Recent Improvements in SPARK: Strong Component Decomposition, Multivalued Objects, and Graphical Interface

W. F. Buhl, A. E. Erdem, F. C. Winkelmann
Simulation Research Group
Lawrence Berkeley Laboratory
University of California
Berkeley, CA 94720 USA

E. F. Sowell
Computer Science Department
California State University at Fullerton
Fullerton, CA 92634 USA

Abstract

The Simulation Problem Analysis Research Kernel (SPARK) environment for simulation of nonlinear differential algebraic systems has been revised to improve modeling convenience, modeling flexibility, and solution efficiency. Solution efficiency has been enhanced by automatic decomposition of the problem into strongly connected components, characterized as separately solvable subproblems. The normally constructed data flow graph in SPARK allows such components to be identified and placed in the correct order for sequential solution resulting in significant speed-up for problems that are not strongly interconnected. Modeling flexibility has been enhanced by adding Multivalued Objects. Whereas conventional SPARK objects represent single equations, and produce a single result, this extension allows more complex objects which themselves solve simultaneous sets of equations for multiple results. The need for such objects arises when submodels are to be solved independently of the SPARK solver; e.g., to use a specially tailored algorithm. With regard to modeling convenience, the graphical user interface now allows model definition by selection and placement of object icons in a graphical window in an X-windows environment. These objects can be connected with macro links comprising multiple problem variables. The resulting problem is then translated into a Network Language Specification file for SPARK processing.

Strong Component Decomposition

The Case for Decomposition

Often, the sets of simultaneous, nonlinear equations required in simulations can be quite large, making solution efficiency an important issue. Since the number of steps required for solution is $O(n^3)$, where n is problem size, one way to seek improved efficiency is to break the problem into several smaller ones, to be solved one at a time. This principle of "divide and conquer" is widely accepted, but it needs to be observed that there are good ways and bad ways to decompose a problem. For example, if we break the problem into two parts A and B that are interdependent, i.e., A depends on variables that are calculated in B and vice versa, solution requires iteration between A and B until convergence is achieved on the coupling variables. This is not a particularly good situation, since at best we lose, through the iteration between the parts, some of what we have gained through size reduction. If the problem is nonlinear, it is also likely that iteration is required internal to A and B, and iteration within iteration is fraught with numerical difficulties. For one thing, the convergence theorems that apply to iteration of sets of nonlinear equations (Ortega 1970) do not apply in this case. Clearly, we are much better off if we can

The authors can be reached at:
Lawrence Berkeley Laboratory
Mailstop: 90-3147
1 Cyclotron Road
Berkeley, CA 94720, U.S.A.
FAX: (510) 486-4089 or 486-5172
Phone: (510) 486-5711
E-mail: fcw%gundog@lbl.gov

break the problem into pieces that are not interdependent, i.e., either A or B can be solved without any results from the other; the independent part is solved first, then the second part, and we are finished. The entire efficiency gain due to size reduction is then realized.

Most simulation programs offer some method of problem decomposition, but usually the decomposition encouraged by these programs is not ideal from the standpoint of solution efficiency. Modular simulators, e.g., TRNSYS (Klein 1988), evaluate portions of the problem internal to the modules, which is a form of decomposition. Wisely, the modules are usually crafted so as to avoid internal iteration since the global solver iterates for convergence of the external variables. However, the decomposition into modules is done manually, and it is common practice to make the divisions along lines suggested by physical component and subsystem boundaries. This leads to modules that combine equations that, while related physically, are not necessarily related in an efficient numerical solution procedure. For example, both mass and energy balance equations must be enforced in a collector component of an air duct system. Often an air duct system problem will have specified mass flows, but will require iteration to satisfy the energy balance equation. Since all embedded equations are executed at every call of the module, un-needed work is performed as the mass equation is needlessly reevaluated at every iteration.

The HVACSIM+ program (Park 1985) has a modular structure like TRNSYS. In addition, the user is allowed to specify a higher form of decomposition unit, called a block, containing several or many modules. There can be iteration both within and among these blocks. Again, since users customarily handcraft the modules and blocks along lines paralleling the physical system, there is little likelihood of achieving independence among them, so the iteration-within-iteration situation arises.

