# Lawrence Berkeley National Laboratory
## Recent Work

### Title
Specifying Dynamic Models in the Simulation Problem Analysis Kernel

### Permalink
https://escholarship.org/uc/item/4s13w9kt

### Authors
Moshier, M.A.
Sowell, E.F.

### Publication Date
1989-10-01

eScholarship.org Powered by the California Digital Library
University of California

LBL-28275



# Lawrence Berkeley Laboratory

UNIVERSITY OF CALIFORNIA

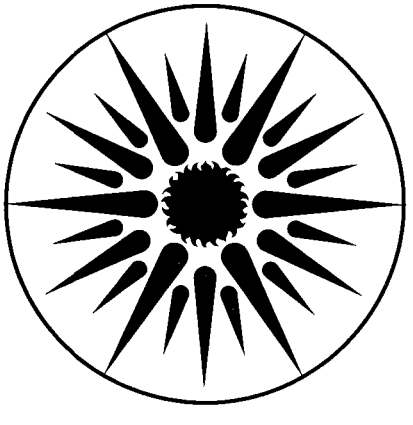
## APPLIED SCIENCE DIVISION

To be presented at the Computer Simulation 1990 Multiconference, San Diego, CA, January 17–19, 1990, and to be published in the Proceedings

## Specifying Dynamic Models in the Simulation Problem Analysis Kernel

M.A. Moshier and E.F. Sowell

October 1989



**APPLIED SCIENCE DIVISION**

| LOAN COPY |
|Circulates |
|for 2 weeks|

Bldg. 50 Library.

LBL-28275
Copy 2

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

## DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

Presented at the Society for
Computer Simulation 1990
Multiconference, San Diego, CA
Jan 17-19, 1990

LBL-28275

October 15, 1989

# Specifying Dynamic Models in the Simulation Problem Analysis Kernel*

M. Andrew Moshier
Program in Computing, University of California, Los Angeles, CA 90024
Edward F. Sowell
Computer Science Dept., California State University, Fullerton, CA

**Abstract**

The Simulation Problem Analysis Kernel (SPANK) is a prototype implementation of the Energy Kernel System (EKS). It is intended to demonstrate some of the features that have been argued as necessary for future simulation systems. In particular, with earlier versions of SPANK, the model designer can specify static behavior as modeled by systems of algebraic equations. More recent work has led to extensions that allow the designer to specify dynamic behavior as modeled by systems of equations together with certain predefined integration methods. One drawback of this approach is that the integration methods must be of a specific kind involving at most a fixed number of values from previous time steps. In this paper, we propose to extend SPANK in a more flexible way to give the designer the ability to specify any of the standard difference methods for initial value problems. These include Runge-Kutta methods as well as predictor-corrector methods.

The development of our extensions begins with a close look at how the static version of SPANK specifies models. On this foundation, we then look at two very simple integration methods, Euler's implicit and explicit methods, for clues as to how we can extend the system description language to admit dynamics. With the extensions to SPANK called for by looking at Euler's methods, we show that Runge-Kutta and predictor-corrector methods can also be specified. From this, we argue that the extensions proposed herein are suitable for specifying arbitrary finite difference integration methods. We conclude with a brief discussion of further extensions that we believe would make the SPANK system more useful for dynamic simulations.

*This work was supported by the Assistant Secretary for Conservation and Renewable Energy, Office of Buildings and Community Systems, Building Systems Division of the U.S. Dept. of Energy under contract No. DE-AC03-76SF00098, and by the National Science Foundation under grant MSM-8614974.

# 1 Introduction

The Simulation Problem Analysis Kernel (SPANK) [SBEW86, SB89] demonstrates an object-oriented method of constructing models of large static problems that closely mimics methods of constructing physical systems. Namely, models are built from (models of) simple components by specifying how the components are connected. Simple components are modeled by systems of algebraic equations. Large systems are modeled by a collection of simple component models together with descriptions of how the individual components are related. The promise of this approach is that the resulting description of a large system is essentially a schematic of the corresponding physical system, and is constructed in similar manner to the physical system. Thus for example, verification of the correctness of the model is vastly simpler than verification that a single monolithic system of equations is correct — we simply verify that the model was constructed to faithfully mimic the construction of the physical system. Similarly motivated simulation environments are described in [Cla88b, Mat88, Mat89b, Mat89a, And89a, And89b, Sah88, SS89]. Also, see [Cla87, Cla88a, Hir88] for further discussions of the motivations for the Energy Kernel System (EKS) of which SPANK can be seen as a prototype.

