate huge expressions. To prevent this, a careful user will avoid requesting such equations to be inverted. This can be done at the SPANK/MACSYMA interface level by declaring the variable appearing to the fourth order as a "bad inverse," and not try to solve for it (unless it is short and simple, as in the Stefan radiation law above).

5. Applications of EKS/US

EKS/US has been tested on a range of simple to complex problems in energy analysis. We briefly describe here a subset of these problems to give the reader a feeling for the scope of applications that are possible. References are given when a more detailed discussion of the problem has been published. Due to space limitations, we show results for only the last case, the lighting/HVAC problem.

5.1 Three-Node Room

Figure 2 shows a very simple room model that was used as a first test of dynamic SPANK [Sowell 1988]. The three nodes correspond to a massless ceiling, a massive floor, and massless room air. The floor and ceiling exchange long-wave radiation and convect to the room air. EKS/US solved the problem of finding the floor and ceiling temperature for time-varying air temperature. The block diagram, in Fig. 2(b), shows the objects for this problem and their links. In addition to the floor, air, and ceiling objects, there are predictor and corrector objects for differential equation integration. Sample results of running this problem are shown in Fig. 2(c).



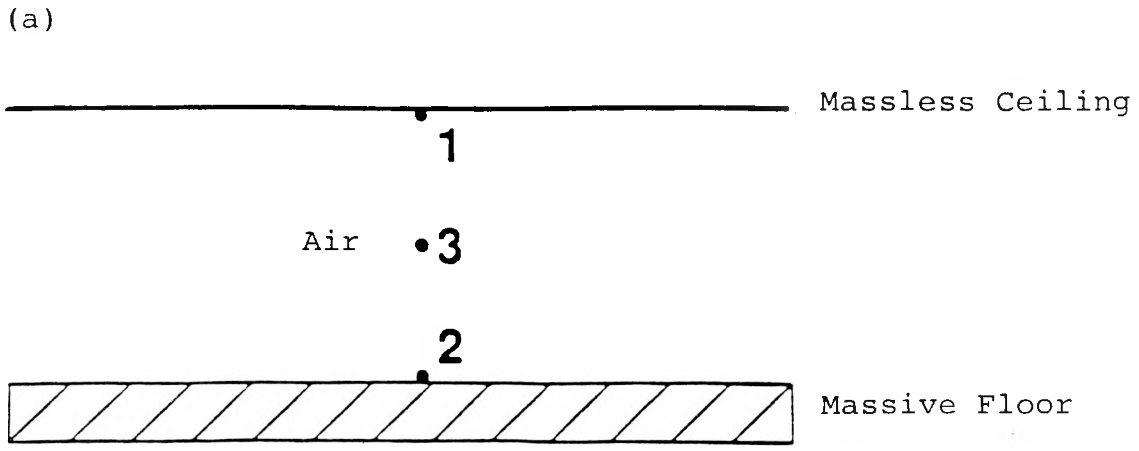


Figure 2: (a) Three-node room model. (b) Block diagram showing objects and links; T = node temperature, q = heat addition rate, h = convective heat transfer coefficient, dt = timestep.

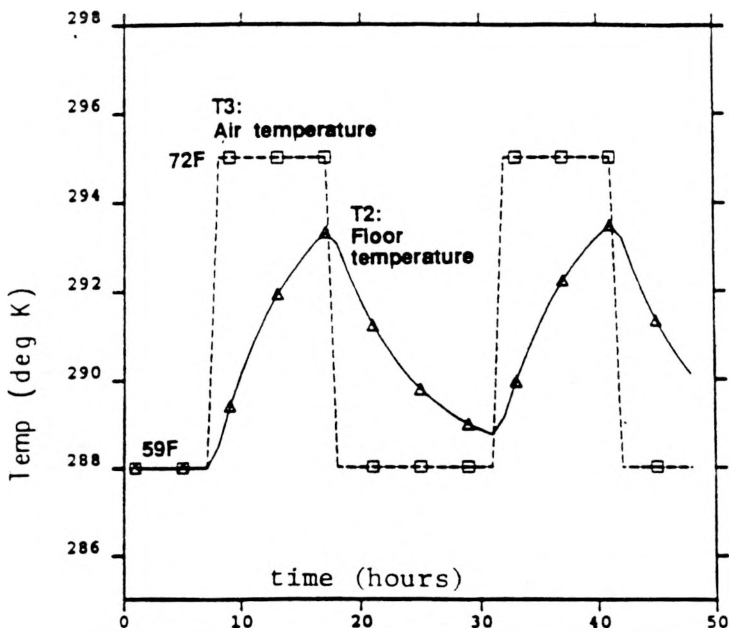


Figure 2(c): Simulation results for 3-node room showing calculated floor temperature for user-input time-varying air temperature.

5.2 Thermal conduction

Finite-difference simulation was done for one-dimensional conduction problems with variable conductivity, mixed boundary conditions, and bulk domain heat generation. Both steady state and dynamic cases were treated with various spatial discretizations. Figure 3 shows a typical configuration in which the heat flux is constant at one end of the conductor and natural convection takes place on the other end.

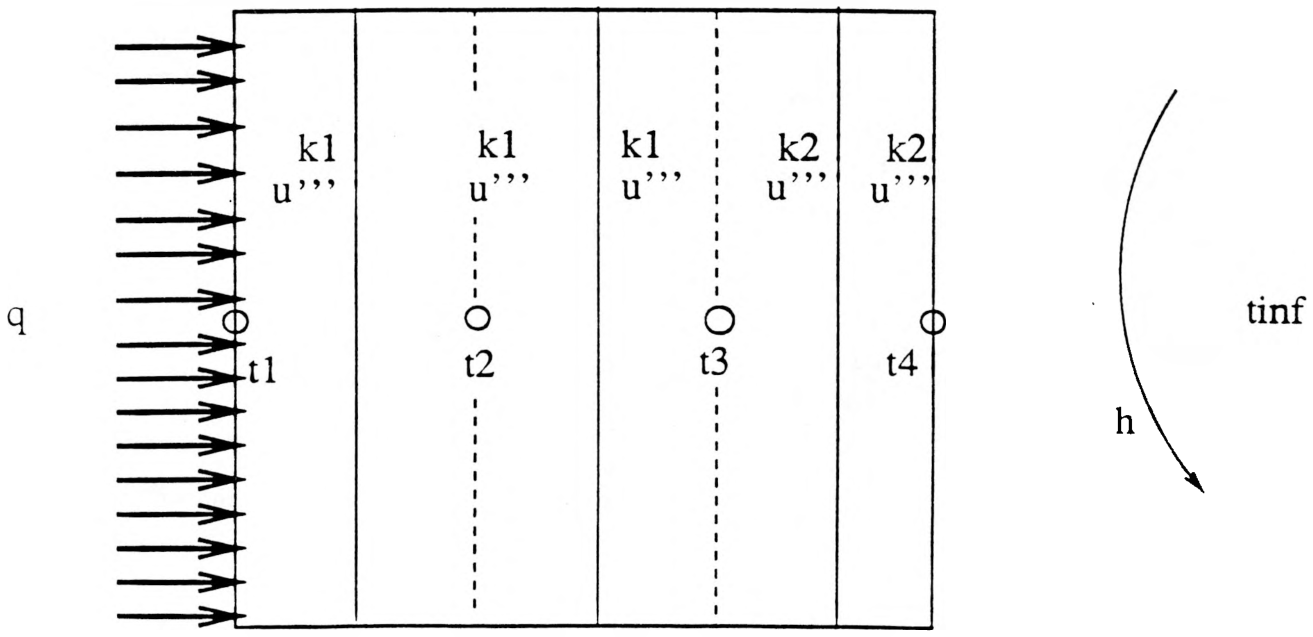


Figure 3: One-dimensional thermal conduction model; q = heat flux, t = temperature, k = conductivity, u = heat generation rate, h = convective heat transfer coefficient, tinf = ambient temperature.

5.3 Steady-state zone convection

Natural convection in a room heated by a radiator was modeled according to the Inard [1988] formalism. As shown in Fig. 4, the room is divided into five cells, each of which has a simple flow pattern. The primitive cell objects are linked into a zone macro object. The convective conductances between subzones are based on empirical correlations. Given the heater output, Qconv, and the temperature at nodes 1, 3, and 5, EKS/US solved for the intercell heat fluxes and the temperature nodes 2 and 4.

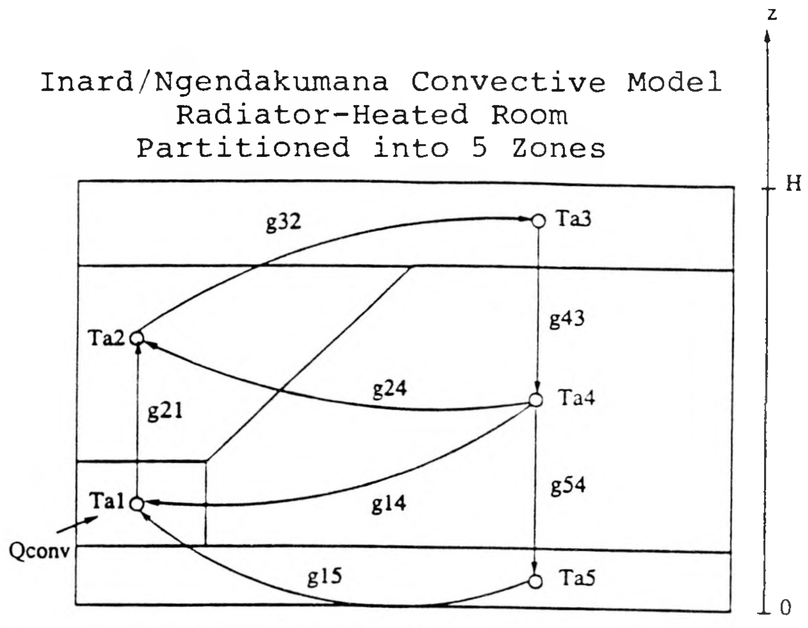


Figure 4: Five-cell model for in-room natural convection; T = temperature, Q = radiator heat, g = intercell convective conductance.

5.4 Multiroom air flow

Figure 5 shows a schematic for air flow between rooms driven by wind pressure and stack effect [Buhl 1989]. A variable number of rooms are connected to each other by a variable number of orifices. The smallest problem solved had one room with six orifices, the largest had 24 rooms with six orifices per room. Pressures on the orifices connected to the outside are input, and the pressure difference at and mass flow through each orifice are obtained. Reduction factors between 10 and 20 were obtained; the number of iterations to solution varied from 8 to 44.

