GENERAL PURPOSE SYMBOLIC SIMULATION TOOLS FOR ELECTRIC NETWORKS

Fernando L. Alvarado, Senior Member and Yenren Liu, Student Member

Electrical and Computer Engineering
The University of Wisconsin-Madison

Abstract

This paper presents research results on the use of computers to solve simulation problems in a way closer to human thinking. With the aid of techniques in AI (Artificial Intelligence), DB (Data Base Systems), and Computer Graphics, a set of general purpose LISP and PASCAL based simulation tools have been developed. Each of these tools solves a specific problem in some stage of simulation. Rule-based and object-oriented symbolic manipulations are extensively used. The tools provide more powerful and accurate modelling capability for complex objects, and permit simplicity and flexibility in implementation. The tools are used to study electrical transient problems, optimal load flow problems, linear control systems, and other simulation problems.

I. INTRODUCTION

The design of a system is not successful until its performance is well tested and optimized. Simulation is very important in the design of systems. It is often the only practical means to test system performance. To do a simulation, an engineer must first find a suitable representation or model for the system in question. A mathematical representation (a set of equations) is most useful. The behavior of many systems can be described by sets of equations. For example, electrical power systems can be described by a set of network equations; vibrations of an elastic system can be described by a set of dynamic equations; and characteristics of a control system can be described by a set of state equations.

Equation sets quickly become too complex for hand computation. Computers and advances in computer techniques have helped solve many problems. Many simulators and simulation programs have been developed to solve a variety of simulation problems. Simulation programs can be very successful and popular in their own application areas. Examples include the EMTP (Electro-Magnetic Transients Program) in Electrical Power Systems, SPICE in Electronics Circuits, and ACSL (Advanced Continuous Simulation Language) in control systems. All of these programs are specific to given classes of simulation problems. None of these can solve equations directly. Every existing simulation program only understands a specific language designed for it. To run a simulation, the engineer has to learn the simulation language first, then represent the system to be simulated in this language by either writing specified code or preparing specified input data. Recent studies have improved user interfaces to simulation programs by the use of graphics and database methods [2,6]. However, a more ambitious goal eluded all up to this time: a truly general purpose simulator that understands the natural language of simulation, namely mathematical equations of all kinds.

This paper presents initial research results toward this goal, a truly general purpose simulation tool, and illustrates the applications of this tool to a variety of power system problems. By integrating techniques in artificial intelligence (AI), data base systems (DB) and computer graphics, a set of symbolic manipulation assisted equation handling tools have been developed under the name SOLVER-Q. Symbolic manipulation means that the computer deals with symbolic expressions: it looks for particular symbols in the expressions, analyzes the meaning of the symbols, reorganizes the expressions and/or generates new expressions. SOLVER-Q can be used as a general purpose simulator which will solve problems in every stage of a simulation, in a way closer to human thinking. An engineer can simulate a design directly using the language of mathematics without learning any specific simulation language. Section 2 is a description of contents and features of SOLVER-Q. Sections 3 and 4 introduce some of the advanced techniques used in SOLVER-Q. Section 5 gives examples of applying SOLVER-Q to simulation problems. Section 6 discusses the weaknesses and advantages of this approach.

II. CONTENTS OF SOLVER-Q

The set of general purpose equation handling tools known as SOLVER-Q consists of a set of equation handling programs, written mostly in LISP and PASCAL. Each of these programs solves a specific problem in some stage of simulation. These programs can be used independently or they can be combined by a user to perform complex simulation tasks. Figure 1 illustrates these programs and their relationship.

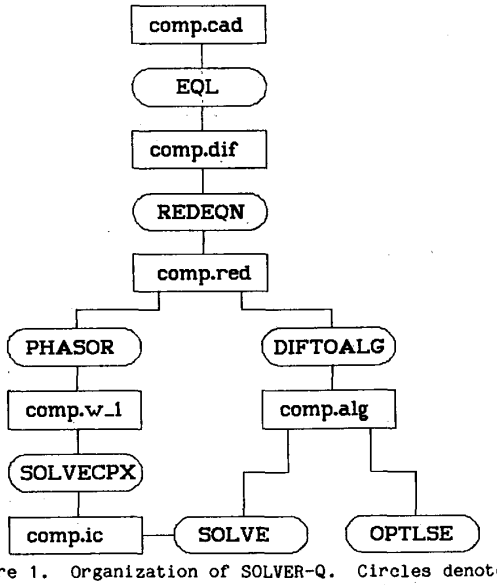

Figure 1. Organization of SOLVER-Q. Circles denote programs, rectangles denote files.

This paper was sponsored by the IEEE Power Engineering Society for presentation at the IEEE Power Industry Computer Application Conference, Montreal, Canada, May 18-21, 1987. Manuscript was published in the *1987 PICA Conference Record.*

0885-8950/88/0500-0689$01.00©1988 IEEE

The following sub-sections describe the function and features of each program within SOLVER-Q by means of examples.

1. EQL: automatic generation of differential equation systems:

Figure 2 illustrates a simple electrical circuit. File comp.cad contains a database representing this circuit (see details of the database in [2]). Figure 3 illustrates the differential equation system generated by EQL for this circuit.