In a later paper [SB88], it is shown that this same method can be extended to dynamic behavior modeled by systems of differential equations. The method proposed there, however, assumed a specific set of predictor-corrector integration formulas. In this paper, we show how the SPANK method can be extended to permit a wide range of integration methods, including user defined methods. The extension involves solving two separate, but related problems. First, SPANK uses the Newton-Raphson method for finding solutions to the algebraic systems, but for systems of differential equations there is no obvious choice for a single method of integration. Hence, we must be able to support many different methods, preferably in such a way as to allow new methods to be added. Second, specification of a system of differential equations ought to follow in the spirit of the earlier system. That is, small processes, e.g., the dynamic behaviors of small components, ought to be modeled by small systems of equations that can be linked together to model larger complicated behaviors. We show that these two problems can be solved together in a very uniform way that allows both components and integration methods to be described in the same formalism.

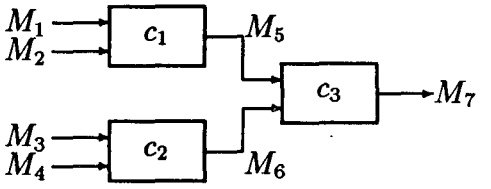



Figure 1: A system of three air collectors

# 2 Overview of SPANK: Static Models

Suppose we are asked to construct a model of the static behavior of a certain configuration of three air collectors, represented schematically in Figure 1. For the sake of simplicity, let us assume that we are only interested in modeling the rate of flow of dry air through the various ducts, given rates at the ducts numbered 1 through 4. Let us further assume that differences in pressure are not of interest. We can model this by the following system of equations.

$$\begin{aligned} 0 &= M_1 + M_2 - M_5 \\ 0 &= M_3 + M_4 - M_6 \\ 0 &= M_5 + M_6 - M_7 \end{aligned} \tag{1}$$

where $M_1$ through $M_4$ are given, and $M_5$ through $M_7$ are unknown mass flows to be found.

Looking at a typical text on numerical analysis, one might get the impression that such a system of equations springs forth complete from the mind of the designer like Pallas Athena from the mind of Zeus. But of course it does not. Rather it is arrived at, more mundanely, in stages with each stage ensuring that some aspect of the physical system is faithfully represented. In SPANK, these stages are reflected in the system description language, so that a description not only accurately specifies a system of equations, but also specifies the design steps taken to arrive at the system. Thus, in order to understand the system description language used in SPANK, we must understand these design steps.

Consider the configuration of air collectors given in Figure 1 again, under the same assumptions. We know that conservation laws require that the

behavior at each collector be modeled by an equation of the form

$$0 = m_1 + m_2 - m_3 \tag{2}$$

where $m_3$ is the rate of flow of the outgoing stream of air and $m_1$ and $m_2$ are the rates of flow of the two incoming streams of air. Clearly, the actual equations of our system must involve problem variables corresponding to the flow rates at the various ducts, rather than the abstract quantities $m_i$. So we refer to Equation 2 as an *equation schema* to distinguish it from particular instances where problem variables have been substituted for the $m_i$.

As the first stage in the design, we chose three instances of the equation schema 2, corresponding to the three components of the physical system.[1] For convenience, we give these three instances the names $c_1$, $c_2$ and $c_3$. Then, we can refer to the variables in equation $c_1$ as $c_1.m_1$, $c_1.m_2$ and $c_1.m_3$, and likewise for the variables in the other equations. Thus, as the result of the first step in the design, we have the following three instances of 2.

- Equation $c_1$ is $0 = c_1.m_1 + c_1.m_2 - c_1.m_3$
- Equation $c_2$ is $0 = c_2.m_1 + c_2.m_2 - c_2.m_3$
- Equation $c_3$ is $0 = c_3.m_1 + c_3.m_2 - c_3.m_3$

Next, we identify the given problem variables with certain variables in the equations. Namely, we make the following identifications:

- Identify $c_1.m_1$ with $M_1$.
- Identify $c_1.m_2$ with $M_2$.
- Identify $c_2.m_1$ with $M_3$.
- Identify $c_2.m_2$ with $M_4$.

Finally, we identify the three unknown problem variables (in general, as many variables as there are equations) with other variables in the equations. Namely, we specify

[1] We assume that the ducts do not contribute any non-trivial behavior to the system. So they are not considered components. If, for example, pressure were taken into account, we might need to consider each duct to be a separate component.

- Identify $c_1.m_3$ and $c_3.m_1$ with $M_5$.
- Identify $c_2.m_3$ and $c_3.m_2$ with $M_6$.
- Identify $c_3.m_3$ with $M_7$.