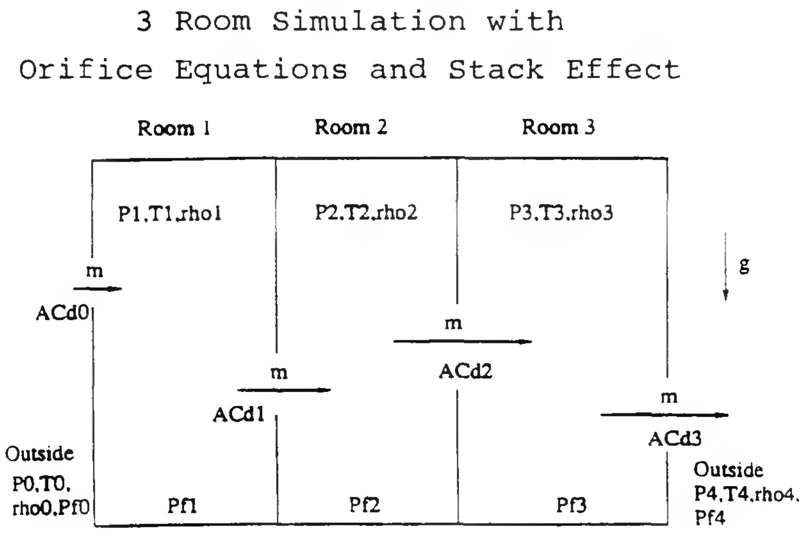


Figure 5: Air flow between rooms driven by wind pressure, stack effect; P = air pressure, T = air temperature, rho = air density, m = mass flow, ACd = effective orifice area.

5.5 Hamburg Cell

The Hamburg Cell, shown in Fig. 6, is an exercise originally used to test the French ZOOM program [Bonin 1987]. We are using it as a test problem to compare EKS/US and ZOOM. The problem consists of a idealized three-zone room enclosed by four three-node walls. Two of the walls face north and have constant outside temperature; the others face south and are exposed to time-varying outside air temperature and solar radiation. Convection between room zones is modeled, but long-wave radiation exchange between room surfaces is neglected. The only nonlinearity is introduced by a room heater that is controlled by the average of the north wall inside surface temperature and the temperature of one of the air cells. Preliminary results show good agreement between EKS/US and ZOOM results on this problem.

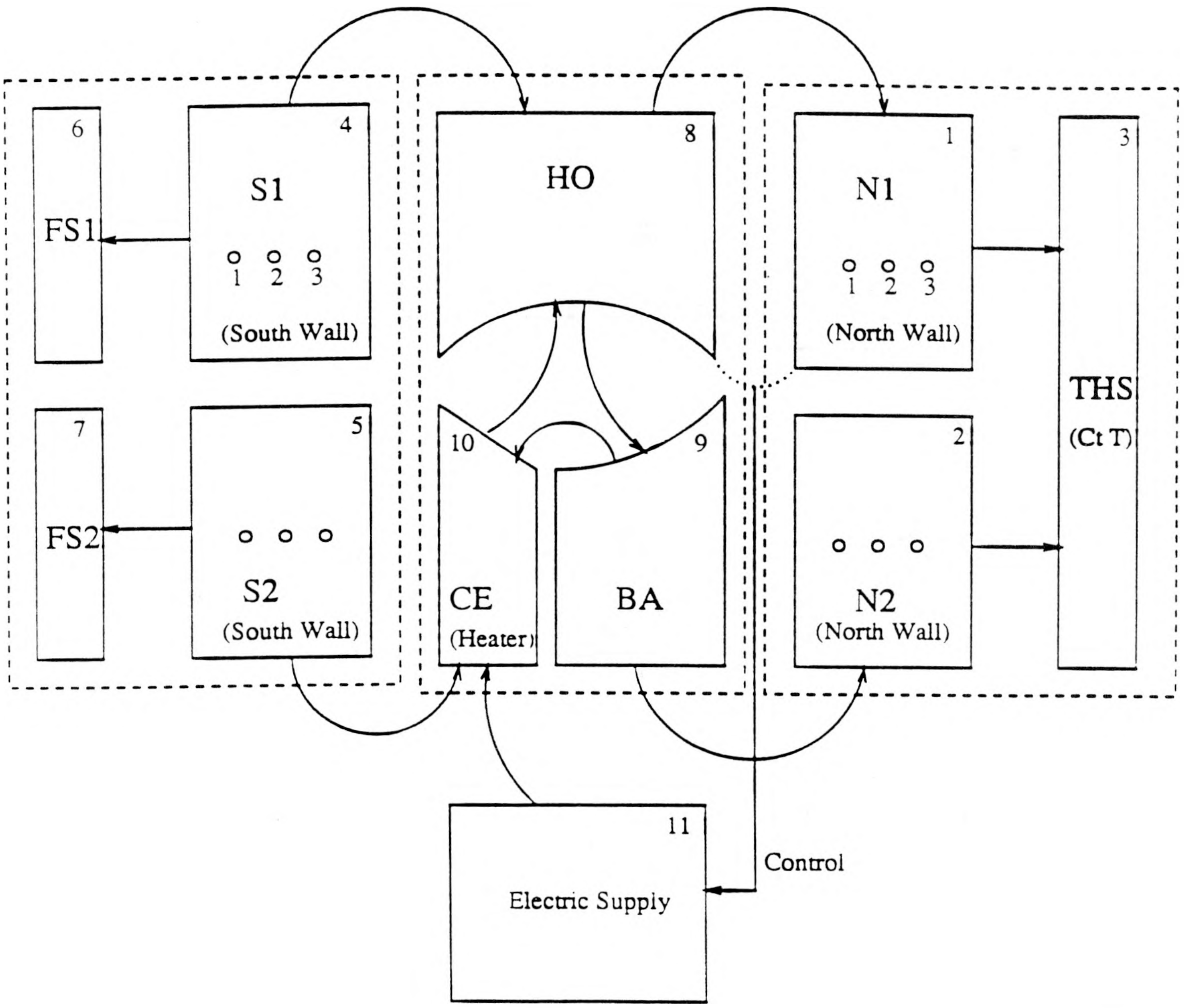


Figure 6: The "Hamburg Cell", an "idealized" three-zone room enclosed by four, three-node walls. Arrows (except for the one labeled "control") indicate energy transfer.

5.6 Desiccant Cooling

Figure 7 shows a hybrid liquid desiccant system that provides cool, dry air to a space [Nataf 1990]. The working fluid is a solution of lithium chloride in water. The system contains an interchanger, a heater, and a cooler (all modeled with the LMTD method), and a regenerator and conditioner (both of which are modeled with a Kathabar equation). It also contains two sumps, one of which is massive and, therefore, dynamic. In the EKS/US object-oriented approach, the conditioner and regenerator are instantiations of a single object class. Similarly, the cooler, heater, and interchanger are instantiations of a single heat exchanger object class. The problem consists of 83 equations. After reduction there were only 9 iteration variables.

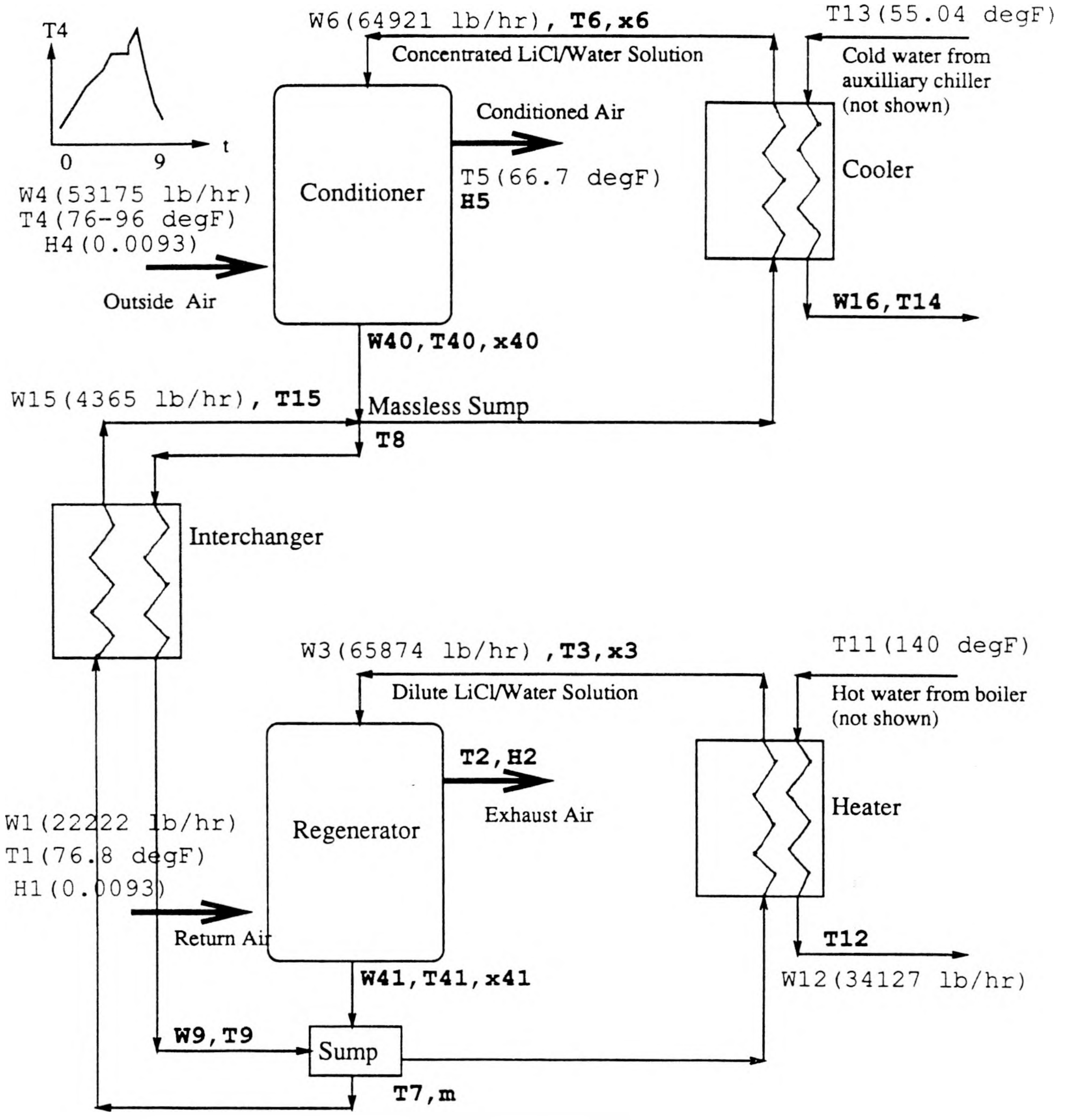


Figure 7: Liquid desiccant cooling system. Unknown variables are shown in boldface and input variables in lighter type, with input values in parentheses. W = mass flow, x = salt concentration, H = humidity ratio, i = specific enthalpy, T = temperature, m = mass of solution in regenerator sump.

5.7 Boiler plus DHW Heater

In this problem, shown schematically in Fig. 8, a boiler and domestic hot water heater are connected to the same chimney. Heat transfer in the chimney is modeled using 1-d finite difference. EKS/US solved for the various temperatures and mass flows given ambient temperature and pressure and the water temperature in the boiler and DHW heater.