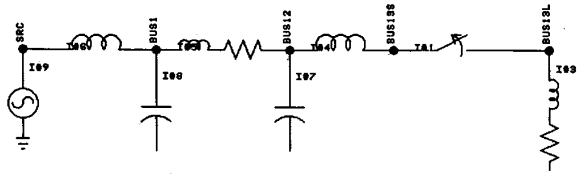

Figure 2. A Simple Circuit.

```
{ Branch Current and Voltage Equations }
{ Branch # 1 }
IF (t_0<1.E-3) THEN i_BUS13S_1_1 ELSE (i_BUS13S_1_1+i_BUS13L_1_2);
IF (t_0<1.E-3) THEN i_BUS13L_1_2 ELSE (v_BUS13S-v_BUS13L);
{ Branch # 3 }
i_BUS13L_3_1 + i_0_3_2;
v_BUS13L - vm_3_1 = 22.61 * i_BUS13L_3_1;
0.001 * 19.72 * D(i_BUS13L_3_1,t) = vm_3_1 - 0.0;
{ Branch # 4 }
i_BUS12_4_1 + i_BUS13S_4_2;
0.001 * 6.0 * D(i_BUS12_4_1, t)=v_BUS12 - v_BUS13S;
{ Branch # 5 }
i_BUS1_5_1 + i_BUS12_5_2;
v_BUS1 - vm_5_1 = 0.05 * i_BUS1_5_1;
0.001 * 2. * D(i_BUS1_5_1,t) = vm_5_1 - v_BUS12;
{ Branch # 6 }
i_SRC_6_1 + i_BUS1_6_2;
0.001 * 6.0 * D(i_SRC_6_1, t)=v_SRC - v_BUS1;
{ Branch # 7 }
i_BUS12_7_1 + i_0_7_2;
D(v_BUS12,t)-D(0.0,t) = 1/(0.8*0.000001) * i_BUS12_7_1;
{ Branch # 8 }
i_BUS1_8_1 + i_0_8_2;
D(v_BUS1,t)-D(0.0,t) = 1/(0.8*0.000001) * i_BUS1_8_1;
{ Branch # 9 }
v_SRC = 56.34 * cos (2 * 3.14159 * (60. * t + 0./360.0));
i_SRC_9_1 + i_0_9_0;
{Nodal Current Equations}
i_SRC_9_1 + i_SRC_6_1;
i_BUS1_6_2 + i_BUS1_8_1 + i_BUS1_5_1;
i_BUS12_7_1 + i_BUS12_4_1 + i_BUS12_5_2;
i_BUS13S_4_2 + i_BUS13S_1_1;
i_BUS13L_1_2 + i_BUS13L_3_1;
```

Figure 3. The Differential Equation System File comp.dif.

The conventions used by SOLVER-Q to write equations are few: equations must end with a semicolon; "^" denotes exponentiation; D(x,t) means dx/dt; and an equation that has no equal sign "=" is assumed to have a zero right hand side.

Equations generated by EQL are based on two rules:

Rule #1: each component in the electrical circuit is representable by a set of equations.

For example, the equations:

```
v1 - v2 = i1 * R;
i1 + i2 = 0;
```

are used to represent a resistor where v1, v2 are two terminal voltages and i1, i2 are currents injected into each of the two terminals of the resistor, respectively. R is the resistance of the resistor.

The equations representing an inductor are:

```
v1 - v2 = L * D(i1, t);
i1 + i2 = 0;
```

and the equations for a capacitor are:

```
D(v1, t) - D(v2, t) = 1/C * i1;
i1 + i2 = 0;
```

where L is the inductance of the inductor and C is the capacitance of the capacitor. Mathematical models for much more complex components are possible.

Rule #2: connections between components establish equations relating component variables to each other.

For the case of electrical circuits, connections of components establish two added constraints to each connection point:

o the voltage variables become equal:

```
v1 = v2 = ...
```

o the current variables add up to zero:

```
i1 + i2 + ... = 0
```

2. REDEQN: reducing equations

The program REDEQN follows a set of formal rules to reduce equations. REDEQN first looks for constant variables, equivalent variables, and complemental variables. REDEQN then performs symbolic substitutions of variables and removes redundant equations from the equation system. Constant variables are those to which a constant value can be assigned. Equivalent variables means that if x1 - x2 = 0 then x1 and x2 are equivalent. Complemental variable means that if x1 + x2 = 0 then x1 and x2 are complemental. In addition to these three explicit rules of variable substitutions, the following rule is also used for deduction of equivalent variables:

```
IF (x1 + x2 = 0) AND (x1 + x3 = 0)
THEN x2 and x3 are equivalent.
```

Figure 4 illustrates the equation system from Figure 3 after processing by REDEQN.

```
{ constant variables :
}
{ equivalent variables :
(I_SRC_6_1                I_0_9_0)
(I_SRC_9_1                I_BUS1_6_2)
(I_BUS13S_1_1             I_BUS12_4_1)
(I_BUS13L_1_2             I_0_3_2)
}
{ complemental variables :
(I_BUS13L_3_1             I_0_3_2)
(I_BUS12_4_1              I_BUS13S_4_2)
(I_BUS1_5_1               I_BUS12_5_2)
(I_SRC_6_1                I_BUS1_6_2)
(I_BUS12_7_1              I_0_7_2)
(I_BUS1_8_1               I_0_8_2)
}
IF ( T_0 < 0.001 ) THEN -I_BUS13S_4_2 ELSE ( -I_BUS13S_4_2 + I_0_3_2 );
IF ( T_0 < 0.001 ) THEN I_0_3_2 ELSE ( V_BUS13S - V_BUS13L );
V_BUS13L - VM_3_1 = 22.61 * -I_0_3_2;
0.001 * 19.72 * D ( -I_0_3_2 , T ) = VM_3_1 - 0.0;
0.001 * 6.0 * D ( -I_BUS13S_4_2 , T ) = V_BUS12 - V_BUS13S;
V_BUS1 - VM_5_1 = 0.05 * -I_BUS12_5_2;
0.001 * 2.0 * D ( -I_BUS12_5_2 , T ) = VM_5_1 - V_BUS12;
0.001 * 6.0 * D ( I_0_9_0 , T ) = V_SRC - V_BUS1;
D ( V_BUS12 , T ) - D ( 0.0 , T ) = 1 / ( 0.8 * 0.000001 ) * -I_0_7_2;
D ( V_BUS1 , T ) - D ( 0.0 , T ) = 1 / ( 0.8 * 0.000001 ) * -I_0_8_2;
V_SRC = 56.34 * COS ( 2 * 3.14159 * ( 60.0 * T + 0.0 / 360.0 ) );
-I_0_9_0 + -I_0_8_2 + -I_BUS12_5_2;
-I_0_7_2 + -I_BUS13S_4_2 + I_BUS12_5_2;
```

Figure 4. The Reduced Differential Equation System File comp.red.

Other rules used by REDEQN perform symbolic equation simplification, combination of similar terms, elimination of redundant ones, and evaluation of numeric functions whenever possible.