So the steps involved in constructing the system in Figure 1 are (1) instantiate equations corresponding to system components, (2) identify equation variables with problem variables, and (3) specify which problem variables are to be solved for. Taken together, these three steps completely specify the desired system of three equations in three unknowns. Correspondingly, the SPANK language supports all three of these steps. First, the language has a construct called `declare`, an example of which is

$$\texttt{declare } c_1, c_2, c_3 \texttt{ collector};$$

The effect of this is to form three instances of the equation schema `collector` and call them $c_1$, $c_2$ and $c_3$.[2]

For given problem variables, the language has a construct called `input`, examples of which are

$$\begin{aligned}&\texttt{input } M_1(c_1.m_1);\\ &\texttt{input } M_2(c_1.m_2);\\ &\texttt{input } M_3(c_2.m_1);\\ &\texttt{input } M_4(c_2.m_2);\end{aligned}$$

The first of these specifies that $M_1$ is a parameter to the system identified with the equation variable $c_1.m_1$. And again, the others read similarly. For the unknowns, the language has a construct called `link`, examples of which are

$$\begin{aligned}&\texttt{link } M_5(c_1.m_3, c_3.m_1);\\ &\texttt{link } M_6(c_2.m_3, c_3.m_2);\\ &\texttt{link } M_7(c_3.m_3);\end{aligned}$$

The first of these specifies that $M_5$ is an unknown identified with the equation variables $c_1.m_3$ and $c_3.m_1$. The other two read similarly. Thus, in SPANK the whole system modeling the collectors is specified as in Figure 2. A brief look at this specification shows that certain well-formedness conditions must

[2]The language also supports introduction of new equation schemata such as `collector`. For our purposes, though, we assume that these are already defined in a library of schemata, corresponding to a catalogue of standard components.

```
declare c1, c2, c3 collector;
input M1(c1.m1);
input M2(c1.m2);
input M3(c2.m1);
input M4(c2.m2);
link M5(c1.m3, c3.m1);
link M6(c2.m3, c3.m2);
link M7(c3.m3);
```

Figure 2: SPANK specification of Figure 1

hold in order to completely specify a system of $n$ equations in $n$ unknowns. Namely, (1) there must be exactly the same number of `link` statements as there are equations declared in any `declare` statements; and (2) each equation variable must occur in exactly one `link` or `input` statement. These are trivial syntactic conditions, which, of course, SPANK checks automatically.

**Digression.** An important technical innovation in SPANK, but one that is not directly relevant to this discussion, is that a system of equations as specified above is subjected to certain graph-theoretic algorithms that attempt to minimize the number of unknowns that must be solved for simultaneously. The benefit of this is that, in general, a solution to a non-linear system of $n$ equations in $n$ unknowns requires on the order of $n^3$ operations for each iteration of Newton-Raphson's method. Hence, reduction of the number of variables involved yields significant gains in efficiency. In fact, because practical systems may involve hundreds of equations, this reduction can easily spell the difference between a tractable and an intractable simulation problem. See [SBEW86, SB89] for more discussion of this.

# 3 Dynamic models

Currently, SPANK supports algebraic equations. But in order to simulate dynamic behavior, we must be able to describe differential equations as well. Of course, the field of differential equations is very large. We cannot hope to deal with all of it. Nevertheless, for the purpose of simulating dynamic behavior, we can make three simplifying assumptions. First, we concern our-

selves only with ordinary differential equations. This corresponds roughly to the assumption that our models involve only one independent variable (presumably, this is time) and are suitably discretized. From the point of view of practical *dynamic* models, this makes sense. Second, we deal only with initial value problems. There is nothing radical in this either — most dynamic simulations are cast as initial value problems, and most well-understood solution methods for differential equations solve initial value problems (see for example [Gea71]).

Consider the problem of numerically solving an initial value problem for a specific ordinary differential equation

$$y' = f(t, y) \tag{3}$$

e.g., $y' = ky + \alpha \sin t$, with initial value $y(t_0) = c$ on an interval $[t_0, s]$. [3] The reader will recall that an *exact* solution to this initial value problem is a function $y(t)$ such that $y(t_0) = c$ and

$$dy(t)/dt = f(t, y(t))$$

for all $t$ in $(t_0, s]$. A numerical solution is, in spirit, a function $\hat{y}(t)$ that approximates $y(t)$. In practice, though, a numerical solution method gives values of the approximation $\hat{y}(t)$ at a series of points. That is, a numerical solution is actually just two sequences $t_0, \ldots, t_n = s$ and $c = \hat{y}_0, ..., \hat{y}_n$ so that for every $k \leq n$, $\hat{y}_k$ approximates $y(t_k)$. We will restrict our attention to the problem of how to specify dynamic models for which such numerical solutions are obtained.

To start, we will consider how to specify a numerical solution to the above initial value problem using two very simple solution methods known as Euler's methods. From this, we propose extensions to the SPANK system definition language which support specification of Euler's methods. Then, having the extensions in place, we consider how to use them to specify other more complicated solution methods: Runge-Kutta methods and predictor-corrector methods. These considerations lead us further to refine our proposed extensions.

