Object-Oriented Programming, Equation-Based Submodels, and System Reduction in SPANK

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

W. Fred Buhl and Jean-Michel Nataf
Simulation Research Group
Applied Science Division
Lawrence Berkeley Laboratory
Berkeley, CA 94720

May 15, 1989

Abstract

Collaborative efforts among building simulation researchers in Europe and the US have resulted in wide acceptance of certain features as necessary attributes of future simulation environments. As identified 'in the Energy Kernel System (EKS), the principal features are those of the object-oriented programming (OOP) paradigm, in which a hierarchy of submodels is readily defined and interconnected to form system models of widely varying purpose, solution methodology, and implementation description. The Simulation Problem Analysis Kernel (SPANK) is an early, prototypical implementation intended to exhibit at least some of the EKS features, including an OOP-like environment. Additionally, SPANK implements a solution process that is based on graph algorithms and achieves solution efficiency through a reduction in size of the iterative problem. This paper enlarges on earlier descriptions of SPANK, attempting to separate and clarify the modeling issues and the solution issues. To this end the nature of the SPANK environment is compared to established OOP environments. The SPANK equation-based objects are contrasted to the assignment- based (input/output

Introduction

The Energy Kernel System (EKS) is a recently proposed object-based environment intended for building simulation studies [HIRSCH 1985, CLARKE 1987, CLARKE 1988A]. The Simulation Problem ANalysis Kernel (SPANK) was developed as an early prototype of such an environment [SOWELL 1986, SOWELL 1988], and other implementations are under development [CLARKE 1988B, SAHLIN 1988].

The basic precept was originally the same for both of these environments: particular building simulation problems are to be defined and solved using *objects* from a library, thus providing much greater flexibility in defining simulation problems than afforded by currently available whole-building simulators such as DOE-2 [BIRDSALL 1985]. However, this precept is subject to widely varying interpretations. The central questions appear to be, what is an object, how are problems defined in terms of these objects, and how do the objects interact to determine a solution. In one view [CLARKE 1988B], objects are relatively large, algorithmic procedures, perhaps extracted from existing programs, and the issues of problem definition, object interaction and solution procedure have yet to be concretely defined. The SPANK object, on the other hand, is a single equation, with the possibility of defining larger *macro objects* (or submodels) by assembling objects and macro objects in an hierarchal manner. This definition of the object is not arbitrary, but was developed to facilitate model development and efficient solution through reduction.

The relative merits of the two interpretations will eventually be decided based on the usefulness of the resulting software. In the meanwhile, this paper is intended to more fully describe and defend the SPANK interpretation. Additionally, we attempt to clarify the distinction between SPANK and existing, component-based simulators such as TRNSYS [KLEIN 1983].

The SPANK Environment and Object-Oriented Programming

The SPANK development effort is aimed at a convenient, flexible and intuitive simulation environment. The object-oriented programming (OOP) paradigm, as usually defined, meets this need in some ways, but not in others. Therefore SPANK makes use of some of the OOP concepts, but not all. In this section we explain where SPANK is object-oriented, where it is not, and why it is not.

Fundamentally, OOP has been described as "...a style of programming based on directly representing physical objects and mental concepts in the machine" [LIEBERMAN 1987]. Thus the computer objects behave and interact with their neighbors so as to mimic their real-world counterparts. Other general descriptions emphasize code reusability: programs are to be assembled from pre-existing software modules. As OOP environments have developed, more specific definitions have emerged.

One of the earliest object-oriented languages (OOL) was SIMULA [BIRTWISTLE 1973], which also introduced the idea of a *class*, meaning a template of an object from which instances could be constructed. The concept has been enlarged upon by various proponents to emphasize features such as the *encapsulation* of data and the methods (procedures) allowed to operate on that data, *information hiding*, and *inheritance*. Later extensions to the OOP paradigm included *message passing* as the means of object interaction.