3. PHASOR: steady state phasor analysis

The program PHASOR is used to study the frequency response of a circuit or to find a steady state solution to the circuit. It converts a time domain differential equation system into phase domain complex algebraic equation sets. It first searches for sinusoidal sources in the differential equation system, then for each frequency (including DC) it generates a separate set of phasor equations. The equation system in Figure 4 has only one frequency, 60 Hz, hence only one set of phasor equations is generated. This is illustrated in Figure 5.

```
W_1 = 376.9908;
I_BUS13S_4_2;
I_0_3_2;
(V_BUS13L - VM_3_1) - (22.61 * (- I_0_3_2));
((0.01972 * (- I_0_3_2)) * (W_1 * (PHASOR 1.57079632))) - VM_3_1;
((0.006 * (- I_BUS13S_4_2)) * (W_1 * (PHASOR 1.57079632))) -
        (V_BUS12 - V_BUS13S);
(V_BUS1 - VM_5_1) - (0.05 * (- I_BUS12_5_2));
((0.002 * (- I_BUS12_5_2)) * (W_1 * (PHASOR 1.57079632))) -
        (VM_5_1 - V_BUS12);
((0.006 * I_0_9_0) * (W_1 * (PHASOR 1.57079632))) - (V_SRC - V_BUS1);
(V_BUS12 * (W_1 * (PHASOR 1.57079632))) - (1250000.0 * (- I_0_7_2));
(V_BUS1 * (W_1 * (PHASOR 1.57079632))) - (1250000.0 * (- I_0_8_2));
V_SRC - (56.34 * (PHASOR 0.0));
((- I_0_9_0) + (- I_0_8_2)) + (- I_BUS12_5_2);
((- I_0_7_2) + (- I_BUS13S_4_2)) + I_BUS12_5_2;
```

Figure 5. The Phasor Equation System.

4. DIFTOALG: conversion to algebraic equations

The program DIFTOALG converts differential equation systems into algebraic equation systems based on any desired numerical integration technique, including all implicit integration methods. The trapezoidal integration rule was chosen for this paper because it leads to stable and accurate solutions and because it has been used extensively in the simulation of power system transients [4,5]. This permits easy comparisons with existing simulation software. Figure 6 illustrates the algebraic equation system generated from Figure 4.

```
IF( T_0 < 0.001 ) THEN - I_BUS13S_4_2 ELSE ( - I_BUS13S_4_2 I_0_3_2 );
IF( T_0 < 0.001 ) THEN I_0_3_2 ELSE ( V_BUS13S - V_BUS13L );
V_BUS13L - VM_3_1 = 22.61 * - I_0_3_2;
0.001 * 19.72 * - DI_0_3_2 = VM_3_1 - 0.0;
0.001 * 6.0 * - DI_BUS13S_4_2 = V_BUS12 - V_BUS13S;
V_BUS1 - VM_5_1 = 0.05 * - I_BUS12_5_2;
0.001 * 2.0 * - DI_BUS12_5_2 = VM_5_1 - V_BUS12;
0.001 * 6.0 * DI_0_9_0 = V_SRC - V_BUS1;
DV_BUS12 - 0 = 1 / ( 0.8 * 0.000001 ) * - I_0_7_2;
DV_BUS1 - 0 = 1 / ( 0.8 * 0.000001 ) * - I_0_8_2;
V_SRC = 56.34 * COS ( 2 * 3.14159 * ( 60.0 * ( T_0 + DT ) + 0.0 / 360.0 ) );
- I_0_9_0 + - I_0_8_2 + - I_BUS12_5_2;
- I_0_7_2 + - I_BUS13S_4_2 + I_BUS12_5_2;
V_BUS1 - V_BUS1_0 - ( DT / 2.0 ) * ( DV_BUS1_0 + DV_BUS1 );
V_BUS12 - V_BUS12_0 - ( DT / 2.0 ) * ( DV_BUS12_0 + DV_BUS12 );
I_0_9_0 - I_0_9_0_0 - ( DT / 2.0 ) * ( DI_0_9_0_0 + DI_0_9_0 );
I_BUS12_5_2 - I_BUS12_5_2_0 - ( DT / 2.0 ) * ( DI_BUS12_5_2_0 +
        DI_BUS12_5_2 );
I_BUS13S_4_2 - I_BUS13S_4_2_0 - ( DT / 2.0 ) * ( DI_BUS13S_4_2_0
        + DI_BUS13S_4_2);
I_0_3_2 - I_0_3_2_0 - ( DT / 2.0 ) * ( DI_0_3_2_0 + DI_0_3_2 );
DV_BUS1_0 = $DV_BUS1 0;
DV_BUS12_0 = $DV_BUS12 0;
DI_0_9_0_0 = $DI_0_9_0 0;
I_0_9_0_0 = $I_0_9_0 0.00000025;
DI_BUS12_5_2_0 = $DI_BUS12_5_2 0;
I_BUS12_5_2_0 = $I_BUS12_5_2 -0.00000025;
V_BUS1_0 = $V_BUS1 56.41698249;
V_BUS12_0 = $V_BUS12 56.42981435;
DI_BUS13S_4_2_0 = $DI_BUS13S_4_2 0;
DI_0_3_2_0 = $DI_0_3_2 0;
I_0_3_2_0 = $I_0_3_2 0;
I_BUS13S_4_2_0 = $I_BUS13S_4_2 0;
T_0 = $T_1 0;
DT = 0.0001;
T_1 = T_0 + DT;
```

Figure 6. The Algebraic Equation System File comp.alg.

5. SOLVE: solving general non-linear equations

The program SOLVE is the central program in SOLVER-Q. It solves simultaneous linear or nonlinear algebraic equation systems using Newton's method and variations [9]. Note that in the last few equations in Figure 6, some variables are prefixed by "$". This implements a time advancing solution. Variables on the left hand side of the equations are initially equal to the numbers on the right hand side. The values of these variables are automatically replaced by values of those which are prefixed by a "$" before the next time step. This enables the program to simulate continuously varying systems. Thus, the same equations that are used to advance time from t to t+Δt can be used to advance time from t+Δt to t+2Δt.

Many modern concepts of computer technology, such as rule-based and object-oriented symbolic manipulations and other advanced computation techniques are used in implementing this program. Sections 3 and 4 introduce these concepts and techniques. Figure 7 gives the resulting solution to the circuit.

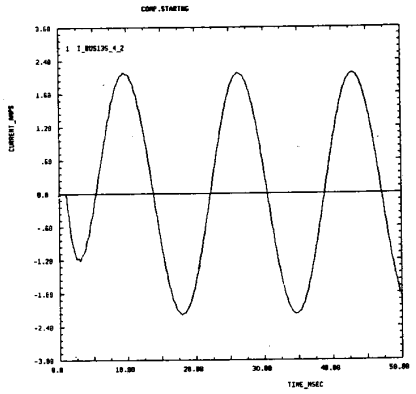

Figure 7. The Solution to the Circuit in Figure 2.

6. SOLVECPX: solving complex algebraic equation systems

SOLVECPX solves arbitrary nonlinear complex equations. Although SOLVECPX can also solve real algebraic equation systems, it is slower than SOLVE. SOLVECPX was derived from SOLVE by replacing all arithmetic operations on real numbers with generic object-oriented operations.