[3] Actually, the equation may also involve variables other than $y'$, $t$ and $y$. But for clarity of exposition, we will use equations of this form.

## 3.1 Euler's methods

For Euler's methods, we make the observation that in the neighborhood of any particular value $t$, the exact solution $y(t)$ is approximated by the function

$$\hat{u}(x) = y(t) + (x - t)f(t, y(t))$$

So we can approximate $y$ numerically with values $\hat{y}_k$ at points $t_k = t_0 + kh$ as follows

$$\begin{aligned} \hat{y}_0 &= c \\ \hat{y}_{k+1} &= \hat{y}_k + hf(t_k, \hat{y}_k) \end{aligned} \tag{4}$$

This is the *explicit* Euler's method. The *implicit* version uses the (approximate) derivative at $t_{k+1}$ instead of at $t_k$. That is,

$$\begin{aligned} \hat{y}_0 &= c \\ \hat{y}_{k+1} &= \hat{y}_k + hf(t_{k+1}, \hat{y}_{k+1}) \end{aligned} \tag{5}$$

For now, let us concentrate on the explicit method. Notice that this method involves solving an algebraic system at each time step $t_{k+1}$. Given that SPANK already supports specifications of algebraic systems, it behooves us to write out the algebraic part of the method in SPANK's language.

Suppose we have an equation schema named *ODE* given by equation 3 and another schema named *Euler* given by

$$y_{next} = y + hy'. \tag{6}$$

Then, we can specify the algebraic part of the explicit Euler's method by Figure 3.

Although this seems to capture the relationships between variables needed to solve the problem, there are some issues not dealt with. Consider the `input` statements declaring $h$ and $t$. Although formally they look essentially the same, the two variables serve very different purposes in the solution. In particular, $t$ varies at each step, whereas $h$ is held constant. The variable $y$ likewise varies at each step, taking its initial value from $c$, and subsequent values from $y_{next}$. So we must specify, for example,

- $y$ initially gets the value $c$.
- $y$ subsequently gets $y_{next}$.

```
declare Eqn ODE;
declare Eu Euler;
link y'(Eqn.y', Eu.y');
link y_next(Eu.y_next);
input h(Eu.h);
input t(Eqn.t);
input y(Eqn.y, Eu.y);
input t_0();
input c();
```

Figure 3: Algebra of Euler's method

What remains is to explain how the intended independent variable $t$ is incremented. For this, we can just require that $t$ increment by a constant value $h$ at each step. So, we presume that a schema named *Timestep* is available of the form

$$t_{next} = t + h$$

and specify that

- $t$ initially gets the value of $t_0$.
- $t$ subsequently gets $t_{next}$.

Putting these together, we extend SPANK with a construct that allow us (1) to declare an input to be one that takes its value from previous steps and (2) to specify how to obtain the values of those inputs. We add a statement `feedback`, an example of which is

```
feedback y(Eqn.y, Eu.y)
    initially c update-by y_next;
```

which means that the variable $y$ is identified with equation variables *Eqn.y* and *Eu.y*. Also, $y$ is initially set to $c$, and at each time step, $y$ gets its value from the latest value of $y_{next}$. The time variable $t$ would be declared similarly with the additional declaration of a *Timestep* object having its variables suitably identified.

Notice that we have not eliminated the original `input` statement. It still has its use for true problem parameters, i.e., values that do not change

```
declare Eqn ODE;
declare Eu Euler;
declare TS Timestep;
feedback t(Eqn.t, TS.t)
    initially t0 update-by tnext;
feedback y(Eqn.y, Eu.y)
    initially c update-by ynext;
link y'(Eqn.y', Eu.y');
link ynext(Eu.ynext);
link tnext(TS.tnext);
input h(Eu.h, TS.h);
input t0();
input c();
```

Figure 4: Euler's explicit method

through the course of the simulation. For example, the values of $h$, $t_0$ and $c$ might be taken as problem parameters. Thus, the solution of the initial value problem by Euler's explicit method can be specified as in Figure 4 in the new syntax. Similarly, Euler's implicit method can be specified as in Figure 5. To convince oneself that the latter specifies Euler's implicit method, consider the algebraic systems that are solved at each step. Namely, this description specifies that the following system of equations be solved at each time step.

$$\begin{aligned} y' &= f(t, y) \\ y &= y_{prev} + Hy' \end{aligned} \tag{7}$$

Now, for the first time step, we have $t = t_0$, $y_{prev} = c$ and $H = 0$, so this reduces to

$$\begin{aligned} \hat{y}'_0 &= f(t_0, c) \\ \hat{y}_0 &= c \end{aligned} \tag{8}$$