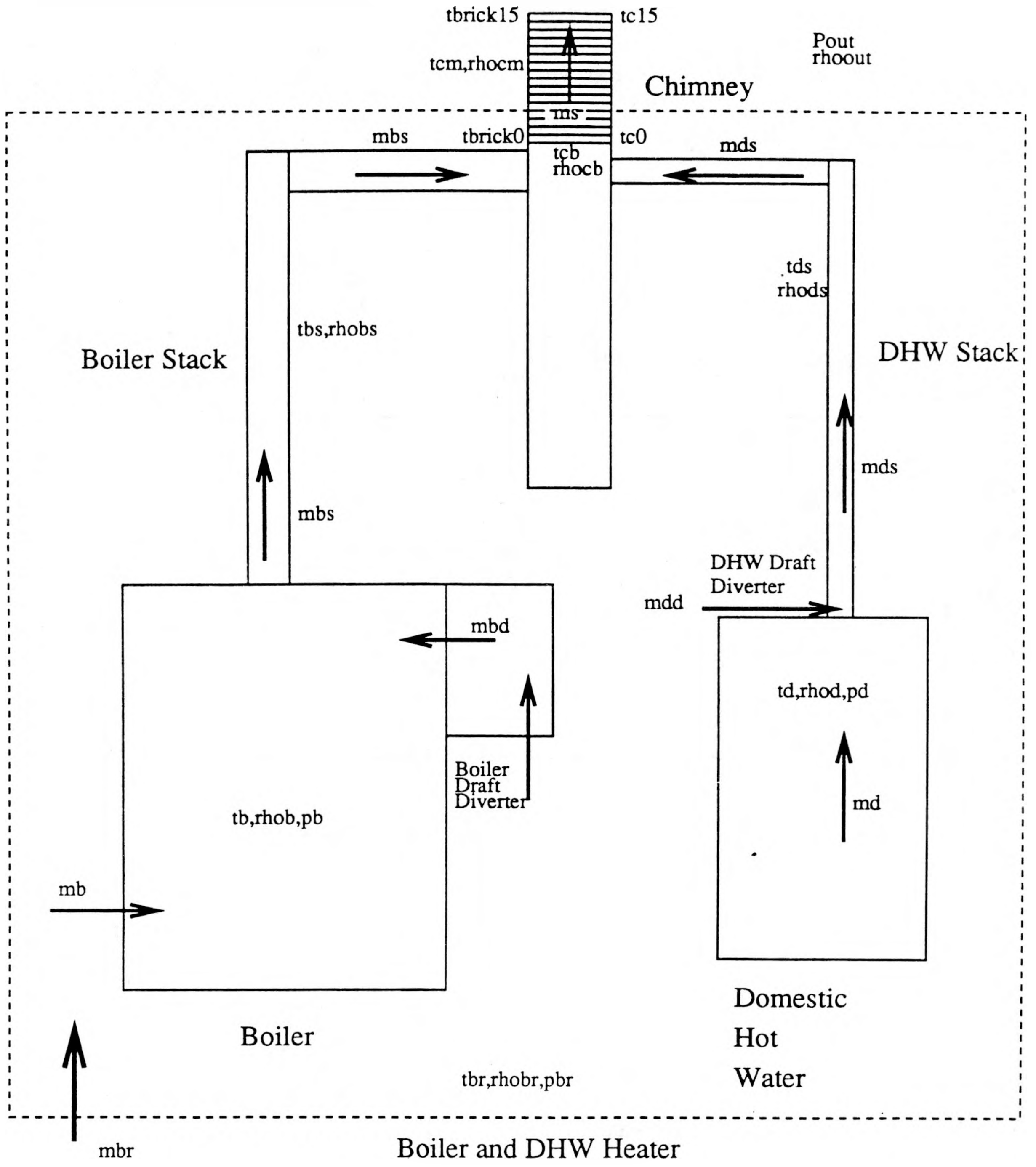


Figure 8: Boiler and domestic hot water heater sharing a common chimney; m = air mass flow, t = temperature, rho = density, p = pressure.

5.8 Constant Volume Reheat System

Figure 9 shows a constant volume reheat system used to test the early, steady-state version of EKS/US [Sowell 1986]. Outside air is mixed with return air and passed through a cooling coil, a heating coil, and a fan to become the zone supply air stream. The zone has sensible heat gain, E_{in}, air infiltration, m_{inf}, and water vapor addition, *water_in*. In addition to the physical components the diagram shows dashed blocks representing "data conversion" objects that transform enthalpy and humidity ratio to drybulb temperature and vice-versa. This problem results in 23 equations and 38 variables, 15 of which were chosen as inputs (the circled variables in the figure), leaving 23 to be solved for. After reduction, this problem has only one iteration variable, the humidity ratio, $w2$, leaving the cooling coil.

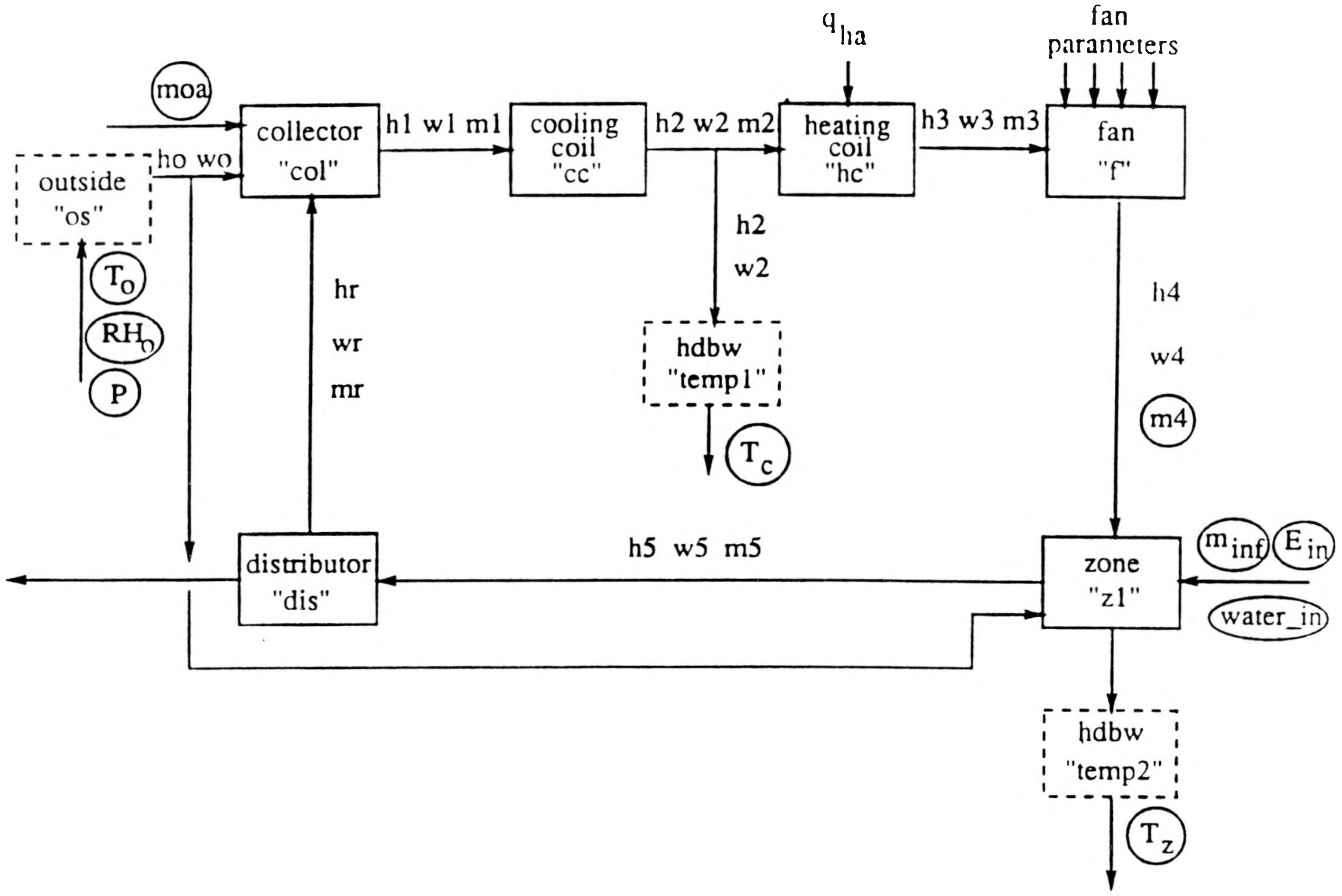


Figure 9: Constant volume reheat system showing problem variables. Inputs are circled, unknowns uncircled. T = temperature, m = mass flow, h = specific enthalpy, w = humidity ratio, E = sensible heat gain, RH = relative humidity, P = pressure.

5.9 VAV Reheat System

Figure 10 shows a variable volume reheat system containing a preheat coil, cooling coil, zone heating coils, supply and return fans, and nonlinear controls. The system can serve an arbitrary number of zones; the 5-zone case is shown in the figure. In the problem analyzed, zone loads were input. For dynamic simulation, there are four iteration variables independent of the number of zones. The reduction factor can therefore be quite high; for example, for 20 zones there are 264 equations and four iteration variables, giving a reduction factor of 66.

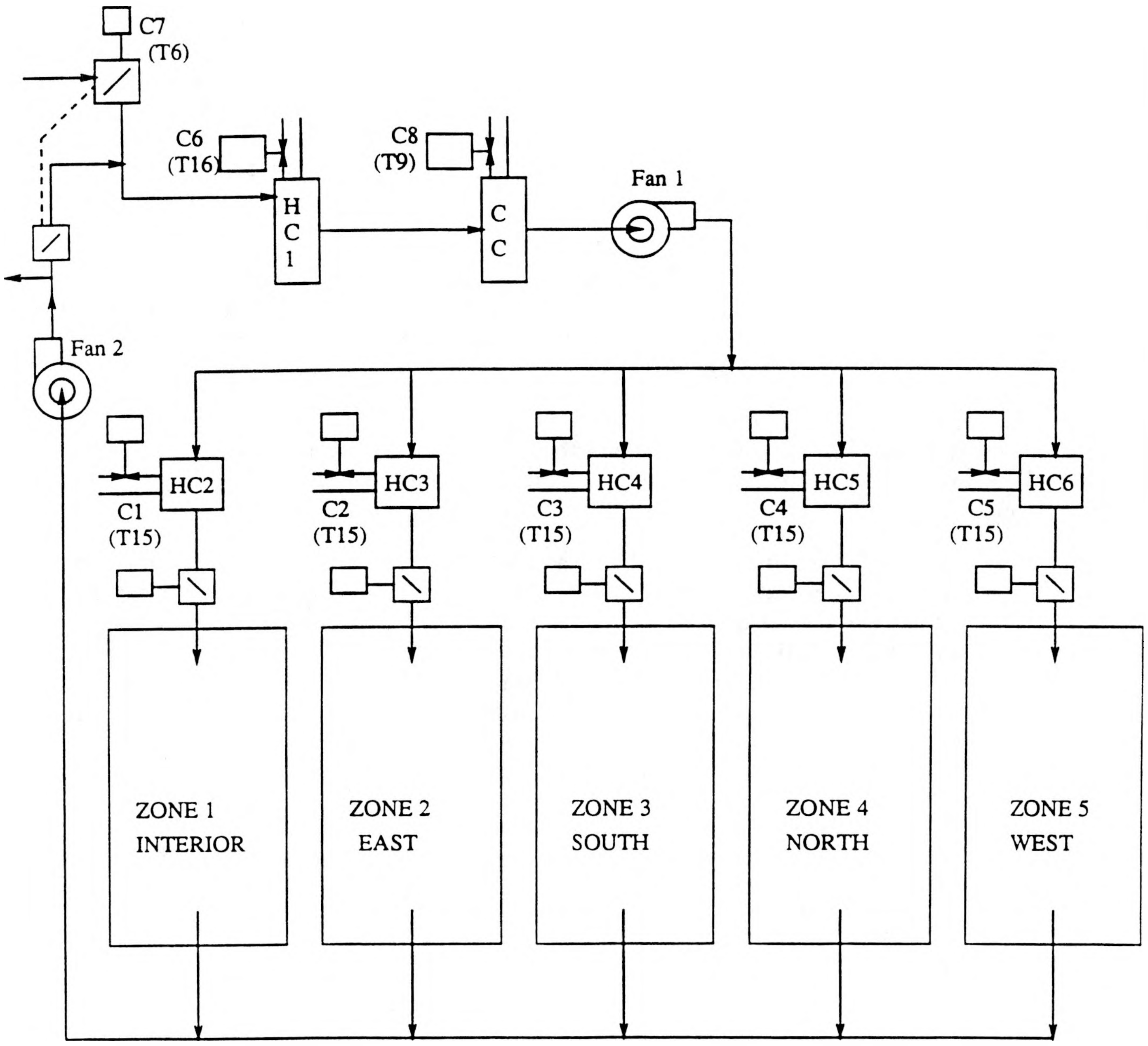


Figure 10: VAV reheat system serving five zones. HC = heating coil, CC = cooling coil, C = control, T = type of control.