The solution to complex phasor equations is of interest in its own right. The solution to phasor equation systems can also be used as an initial condition of the differential equation system for transients studies. SOLVECPX recognizes both rectangular and polar forms of complex number. (RECT real imag) or simply (real imag) represent rectangular form and (PHASOR mag angle) represents polar form. Figure 8 is the solution to the phase equation system and Figure 9 is the initial condition file automatically generated by SOLVECPX.

```
W_1 = 376.9908
I_BUS13S_4_2 = 0.0
I_0_3_2 = 0.0
V_BUS13L = 0.0
VM_3_1 = 0.0
V_BUS12 = (56.42981435 -0.00085171)
V_BUS13S = (56.42981435 -0.00085171)
V_BUS1 = (56.41698249 -0.00000058)
VM_5_1 = (56.41698248 -0.00085152)
I_BUS12_5_2 = (-0.00000025 -0.01701881)
I_0_9_0 = (0.00000025 0.03403376)
V_SRC = (56.34 0.0)
I_0_7_2 = (-0.00000025 -0.01701881)
I_0_8_2 = (-2.91284366e-10 -0.01701494)
```

Figure 8. The Solution to the Phasor equation system in File comp.w_1.

```
DT ~ 0.0001;
W_1 ~ 376.9908;
I_BUS13S_4_2 ~ 0.0;
I_0_3_2 ~ 0.0;
V_BUS13L ~ 0.0;
VM_3_1 ~ 0.0;
V_BUS12 ~ 56.42981435;
V_BUS13S ~ 56.42981435;
V_BUS1 ~ 56.41698249;
VM_5_1 ~ 56.41698248;
I_BUS12_5_2 ~ -0.00000025;
I_0_9_0 ~ 0.00000025;
V_SRC ~ 56.34;
I_0_7_2 ~ -0.00000025;
I_0_8_2 ~ -2.91284366e-10;
```

Figure 9. The Initial Condition File comp.ic generated by SOLVECPX.

7. OPTLSE: optimization and regression

OPTLSE solves optimization problems with both equality constraints and inequality constraints using the Lagrange multipliers method. The Lagrange multipliers method is a means of reducing constrained optimization problems to unconstrained ones. The problem:

$$\text{minimize } f(\bar{x})$$
$$\text{subject to } h(\bar{x}) = 0;$$

can be reduced to finding a stationary point of the Lagrangian:

$$L = f + \bar{\mu}^{t}\,\bar{h}$$

This can be done by setting:

$$\frac{\partial L}{\partial \bar{x}} = 0$$

$$\frac{\partial L}{\partial \bar{h}} = 0$$

This entire process is performed first symbolically and then numerically by OPTLSE. OPTLSE also solves state estimation and regression problems using the least square errors rule. Examples of application of OPTLSE are given in Section 4.

III. SYMBOLIC MANIPULATIONS

Many modern concepts of computer technology are used in the implementation of the general purpose equation handling system SOLVER-Q. Rule-based and object-oriented symbolic manipulations are extensively used.

Symbolic manipulation means that a program uses symbolic expressions as data. Based on some rules, the program recognizes particular symbolic expressions, tears them apart, reassembles them and generates new symbolic expressions. A rule base or knowledge base system is necessary to conduct these symbolic manipulations. Rule-based systems are the best means for representing the problem-solving approach of human experts. The knowledge base contains a set of situation-action or cause-effect rules. These rules are usually expressed in the "IF-THEN" or "IF-THEN-ELSE" statement form. Problem-solving is achieved by selecting proper rules and applying these rules. The formation of Jacobians and Lagrangians involves a large amount of symbolic differentiation operations. The rules that conduct the differentiation are well-defined in a mathematical sense. For each primary mathematical function, there is a well-defined differentiation rule. The differentiation rules for arithmetic operations and for compound functions are available. Therefore, the differentiation of any complex function or operation can be derived by recursively applying these fundamental rules. To implement these differentiation rules, it is possible to define a function "differentiate" in the "IF-THEN-ELSE" form. In LISP terminology [8], this can be expressed as:

```
(defun differentiate (E x)
  (cond ((atom E) (cond ((equal E x) 1)
                        (t 0)))
        ((or (equal (operator E) '+) (equal (operator E) '-))
         '(,(operator E)      ,(differentiate (arg1 E) x)
                              ,(differentiate (arg2 E) x)))
        ((equal (operator E) '*)
         '(+ (* ,(arg1 E)     ,(differentiate (arg2 E) x))
             (* ,(arg2 E)     ,(differentiate (arg1 E) x))))
        ((equal (operator E) '/)
         '(/ (- (* ,(arg2 E)  ,(differentiate (arg1 E) x))
                (* ,(arg1 E)  ,(differentiate (arg2 E) x)))
            (square (arg2 E))))
        ((equal (operator E) 'sin)
         '(* (cos ,(arg1 E))  ,(differentiate (arg1 E) x))))
        .
        .
        .)
```

This means the following:

```
dx/dx = 1
d(constant)/dx = 0
d(y1 ± y2)/dx = dy1/dx ± dy2/dx;
d(y1 * y2)/dx = y1 * dy2/dx + y2 * dy1/dx
d(y1 / y2)/dx = (y2*dy1/dx - y1*dy2/dx)/y2^2
d(sin(y))/dx = cos(y) * dy/dx
      .
      .
      .
```

"operator", "arg1" and "arg2" are defined as follows:

```
(defun operator (E) (car E))
(defun arg1 (E) (cadr E))
(defun arg2 (E) (caddr E))
```

In LISP, all operations use prefix notation, that is, the first item in a LISP expression is an operator, followed by the first argument and then the second argument. "car", "cadr" and "caddr" represent the 1st, 2nd and 3rd element of a list, respectively.

Object-oriented programming is more efficient than "IF-THEN-ELSE" rules for implementing differentiation rules. In the object-oriented approach, a generic operation "differentiate" is defined:

```
(defun differentiate (E x)
  (cond ((atom E) (cond ((equal E x) 1)
                        (t 0)))
        (t (apply (get (operator E) 'differentiate) (list E X)))))
```

Associated with each type of arithmetic operation or primary function an object-oriented differentiate operation is defined as follows:

```
(setf (get '+ 'differentiate)
   #'(lambda (E x) '(+ ,(differentiate (arg1 E) x)
                       ,(differentiate (arg2 E) x))))

(setf (get '- 'differentiate)
   #'(lambda (E x) '(- ,(differentiate (arg1 E) x)
                       ,(differentiate (arg2 E) x))))

(setf (get '* 'differentiate)
   #'(lambda (E x) '(+ (* ,(arg2 E)  ,(differentiate (arg1 E)))
                       (* ,(arg1 E)  ,(differentiate (arg2 E) x)))))

(setf (get '/ 'differentiate)
   #'(lambda (E x) '(/ (- (* ,(arg2 E)  ,( differentiate (arg1 E) x))
                          (* ,(arg1 E)  ,( differentiate (arg2 E) x)))
                       (square ,(arg2 E)))))

(setf (get 'sin 'differentiate)
   #'(lambda (E x) '(* (cos ,(arg1 E))  ,( differentiate (arg1 E) x))))
```

where the "lambda" notation is a way of associating some formal parameters (such as "e" and "x") with a piece of code without the need for a separate function name.