This gives us the correct first value of $\hat{y}_0 = c$. For the $k+1$ step, we have $t = t_k + h = t_{k+1}$, $y_{prev} = y_k$ and $H = h$. So this reduces to

$$\begin{aligned} \hat{y}'_{k+1} &= f(t_{k+1}, \hat{y}_{k+1}) \\ \hat{y}_{k+1} &= \hat{y}_k + h\hat{y}'_{k+1} \end{aligned} \tag{9}$$

Again, this gives us the correct value of $\hat{y}_{k+1}$.

```
declare Eqn ODE;
declare Eu Euler;
declare TS Timestep;
feedback t(Eqn.t, TS.t)
    initially t0 update-by tnext;
feedback yprev(Eu.y)
    initially c update-by y;
feedback H(Eu.h)
    initially 0 update-by h;
link y'(Eqn.y', Eu.y');
link y(Eqn.y, Eu.ynext);
link tnext(TS.tnext);
input h(TS.h);
input t0();
input c();
```

Figure 5: Euler's implicit method

**Digression.** A word about our choice of the indicator `feedback` is in order. Consider the algebraic part of Euler's explicit method again (Figure 3). We can think of this as specifying a function from values for the `input` variables (and initial estimates for the `link` variables) to values for the `link` variables that solve the system of equations. Now, the feedback in the system is clear. Namely, the `feedback` statements indicate how outputs returned by the solving function are to be fed back to the function as inputs for the next iteration.

## 3.2 Runge-Kutta methods

Here we show that the proposed extensions to SPANK, called for by a careful look at Euler's methods, actually suffice to permit us to specify Runge-Kutta solution methods as well. We demonstrate the idea by considering a Runge-Kutta method of order 2. The technique easily generalizes to other Runge-Kutta methods.

Essentially, Runge-Kutta methods work by finding some fixed number of approximations of $y(t_{k+1})$ and then taking a weighted average of these as the

value of $\hat{y}_{k+1}$. There are many different Runge-Kutta methods, but they all use this general strategy.

Without discussing the theoretical justification any further, one Runge-Kutta method of order 2 is given by the following.

$$\begin{aligned} \hat{y}_0 &= c \\ u_{k+1} &= \hat{y}_k + hf(t_k, \hat{y}_k) \\ v_{k+1} &= \hat{y}_k + hf(t_{k+1}, u_{k+1}) \\ \hat{y}_{k+1} &= \tfrac{1}{2}(u_{k+1} + v_{k+1}). \end{aligned} \tag{10}$$

That is, $u_{k+1}$ and $v_{k+1}$ are approximations of $y(t_{k+1})$, and $\hat{y}_{k+1}$ is taken as the simple average of the two. Notice that this method uses equations already found in Euler's method together with an equation that specifies the average of the two approximations. So suppose along with the schemata *ODE* and *Euler*, we have a schema named *Avg2* as follows

$$avg = \frac{1}{2}(x_1 + x_2). \tag{11}$$

Then we can specify the method as in Figure 6. Although this specification is significantly more complicated than those for Euler's methods, it certainly does accurately describe the desired method. Also, the basic technique of specifying complicated interactions amongst the problem variables is clearly general enough to support higher order Runge-Kutta methods. As an aside, the method described here is an explicit one — implicit Runge-Kutta methods can be developed in the same spirit as the implicit Euler's method. Although implicit methods are more difficult to solve (they require solving implicit algebraic equations), they can be specified in our language with no additional constructs.

## 3.3 Predictor-corrector methods

So far, we have considered only solution methods that use the value $\hat{y}_k$ to obtain the next value $\hat{y}_{k+1}$. These are called *single step* methods. More accurate approximations can be obtained by using information about several previous values instead of just one. To demonstrate how such *multiple step* methods can be specified in SPANK, we consider a class of very common methods called *predictor-corrector* methods. In essence, a predictor-corrector

```
declare Eqn1, Eqn2 ODE;
declare Eu1, Eu2 Euler;
declare TS Timestep;
declare A Avg2;
link u(Eu1.ynext, Eqn2.y, A.x1);
link v(Eu2.ynext, A.x2);
link y'1(Eqn1.y', Eu1.y');
link y'2(Eqn2.y'.Eu2.y');
link ynext(A.avg);
link tnext(TS.tnext, Eqn2.t);
feedback t(Eqn1.t)
    initially t0 update-by tnext;
feedback y(Eqn1.y)
    initially c update-by ynext;
input h(Eu1.h, Eu2.h, TS.h);
input c();
```

Figure 6: 2nd order Runge-Kutta method

method involves finding the next value $\hat{y}_{k+1}$ by solving an implicit equation in $\hat{y}_{k+1}$ — this equation is the *corrector*. Because this equation is implicit it must be solved by some iterative method (Newton-Raphson). Because of the expense of this iteration, it is in our interest to start the iteration with an estimate of $\hat{y}_{k+1}$ which is as close as possible to the solution. This initial estimate is given by an explicit equation, the *predictor*.