5.10 Lighting/HVAC Problem

Figure 11 shows the schematic of a model used to study lighting/HVAC interactions [Sowell 1990]. Lighting is provided by fluorescent lamps in the plenum space of a 10,000-ft^2 room. A translucent ceiling lens separates the plenum from the room below. Supply air enters the room, mixes with the room air, then exhausts to the plenum through small openings in the ceiling lens. Input power leaves the lamp by shortwave (visible) and longwave (infrared) radiation and by convection to the plenum air. The radiative portion undergoes interreflection and transmission, and is ultimately absorbed by surfaces in the plenum and the room. If the plenum air temperature is greater than the room temperature, some or all of the convective portion can also escape the plenum by conduction through the transparent ceiling to the room air. Ultimately, all lamp power must be removed by the airstream after convective transfer from the various solid surfaces in the room and plenum. We wish to determine the surface and air temperatures, and the heat removal rate in the room and plenum. Naturally, these will be functions of the mass flow rate of air and the supply air temperature.

For simplicity, we assumed that the dimensions in the horizontal plane are large relative to room and plenum height, thus making losses through walls negligible. It is also assumed that the floor and ceiling are adiabatic, i.e., that no heat transfer occurs between the ceiling and the room above or the floor and the plenum below. View factors for radiation exchange were calculated with a separate program.

The convective heat transfer coefficients used assume free convection and were taken to be constant. A later improvement to the model used recently measured correlations [Spitler 1991] giving these coefficients as a function of supply air jet momentum.

The above problem can be formulated as an n-node network in which each node is viewed as a surface that can emit, absorb, reflect, and transmit radiant energy in the short and long wave bands. Also, nodes can interact through surface-to-air convection, and through bulk flow convection. The system variables include node temperatures, short and long wave radiosities and irradiations at each node. The basic physical laws governing the system are those of diffuse radiative transfer, convective heat transfer, and conservation of energy and mass [Sowell 1973].

The block diagram, Figure 12, shows the macro objects for this problem and their connections. The equations corresponding to these objects are given in [Sowell 1990]. By virtue of designation of particular system variables as "inputs", Fig. 12 also represents a particular "problem". One problem that can be represented (which corresponds to case (1), below) is:

Given:
All geometric and property data, and convection coefficients.
The short wave emission at each surface, J0S.
The source energy addition/removal rates at all surface nodes and plenum air node, Q0(1)—Q0(6).
The temperature at the room air node, T(7).

Find:
The temperatures at all surface nodes and plenum air node, T(1)—T(6).
The heat addition/removal rate at the room air node, Q0(7).

The short and long wave radiosities and irradiations at each node.

An important feature of EKS/US is that different problems on the same system can be specified *without structural changes in the model.* For example, if we wished to specify a surface temperature and solve for the required heat addition/removal rate we could simply designate a different input set.

Dynamic simulation results for some of the problem variables are shown in Fig. 13. Two cases are shown: fixed room air temperature and fixed supply air temperature. For this study the air flow rate was set at 1.0 cfm/ft^2. A run period of 200 hours was chosen, with a time step of 6 minutes. Initially, all of the node temperatures are near the steady-state lights on condition. Then, at time zero, the lights are turned off and remain off for 50 hours, during which time the system approaches a steady-state lights off condition. The lights are then switched on with an input power of 3.5 W/ft^2.

The general behavior observed in Fig. 13 is an initial decrease in temperatures, followed by an asymptotic approach to equilibrium lights-off values, then a relatively rapid increase at 50 hours when the lights are turned on, followed by an asymptotic approach to equilibrium lights-on values. The initial decrease is due to the fact that the temperature starting values chosen for the simulation were above the equilibrium lights-off values.

This example shows that EKS/US can be used to solve complex, nonlinear dynamic heat transfer problems involving simultaneous radiative, conductive and convective processes.

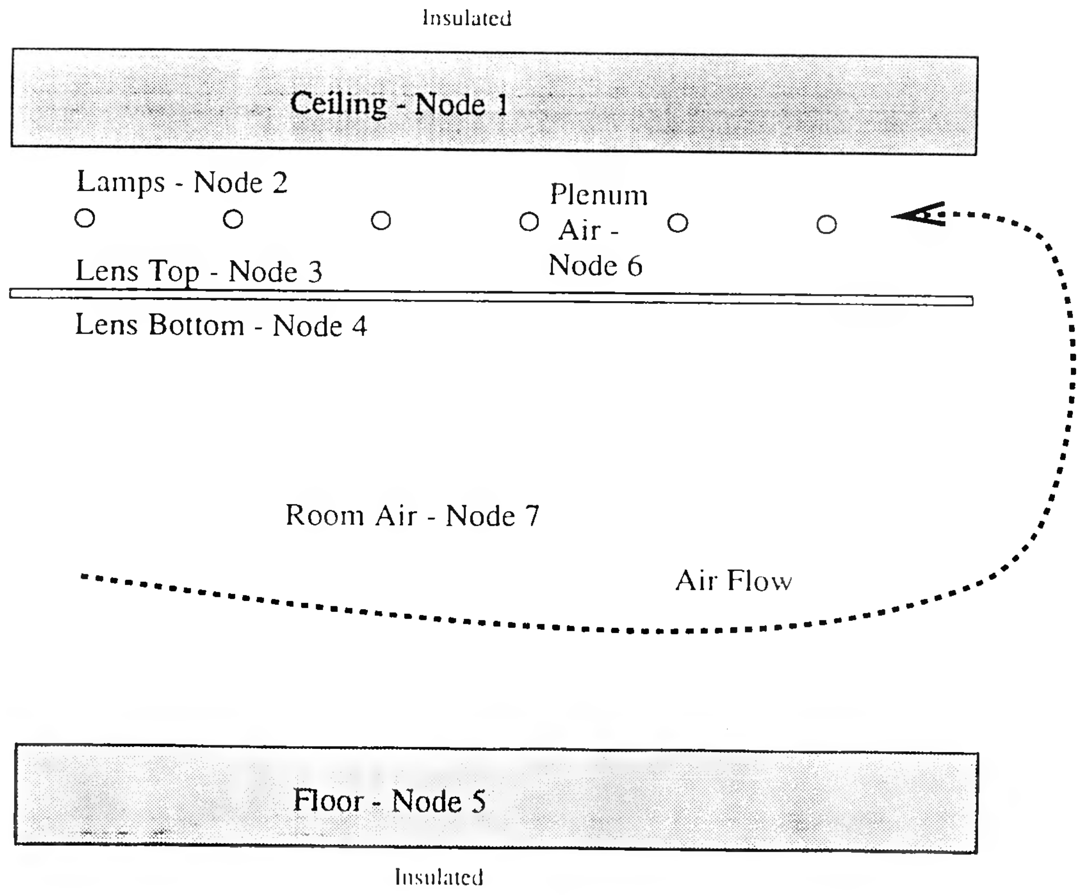


Figure 11: Lighting heat transfer problem: vertical section through room and plenum.

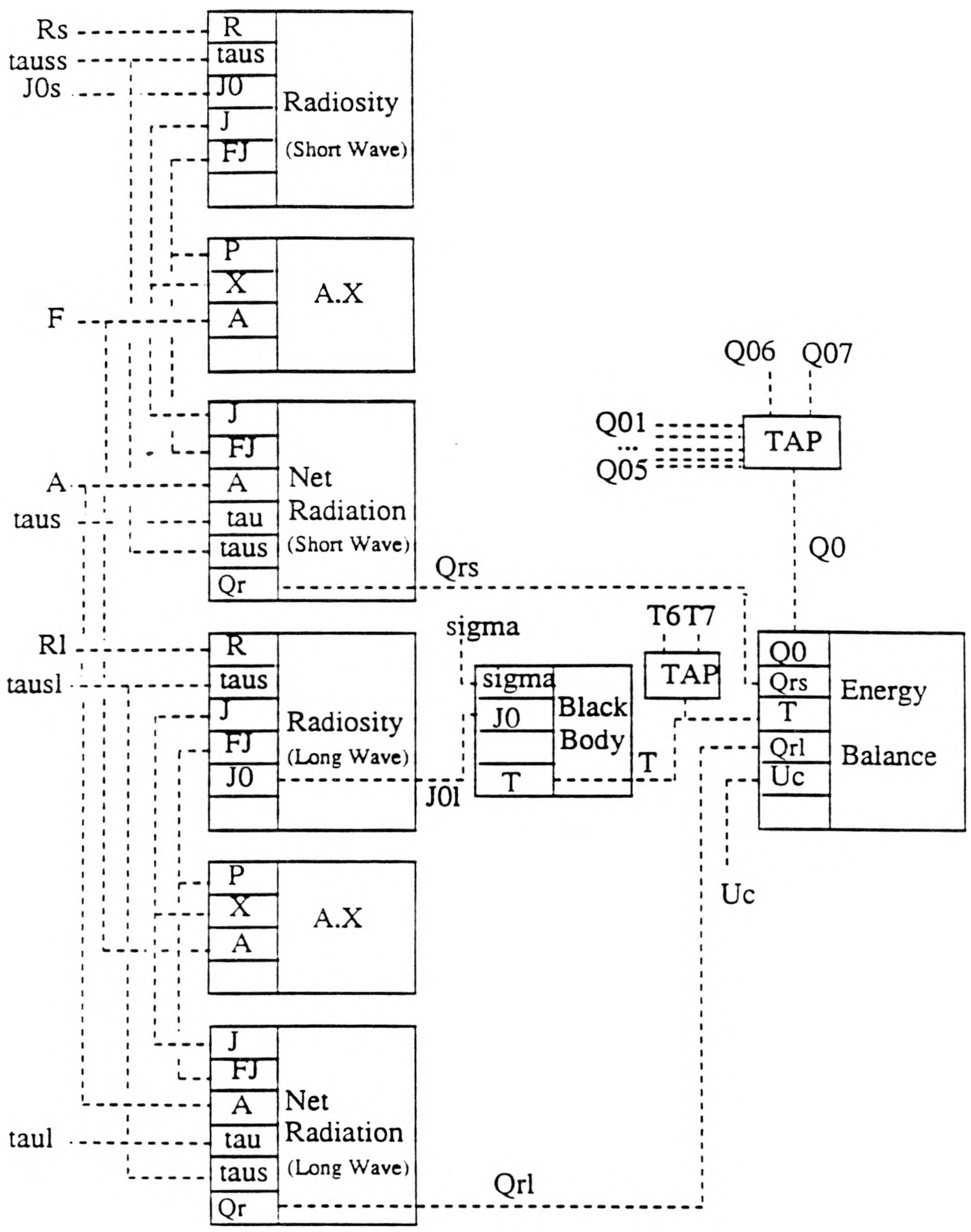


Figure 12: Block diagram showing objects for the lighting heat transfer problem. Dashed lines indicate inputs or system variables shared by objects. T = temperature, J = radiosity, FJ = irradiation, R = reflectance, tau = transmittance, A = area, U = conductance, Q0 = heat addition rate, Qr = net radiant heat transfer rate.