SPARK Problem Decomposition

The Simulation Problem Analysis Research Kernel (SPARK) differs radically from other simulation programs with regard to problem decomposition. This is because the fundamental unit of a SPARK model is an object containing a single equation rather than a module. Large modules, called macro objects, are allowed, but are only temporary assemblies of equation objects provided for convenience of the modeler. They are disassembled into equation objects as processing begins, so that the SPARK solver has access to individual equations and variables. From the equations and variables a data flow graph is constructed, representing all data dependency throughout the problem.

This structure allows decomposition to be done so as to maximize solution efficiency using well known algorithms on graphs. Separately solvable parts of the problem are detected and placed in the correct order for solution without iteration among the parts. The basic principles of the SPARK decomposition method can be demonstrated with a simple example. Figure 1 shows a data flow graph that might be constructed by SPARK from a problem specification.

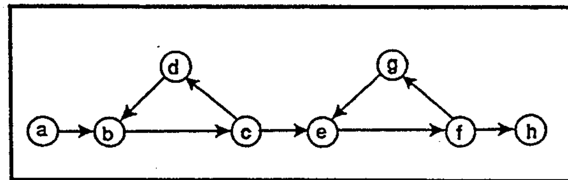

Figure 1. Problem data flow graph

Here we see a graph with 8 vertices and 9 directed edges. Each vertex represents an equation (perhaps nonlinear), producing a single variable that is propagated to other vertices (i.e., equations) along the edges of the graph. Inspection of this graph reveals much about the proper solution process. It can be seen that iteration will be required to solve it because there are cycles (closed paths) in the graph (e.g., solving c requires solving b, which requires solving d, which requires solving c, ...). Also, it is evident that at least two vertices are required to break all cycles. A little thought will reveal that this means at least two iteration variables will be required, e.g., the ones calculated at d and g. Assuming that a Newton-Raphson iteration method is employed to calculate new values, this means that a 2x2 linear problem has to be solved at each iteration if we solve the entire problem together. However, closer examination reveals that the problem partitions nicely into four non-interdependent parts, represented by vertex sets {a},{b,c,d},{e,f,g}, and {h}. Obviously, we should begin by solving {a}, since it has no inputs from other parts of the problem*. Then we could simultaneously solve the set {b,c,d}, an iterative problem in one variable. Once this is finished the {e,f,g} part could be solved iteratively, and finally the singleton {h}. It is thereby seen that the problem can be solved by two sequential, iterative problems with a single iterant in each, and evaluation of two single equations. What we did above by visual inspection is easily automated with well-known graph algorithms. Graph terminology uses the term component to represent a particular subset of edges and vertices of a graph. A component in which ever vertex can be reached from every other vertex, and is maximal in the sense that no other vertices could be included without losing this reachability, is said to be a strongly connected component, or strong component for short. In simple terms, it means that while you can reach any vertex in the strong component from any vertex within it, you cannot get back to it once you leave. It will be observed that this precisely describes the conditions for non-interdependent problem components. Therefore, since there are simple algorithms that identify the strong components of a directed graph (Aho, Hopcroft, and Ullman 1983), computation problems can be automatically decomposed into non-interdependent sub-problems. This is

* Exogenous values possibly required at various vertices are not shown.

what is done in the most recent version of SPARK.

Let us compare this solution approach to the alternatives. One approach would be to put the equation at every vertex in residual form and treat every variable as an iterant. This would require solution of a set of eight linear equations at each Newton iteration, requiring roughly 8^3 steps per iteration. Although done by many solvers, this is clearly far from optimal. Merely by using a small cut set algorithm we can reduce the problem to two iterants, requiring only 2^3 steps per iteration, as has always been done in SPARK. This gives a theoretical solution time reduction of 64:1. By strong component decomposition we instead solve two single equations completely outside of any iteration, and two 1-dimensional iterative problems requiring 1^3 steps per iteration. Ignoring the singletons, this is a 4:1 reduction relative to the normal SPARK cut set reduction, and 256:1 relative to the residual approach. While these reductions will not be fully realized due to overhead (see below), they provide strong motivation for pursuing the approach. Other than SPARK, the only program of which we are aware that decomposes the problem into strong components is EES (Klein 1991). However, the current version of EES allows algebraic problems only, so is not directly useful as a simulation tool.