As an example, consider the implicit Euler's method again as given in Figure 5. The corresponding equation is implicit in the unknown `y`. Newton-Raphson's method requires that we give an estimate of `y` to begin the iteration. Hence, the above specification leaves out an important datum: the initial estimate of $y$. We could simply assume that the initial estimates are carried over from values at the previous step. But, we would obtain a better initial estimate of $y$ by using the explicit Euler's rule to predict the value of $y$. Thus, the `link` statements ought to be modified to permit us to specify an initial estimate. An example of the extended `link` statement is

```
link y(Eqn.y, Eu.ynext, Eupredict.y)
    initially c predict-by ynext;
```

```
declare Eqn ODE;
declare Eu, Eu_predict Euler;
declare TS Timestep;
feedback t(Eqn.t, TS.t)
    initially t_0 update-by t_next;
feedback y_prev(Eu.y)
    initially c update-by y;
feedback H(Eu.h)
    initially 0 update-by h;
link y'(Eqn.y', Eu.y', Eu_predict.y');
link y(Eqn.y, Eu.y_next, Eu_predict.y)
    initially c predict-by y_next;
link y_next(Eu_predict.y_next);
link t_next(TS.t_next);
input h(Eu_predict.h, TS.h);
input t_0();
input c();
```

Figure 7: Euler's implicit method with predictor

This statement is taken to mean the same thing as the original version with the additional information that Newton-Raphson iteration ought to begin with $y$ set to $c$ for the first time step, and to the current value of $y_{next}$ thereafter. Now, a more complete description of the implicit method, using the explicit method to give us the initial estimates of $y$ is given by Figure 7.

The effect of this change is simply to calculate in $y_{next}$ the value that the explicit method would give as the next value of $y$, and then use that value as the prediction of $y$ in the iterative solution to the implicit method. Notice also that the $y_{next}$ link does not have an initial value and prediction. The reason for this is simple. The equations are explicit in $y_{next}$ so the number of iterations of Newton-Raphson are not affected by the initial value of $y_{next}$. Thus we take a `link` statement such as

$$\texttt{link } y_{next}(Eu_{predict}.y_{next});$$

to be an abbreviation of

$$\texttt{link } y_{next}(Eu_{predict}.y_{next})$$
$$\texttt{initially 0 predict-by } y_{next};$$

In that case, the above correctly specifies the desired method.

This revision of the implicit Euler's method demonstrates a very simple predictor-corrector method (in this case, a single-step method). That is, we use an explicit rule to predict a value for $y$ and then use an implicit rule to correct that prediction.

Multi-step predictor-corrector methods involve keeping track of values $\hat{y}_k$, $\hat{y}_{k-1}, \ldots, \hat{y}_{k-m}$ at each step, for some fixed $m$. A well-known example of this is Milne's method. To obtain a value of $\hat{y}_{k+1}$, this method requires values of $\hat{y}_{k-3}$ and $\hat{y}_{k-1}$, together with values of the derivatives $f(t_k, \hat{y}_k)$, $f(t_{k-1}, \hat{y}_{k-1})$ and $f(t_{k-2}, \hat{y}_{k-2})$. Let us refer to $f(t_i, \hat{y}_i)$ as the value $f_i$. Then Milne's method predicts the value of $\hat{y}_{k+1}$ by the following explicit equation schema (call it *Milne-pred*).

$$\hat{y}_{k+1} = y_{k-3} + \frac{4h}{3}(2f_k - f_{k-1} + 2f_{k-2}) \tag{12}$$

And then corrects by the following implicit equation schema (call it *Milne-corr*).

$$\hat{y}_{k+1} = y_{k-1} + \frac{h}{3}(f_{k+1} + 4f_k + f_{k-1}) \tag{13}$$

Putting this into the system description language of SPANK we have Figure 8. As with the methods already discussed, we can verify that this correctly specifies Milne's method.

# 4 Systems of equations

In the preceding section, we concentrated on specifications of solution methods for single equation problems. But our language can easily be used to describe solution methods for systems of ordinary differential equations as well. Namely, we have only to observe that all of the common solution techniques generalize in a very simple way, by applying the method to each equation separately. We can demonstrate the idea using the explicit Euler's