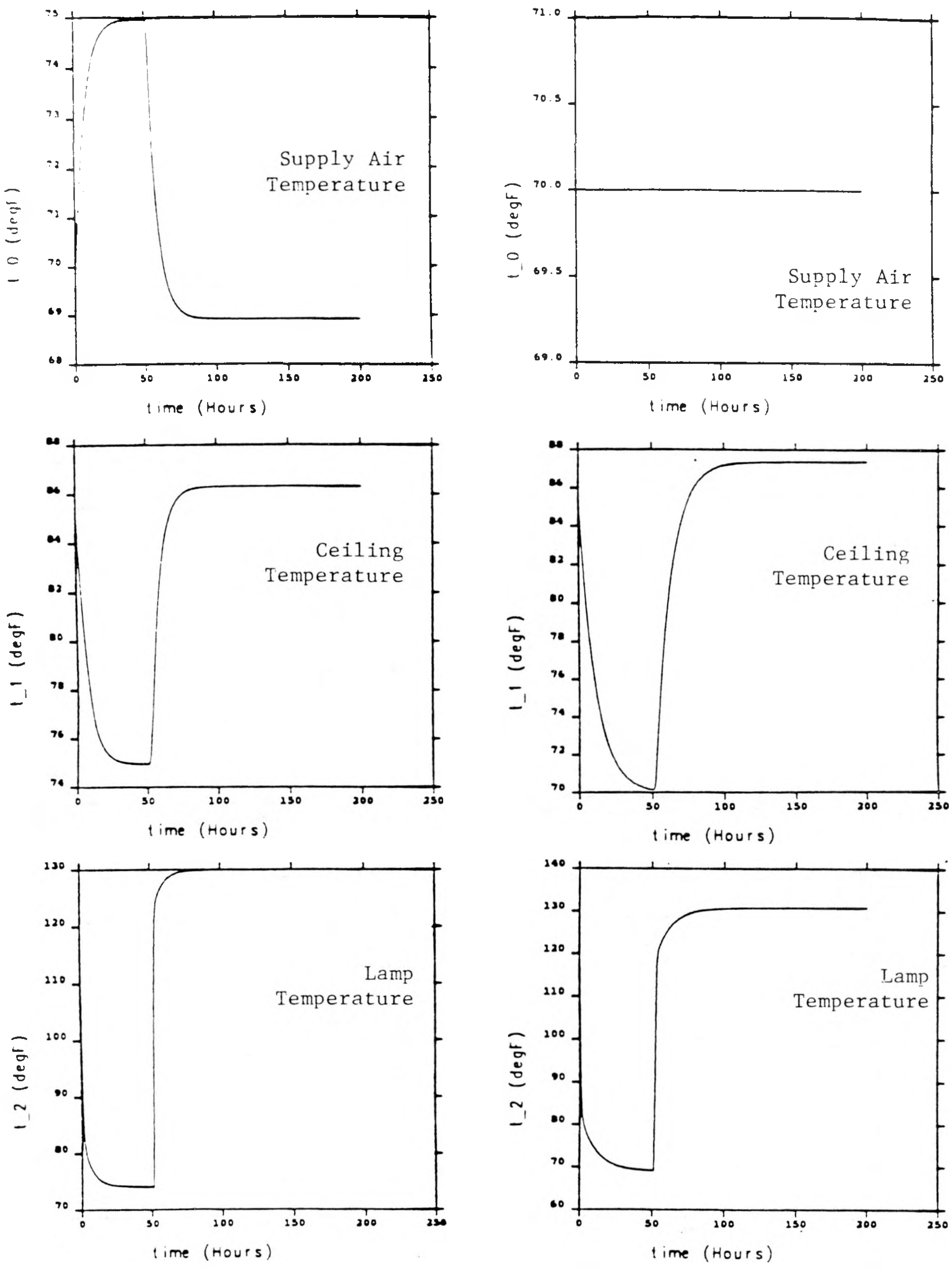


Figure 13: Simulation results for the lighting heat transfer problem. The lights are turned on at t = 50 hours. The supply air flowrate is fixed at 1.0 CFM/sf. For the left-hand graphs, supply air temperature varies to maintain a fixed 75F room air temperature. For the right-hand graphs the supply air temperature is fixed at 70F.

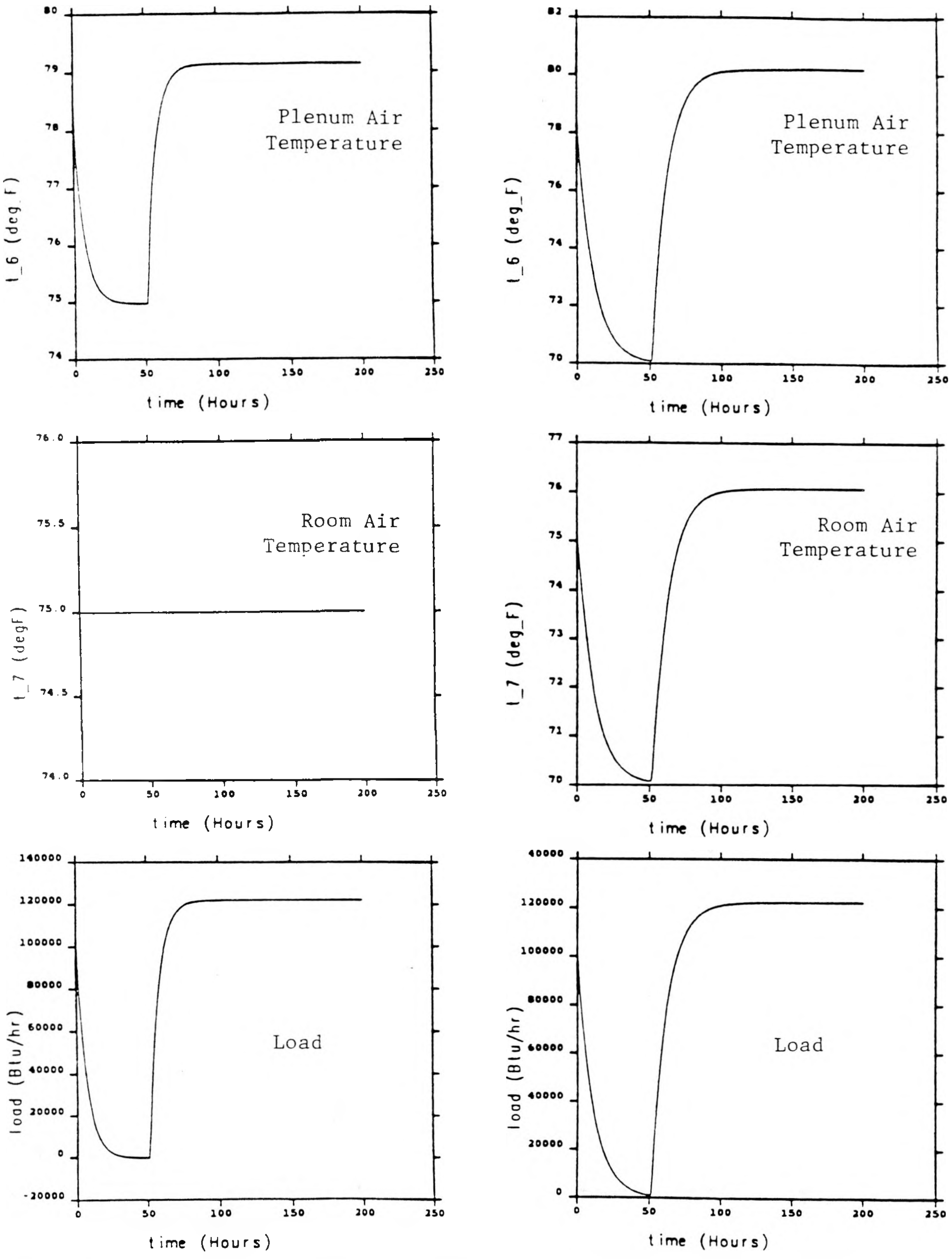


Figure 13: Simulation results for the lighting heat transfer problem. (Cont.)

6. Semantic Extensions

6.1 Current Limitations

The original design of SPANK was based on static models. As such only algebraic systems could be specified. As demonstrated above, we were able to implement significant dynamic simulation capability with minor modifications to the original syntax. However, the user is currently limited to a small range of numerical integration methods, namely those with predictors and correctors employing three or fewer previous values of variables and derivatives and a fixed, global time-step. Although Runge-Kutta integrators can be specified, doing so is awkward, requiring the integrator object to involve elements of the particular problem rather than being a semantically distinct entity. More complex integration schemes, including those with separate start-up methods, cannot be specified. Moreover, certain kinds of dynamic systems cannot be specified, such as those with some constraints applying only at certain times or under certain conditions depending on system state.

Other current limitations, unrelated to dynamics, have to do with the way objects, macro objects, and problems are specified. The current implementation lacks uniformity in the way these entities are seen by the user, imposing unnecessary burdens on the user to keep track of the differences between various constructs which semantically ought to be treated the same. Similarly, in the current implementation there are artificial differences between "scalar" values, such as temperature, and "compound" values, such as air flow, which are characterized by several variables, e.g, temperature, humidity ratio, pressure, etc. It is often the case that statement of a problem is more naturally expressed in terms of such compound values, but the current implementation forces the user to decompose them into their constituent scalar values.

Consideration of these needs led to reevaluation of the semantics of dynamic simulation as the first step toward a completely new specification language. Below we present a specification for this new language, called the Component Definition Language (CDL).

6.2 Component Definition Language (CDL)

In the following section, we describe a grammar for CDL along with an informal semantic specification. We use certain conventions for describing the grammar. In particular, keywords are always typed in bold face, e.g., **object** is a keyword. Likewise, punctuation marks in the object language are typed in boldface. Thus, "**(**" is an object language punctuation mark as distinct from "(", which serves to group constructs together in the grammar. Syntactic variables (think of them as names for syntactic categories) are denoted by italic typeface enclosed in angle brackets, e.g., <*type*> is a certain syntactic category. In the grammar, a construct with a superscript asterisk means zero or more occurrences of the construct, a superscript plus means one or more occurrences. Vertical bars separating constructs means exactly one of the constructs must occur. Finally, a construct in square brackets is optional.

6.2.1 The Basic Semantic Categories

The semantic entities of CDL fall into seven basic categories: kinds, classes, objects, types, values, variables, and connections. Roughly speaking, the relation of class to kind is the same as that of value to type. That is, a type is a certain collection of values all having similar shape. Likewise, a kind is a certain collection of classes all having similar shape.