Uniform implementations of the object-oriented paradigm, such as Smalltalk [GOLDBERG 1983] use all of these features, and require uniform usage of the paradigm, so that even normal arithmetic is treated as message passing to numerical objects. Such strict adherence leads to syntax that appears inverted and nonintuitive. Other implementations, such as C++ [STROUSTRUP 1986], support all of the OOP features, but also allow usage of normal assignment-based procedural constructs. At the other end of the spectrum, some simulation programs have been described as "object-oriented" while providing only simple modularity.

SPANK fits the object-oriented description most closely in the fundamental sense. That is, SPANK views simulation problems as collections of interacting objects, where there is a close correspondence between the programming entity and the physical or conceptual entity being modeled.

The fundamental object in SPANK is the equation. As such, it has as its private data the variables involved in the equation. Since these variables are not classified according to input/output designations, they are called simply *interface variables*. A particular equation can usually be manipulated algebraically — manually or by symbolic computation — to yield an explicit formula for one or more of its interface variables. These formulas are called the *inverses* of the equation. The set of inverses can be viewed as the *methods* of the object.

An example will clarify these ideas. The mathematical model for a flow collector is:

$m3 - m1 - m2 = 0$ (where mj is a mass flow rate)

The inverses of this equation are:

$m1 = m3 - m2$
$m2 = m3 - m1$
$m3 = m1 + m2$

Thus, the collector object is:

Interface:
m1, m2, m3

Methods:
M1: $m1 = m3 - m2$
M2: $m2 = m3 - m1$
M3: $m3 = m1 + m2$

Some solution strategy for the overall simulation problem determines which of the methods is needed, and the list of objects needing this result. In a uniform object-oriented environment, this solution process would also be an object or objects. However SPANK does not carry the object-oriented paradigm to this extent. Instead, the object interconnections are specified by means of a simple Network Specification Language (NSL). NSL is similar to input languages for earlier simulators such as TRNSYS [KLEIN 1983] in that it specifies object interconnectivity. However, it differs in that information flow direction is not implied. This feature is in keeping with the input/output-free nature of SPANK objects. Problem definition in NSL is described in earlier papers [SOWELL 1986, SOWELL 1988] so here we show only a simple example for convenience in the current discussion.

A two-object problem might be specified as:

```
define obj_class_a obj1;
define obj_class_b obj2;

input ap(obj2.a);
input xp(obj1.x);

link  yp(obj1.y, obj2.u);
link  zp(obj1.z, obj2.v);
```

Here obj_class_a and obj_class_b are two different object types, each with three interface variables. The interface variables for obj_class_a are x, y, and z, while those for obj_class_b are a, u, and v. The first *define* statement creates an instance of obj_class_a called obj1, while the second makes obj2 an instance of obj_class_b. The two *input* statements declare ap and xp as required external input to the problem, giving values to the obj2.a and obj1.x interface variables respectively. The *link* statement acts to define *friend* relationships among objects. The first link declares a problem variable yp that is linked to both obj1.y and obj2.u. This means that y of obj1 is forced to be the same as u of obj2. In a like manner zp is indicated to be a problem variable equivalent to both obj1.z and obj2.v. This problem specification is placed in a file which becomes input to the SPANK processor.

It is apparent that NSL retains object orientation in the basic sense. That is, the objects can correspond to entities in the modeled system (modularity); the methods by which the model equations are enforced are hidden (information hiding, data abstraction); objects are instances of classes (code reuse); and although not evident in this example, macro objects can be constructed from objects and other macro objects (multiple inheritance, code reuse, information hiding). On the other hand, there is not much similarity to the message passing syntax of object-oriented languages such as Smalltalk. That is, we do not see in NSL any indication of obj1 "sending a message" to obj2. This is because the SPANK processor acts as a "hardwired" executive that determines the solution sequence and carries it out. In a true object-oriented simulation environment, the solution methodology and control would be embodied in objects and would be subject to user selection and control. We hope to improve SPANK flexibility in the future by the introduction of general integrator and iteration objects. However, it is not clear that strict adherence to the OOP paradigm is the best means of providing this flexibility.