This object-oriented approach has advantages over the traditional "IF-THEN-ELSE" approach. It achieves run-time efficiency because it provides direct access to the code to be executed without conditional comparisons. Moreover, by hiding the details of implementation from the generic operation "differentiate", the object-oriented approach achieves a higher level of abstraction: it is much easier to make modifications to an object-oriented rule than to a big set of "IF-THEN-ELSE" rules, and it is easy to add more types of new functions and operations to the generic operation. Other advantages of using the object-oriented approach to implement a rule-based system are pointed out in the literature [1,3].

After recursive application of the differentiation rules to the expressions, the ultimate expressions may become untolerably large and complex. It is important to simplify these expressions to save storage space and computing time. A set of simplification rules are implemented using the object-oriented approach. The following are some examples of rules used to conduct the symbolic simplification manipulations:

```
(defun simplify (E)
   (cond ((atom E) E)
         (t (apply (get (car E) 'simpcode) (list E)))))

; rule to simplify addition operation

(setf (get '+ 'simpcode) 'simpplus)
(defun simpplus (E)
   (let ((arg1 (simplify (cadr E))) (arg2 (simplify (caddr E))))
   (cond ((and (numberp arg1) (numberp arg2)) (+ arg1 arg2))
         ((zerop arg1) arg2)
         ((zerop arg2) arg1)
         (t (list (car E) arg1 arg2)))))

; rule to simplify subtraction operation

(setf (get '- 'simpcode) 'simpdifference)
(defun simpdifference (E)
   (let ((arg1 (simplify (cadr E))) (arg2 (simplify (caddr E))))
   (cond ((and (numberp arg1) (numberp arg2)) (- arg1 arg2))
         ((equal arg1 arg2) 0.0)
         ((zerop arg2) arg1)
         ((zerop arg1) (list 'minus arg2))
         (t (list (car E) arg1 arg2)))))
```

Rule-based and object-oriented symbolic manipulations are used in other programs within SOLVER-Q. For example; the trapezoidal integration rule in DIFTOALG, the rule used to find equivalent variables in REDEQN, the generic arithmetic operations in SOLVECPX, and the Lagrange multipliers method and the least square errors rule in OTPLSE.

IV. DIRECT MODELLING METHOD

SOLVER-Q represents a direct and powerful modelling environment for simulation. SOLVER-Q uses a direct modelling method: use equations for the modelling of both components of a system and relationships between components. By putting all these equations together, the resulting simultaneous equation set is a mathematical model for the system. This is a very simple and powerful method. Many components of a system are easily and accurately modelled by differential and logic equations. The connection between components are also expressed as equations. Section 2.1 described the models for resistors, inductors and capacitors, and the connection rules for electrical circuits. The following are additional examples of models for components of electrical power systems.

A switch is modelled by two logical equations:

```
if t < t_close then i1 else i1 + i2;
if t < t_close then i2 else v1 - v2;
```

The model for a diode is:

```
if (v1 - v2 < 0) and (i1 < i_hold)
then i1 else i1 + i2;
if (v1 - v2 < 0) and (i1 < i_hold)
then i2 else v1 - v2 = i1 * r_in;
```

The diode can be modelled as an ideal switch with a small linear or nonlinear resistance r_in when it is conducting.

The model for SiC gapped arrester is:

```
IF (V4 < 150000.) THEN I4 = 0.0 + (16.1-0.0)/(150000.-0.0)*(V4-0.0) ELSE
IF (V4 < 200000.) THEN I4 = 16.1 + (94.5-16.1)/(200000.-150000.)*(V4-150000.) ELSE
IF (V4 < 220000.) THEN I4 = 94.5 + (167.3-94.5)/(220000.-200000.)*(V4-200000.) ELSE
IF (V4 < 240000.) THEN I4 = 167.3 + (282.1-167.3)/(240000.-220000.)*(V4-220000.) ELSE
IF (V4 < 260000.) THEN I4 = 282.1 + (455.0-282.1)/(260000.-240000.)*(V4-240000.) ELSE
IF (V4 < 280000.) THEN I4 = 455.0 + (711.2-455.0)/(280000.-260000.)*(V4-260000.) ELSE
IF (V4 < 300000.) THEN I4 = 711.2 + (1076.0-711.2)/(300000.-280000.)*(V4-280000.) ELSE
IF (V4 < 400000.) THEN I4 = 1076.0 + (6046.0-1076.0)/(400000.-300000.)*(V4-300000.) ELSE
I4 = 6046.0;
IF vm_4_1 - 0.0 < 0 THEN V4 + vm_4_1 - 0.0 ELSE V4 - vm_4_1 + 0.0;
IF vm_4_1 - 0.0 < 0 THEN i_REC_4_1 + I4 ELSE i_REC_4_1 - I4;
IF v_REC < 0 THEN V4g + v_REC ELSE V4g - v_REC;
IF B4_1 > 0 and B4_2 > 0 THEN v_REC - vm_4_1 ELSE i_REC_4_1;
IF B4_1 > 0 and B4_2 > 0 THEN i_REC_4_1 + i_0_4_2 ELSE i_0_4_2;
IF (V4g >= 234740.) and (B4_1 > 0) THEN B4_2 - 1.0 ELSE B4_2 - B4_4;
IF (I_rec_4_1 < 0.0) and (B4_2 > 0) THEN B4_1 + 1 ELSE B4_1 - B4_3;
B4_4 = $B4_2 -1.0;
B4_3 = $B4_1 1.0;
```

This model describes not only the nonlinear but also the flash over characteristics of the SiC gapped arrester. The first equation represents the piecewise-linear characteristics of a nonlinearity. The remainder of the equations contain the logic for flash-over characteristics. In this model if the amplitude of the voltage over the arrester is greater than 234700, the gap will flash over; then the gap will discharge, clear once and then stay permanently open thereafter.