The semantic notion of a value is fairly clear. Likewise, the notion of a variable in CDL is essentially the familiar notion of a variable in programming languages.

Types are built up inductively from a collection of simple types (double, real, int, etc.) together with a construct essentially like the "struct" type in C. Any value must fall into one of these types. Likewise, any variable has an associated type constraining the possible values for that variable. The kinds are also built up inductively from structured types together with a construct that describes functions from kinds to structured types.

The semantic intuition for objects is that they correspond to physically real objects obeying certain laws, or constraints. For example, an object might correspond to a specific fan in a system. And there might be more than one fan obeying the same constraints. By contrast, a class corresponds to a collection of all similarly behaving objects. So we could have a class named *fan* which embodies the physical specifications of all fans of a particular sort. Then, we might have objects *fan-a* and *fan-b* both of the class *fan*. Thus *fan-a* and *fan-b* are distinct objects (so they may be in different states at a given time), yet they both obey the same laws. Somewhat more formally, in the simple case a class is a collection of laws. However, a class may depend on other classes in its definition. So, in general, a class is a function from *n*-tuples of classes to a collection of laws. [N.B. *n* may be zero here, taking care of the simple case.] An object is a variable of a structured type, constrained by the laws of some class.

A connection is an equality constraint between (fields of) variables, together with an indication of the role that the constrained variables play in a network. In particular, a connection tells us where the value for that variable is obtained, i.e., from exogenous sources, by feedback from solution of the network, as unknowns in the network that can be solved iteratively, or as unknowns that must be solved explicitly.

6.2.2 Naming Things

As usual, we have to provide some sort of collection of names for the entities of a category. For most purposes the collection of C identifiers will suffice. So we have our first (informal) grammar rule:

<identifier> ::= The usual C identifiers

A variable is named by an identifier, as are objects and classes.

[N.B. An object will go by the same name as the variable of which it is composed.] The names of types are built inductively following the inductive definition of types.

<type> ::= *<simple-type>* | *<struct-type>*
<simple-type> ::= **double** | **int** | **bool** | **...**
<struct-type> ::= (*<typed-id>*(,*<typed-id>*)*)
<typed-id> ::= *<identifier>* [*<type>*]

If a *<typed-id>* is an *<identifier>* only, it is implicitly assumed to be of type **double**.

Because *<type>* expressions can be rather verbose, we also allow abbreviations to be defined by the following construct.

<type-def> ::= **type***<identifier>* = *<type>*

And we allow *<struct-type>* to use these abbreviations. Thus, we add a clause to the grammar rule for *<struct-type>*:

<struct-type> ::= *<identifier>* | *<type-id>* (,*<type-id>*)*

Similarly, kinds are defined inductively, allowing for defined abbreviations.

<kind> ::= *<identifier>* | *<struct-type>* | [*<kind-list>*>>*<struct-type>*]
<kind-list> ::= *<kind>*(* *<kind>*)*
<kind-def> ::= **kind** *<identifier>* = *<kind>*

If **x** names a variable of structured type that has a field named **field**, then we can indicate the value of that field by writing **x.field**. In general, names of values obtained in this way are called descriptors.

<descriptor> ::= *<identifier>* (.*<identifier>*)*

Connections do not have to be named. However, if the constrained fields are to be used as a single unit elsewhere, then they must share a name. So a connection can optionally be named by a simple *<identifier>*. The effect of this is to associate a variable with the name *<identifier>* with the connection.

6.2.3 Declaring Objects

An object is declared by specifying its name, and its associated class. Remember that a class may depend on other classes, so specifying a class may involve parameters. Thus,

<declaration> ::= **declare** *<identifier>*(,*<identifier>*)[*<param-list>*];
<class-instance> ::= (*<identifier>* | *<class>*)[*<param-list>*]
<param-list> ::= [*<class-instance>* (;*<class-instance>*)*]

6.2.4 Making Connections

To specify a connection, we give a keyword indicating the relation of the constrained fields to the advancement of time, followed optionally by a name for the connection, followed by a list of fields of variables (typically, fields of objects) that are to be equated, and finally followed by a specification of how the value of the connection should be obtained from previous time steps (if this is appropriate).

There are five sorts of connections: inputs, feedbacks, unknowns, clocks, and signals. Inputs are essentially initial values. They do not change over time. Feedbacks are values that cannot be solved for; they are used to communicate values from one time step to the next. Unknowns are values that are suitable for solving at a time step. Clocks are mechanisms for advancing the system time. Signals are values similar to unknowns, but which are not allowed to enter into the iterative solution for unknowns. Typically, signals will be of some discrete type, e.g., boolean, so that Newton-Raphson would not make sense if it involved values of that type. In addition to these five sorts of connection, we allow for a "link" connection, which simply inherits its sort from the fields it equates. The grammar for the connections is this.

<connection> ::= *<link>* | *<unknown>* | *<feedback>*

		<clock> | *<input>* | *<signal>*
<link>	::=	**link** *<connection-id>*(*<descriptor-list>*);
<unknown>	::=	**unknown** *<connection-id>*(*<descriptor-list>*) **predict-init** *<expr>***predict-next** *<expr>*;
<feedback>	::=	**feedback** *<connection-id>*(*<descriptor-list>*) **init** *<expr>***next** *<expr>*;
<clock>	::=	**clock** *<connection-id>*(*<descriptor-list>*) **init** *<expr>***next** *<expr>*;
<input>	::=	**input** *<connection-id>*(*<descriptor-list>*);
<signal>	::=	**signal** *<connection-id>*(*<descriptor-list>*);
<descriptor-list>	::=	*<descriptor>*(, *<descriptor>*)*
<connection-id>	::=	*<identifier>* | *<typed-id>*
<expr>	::=	Any C expression with variable names drawn from the names of connections.

We assume that several clocks can be extant in a simulation. This means that the current time should be available to the system as a specially named variable, say **current-time.** The value of a clock connection will advance only when it is scheduled to tick. Thus, if **t** is a clock connection, then the boolean expression **current-time** = **t** will evaluate to true if and only if the clock **t** has just ticked.

6.2.5 Defining classes

A class is defined by giving a *<struct-type>* called the *<interface>*, and then specifying constraints on values of the interface type. Typically, the interface has two parts: the object interface and the class interface (the class interface may be empty). The object interface simply tells us the type of objects of the defined class. The class interface tell us that the class itself has a variable associated with it. This is for the purpose of specifying information shared amongst all objects of a particular class. The class interface is similar in spirit to the notion of a class variable in Smalltalk, except that class variables in Smalltalk are typically hidden from all objects outside the class, whereas a class interface is necessarily visible to the rest of the system.

The grammar for class definitions is the following.

<class-def>	::=	*<identifier>*=*<class>* *<identifier>*;
<class>	::=	(*<simple-def>* | *<macro-def>* | *<switch-def>*
<simple-def>	::=	**simple class** *<interface>* *<inverse>****end**
<macro-def>	::=	**class**[*<param-spec>*] *<struct-type>* [**class interface** *<struct-type>*] *<library>** *<definition>** *<declaration>** *<connection>** *<equation>** **end**
<switch-def>	::=	**switch**[*<param-spec>*] *<struct-type>* [**class interface** *<struct-type>*]**is**

```
                        <library>*
                        <definition>*
                        <cases>*
                      end <identifier>;
<definition>   ::=     | <interface-def> | <kind-def>
```

The syntactic category *<library>* will be explained below. The *<param-spec>* part of this definition indicates (optionally) the kinds of classes on which the macro class depends. So, a *<param-spec>* is given by

```
<param-spec> ::= [<identifier>:<kind>(;<identifier>:<kind>)*]
```

And *<cases>* is essentially like the *switch* construct in C.

```
<cases> ::= (<bool-expr>:<class-instance>;)*
            else: <class-instance>;
```

Here *<bool-expr>* is just an *<expr>* that returns a boolean value. The semantics of a switch is that at each time step, the boolean expressions are evaluated in order until the first true expression is found. Then the switch class is constrained as if it were defined by the accompanying declared class. If all expressions are false, the "**else:**" class is used instead.

6.2.6 Equational Constraints

In defining a macro class, we can specify that certain variables are constrained by an equation. The effect of this is essentially to define an anonymous simple class and an anonymous object of that class, the interface of which is connected to the variable occurring in the equation.

```
<equation> ::= eqn <expression>=<expression>end eqn;
```

6.2.7 Libraries

For the sake of modularity, a collection of definitions can be stored in a separate file to be used in other definitions. So, a *library* is simply a file containing *<definition>**. To refer to a library in another file, we have the construct

```
<library> ::= library <filename>;
```

where *<filename>* is the name of a file containing a library.

6.2.8 Systems

A system is a special macro class, analogous to the main procedure in a C program. One and only one system must be specified in any simulation. When SPANK runs a system, it instantiates an object of the system class with initial values determined by the user, and then runs the simulation. A system is specified by the following.

```
<system>   ::=   <library>*
                 system <identifier>[<interface>]is
                     <library>*
                     <definition>*
                     <declaration>*
                     <connection>*
                 end <identifier>.
```

6.3 Example

The ideas formalized above are made concrete in the example shown in the Appendix. There we show a CDL problem specification for the three-node room problem described in Sec. 5.1, **Three Node Room**. Comments in the code should allow the dedicated reader to see how the CDL specifies the problem. We will not describe the example line by line, but a few comments are in order.