SPANK and Component Based-Simulators

The object-oriented features of SPANK superficially resemble older component-based simulation environments such as TRNSYS. Both SPANK and TRNSYS are based on the idea of assembling simulation problems from pieces of existing code by use of simple interconnection languages. In SPANK, object interfaces are linked together using the *link* verb, while in TRNSYS one connects the outputs of one component to the inputs of others using component and variable numbers. Observe that in addition to allowing code reuse, either environment provides the modeler with the important advantage of *nonprocedural* specification of the problem. That is, the simulation problem is defined and solved without having to specify a solution sequence. Therefore either environment is superior to the more primitive technique of procedural programming (e.g., in FORTRAN), even if subroutine libraries are employed to achieve code reuse.

Despite these common features, there are significant differences between component-based simulators and object-oriented environments like SPANK or IDA (formerly MODSIM) [SAHLIN 1988]. These differences impact the component model development, component model validation, problem definition, and problem solution phases of simulation.

From the modeling perspective, the most important difference is that SPANK objects are *equation-based* or *declarative*, while TRNSYS components are *assignment-based* or *procedural* [MATTSSON 1988]. Assignment-based components have two disadvantages when compared to equation-based objects. First, assignment-based components are inherently procedural algorithms and therefore require prescribed inputs. A particular implementation is then limited to problem usage where the needed inputs are available from outputs of other components. Therefore the definition of the components limits the class of simulation problems that can be solved. On the other hand, when equation-based objects are used, there are no prescribed inputs: all interface variables are candidate inputs. Therefore a single implementation of an object serves all possible problem usages. The second disadvantage attributable to assignment-based component models is that because they are procedural, the modeler must devise an algorithm. Object-oriented environments, on the other hand, are declarative, meaning that the modeler need only give the equations that must be enforced (in any order). That is, the object itself does not describe how the solution is to be achieved.

Another advantage of object-oriented models is hierarchal decomposition, whereby complex objects are formed from simpler ones. In SPANK, for example, one defines a mass balance object and an energy balance object. These can then be combined into a macro object representing a collector. Moreover, each macro object can also be used to define other macro objects, giving another form of code reuse not available in component-based environments. Assignment-based component models do not allow macro objects, so each new need for a component model must be met with a separate code module, often duplicating code in existing modules. In addition to creating larger run-time modules, this makes maintenance more difficult, since each module must be updated separately.

Validation also favors the equation-based, object-oriented approach. First, the component model implementation, being nothing more than the equations of the underlying mathematical model, is easier to manually check than an algorithm. Also, the declarative nature of SPANK objects allows automatic problem analysis, such as matching of equations to variables. As a byproduct of this matching, unequal numbers of equations and variables are easily detected. Assignment-based, algorithmic component models do not allow such detection. Because declarative languages in general lend themselves to automatic correctness proofs, we might expect a greater degree of automated validation of equation-based object models in the future.

There are also differences in solution strategy between SPANK and component-based environments. Component-based solvers most often follow a *sequential-modular* solution strategy [CHEN 1985] in which each component (most often comprising several equations) is executed in turn. Recyclic problems require iteration, so the sequence is repeated, using some method for updating the cyclic variables, until convergence is achieved. TRNSYS, for example, employs successive substitution. One problem with this method is that the "packaging" of equations into component modules reflects the user's view of the physical system, and not necessarily the order in which the equations should be executed in an efficient solution process. For example, each component model may have a mass and an energy balance equation. In many simulation problems the mass flows can be solved independently, without iteration. In the sequential-modular approach, however, both mass and energy equations are executed iteratively. In SPANK each equation is treated separately. This allows graph theoretic techniques

to be used to find an efficient solution sequence with a minimum number of variables involved in the iteration. Also, the more efficient Newton-Raphson method can conveniently be used in place of successive substitution.