Implementation

Because the implementation of strong component (SC) decomposition was a straightforward extension, we begin here by a review of the basic SPARK methodology. First, SPARK creates a bipartite graph in which the equation objects are represented in one set of vertices, and the problem variables in the other. A matching algorithm (McHugh 1990) is then applied, yielding an equation for each variable. Subsequently, a directed graph is created, not unlike the simple one depicted in Figure 1. This graph is processed by a cut set algorithm (Levy and Low 1988), yielding a small set of vertices that cut all cycles. For each vertex in the cut set, a new vertex, called a break, is added to the graph and all edges emanating from the original cut set vertex are caused to emanate instead from the break vertex. This creates an acyclic graph as shown in Figure 2.

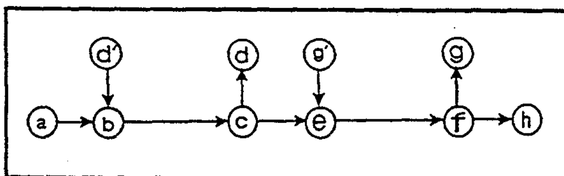

Figure 2. Acyclic data flow graph.

A topological sort (Aho, Hopcroft, and Ullman 1983) is then done on the vertices of the acyclic graph, yielding a visitation order; evaluating the equation objects in this order yields a new value for each original cut set vertex for a set of "guess" values at the break vertices. This is called "driving" the data flow graph. The differences between the guess and calculated values are treated as the "functions" which the Newton-Raphson iteration forces to zero.

The shortcoming of the above process is that it results in firing the entire graph at each iteration. The purpose of the revisions was to instead detect the strong components and treat each one separately, but in the same manner as just described.

The implementation attempted to make maximum use of code and data structures in the existing SPARK. Consequently, the parser which interprets the problem specification file and generates the basic problem data structures, remained unchanged. Also unchanged were the basic graph structures and algorithms. Since matching precedes construction of the data flow graph, and hence strong component decomposition, the matching algorithm also survived unchanged. After the matching, however, it was apparent that extensive changes would be necessary either in the basic structure of the data, or in the algorithms. We could either build a completely separate data flow structure for each strong component, or leave the existing whole-problem structure in place and simple tag the objects to identify their strong component membership. The former would require major changes in data structures with minimum algorithm changes, while the latter would be the reverse. The latter approach was selected because the data flow structure is rather complex, while the algorithms are straightforward.

With this approach, the directed problem graph is created as before. However, before doing the cut set, an SC algorithm is applied to the directed problem graph, marking each object with the its strong component number. Notably, the algorithm is such that the strong components are discovered in a reverse topological ordering, so along with the decomposition we learn the correct solution order. For efficient access, object pointers are collected into an array for each strong component. Then the cut set algorithm is applied to the entire problem and the break vertices are added exactly as before, but now marking them with the strong component number. This yields a whole-problem, acyclic data flow graph exactly as shown in Figure 2, except now each vertex is marked with a strong component number. The solve procedure, which previously subjected the entire data flow graph to the Newton-Raphson process, now contains a loop over the strong components in topological order. In a particular pass through this loop, only the vertices in the corresponding strong component are processed. Thus we solve the problem in pieces, although it is all stored in the same data structure.

Benchmark Testing

In order to see how much solution performance was improved, a problem was constructed that should greatly benefit from strong component decomposition. This problem consisted of ten simple zones connected in series by a single air stream, Figure 3.

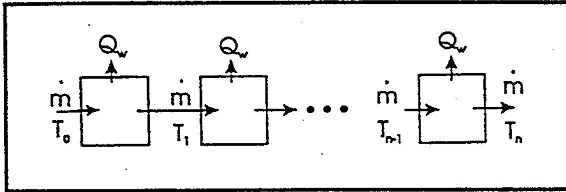

Figure 3. Rooms in series.