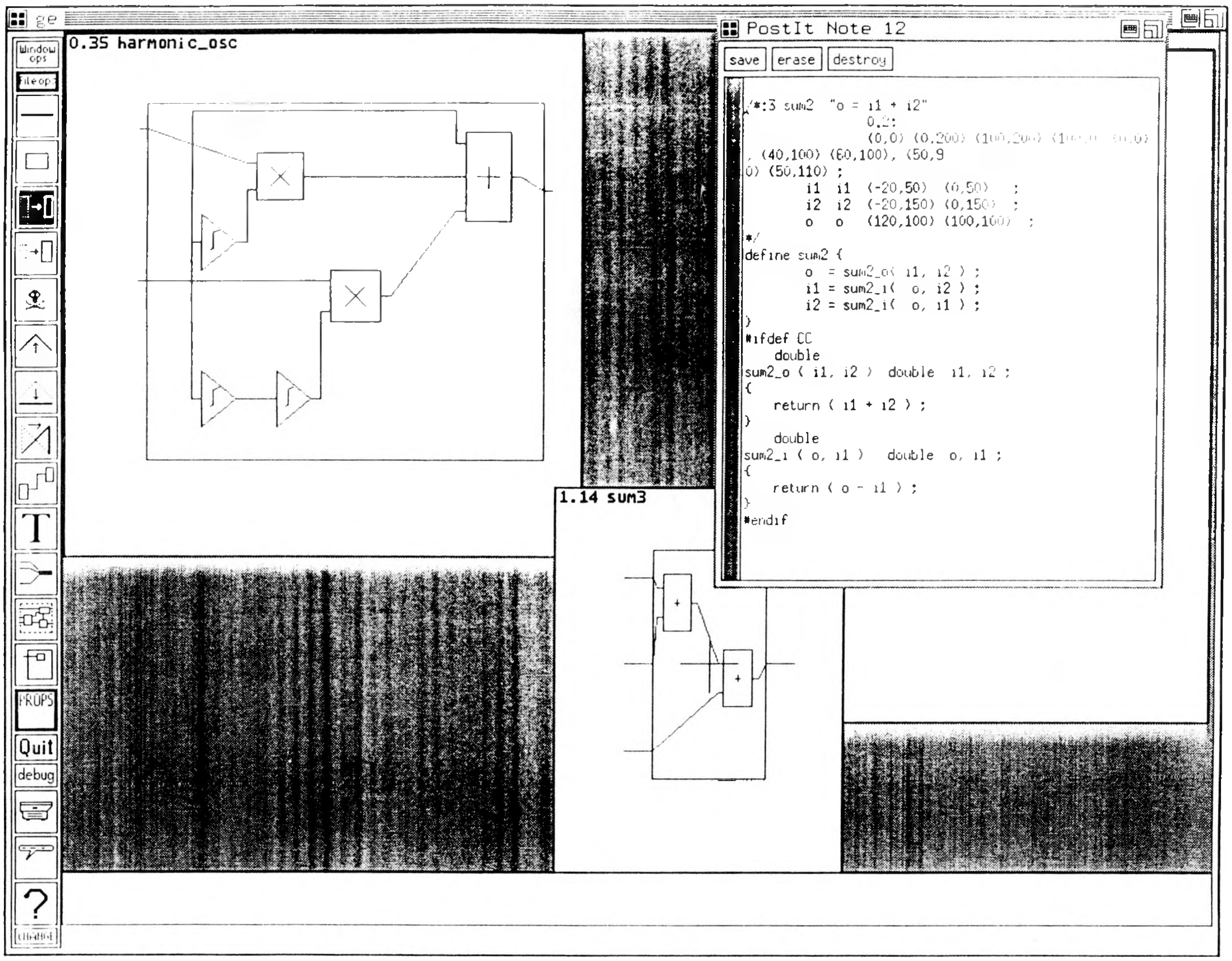
First, note that the system definition (called "room") is completely in terms of objects that have intuitive meanings, strongly coupled to the physics of the problem. Numerical details are contained within the objects, out of view at this level. Yet the knowledgeable user can, for example, change to a different integration method, presently Milne4, by changing the argument in the declaration of the massive object, "floor". Also note that we can link the room interface variables h, alpha, sigma, T, T_air, and dt directly to interfaces of objects comprising the system wherever needed; this is exactly the same as when defining a class in terms of simple classes (or other classes), thus demonstrating the intended seamless transition from class definitions to problem definitions.

Classes used in the system definition are defined in separate CDL files referenced with the keyword **library**. These files are included in the Appendix. For example, energy.cdl contains all classes pertaining to the problem physics, while Milne4 has those for the Milne fourth order integration method. In energy.cdl we see how simple classes are defined as a single equation. This equation is placed directly in the CDL file, in contrast to the current SPANK implementation which requires an intermediate C function definition. In the same file we see the class "air" defined in terms of the simple class "conductive_heat", augmented with one equation. In the class "massive" we see that classes can also employ other classes in their definition.

7. Graphical User Interface

Currently users of EKS/US must express their problems textually using the Network Specification Language. While this language has served well for the development and testing of the program, it leaves much to be desired as an intuitive and efficient user interface. Currently under development is a graphical user interface called the Kernel Graphical Editor (KGE) that will come closer to these goals.

The basic idea of the KGE is that objects, macro objects, and problems are specified by the user by manipulation of screen icons. Available object classes are selected from libraries, using a browser, and then appear as icons on a menu, from which they can be selected (instantiated) and placed anywhere on the screen. Once placed, they can be interconnected to form a macro object or a problem. The objects can also be moved, deleted, or modified in any way. Also, any object can be expanded to show internal structure when needed. When the problem image is complete, the KGE will create a CDL file for SPANK processing. The implementation employs the X-Windows system in order to allow maximum portability. Figure 14 shows a preliminary KGE screen.



```
/*:3 sum2  "o = i1 + i2"
              0.2:
              (0,0) (0,200) (100,200) (100,0) (0,0)
, (40,100) (60,100), (50,9
0) (50,110) ;
        i1  i1  (-20,50)  (0,50)   ;
        i2  i2  (-20,150) (0,150)  ;
        o   o   (120,100) (100,100)  ;
*/
define sum2 {
        o  = sum2_o( i1, i2 ) ;
        i1 = sum2_i(  o, i2 ) ;
        i2 = sum2_i(  o, i1 ) ;
}
#ifdef CC
    double
sum2_o ( i1, i2 )  double  i1, i2 ;
{
    return ( i1 + i2 ) ;
}
    double
sum2_i ( o, i1 )   double  o, i1 ;
{
    return ( o - i1 ) ;
}
#endif
```

Figure 14: Example screen from the Kernel Graphical Editor (KGE), the graphical user interface for EKS/US. The three windows show: harmonic oscillator problem with multiplier, sum, and integrator objects and links (upper left); the "sum" macro object showing its constituent objects (lower middle); textual input for the "sum" object with associated C code. Buttons along the left side of the screen perform operations such as positioning objects in a window, drawing links between objects, and grouping objects into macro objects.

8. Conclusions

The current state of the U.S. Energy Kernel System has been reviewed, and its relationship to the Simulation Problem Analysis Kernel (SPANK) has been described. It currently has the capability to simulate general differential-algebraic systems, with modest flexibility in specification of numerical methods to be used. Also, objects, macro objects, and problems can be described in concise textual form and symbolically manipulated to create needed SPANK and C code for the simulation. Ten application problems that have been solved were briefly discussed. Finally, we described current work aimed at improving EKS/US capability and user interaction mechanisms. The Component Definition Language is the result of reassessing the semantics of dynamic model specification and, when implemented, will allow more complex system models to be expressed, as well as affording greater flexibility in specifying numerical methods. The Kernel Graphical Editor, currently under development using the X-Windows protocols, will allow users to define simulation problems on the computer screen using pointing devices, rather than expressing the problem in a textual language.

EKS/US will be released for public use in 1992/93 after we have completed the user interface, implemented the Component Definition Language, and built up the object library. In parallel, we plan to integrate the EKS/US approach into the SYSTEMS and PLANT portions of the existing DOE-2 hourly energy analysis program [BIRDSALL 1990]. The resulting program, to be called DOE-3, will allow object-oriented techniques to be used in the context of a whole-building program that many users are already familiar with. With DOE-3 users will be able to configure and model advanced HVAC components and systems that cannot be simulated with DOE-2, while retaining DOE-2's powerful LOADS program.

9. References

Aho 1983	Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman, **Data Structures and Algorithms,** Addison-Wesley.
Anderson 1986	Anderson, J.L., *A Network Definition and Solution of Simulation Problems,* Lawrence Berkeley Laboratory report LBL-21522.
Birdsall 1990	B. Birdsall, W.F. Buhl, K.L. Ellington, A.E. Erdem and F.C. Winkelmann, *Overview of the DOE-2 Building Energy Analysis Program, Version 2.1D,* Lawrence Berkeley Laboratory report LBL-19735.
Bonin 1987	Bonin, J.L., J.Y. Grandpiex, A. El Hasnaoui and J.L. Joly, *Coupling Analysis in Building Thermal Simulation: The ZOOM Program,* Proc. Int'l Solar Energy Society Conference, Hamburg.
Buhl 1989	Buhl, W.F., E.F. Sowell and J.-M. Nataf, *Object-Oriented Programming, Equation-Based Submodels, and System Reduction in SPANK,* Proc. Building Simulation '89, Vancouver, B.C.; Lawrence Berkeley Laboratory report LBL-28272.

Clarke 1986	Clarke, J.A., *The Energy Kernel System: A Technical Overview,* Proc. Second International Conference on System Simulation in Buildings, Liege.
Clarke 1987	Clarke, J.A., *The Energy Kernel System: An Overview in Support of Three Grant Proposals,* Energy Simulation Research Unit, University of Strathclyde, U.K.
Elmqvist 1978	Elmqvist, H., *A Structured Model Language for Large Continuous Systems,* Ph.D. Thesis. Report CODEN: LUTFD2/(TFRT-1015), Dept. of Automatic Control, Lund Institute of Technology, Sweden.
Even 1979	Even, S., **Graph Algorithms,** Computer Science Press. Inc., Potomac, Maryland.
Hirsch 1985	*A Plan for the Development of the Next Generation Building Energy Analysis Computer Software*, Proc. Building Simulation '89, Vancouver, B.C.; Lawrence Berkeley Laboratory report LBL-19830.
HVACSIM$^+$ 1985	*HVACSIM$^+$ Building Systems and Equipment Simulation Program (Reference Manual and Users Guide),* U.S. Department of Commerce, National Institute of Science and Technology, Gaithersburg, MD.
Inard 1988	Inard, C. and N. Molle, *Etude du Couplage Thermique Entre Des Corps de Chauffe et un Local* CETIAT, BP 6084, F-69104, Villeurbanne, France, 1988.
Johnson 1988	Johnson, D., *Matching Algorithms for Equation Selection,* Lawrence Berkeley Laboratory report LBL-28276.
Karp 1972	Karp, R.M., *Reducibility Among Combinatorial Problems,* in R.E. Miller and J.W. Tatcher's **Complexity of Computer Computations**, Plenum Press, New York, pp 85-103, 1972.
LBL 1985	*A Proposal to Develop a Kernel System for the Next Generation of Building Energy Simulation Software,* Simulation Research Group, Lawrence Berkeley Laboratory, Internal Report.
Levy 1986	Levy, H., *A Comparison of Low Complexity Algorithms for Finding Small Cycle Cut sets,* Proc. 24th Annual Allerton Conference on Communications, Control, and Computation, Allerton House, Monticello, IL, pp.49-58.
Levy 1988	Levy, H. and D.W. Low, *A Contraction Algorithm for Finding Small Cycle Cut sets,* J. Algorithms, 9, pp.470-493.
MACSYMA 1983	*MACSYMA Reference Manual, Version 10,* Mathlab Group, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

MAPLE 1985	Char, B.W. et.al., *MAPLE User's Guide: 1st Leaves, A Tutorial Introduction to MAPLE and the MAPLE Reference Manual*, 4th Edition. Series Title: **WATCON Series in Computer Science and Computer Applications.** WATCOM Publications, Ltd., Waterloo, Ontario, Canada, 1985.
Mattsson 1989	Mattsson, S.V., *Concepts Supporting Reuse of Models*, Proc. Building Simulation '89, Vancouver, B.C.
Moshier 1990	Moshier, M.A., *Specifying Dynamic Models in the Simulation Problem Analysis Kernel,* Proc. Modeling and Simulation on Microcomputers, Society for Computer Simulation, San Diego: Lawrence Berkeley Laboratory report LBL-28275, October 1989.
Nataf 1990	Nataf, J.-M. and F.C. Winkelmann, *Dynamic Simulation of Liquid Desiccant Cooling System Using the Simulation Problem Analysis Kernel*, Lawrence Berkeley Laboratory report — in preparation.
Rand 1984	Rand, R.H., **Computer Algebra in Applied Mathematics: An Introduction to MACSYMA.** Pitman Advanced Publishing Program, Boston.
REDUCE 1987	*REDUCE-3 User's Manual, Version 3.3*, Rand Corporation, Pub. CP78(7/87).
Sahlin 1988	Sahlin, P., *MODSIM: a Program for Dynamical Modeling and Simulation of Continuous Systems,* Report from the Swedish Institute of Applied Mathematics, P.O. Box 26300, S-100 41 Stockholm, Sweden.
Sowell 1973	Sowell, E.F. and P.F. O'Brian, *The Transport of Lighting Energy,*, ASHRAE Trans. pt.2.
Sowell 1984	Sowell, E.F., et al., *Generation of Building Energy System Models,* ASHRAE Trans. vol. 90.
Sowell 1986	Sowell, E.F., W.F. Buhl, A.E. Erdem, and F.C. Winkelmann, *A Prototype Object-Based System for HVAC Simulation,* Proc. Second International Conference on System Simulation in Buildings, Liege; Lawrence Berkeley Laboratory report LBL-22106.
Sowell 1988	Sowell, E.F. and W.F. Buhl, *Dynamic Extension of the Simulation Problem Analysis Kernel (SPANK),* Proc. User-1 Conference, Ostend, Belgium; Lawrence Berkeley Laboratory report LBL-26262.
Sowell 1989	Sowell, E.F. and P. Sahlin, *Neutral Format and Automatic Translation for Building Simulation Submodels*, Proc. of Building Simulation '89, Vancouver, B.C.; Lawrence Berkeley

Laboratory report LBL-28274.