Reduction and Sparse Matrix Techniques

To solve a typical simulation problem, a set of algebraic equations must be solved simultaneously at each point in time. One way of carrying out the solution is to express each equation in residual form and iterate using the entire set of problem variables as iterates. With this approach the size of the iteration vector is the number of problem variables, n, and if a Jacobian is used in updating the iteration vector, it is n by n. Because the individual equations in simulation problems typically have far fewer than n variables present, the Jacobian usually will be sparse, and sparse techniques are therefore often employed in the updating process [PISSANETZKY 1984].

As an alternate solution strategy, SPANK uses equation-level information about the problem structure to determine a small set of variables that can be iterated. This is called *reduction*, because the iteration vector and Jacobian are then smaller. In a later section of this paper we show that for HVAC systems the reduction can be significant.

It can be shown that normal matrix methods are in general not competitive with graph-theoretic reduction. However, sparse techniques may be equally efficient if properly implemented. We note that there is a simple form of reduction that is nearly always beneficial, and should be done even if sparse matrix techniques are to be used. By examination of data dependencies in the equation set, the equations can be divided into three nonintersecting sets: the *pre-set*, consisting of equations that can be solved directly without iteration and not involving any variables in the iteration set; the *iteration-set*, consisting of those that will be involved in iteration; and the *post-set*, containing the equations that can be directly solved once the solutions of the pre-set and post-set equations have been obtained. The problem can be so partitioned by straightforward means, e.g., sorting the equations, or breaking the problem graph into strongly connected components. The size of the iteration vector is then the size of the iteration-set. If this partitioning is not done, regardless of the solution method, we pay the penalty of calculating pre- and post-set function and derivative values repeatedly when we could calculate these functions only once and their derivatives not at all. If the matrix approach (sparse or otherwise) is used, we also work with a larger Jacobian and have to do updates on a larger iteration vector.

Focusing now on the iteration-set only, we can compare further reduction with employing sparse techniques. Both approaches are described elsewhere, so here we provide only brief descriptions. In further reduction we identify a small number of variables that cut all cycles, called the *cut-set*. Note that the cut-set is smaller than the iteration-set. The solution process involves calculating and solving a reduced Jacobian, of size equal to the cut set. A standard linear solver can be used — Gaussian elimination, for instance — because the Jacobian will be small and dense.

With sparse methods, the Jacobian is the size of the iteration set, and is stored in some manner to avoid storage of zero elements. Further, steps are taken to avoid arithmetic operations with the zero elements, and to preserve sparsity. These steps constitute the more or less standard sparse techniques [PISSANETZKY 1984]. If these are the only steps taken and the Jacobian is not of some special structure, the reduction method will be more efficient because the basic matrix size remains the same. However, some sparse solvers [SODERLIND 1988] go further, and have the potential for efficiency equal to the reduction method. Briefly, row-column interchanges are performed in an effort to achieve a block-bordered upper triangular form, with the borders as small as possible. When this is done, the variables represented in the right border columns can be determined working only with a matrix of size equal to the number of border variables, and further, the remaining variables can be thereafter determined in a manner not unlike back substitution; i.e., in a direct manner without further simultaneous solution. Thus the entire set has been solved without simultaneous solution of the iteration-set size Jacobian.

It is interesting to note that when pursued sufficiently far, the sparse matrix techniques begin to look very much like the graph-theoretic reduction technique. Indeed, it can be shown that the border variables noted above are often one and the same with the minimum cut-set found by SPANK. Also, the computations involved in back substitution are substantially the same as those required in calculation of the elements of the reduced Jacobian. The only difference then appears to be in how the border or cut-set variables are discovered. When the graph representation is used, cut-set algorithms are readily available [LEVY 1983, SHAMIR 1979, ROSEN 1982]. The algorithms actually used to achieve a small border set have not been documented, but it is clear that the cut-set algorithms could be applied. Finally, when we observe that sparse storage techniques are or can be identical to the structures used to store graphs, the differences begin to appear superficial, so that whether one refers to sparse matrices and border sets or graphs and cut-sets depends primarily on point of view. If this proves to be the case, we suggest that the graph is the natural way to represent a set of sparse nonlinear equations. Indeed, Pissanetzky points out that "A sparse matrix should not be thought of as a matrix at all, but as a graph." The matrix, which is fundamentally a linear construct and only efficient when densely filled, is the ideal structure for the reduced Jacobian at the core of the solver.