An important aspect of the direct modelling method is that it is independent of simulation programs or algorithms used. Most traditional modelling methods are program or algorithm dependent. For example, the modelling method in the EMTP is based on the trapezoidal integration rule and is embedded into the

program. Should the designer of EMTP find another algorithm better than the trapezoidal integration rule or wish to add new models for more components, the EMTP program must be modified. The user can not change both models and algorithms in traditional simulation programs. By separating the models from the program, the direct modelling method provides a great deal of flexibility and portability. The user (not the program designer) can easily modify an old model based on a new algorithm or add a new model without changing the simulation programs. The connection rules which govern the relationships between components can also be replaced. By adding different models for components and connection rules of different systems, SOLVER-Q can be used as a general purpose simulator which is able to solve a variety of simulation problems. The following is an example of how easy it is to modify the model for a component. The single phase transmission line was modelled by a lossless line which can be represented in these equations:

```
z3 = sqrt(1000.*.9238/.0126);
t3 = 193.1 * sqrt(1.E-9*3.222 * 0.00787);
i_SEND_3_1 = v_SEND/z3 - (v_REC delay t3)/z3 - (i_REC_3_2 delay t3);
i_REC_3_2 = v_REC/z3 - (v_SEND delay t3)/z3 - (i_SEND_3_1 delay t3);
```

This model ignores the series resistance and results in an unrealistic solution to many problems. Just by replacing the equations above with the equations below, a more accurate model for lossy lines is established:

```
z3 = sqrt(1000.*3.222/.00787) + 0.25 * r3;
r3 = 0.3167 * 193.1;
t3 = 193.1 * sqrt(1.E-9*3.222 * .00787);
h3 = (z3 - 0.25 * r3) / (z3 + 0.25 * r3);
i_SEND_3_1 = v_SEND/z3 + 0.5 * (1 + h3) * I3k + 0.5 * (1 - h3) * I3m;
i_REC_3_2 = v_REC/z3 + 0.5 * (1 + h3) * I3m + 0.5 * (1 - h3) * I3k;
I3k = - ( v_REC delay t3)/z3 - ( i_REC_3_2 delay t3);
I3m = - ( v_SEND delay t3)/z3 - ( i_SEND_3_1 delay t3);
```

where "delay" is a function defined to find a past value of a variable.

Many advanced computation techniques are used in SOLVE. These include sparse matrix methods, optimal ordering and factorization, fast repeat solution techniques, and others.

V. APPLICATIONS OF SOLVER-Q

The example in Section 2 demonstrated how SOLVER-Q solves a simulation problem in electrical power transients. The solution to this problem is identical to that from the EMTP (see [2]) because both programs use the exact same modelling methods and algorithms.

Other electro-magnetic transient phenomena can be simulated using SOLVER-Q. Figure 10 illustrates the energization of a 120 mile single phase transmission line terminated with a SiC gapped lightning arrester. For this case, SOLVER-Q obtains a solution very close to the solution from the EMTP. Figures 11 and 12 show results from both programs.

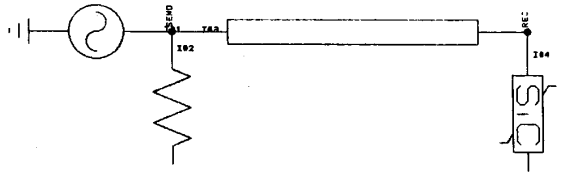

Figure 10. Energization of a line with a SiC gapped arrester.

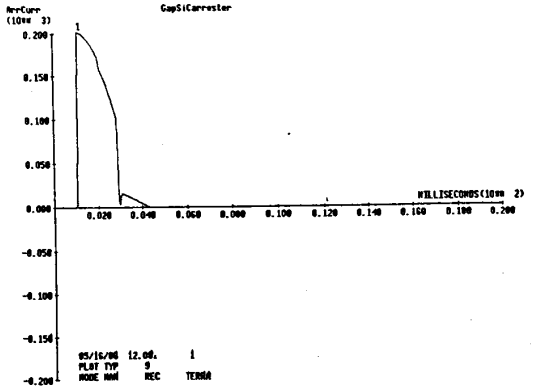

Figure 11. Results from the EMTP.

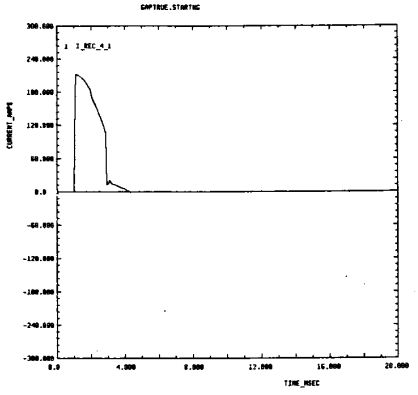

Figure 12. Results from SOLVER-Q.

Figure 13 shows a simple three phase bridge rectifier with a resistive load. For this case, both the EMTP and SOLVER-Q give a sixth order harmonic wave of current flowing through the load resistor. (See Figure 14).

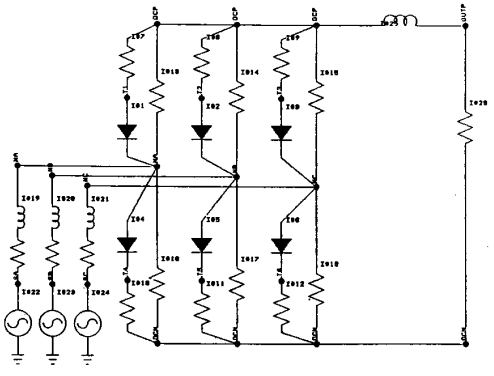

Figure 13. A Simple Three Phase Bridge Rectifier.

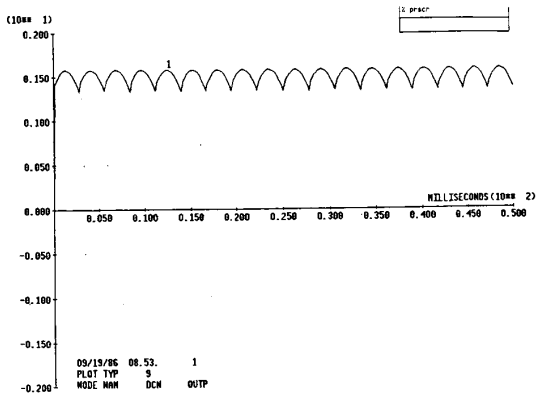

Figure 14. Load Current, Result from EMTP.

Figure 15 adds a filter between the bridge and the load in Figure 14. The filter consists of two capacitors and an inductor. The EMTP fails to solve this case. However, SOLVER-Q gives a reasonable solution as shown in Figure 16.