Sowell 1990	Sowell, E.F., J.-M. Nataf and F.C. Winkelmann, *Radiant Transfer Due to Lighting: An Example of Symbolic Model Generation for SPANK,* October 1989, rev. January 1990. Proc. Society for Computer Simulation 1990 Western Multiconference, San Diego; Lawrence Berkeley Laboratory report LBL-28273.
Spitler 1991	Spitler, J., C. Pedersen, D. Fisher, P. Menne and J. Cantillo, *An Experimental Facility for Investigation of Interior Convective Heat Transfer*, ASHRAE Transactions, Vol.97, Pt.1, 1991. and Spitler, J., C. Pedersen and D. Fisher, *Interior Convective Heat Transfer in Buildings with Large Ventilative Flow Rates*, ASHRAE Transactions, Vol.97, Pt.1, 1991.
TRNSYS 1983	*A Transient Simulation Program,* Solar Energy Laboratory, University of Wisconsin.

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

10. Appendix: Example of Problem Specification in CDL

```
/* File: room.cdl */

/* A system modeling energy balance in a room with
      (1) massive floor,
      (2) massless ceiling,
      (3) height/floor-area negligible.

as described in Sec. 5.1 and in [Sowell 1988].
We assume that the floor, air and ceiling are held at a constant temperature T0 prior
to the simulation; and at time t0 the air temperature is instantaneously changed to
T_air, and is held constant thereafter. The model then simulates the ensuing loads.


(1) The ceiling is modeled by the energy balance equation for a massless object:
      0 = sigma*(T_rad**4 - T**4) + h*(T_air - T);

(2) The air is modeled by the air energy balance equation

      qo = h*(T_surface1 - T) + h*(T_surface2 - T);

(3) The ceiling is modeled by the energy balance differential equation for a
massive object:

      alpha*T' = (sigma*(T_rad**4 - T**4) + h*(T_air - T)

      where T = integral of T' dt;

with
  h     = convective film coefficient
  alpha = floor thermal capacitance
  sigma = Stefan-Boltzmann constant

In this file, the integration in (3) is done by a 4th order Milne method. Comments
indicate exactly where changes must be made to change to another integration
method.
*/
library stdio.cdl      /* a library that implements the standard i/o */
library energy.cdl      /* read in the energy balance objects */
library Milne4.cdl      /* read in the 4th order Milne method. Change this to
                           "library RungeK2.cdl" for 2nd order Runge-Kutta */
```

```
system room(h, alpha, sigma,  T,  T_air,  T0,  t0, dt)
 declare ceiling  massless;
 declare air air;
 declare floor massive[Milne4];  /* Replace "Milne4" with "RungeK2"
for 2nd order Runge-Kutta */
 declare report_load reporter;   /* reporter is an output class defined in stdio.cdl
                                 that records its interface at each time step */

 input h(room.h, floor.h, air.h, ceiling.h); /* convective film coefficient */
 input alpha(room.alpha, floor.alpha);   /* floor thermal capacitance */
 input sig(room.sigma, floor.sigma, ceiling.sigma);/* Stefan-Boltzmann constant */
 input T0(room.T0, floor.T0);  /* As everything else is massless,  the floor is
                                the only object that "remembers" the
                                temperature prior to simulation time */
 input T_air(room.T_air, floor.T_air, air.T, ceiling.T_air); /* air node temperature */
 input dt(room.dt, floor.dt);        /* time step (in hours) */


 link qo_air(air.q, report_load.x);     /* load */
 link T_floor(floor.T, air.T_surface1, ceiling.T_rad); /* floor temperature */
 link T_ceiling(floor.T_rad, air.T_surface2, ceiling.T); /* ceiling temperature */
 link t(floor.time,  report_load.time);  /* communicate the time from
                                            the floor to the reporter */

end room.


/*-----------------------------------------------------*/
/* File: energy.cdl */
/*

This file contains definitions for various heat balance equations.  As of now,  we have
three kinds implemented: massless,  air and massive.  The definitions should make
the underlying models evident.

CONVENTION: Loads transfers will always be measured as positive values indicat-
ing incoming heat.

*/
radiant_heat = simple class (T, T_rad, q, sigma)  /* radiant heat transfer */
  q = sigma*(T_rad**4 - T**4)
end radiant_heat;


conductive_heat = simple class(T, T_cont, h, q) is        /* conductive heat transfer */
  q  = h*(T_cont-T)
end conductive_heat;
```

```
air = class (T, T_surface1, T_surface2, h, q)

/* An air object obeys the heat balance:

      q = h*(T1-T) + h*(T2-T)

where q is the load,
     T is the temp of the air object,
     T1 and T2 are temps of surfaces.
*/

  declare s1cond,  s2cond conductive_heat;

  link T(air.T, s1cond.T, s2cond.T);
  link T_surface1(air.T_surface1, s1cond.T_cont);
  link T_surface2(air.T_surface2, s1cond.T_cont);
  link h(air.h, s1cond.h, s2cond.h);

  eqn
    air.q = s1.q + s2.q
  end eqn;
end air;

massless = class(T, T_rad, T_air, h, sigma)

/* A massless object obeys the heat balance:

      0= sigma*(T_rad**4 - T**4) + h*(T_air - T)

where T is temp of the object,
     T_air is temp of air,
     T_rad is temp of nearby radiator
*/

  declare r radiant_heat;      /*  q = sigma*(T_rad**4 - T**4) */
  declare c conductive_heat;   /*  q = h*(T_cont-T) */

  link h(massless.h, cv.h);
  link sigma(massless.sigma, rd.sigma);
  link T(massless.T, cv.T, rd.T);
  link T2(massless.T_rad, rd.T);
  link T3(massless.T_air, cv.T_cont);

  eqn
    0 = r.q + c.q
  end eqn;
end massless;
```

```
massive = class[Int[ODE(y, y', t)](y, y', t, dt, y0, t0)](T, T_rad, T_air, t, dt, T0, t0, h, sigma, alpha)

/* A massive object obeys the heat balance equation:

      alpha*T' = sigma*(T_rad**4 - T**4) + h*(T_air-T)

where T is temp of the object,
     T_air is temp of the surrounding air
     T_rad is temp of nearby radiator

Because this is a dynamic object (involving T'), it is only well defined when given a
method of integration Int. The class Int has the interface (y, y', t, dt, y0, t0) and
depends on a class ODE with interface (y, y', t). Specifically, this definition assumes
that the integrator doesn't require any start-up values beyond the initial conditions:
(y0, t0).

*/
Mass_ode = class(y, y', t) class interface (T_rad, T_air, h, sigma, alpha)
   declare r radiant_heat;    /* q = sigma*(T_rad**4 - T**4) */
   declare c conductive_heat; /* q = h*(T_cont-T) */

   link sigma(Mass_ode.sigma, r.sigma);
   link h(Mass_ode.h, c.h);
   link y(Mass_ode.y, rd.T, c.T);    /* T is renamed y for the ODE */
   link T_rad(Mass_ode.T_rad, r.T_rad);
   link T_air(Mass_ode.T_air, c.T_cont);

   eqn
         /* T' is named y' for the ODE */
     Mass_ode.alpha * Mass_ode.y' = r.q + c.q
   end eqn;

  end Mass_ode;

  declare mass Int[Mass_ode]; /* the mass object integrates y by the
                                 method implemented in the class Int */

  link T(massive.T, mass.y);  /* the integrated variable y is really T */
  link T_air(massive.T_air, Mass_ode.T_air);
  link T_rad(massive.T_rad, Mass_ode.T_rad);
  link t(massive.t, mass.t);
  link dt(massive.dt, mass.dt);
  link t0(massive.t0, mass.t0);
  link T0(massive.T0, mass.y0);
  link sigma(massive.sigma, Mass_ode.sigma);
  link alpha(massive.alpha, Mass_ode.alpha);
  link h(massive.h, Mass_ode.h);
  link T'(massive.T', mass.y');
end massive;
```

```
/*----------------------------------------------------------*/
/* File: Milne4.cdl  */
/*

 In this file we implement a 4th Order Milne integration method.
 See Conte and DeBoor p385 for an explanation of the method.
*/

type diff_eq_type = (y, y', t);
type int_diff_eq_type = (y, t, y0, t0, dt);

Milne4 = class[ODE:diff_eq_type](int_diff_eq_type)

  declare eq, eq_next of class ODE; /* eq is used in the corrector part,
                                     eq_next in the predictor part */
  declare p simple class(y_kp1, y_km3, f_k, f_km1, f_km2, dt) /* 4th order Milne predictor */
    y_kp1 = y_km3 + 4*dt*(2*f_k - f_km1 + 2*f_mk2)/3;
  end p;

  declare c simple class(y_kp1, y_km1, f_kp1, f_k, f_km1, dt) /* 4th order Milne corrector */
    y_kp1 = y_km1 + dt*(f_kp1 + 4*f_k + f_km1)/3;
  end c;

  declare timestep sum;

  unknown y(Milne4.y, c.y_kp1, eq.y) init y0 predict y_next; /*y is solved for by corrector */
  unknown y'(c.f_kp1, p.f_k, eq.y') init 0 predict y'_next; /* y' is solved for by corrector */
  clock t(Milne4.t, eq.t) init t0 next t_next;

  link dt(Milne4.dt, c.dt, p.dt);   /* use constant time step of dt */
  link y0(Milne4.y0);               /* initial value of y */
  link t0(Milne4.t0);               /* simulation start time */

  feedback y_km1() init y0 next y;      /* cascade historical values of y */
  feedback y_km2(c.y_km1) init y0 next y_km1;
  feedback y_km3(p.y_km3) init y0 next y_km2;

  feedback y'_km1(c.f_k, p.f_km1) init 0 next y'; /* cascade historical values of y' */
  feedback y'_km2(c.f_km1, p.f_km2) init 0 next y'_km1;

  unknown y_next(p.y_kp1, eq_next.y);   /* predicted next value of y */
  unknown t_next(eq_next.t);            /* next time */
  unknown y'_next(eq_next.y');    /* predicted next value of y'
                                   (calculated from y_next and t_next */

  eqn
    t_next = t + dt
  end eqn;
end Milne4;
```