declare *Eqn ODE*;
declare *P Milne-pred*;
declare *C Milne-corr*;
declare *TS Timestep*;
feedback $t(Eqn.t, TS.t)$
    initially $t_0$ update-by $t_{next}$;
link $y_{k+1}(C.y_{k+1}, Eqn.y)$
    initially $y_3$ predict-by $y_{next}$;
feedback $y_k()$
    initially $y_2$ update-by $y_{k+1}$;
feedback $y_{k-1}(C.y_{k-1})$
    initially $y_1$ update-by $y_k$;
feedback $y_{k-2}(P.y_{k-3})$
    initially $y_0$ update-by $y_{k-1}$;
link $f_{k+1}(C.f_{k+1}, Eqn.y', P.f_k)$
    initially 0 predict-by $f$;
feedback $f_k(P.f_{k-1}, C.f_k)$
    initially $y_2'$ update-by $f_{k+1}$;
feedback $f_{k-1}(P.f_{k-2}, C.f_{k-1})$
    initially $y_1'$ update-by $f_k$;
link $y_{next}(P.y_{k+1})$;
link $t_{next}(TS.t_{next})$;
input $h(P.h, C.h, TS.h)$;
input $y_0()$;
input $y_1()$;
input $y_2()$;
input $y_3()$;
input $y_1'()$;
input $y_2'()$;

Figure 8: 4th order Milne's method

method. Suppose we want to solve the initial value problem for the following system of ordinary differential equations.

$$\begin{aligned} y_1' &= f_1(t, y_1, \ldots, y_n) \\ &\vdots \\ y_n' &= f_n(t, y_1, \ldots, y_n) \end{aligned} \tag{14}$$

together with initial values for the $y$ functions: $y_1(t_0) = c_1, \ldots, y_n(t_0) = c_n$. Furthermore, suppose we have equation schemata $ODE_1, \ldots, ODE_n$ corresponding to the equations $y_1' = f_1(t, y_1, \ldots, y_n), \ldots, y_n' = f_n(t, y_1, \ldots, y_n)$. Then we can specify the solution as in Figure 9.

Of course, similar techniques allow us to specify other solution methods for systems of equations. Moreover, by the standard technique we can describe higher-order systems by factoring into larger systems of first-order systems.

# 5 Further Extensions

One clear omission in our proposed extensions is the ability to specify solution methods in such a way as to be independent of the particular ODE being solved. That is, for example each of the methods discussed above involves declaring one or more instances of an ODE. We do not need to know anything about the ODE schemata except that it have variables $t$, $y$ and $y'$ which are interpreted in the obvious way. So far as the solution method is concerned, the requirements are purely formal, having only to do with the form of the equation schema. It is reasonable to expect that we should be able to encapsulate the description of a solution method in a similar way to the encapsulation of a physical law in an equation schemata. One difference, however, is that the resulting description would have to be parametric with respect to the ODE. Thus, a reasonable further extension to SPANK would be a means of encapsulating systems of equations such as the explicit Euler's method above.

Namely, we would like to be able to declare a new parametric schema, say *explicit-Euler* in such a way that for any particular schema $ODE$ (with schematic variables $t$, $y$, and $y'$) the statement

`declare` $S$ *explicit-Euler*($ODE$);

```
declare Eqn_1 ODE_1;
declare Eqn_2 ODE_2;
⋮
declare Eqn_n ODE_n;
declare Eu_1, ..., Eu_n Euler;
feedback y_1(Eqn_1.y_1, ..., Eqn_n.y_1, Eu_1.y)
    initially c_1 update-by yn_1;
feedback y_2(Eqn_1.y_2, ..., Eqn_n.y_2, Eu_2.y)
    initially c_2 update-by yn_2;
⋮
feedback y_n(Eqn_1.y_n, ..., Eqn_n.y_n, Eu_n.y)
    initially c_n update-by yn_n;
independent t(Eqn_1.t, ..., Eqn_n.t)
    initially t_0 increment-by h;
link y'_1(Eqn_1.y', Eu_1.y');
link y'_2(Eqn_2.y', Eu_2.y');
⋮
link y'_n(Eqn_n.y', Eu_n.y');
link yn_1(Eu_1.y_next);
link yn_2(Eu_2.y_next);
⋮
link yn_n(Eu_n.y_next);
input h(Eu_1.h, ..., Eu_n.h);
```

Figure 9: A system of differential equations

would introduce a new set of equations to the system with interface variables $S.y$, $S.y'$, etc. which solves the ordinary differential equation $ODE$ by Euler's explicit method. In this way, we would be able to build up a library of solution methods, and thus keep the parts of a system that are due to the choice of solution method separate from the parts that are due to the physical system being modeled.

We expect this notion of parametric schemata to be of particular utility when specifying systems of ODEs, though we have not worked out the details sufficiently to say how these systems would be expressed.

# 6 Conclusion

We have shown that the approach to modeling static behavior in the Simulation Problem Analysis Kernel can be extended in a principled way to modeling dynamic behavior by adding to the description language constructs that specify how time advances and how values for time dependent variables are carried from one time step to the next. In particular, in this paper we have shown that with our simple extensions one can specify a wide variety of solution methods for solving initial value problems for ordinary differential equations.