The zone model (a macro object) was such that a single break variable was required. Without strong component decomposition, 10 iteration variables are required, and a 10x10 linear set has to be solved at each Newton-Raphson iteration. With strong component decomposition, there are instead 10 sequential problems to be solved, each with a single break variable and a 1x1 linear set. Of course, the results produced agree precisely with those produced by earlier versions of SPARK (see Figure 4).

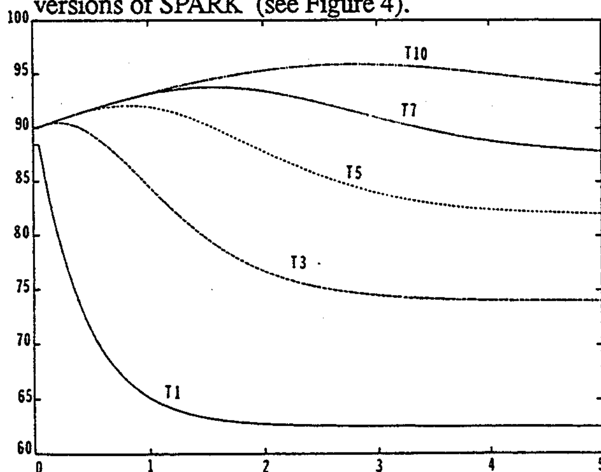

Figure 4. Temperature (F) of rooms 1, 3, 5, 7, and 10 vs. Time (h)

The observed speed-up for solving this problem at 0.1 hour intervals over a 1000 hour simulation period was between approximately 3 and 4, depending on the processor used (Sun Sparc 2: 2.65, Gateway 2000 486/33: 3.73).

If the entire execution time was attributable to solving the linear set within the Newton-Raphson loop, we would expect a theoretical speed up for this problem of 1x103:10x13=100:1. However, many other operations must be carried out, not all of which are affected by the decomposition. If we only look at the time spent in solving the linear set, we see a speed-up of 16:1, still somewhat lower than the theoretical value. One reason is that with decomposition there are more calls to the functions that carry out the Newton-Raphson process. We are investigating ways to improve performance in this area of the code.

With the aid of profiling tools, we also discovered other places where SPARK performance could be significantly improved. In particular, a great deal of time is spent moving arguments into place for the function calls. This is not unexpected, since every equation results in a function call. However, it appears that a simple change in addressing could achieve an additional speed-up by another factor of 2. This change is currently under investigation.

Multivalued Objects

The Need for Multivalued Objects in an Equation-Based Environment

Much of the knowledge and expertise of the building simulation community is embodied in FORTRAN code in such venerable programs as BLAST, DOE-2, TRNSYS, and HVACSIM+. One of the great deficiencies of the new simulation environments such as SPARK and IDA (IDA 1988) is the lack of complete building component libraries needed for building simulation. The developers of such environments tend to concentrate on the more exciting parts of the development task and allow the creation of component libraries to languish. Unfortunately, it can be difficult to use the old component models directly in the new environments. SPARK, for instance, needs its objects and macro objects in equation-based form. Furthermore each SPARK object is initially input-output free. When SPARK does assign inputs and outputs, only one output per object is allowed. In contrast, FORTRAN subroutines are algorithmic, have assigned inputs and outputs, and can have multiple outputs.

The new simulation environments may also not be ideal for solving certain types of problems. There is no reason to expect, for instance, that SPARK is an ideal method for studying forced or natural convection in a room. It is likely that such a study would employ an existing stand-alone FORTRAN program. It would be very useful, however, if there were a natural way in which such a program could be included as a subproblem in a larger SPARK simulation problem. The hard-coded convection model might form part of a larger SPARK zone model, for instance.

For the above reasons the SPARK simulation environment is being extended to include multivalued objects; i.e., objects that for a given set of inputs produce multiple outputs. These objects violate the basic philosophy of SPARK and inhibit SPARK's methods for reducing the effective problem size. However for practical reasons such objects may be needed for a particular problem and there is a fairly natural way of including them in the SPARK data structure.