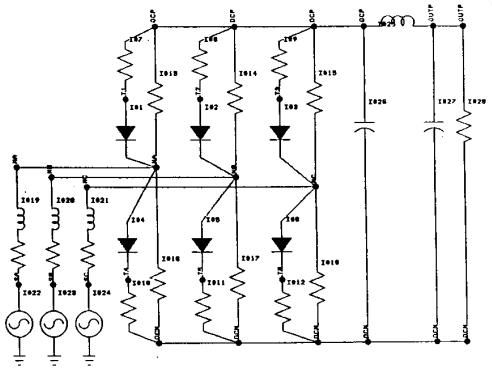

Figure 15. A Simple Three Phase Bridge Rectifier with a Filter.

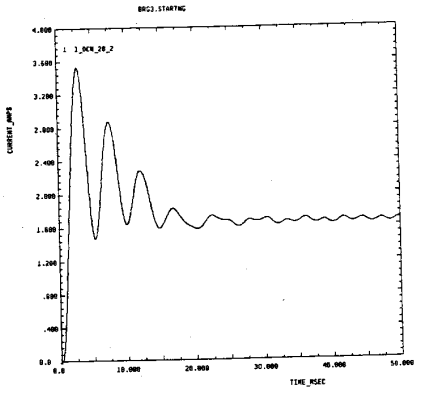

Figure 16. Load Current, Result from SOLVER-Q.

The program OPTLSE has been successfully used to study power flow and optimal power flow problems. Below is a simple example of how SOLVER-Q solves optimization problems.

This is an inequality constraint optimization problem:

```
cost:
x1 - x2^2 - x3;
constraints:
x1 - x2^2 + 2 * x2 * x3 - x3^2 + 3 >= 0;
x1 >= 0 ;
2 * x1 - x2 + x3 = 0;
```

The first two constraints are inequality ones. OPTLSE converts these inequality constraints into equality constraints by introducing an auxiliary non-negative slack variable "δ_i^2" in the constraints equations. The two inequality constraints become:

```
x1 - x2^2 + 2 * x2 * x3 - x3^2 + 3 - δ_1^2 = 0;
x1 - δ_2^2 = 0.
```

Then, based on the Lagrange multipliers method, the solution to this problem is found as follows:

```
X1 = -0.00000034;
X2 = -0.5;
X3 = -0.49999931;
δ_1 = 1.73205068;
δ_2 = -0.00000114;
μ_3 = -2.99999942;
μ_2 = -0.00000001;
μ_1 = 0.99999972;
```

where the μ's are the Lagrange multiplers introduced by OPTLSE.

The program OPTLSE can also solve regression problems. The following is an example of a regression problem:

```
5*b1+18*b2+100*b3+a=8;
7*b1+25*b2+700*b3+a=14;
3*b1+34*b2+700*b3+a=14;
5*b1+17*b2+400*b3+a=8;
6*b1+28*b2+700*b3+a=12;
2*b1+26*b2+700*b3+a=10;
4*b1+10*b2+400*b3+a=8;
7*b1*19*b2+700*b3+a=12;
5*b1+9*b2+400*b3+a=8;
4*b1+14*b2+400*b3+a=10;
```

These equations can be solved based on a least square errors criteria:

```
b1 = 0.25456616;
b2 = 0.15677285;
b3 = 0.00495281;
a = 3.41243595;
```

The program OPTLSE can automatically determine whether a problem is an optimization or a regression problem.

VI. CONCLUSIONS

A general purpose simulator based on SOLVER-Q is currently implemented in the APOLLO computer. The main weakness noted is slow speed. The compiled APOLLO LISP version of SOLVER-Q is still about 10 times slower than the EMTP. For example, it took about 8 minutes for SOLVER-Q to solve the problem in Figure 1 (34 equations) for 1000 time steps. The EMTP solves the same problem in one minute. To solve the problem in Figure 15 (67 equations) for 1000 time steps, it took about one hour. A considerable improvement is possible when

the inner loop computations (non-symbolic manipulations) are recoded in other programming languages such as PASCAL or FORTRAN; or SOLVER-Q is transferred into a LISP machine. A PASCAL version of SOLVER-Q running in an IBM PC environment has shown to be quite fast and effective.

The most striking advantage of the SOLVER-Q environment is the way in which SOLVER-Q solves simulation problems. With the aid of graphics interfaces, an engineer can fulfill all design phases: drawing of a design diagram, verifying and testing the design and then modifying the design, all in one integrated environment. The ideas leading to the development of SOLVER-Q coincide with the ideas that promote the research and development of so-called fifth generation computers. According to P. C. Treleaven [7], the fifth generation computers will support knowledge based expert system applications. Human interaction will involve natural language, speech and images. Programming will use very high level languages. Architectures will support concurrency, utilizing concepts such as data flow. Implementations will use the latest VLSI technology. SOLVER-Q uses a very natural language (mathematical expressions) as input, and it incorporates a knowledge based expert system. The use of LISP to implement SOLVER-Q enables the use of parallelism and data-driven architectures. SOLVER-Q is an initial attempt to have a computer solve simulation problems in a more intelligent manner. Extensions of SOLVER-Q are quite natural ways for simulation software technology to evolve.

A second advantage of SOLVER-Q is its direct modelling capability: it deals directly with equations. By using algebraic, differential and logical equations, many complex components of a system are modelled easily and accurately. The relationship between these components are also modelled by explicit connection equations. Combining these equations together, a system can be represented in a unified way. This direct modelling method is independent of program. The user can add new models for new components, or make modifications to old models, or replace the connection rules without changing the simulation program. This direct modelling method is powerful and flexible. It enables SOLVER-Q to be used as a truly general purpose simulator.

A third advantage of SOLVER-Q is its simplicity and flexibility. Simulation of changes in system structure, (such as, closing or opening switches, conduction or extinction of diodes, or flashover of arresters) can be readily represented. SOLVER-Q does not require any special care. It automatically updates the Jacobians as needed to reflect changes in system structures. SOLVER-Q's dynamic generation of variable names provides great flexibility in the implementation and symbolic formations of Jacobians and Lagrangians.

A final advantage of SOLVER-Q is its compactnesss. The central program SOLVE of SOLVER-Q contains only about 1300 lines of LISP code, including comments. The total size of all lisp programs in SOLVER-Q does not exceed 2500 lines because many functions are common to all these programs. The PC Pascal of SOLVER-Q consists of about 9000 lines of code (including a full screen editor).