Some of the important attributes of static SPANK preserved with these extensions are the following. One, the underlying system of algebraic equations involved in a differential equation solution is given explicitly. Thus, the implementation can easily take advantage of existing code for static SPANK. Two, the basic intuition that a system can be built up component-wise by specifying how the various components relate is preserved in the extensions. Thus, even complicated models involving complicated solution methods can be verified in a modular way. Three, the description language is essentially declarative, i.e., neutral with respect to the particular algorithms used to implement SPANK. This is important in that it allows a model designer to concentrate on the already difficult task of accurately describing the physical system without also having to deal with the task of implementing a particular solution algorithm. In light of these observations, we expect that the extensions proposed here will form the basis for a very general simulation system suitable for modeling large and otherwise unwieldy dynamic systems.

# References


[And89a] Andersson, M. An object-oriented modelling environment. In *The 1989 European Simulation Multiconference*, Rome, Italy, 1989. Available from Lund Inst. of Technology, P.O. Box 118, S-221 00, Lund, Sweden.

[And89b] Andersson, M. Omola – an object-oriented modelling language. Technical report, Lund Institute of Technology, P.O. Box 118, S 221 00, Lund, Sweden, 1989.

[Cla87] Clarke, J. A. The Energy Kernel System: An overview in support of three grant proposals. Technical report, Energy Simulation Research Unit, Univ. of Strathclyde, 16 Richmond St, Glasgow, G1 1XQ, U. K., 1987.

[Cla88a] Clarke, J. A. The Energy Kernel System. *Building and Energy*, 1988.

[Cla88b] Clarke, J. A. An object-oriented approach to modeling. In *Proceedings of User-1*, Ostend, Belgium, 1988. Available from Soc. for Computer Simulation, P.O. Box 17900, San Diego, CA 92117.

[Gea71] Gear, C. William *Numerical Initial Value Problems in Ordinary Differential Equations.* Prentice-Hall, 1971.

[Hir88] Hirsch, J. J. A plan for the development of the next generation building energy analysis computer software. In *Proceedings of the First Building Energy Simulation Conference*, Seattle, U.S.A., 1988. Available from Soc. for Computer Simulation, P.O. Box 17900, San Diego, CA 92117.

[Mat88] Mattsson, S. E. On model structuring concepts. In *Proceedings of the 4th IFAC Symposium on Computer-Aided Design in Control Systems (CADCS)*, P.R. China, 1988. Available from Lund Inst. of Technology, P.O. Box 118, S-221 00, Lund, Sweden.

[Mat89a] Mattsson, S. E. Concepts supporting reuse of models. In *Proceedings of Building Simulation '89*, pages 175–180, Vancouver,

Canada, 1989. Available from International Building Performance Simulation Association, P.O. Box 282, Orleans, Ontario, K1c 1S7, Canada.

[Mat89b] Mattsson, S. E. Modeling of interactions between submodels. In *The 1989 European Simulation Multiconference*, Rome, Italy, 1989. Available from Lund Inst. of Technology, P.O. Box 118, S-221 00, Lund, Sweden.

[Sah88] Sahlin, P. Modsim, a program for dynamical modeling and simulation of continuous systems. Technical report, Institute of Applied Mathematics, Institute of Applied Mathematics. P.O. Box 26300, S-100 41 Stockholm, Sweden, 1988.

[SB88] Sowell, E. F. and W. F. Buhl. Dynamic extension of the simulation problem analysys kernel (SPANK). In *Proceedings of User-1*, Ostend, Belgium, 1988. Available from Soc. for Computer Simulation, P.O. Box 17900, San Diego, CA 92117.

[SB89] Sowell, E. F. and W. F. Buhl. Object-oriented programming, equation-based submodels, and system reduction in SPANK. In *Proceedings of Building Simulation '89*, pages 141–146, Vancouver, Canada, 1989. Available from International Building Performance Simulation Association, P.O. Box 282, Orleans, Ontario, K1c 1S7, Canada.

[SBEW86] Sowell, E. F., W. F. Buhl, A. E. Erdem, and F. C. Winkelman. A prototype object-based system for HVAC simulation. In *The Second International Conference on System Simulation in Buildings*, Liege, Belgium, 1986. Available from Laboratory of Thermodynamics, Univ. of Liege, B-4000, Liege, Belgium.

[SS89] Sahlin, P. and E. F. Sowell. A neutral format for building simulation models. In *Proceedings of Building Simulation '89*, pages 147–154, Vancouver, Canada, 1989. Available from International Building Performance Simulation Association, P.O. Box 282, Orleans, Ontario, K1c 1S7, Canada.

LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
INFORMATION RESOURCES DEPARTMENT
1 CYCLOTRON ROAD
BERKELEY, CALIFORNIA 94720