Multivalued Objects in SPARK

In SPARK the strategy for handling multivalued objects is to replace a multivalued object having n inputs and m outputs with m objects, each having n inputs and 1 output. This will be done in a way transparent to the user. That is, the problem graph seen by the user either via the graphical interface or as a Network Specification Language (NSL) problem file will be unchanged, but the solver portion of SPARK will change the internal representation of the problem by replacing multivalued objects with sets of single-valued objects. Since multivalued objects are special in the sense that they require special processing by the solver, they need to be denoted as special in NSL by the addition of a new class type. Usually, in SPARK, a class represents an equation and is created with the NSL *define* keyword. There are other class types, however. A class representing a predictor

formula is created by the *define_pre* keyword for instance. When an object is instantiated, it inherits its type from its class. In NSL a class of multivalued objects will be created by using the new keyword *define_mult_out* as in the following example.

```
define_mult_out   example(in1,in2,in3,out1,out2)
    double in1,in2,in3,out1,out2;
{
   out1 = examplef(in1,in2,in3);
   out2 = examplef(in1,in2,in3);
}
```

Here a class *example* of type MULT_OUT is created. The C function associated with it is *examplef*. It will be invoked with an argument list in1,in2,in3,out1, out2; that is, the argument list will contain the inputs followed by the outputs. In most cases *examplef* will just call the FORTRAN subroutine or function needed to model or solve this portion of the problem.

In addition to this explicit change to NSL, the underlying data structure must also be enhanced. In particular, a vector of integers *obj_list* will be added to the object data structure. In the solver, this list will contain the unique object numbers of all the objects in a set of single-valued objects that replace a multivalued object.

The SPARK solver will start with the problem graph defined by the user. The solver changes this graph by replacing each multivalued object in the problem with a set of single output objects, each having the same inputs as the original object but having only one of its outputs. The outputs of these new objects are preassigned, so they do not have to go through the matching process. Once the normal objects have been matched with variables in the usual way, the directed problem graph, containing all the objects, can then proceed through cut set reduction normally and the final SPARK graph, the data flow graph, can be created as usual. The final step before numerical solution of the problem is to create the firing list: a list of objects in the order in which they will be executed. When an object is about to be added to the list, SPARK will check whether its type is MULT_OUT. If it is, all the objects in *obj_list* are added to the firing list in sequence. However, only the first of these objects will have the ability to execute the C function associated with the original multivalued object. When this first object is "fired" it will execute the function and the outputs of the function will be saved. When the subsequent objects in *obj_list* are fired, they will obtain their output from these saved output values. Thus, the function associated with the original multivalued object is only fired once in each traversal of the firing list.

SPARK Graphical Editor

The SPARK Graphical Editor (SGE) was developed to overcome the difficulties in developing and linking classes using the textual SPARK Network Specification Language. This language is "bottom up", i.e. you first define the lower level classes and then use them to build higher level classes. SGE facilitates "top down" creation of classes by simultaneously allowing higher level and lower level editing using the multiple windows capability. For example, a class can be temporarily defined by simply drawing its icon and its ports. It can then be instantiated on screen and linked to other objects, deferring its full definition (specifying its underlying equations) to a later time.

SGE is a multi-window, multi-buffer editor running under X-Windows that allows creation of objects, macro objects, and entire simulation problems by manipulating screen icons. A sample screen from the SGE is shown in Figure 5. Classes selected from a library can be instantiated as objects and placed anywhere on the screen. Once placed, they can be interconnected to form a network that represents the simulation problem. The objects can be moved, deleted, modified, or expanded to show internal structure. When the problem is complete, SGE creates a problem specification file that is sent to the SPARK kernel for matching, reduction, and solution.

In SGE, icons are used to represent objects. An icon consists of a polygonal outline that gives a pictorial representation of the object (such as a fan), plus tick marks that represent ports. The ports are the object's variables that can be linked to other objects.

A class is created with SGE as follows (see Figure 5 for numbered items): From the **file ops** menu select **edit file** and give the class a name. This will open an empty window with the class name in the upper left-hand corner. Click the **icon-outline** button (12). Draw the icon outline in the window using line segments. Click the **make-port** button (14) to draw the ports (by clicking on the icon outline) and to name the ports.