VII. REFERENCES

1. H. H. Adelsberger, "Rule-based object-oriented Simulation Systems", AI Applied to Simulation, Simulation Series Vol. 18, No. 1, Feb. 1986, pp. 107-112.

2. F. L. Alvarado, R. H. Lasseter and Y. Liu, "An Integrated Engineering Simulation Environment", to be presented at this conference.

3. J. W. Blakemore, S. B. Dolins and P. R. Thrifts, "A General Purpose Robotic Vehicle Simulator", Intelligent Simulation Environments, Simulation Series Vol. 17, No. 1, Jan. 1986, pp. 151-161.

4. H. W. Dommel, "Digital Computer Solution of Electromagnetic Transients in Single and Multiphase Networks", IEEE Trans. on PAS, Vol. PAS-88, No. 4, April 1969, pp. 388-395.

5. W. S. Meyer, "Present Day Digital Computer Solution Procedures and Programs", IEEE TUTORIAL COURSE on Digital Simulation of Electrical Transient Phenomena, Chap. 2, 81 EH0173-5-PWR.

6. R. E. Shannon, "Intelligent Simulation Environments", AI Applied to Simulation, Simulation Series Vol. 18, No. 1, Feb. 1986, pp. 150-156.

7. P. C. Treleaven, "The New Generation of Computer Architecture", Proceedings of 10th Int. Symp. on Computer Architecture, June 1983, pp. 402-409.

8. P. H. Winston, "LISP, Second Edition", Addison-Wesley Publishing Company, Inc., 1984.

9. D. A. Zein, "Solution of a Set of Nonlinear Algebraic Equations for General Purpose CAD Programs", IEEE CIRCUITS AND DEVICES MAGAZINE, Sept. 1985, pp. 7-20.

Discussion

Andreas F. Neyer, Karl Imhof, and **Felix F. Wu** (University of California, Berkeley, CA): This paper is an important contribution because it is the first time that the object-oriented programming style is introduced for a power engineering application. It combines artificial intelligence techniques, data base systems, and computer graphics into a comprehensive simulation tool for electrical networks.

In the paper the object-oriented programming style is compared to the rule-based programming. The terms "rule-based" and "object-oriented" in the comparison seem to be used in a too narrow sense. In the paper the rule-based approach is explained with the function "differentiate." The rule-based implementation uses the "cond" clause which is the Lisp equivalent of the "IF-THEN-ELSE" statement in other languages, e.g., Pascal. The "cond" clause is a language statement that can be directly executed. It prescribes in which sequence the comparisons have to be performed (procedural programming). This means the differentiation rules are built into the code which accounts for the inflexibility of the approach. However, it is important to note that this approach is not the declarative approach commonly employed in rule-based expert systems. In a declarative representation the rules are data rather than code and describe how the problem has to be solved. Rules in an expert system cannot be directly executed, but have to be interpreted by an inference engine. There is no

"ELSE" statement that prescribes which rule has to be taken next. The selection of the rules for execution is the job of the inference engine. Hence the rules of an expert system are independent modules which can be easily modified.

In the object-oriented version of the function "differentiate" the authors make use of the capability of Lisp to assign properties to all interned symbols such as "+", "−", "/". For example, they assign the function of how to do the differentiation for addition as a property to the symbol "+". Therefore the implementation of "differentiate" (e.g., addition, division, etc.) has been effectively hidden. Information hiding is a concept that is applied in object-oriented languages and many other modern programming languages as well, e.g., Lisp. Lisp is not really considered an object-oriented programming language because it does not offer the mechanisms of classification and inheritance. In an object-oriented programming language the objects are organized into classes. Procedures are defined on a class and they are inherited by all the subclasses. Objects are responsible for the data modeling and all the processing. Instead of passing the data to the procedures as in conventional languages, the objects perform operations on themselves. The objects can be viewed as independent software modules that communicate with each other over a well-defined interface (information hiding).

We agree with the authors that the object-oriented approach is an ideal software tool for power system simulations. We believe that the full range of features in object-oriented programming languages allows to build more flexible and manageable software systems for many power engineering applications.

Fernando L. Alvarado and **Yenren Liu** (The University of Wisconsin, Madison, WI): We thank the discussers for their interest in this paper. This paper has described the use of certain AI-type tools for the specific purpose of improving upon programming tasks that have been previously done with traditional programming tools. In this sense, the paper is limited in scope and not intended as a broad paper on artificial intelligence. We agree that the terms "rule-based" and "object-oriented" are used only in a narrow sense.

Consider first the discussers' comments on "rule-based" programming techniques. The problem at hand did not warrant the full power available under a complete rule-based expert system with a separate inference engine. The rules that conduct symbolic differentiation are well defined. The implementation of these rules is straightforward. The rules for differentiation of mathematical operators and functions are applied directly without heuristics or anything but a trivial inference engine. For this reason, we have used the term "symbolic manipulation" instead of "expert system" in the paper. Rules have been coded as executable statements for expediency, but it is easy to move these rules from code to data because both code and data have the same format in Lisp. We agree that in a true rule-based expert system rules must be coded without the "ELSE" clause, and that it is the inference engine's role to decide on the sequencing of rules. We thank the discussers for helping clarify this point.

Symbolic simplification is more amenable to a true rule-based environment, but because we have elected to use only a small amount of symbolic simplification, these rules have also been coded in the paper as a simple "flat" rule base in which we expect direct application of individual rules to the task of expression simplification. If we want to implement a more sophisticated simplification mechanism or a symbolic integration environment, we must then have a true rule-based expert system which uses an inference engine and a heuristics rule base. Because of the manner in which we have proceeded, however, this becomes now an evolutionary change rather than a total rewrite of the code.

The term "object-oriented programming" refers to a programming style where the programmer defines an environment consisting of interacting objects, describes the function of every object in generic and specific terms. A key feature of the object-oriented environment is the hiding of the details of implementation. Another key aspect is inheritance of properties, so that new objects do not need complete descriptions but inherit many properties from more general classes of objects. We recognize that Lisp itself is not considered an object-oriented language. However, important extensions to Lisp have made it a true object-oriented language, while retaining all the advantages of the underlying language. We refer specifically to the Flavors environment of the Symbolics 3600 Common Lisp.

We agree fully with the discussers that these new programming tools will form the backbone for many new sophisticated software systems in power systems applications. We also recognize that we have used only a small portion of what AI has to offer. We would find it almost impossible to have performed these tasks with conventional languages and tools. We argue further that the evolution of software in engineering applications will have to continue to be driven by the specific requirements of important application areas such as the one presented here, more than by the glamour of the specific software environment itself.