Problem Size Reduction with SPANK

As has been noted above, the declarative nature of the SPANK objects results in the possibility of automatic manipulation of the problem graph. A large variety of such manipulations can be envisaged, and will no doubt be explored in the future. In the current version of

SPANK the automatic manipulation takes the form of matching (choosing an output variable for each object) and cut-set size reduction. Neither matching nor cut-set reduction have unique algorithms. In SPANK specific algorithm choices have been made and test problems have been used to investigate the algorithms' efficiency. The actual algorithms used in SPANK for matching and cut-set reduction will be described in a future paper. In this section, we wish to describe the results achieved by SPANK on a variety of test problems.

One set of test problems consists of HVAC secondary systems. In a previous paper [SOWELL 1986] the solution of a very simple (1 zone) HVAC problem was demonstrated. We have gone on to model problems of increasing complexity, ending up with problems that are of similar complexity to those that would be simulated in a typical small DOE-2 [BIRDSALL 1985] or BLAST [BLAST 1979] run. Table 1 shows the results from two such systems: a variable air volume system serving 5 zones, and a return air reheat system serving 5 zones. The problem sizes involving in excess of 100 variables are reduced to cut-sets of just three variables. This is satisfying but not very surprising. Anyone who has written a Fortran model of such a system could readily predict the number and choice of break variables. What is important is that the break variables were chosen automatically, and that such a choice can be made for a wide variety of systems with no effort by the modeler. While a modeler may easily be able to choose break variables for a familiar system such as variable air volume, the task may be much more difficult for an unfamiliar system. Moreover, SPANK has relieved us of developing the solution algorithm.

Table 1

SPANK Reduction Results - Secondary HVAC Systems

No. of problem variables	No. of break variables	No. of iterations to solution	
148	3	1	(Return Air Reheat System)
100	3	3	(VAV System)

A second set of test problems consists of multiroom air flow simulations. A variable number of rooms are connected to each other by a variable number of orifices. The smallest problem has one room with six orifices, the largest has 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. Results are shown in Table 2, and once again the problem size reduction is very satisfactory.

Table 2

SPANK Reduction Results - Multiroom Air Flow

No. of problem variables	No. of break variables	No. of iterations to solution
13	1	12
72	4	8
84	4	23
96	4	11
108	4	11
118	4	11
141	9	19
207	9	14
248	24	44
95	4	8

A third set of problems involved modeling the steady state heat flow through a one dimensional multi-layer wall. This is a simple problem with a known solution. Although SPANK was able to solve problems of this type, Table 3 shows that the problem size reduction that was obtained was only approximately 2 to 1. This demonstrates that the SPANK matching and cut-set reduction algorithms may not be the most appropriate ones for all types of problems. Future versions of SPANK may use different algorithms for different problem types.

Table 3

SPANK Reduction Results - Thermal Conduction

No. of problem variables	No. of break variables	No. of iterations to solution
151	70	3
16	8	3
31	15	3
4	2	2
61	30	4
7	3	2

Tables 1-3 demonstrate that SPANK's automatic problem size reduction techniques work well on some types of problems. We have not demonstrated how efficient SPANK is in comparison to other techniques with regard to overall solution time. Certainly if SPANK run-times were compared to run-times obtained by including all the problem variables in the cut-set, the increased efficiency — at least for the first two problem types — would be dramatic. But such a comparison would be misleading, since reduced cut-sets are either chosen by hand, when creating special purpose Fortran simulations, or the problem size is effectively reduced by using sparse matrix techniques. It remains to be shown whether techniques like those used in SPANK will prove to be the most efficient on a variety of problems. Such a demonstration will be possible only by solving identical test problems with a variety of simulation programs.