If this is a primitive class (one with a single equation), open a text window and enter the differential or algebraic equation for the class. (A text window is opened by clicking on the class name and selecting 'text-window' from the resulting pop-up menu.) If the MACSYMA symbolic processor (MIT 1983) is available, invoke it to automatically produce the C-functions for the equation inverses (which are needed by the SPARK solver). Otherwise type in the C-functions by hand in the text window.

To create a macro class (one with two or more coupled equations) first instantiate two or more existing classes and then link the resulting objects together in a graphical editing window (20). The procedure is as follows: To instantiate a class, select the **make-object** button (1), then click on any icon, such as (24), that is in the directory window or in any other window. This allows a copy of the icon to be dragged to the editing window (20). A second click instantiates the class and places the resulting object's icon (23) in the editing window.

The next step in creating the macro class is to link the objects. Select the **make-link** button (5). Click on a port of an object in the editing window to start the link. Click on the port of another object (or on another already-existing link) to end the link. SGE

will give a warning if the link is illegal (for example, if you connect a temperature port to a flow port). (Clicking on a port after clicking the "connect hints" button (19) will highlight the other ports to which this port can be legally connected.) The **make-link** operation can also be used to connect external ports of the macro class (21, for example) to ports of objects inside the class (22, for example). After all of the links are made, the resulting macro class can be saved in the library using options under **file ops**.

Other editing buttons are available to **move** an object (2), **resize** an object (3), **delete** an object or a link (4), **reroute** a link (9); **edit** a link (6,7,8); **name** an item in a window (10); or **attach comments** to an item (15). The **query** button (16) gives detailed information on an item. For example, querying an object gives the object name, class name, class description, and scale factor for the icon display.

Two other buttons are used to specify inputs (17) and outputs (18) of a problem. Inputs can be data read from a file or values you are prompted to enter. Output can be shown as graphs or meters attached to the icons on the screen, or output can be written to a file for later display.

Conclusion

Early versions of SPARK established the viability of object oriented modeling and graph theoretic solution techniques for building simulation. This paper has addressed three important extensions. Strong component decomposition enhances the solution efficiency for a wide class of problems. Multivalued objects allow SPARK models to incorporate submodels that calculate multiple variables using conventional, procedural code. Finally, the user interface is enhanced by a graphical problem editor in a modern window environment.

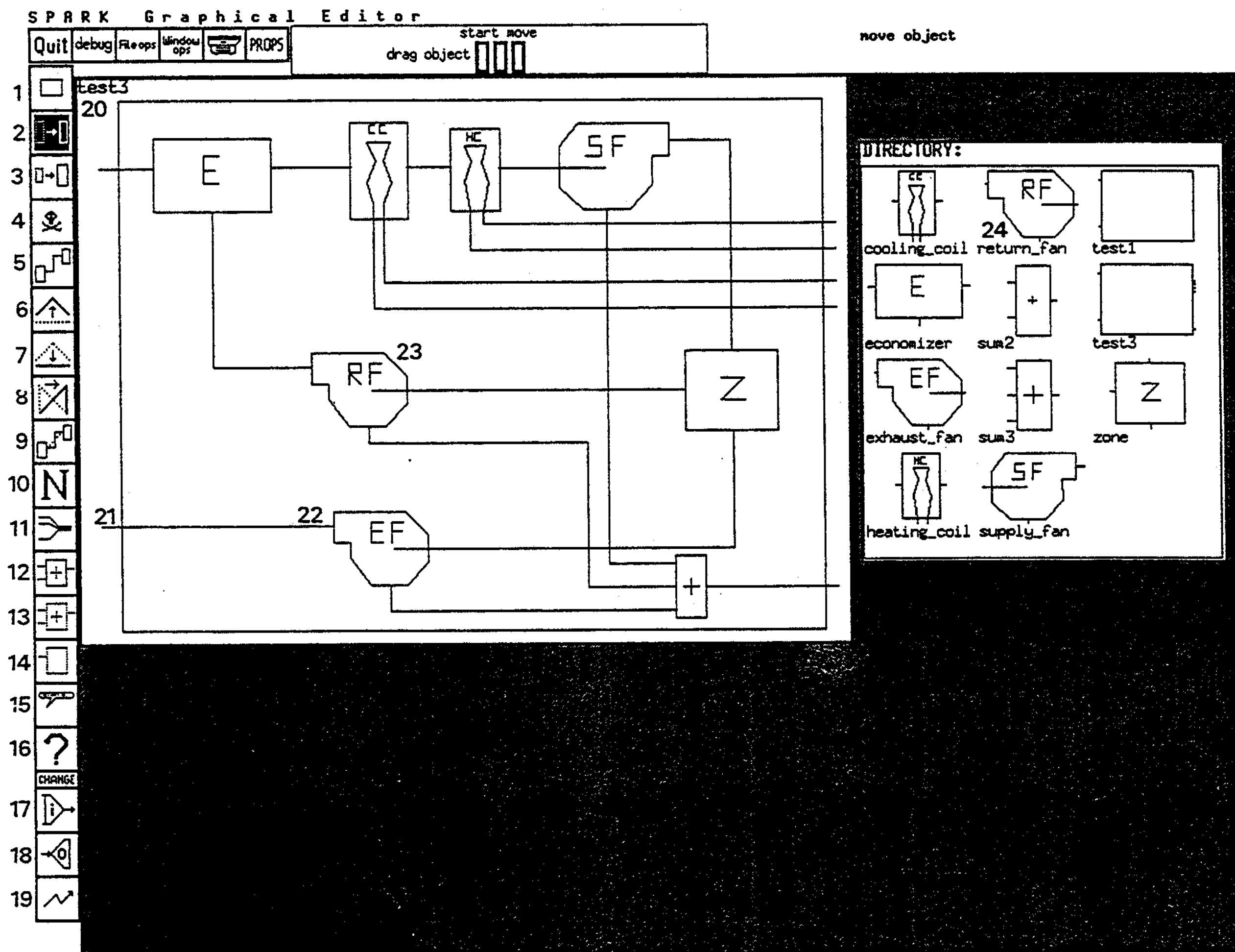

Figure 5:
Sample screen from the SPARK Graphical Editor. DIRECTORY is a (partial) library of classes representing HVAC components like fans and coils. A system model is created by dragging these classes into the editing window (20) and linking them. Numbered items are described in the text.

References

Aho 1983
Aho, A.V., J.E. Hopcroft, and J.D. Ullman. 1983. *Data Structures and Algorithms.* Addison Wesley, Reading, MA.

IDA 1988
Sahlin, P. 1988. (formerly MODSIM) "MODSIM, a Program for Dynamical Modeling and Simulation of Continuous Systems." Technical Report from the Institute of Applied Mathematics, P.O. Box 26300, S-100 41, Stockholm.

Klein 1991
Klein, S.A. 1991. *Engineering Equation Solver (EES),* F-Chart Software, Madison, WI.

Klein 1988
Klein, S.A. 1988. "TRNSYS - A Transient System Simulation Program." Technical Report of the Solar Energy Laboratory, Univ. of Wisconsin, Madison, WI.

Levy and Low 1988
Levy, H., and D.W. Low. 1988. "Contraction Algorithm for Finding Small Cycle Cut Sets." *J. Algorithms,* 9 : 470-493.

McHugh 1990
McHugh, J. 1990. *Algorithmic Graph Theory.* Prentice Hall, Englewood Cliffs N.J.

MIT 1983
MACSYMA Reference Manual, version 10, Mathlab Group, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Ortega 1970
Ortega, J.M. and W.C. Rheinboldt. 1970. *Iterative Solutions of Nonlinear Equations In Several Variables.* Academic Press, New York

Park, Clark, and Kelly 1985
Park, C., D.R. Clark, and G.E. Kelly. 1985. "An Overview of HVACSIM+, A Dynamic Building/HVAC Control Systems Simulation Program." In *Proceedings of the 1985 International Building Performance Simulation Association Conference* (Seattle, WA, Dec 3-6).