REFERENCES

BLAST 1979
Building Loads Analysis and System Thermodynamics (BLAST) Program Users Manual, Volume I, CERL Report E-153 (1979).

BIRDSALL 1985
Birdsall, B.E., et al., "The DOE-2 Computer Program for Thermal Simulation of Buildings" in *Energy Sources: Conservation and Renewables,* Appendix E, American Institute of Physics Conference Series (1985).

BIRTWISTLE 1973
Birtwistle, G.M., Dahl, O-J., Myhrhaug, B. and Nygaard, K. *Simula Begin,* Van Nostrand Reinhold (1973).

CHEN 1985
Chen, H-S, and Stadtherr, M.A.,
"A Simultaneous-Modular Approach to Flowsheeting and Optimization," *AIChE Journal* **31**, 1843 (1985).

CLARKE 1987
Clarke, J.A. "The Energy Kernel System: An Overview in Support of Three Grant Proposals," , Energy Simulation Research Unit, Univ. of Strathclyde (1987).

CLARKE 1988A
Clarke, J.A., "The Energy Kernel System," *Energy and Buildings* **10** 259 (1988).

CLARKE 1988B
Clarke, J.A., "An Object-oriented Approach to Modeling," Proceedings of User-1, Ostend September 6-8, 1988.

GOLDBERG 1983
Goldberg, A. and Robsen, D. *Smalltalk-80: the Language and Its Implementation,* Addison-Wesley (1983).

HIRSCH 1985
Hirsch, J.J. "A Plan for the Development of the Next Generation Building Energy Analysis Computer Software," Proceedings of the First Building Energy Simulation Conference, Seattle (1985).

KLEIN 1983
Klein, S.A., et al., "TRNSYS — A Transient Simulation Program," University of Wisconsin-Madison Engineering Experiment Station Report 38-12, Version 12.1 (1983).

LEVY 1983
Levy, H. and D. W. Low, "A New Algorithm for Finding Small Cycle Cut-sets," IBM Los Angeles Scientific Center report G320-2721 (1983).

LIEBERMAN 1987
Lieberman, Henry, in *Encyclopedia of Artificial Intelligence,* John Wiley and Sons, 452 (1987).

MATTSSON 1988
Mattsson, S.E. "On Model Structuring Concepts," presented at 4th IFAC Symposium on Computer-Aided Design in Control Systems (CADCS), P. R. China, (1988).

PISSANETZKY 1984
Pissanetzky, Sergio, *Sparse Matrix Technology,* Academic Press (1984).

ROSEN 1982
Rosen, B.K. "Robust Linear Algorithms for Cut-sets," *J. of Algorithms* **3**, 205 (1982).

SAHLIN 1988
Sahlin, Per, "MODSIM, a Program for Dynamical Modeling and Simulation of Continuous Systems," Report from the Institute of Applied Mathematics, P.O. Box 26300, S-100 41, Stockholm, Sweden.

SHAMIR 1979
Shamir, A. "A Linear Time Algorithm for Finding Minimum Cut-sets in Reducible Graphs," *SIAM J. Computing* **8**, 645 (1979).

SODERLIND 1988
Söderlind, G., Eriksson, L.O., and Bring, A., "Numerical Methods for the Simulation of Modular Dynamical Systems," Report from the Swedish Institute of Applied Mathematics, P.O. Box, 26300, S-100 41, Stockholm, Sweden.

SOWELL 1986
Sowell, E.F., W.F. Buhl, A. E. Erdem, and F. C. Winkelmann, "A Prototype Object-based System for HVAC Simulation," presented at the Second International Conference on System Simulation in Buildings, Liege, Belgium, December, 1986. LBL-22106.

SOWELL 1988
Sowell, E.F., and Buhl, W.F., "Dynamic Extension of the Simulation Problem Analysis Kernel (SPANK)," Proceedings of User-1, Ostend, September 6-8 (1988).

STROUSTRUP 1986
Stroustrup, B., *The C++ Programming Language,* Addison-Wesley (1986).