Evaluating Algorithmic Properties of Dynamic Simulation Models by Operating Overloading

D. Juárez-Romero[1]*, J. M. Molina-Espinoza[2] , J. R. Zamora-Moctezuma[1], R. Leder[1]

[1]Centro de Investigación en Ingeniera y Ciencias Aplicadas
Universidad Autónoma del Estado de Morelos
Av. Universidad # 1001, Col. Chamilpa,Cuernavaca, Morelos, México 62210
Tel/FAX 777-32970.84,
[2]ITESM-CCM, Calle del Punte 222 Col. Ejidos de Huipulco Tlalpan, 14380, México D.F.,
e-mail djuarezr7@gmail.com, jose.molina@itesm.mx

Abstract. A framework is presented to obtain selective properties of computational models for dynamic simulation. This type of models contains conditional clauses, cycles and arithmetic functions contained in procedures. This framework considers this algorithmic characteristics of computational models, it analyses expressions according to its computational graph, with a similar principle to the techniques used for automatic differentiation of an algorithm. Since the results of the analysis depends on the conditional clauses selected (to limit the equations to specific working regions), a branch testing technique is used to cover all the feasible forms of model behavior.

Three types of model analysis tools are developed: a) to verify that global and local variables are properly assigned/referenced in a model, b) to obtain the dependence of variables in an equation and c) to detect potential arithmetic errors during execution. This framework is implemented with the C++ programming language. It applied to the analysis of computer models to represent the dynamics of power generation cycles.

Keywords: Computational Model, Automatic Model Analysis, Model Testing, Analysis by operator overloading.

1. Introduction

Numerical simulation has become an accepted mean to study the detailed behavior of an industrial plant, to design changes in other units or systems, and to resolve plant startup and operational problems. Our experience with the development of training simulators based in subsystems (like feed-water, main-steam, air-gas) [1], and the demand for new training simulators, allowed us to identify the need of producing simulation models which can be assembled to build a given configuration [2]. A proposal to meet this need was to build reusable computer models of operating units (that is independent pieces of equipment which execute a specific task, for instance, heating, pumping, splitting), whose external variables can be measured and validated.

M.A Cruz-Chávez, J.C Zavala Díaz(Eds):CICos2009, ISBN:978-607-00-1970-8, pp. 250 - 271, 2009.

To appropriately represent the behavior of these units, their models require: to perform in a wide range of working conditions (for instance, a drum boiler should work under filling, emptying, heating, boiling, pressurizing and steaming mode); to be connected under several configurations (forward-flow, reverse-flow, oscillating-flow); to be robust under abnormal situations (like pressure-loss, or pipe rupture). As a result, these models contain conditional clauses which limit the validity of an equation in specific working regions. Additionally, these models contain procedures to evaluate consistently physical properties of fluids consistently at given working conditions. Their mathematical aspect is presented in fig.1.1.

Global variables were needed to allow that all the models have the same invoking arguments to be ensembled and solved by a numerical solver, but compilers did not check some errors in their use. Local variables are used to evaluate intermediate variables used to avoid repeated calculations and enhance clarity of the code. Advanced compiler techniques analyze the behavior of local/static variables, but analyzing the behavior of global variables is more complicated. During the development of this type of computer models, frequent errors in the model's code were found:

- Global variables were assigned both by the model and by the solution method (this error is as dangerous as changing the control-index inside an iterative cycle),
- Local variables were referenced without a previous assigned value.

The following characteristics are useful to assess the quality of a model:

- A table of assignments and references of variables in an equation. This table is used to verify that every intermediate variable is referenced only after it has been assigned a value; otherwise, undefined behavior could result during execution. This table is also used to verify that state variables are not assigned, since they are expected to be modified *only* by the numerical solver.
- The value of inputs variables, that could generate arithmetic overflow, underflow or invalid arguments of arithmetic functions, to prevent the failure of the simulation, and to suggest ways to overcome these possible failure conditions.
- The dependence matrix that is used to detect which variables could cause any difficulty in the iteration matrix.

As we shall see ahead, some of these requirements of the model's code are not due to misuse of the computer language, but such characteristics are needed to ensure the quality of a model.

Section 2 presents a computer framework to analyze this type of models. In Section 3, three types of selective analyzers are presented using this framework: a) to evaluate assignment and reference of variables, b) to evaluate states and structure of the equations, and c) to evaluate measurements of reliability and efficiency; also some results obtained by applying this type of analysis to process units are presented. Finally, some conclusions are drawn from our experience with this framework.

1.1. Mathematical Model Description

The model equations representing the behavior of a process unit can be grouped in mathematical form as:

$$\mathbf{v} = a(\mathbf{p}, \mathbf{u}, \mathbf{x}, \mathbf{y}) \qquad \text{Intermediate equations} \qquad (1)$$

$$\mathbf{r_d} = A(\mathbf{y})\frac{dy(t)}{dt} - f(t, \mathbf{p}, \mathbf{u}, \mathbf{v}, \mathbf{x}, \mathbf{y}) \qquad \text{Differential equations}$$

$$\mathbf{r_a} = g(\mathbf{p}, \mathbf{u}, \mathbf{v}, \mathbf{x}, \mathbf{y}) \qquad \text{Algebraic equations}$$

Where the mathematical type of variables is:

t ≡ time. **p** ≡unit parameters, which do not vary with time. **u(t)** ≡ input variables **v(t)** ≡ intermediate variables, which are used to store the value of some computations.

x(t) ≡ algebraic state variables, which do not have accumulation, but change with time.

y(t) ≡ differential state variables.$\mathbf{r_a}$ ≡ algebraic residuals.,$\mathbf{r_d}$ ≡ differential residuals.

a, f, g, are real valued functions.,A(y) is a nonsingular real matrix.

1.1.1. Mathematical Model Solution

Given: t, **u**, **x*** (starting value), and $\mathbf{y^0}$ (initial value) , **v** intermediate variables are evaluated (eqn. 1.1), and the balance equations, expressed as residuals (eqns. 1.2,1.3) are solved iteratively, as a coupled set of differential-algebraic equations for, **x(t),** and **y(t).** The *procedural* form (explicit) of intermediate equations allows in mediate calculation, while the *declarative* form (implicit) of the balance equations allows flexibility in its working conditions.

1.1.2. Naming of Procedures of a Model

Every process unit is composed by a set of procedures with predefined names:

`Scale()`	Defines geometry and capacity parameters, **p**
`Starts(Time)`	Sets initial values, **x***, and starting conditions, $\mathbf{y^0}$.
`Behave(Time, EeR[])`	Evaluates the intermediate variables, constraints, and evaluates the vector of residuals $\mathbf{r_a}$, $\mathbf{r_d}$ as final variables ata given time.
`Operate (Time)`	Modifies manipulated (external) variables, **u.**
`Signal(Time)`	Evaluate measured variables, **z**

Functions naming delimits which part of the source code to analyze. The analysis is carried only in the **`Behave(double t, double EeR[])`** procedure and the

functions called by it. Other procedures to evaluate physical properties can also be coded by the user

1.1.3. Naming of Variables of a Model

The names of variables are used not only to clarify the purpose of the variable but also to identify the type of a variable required in the algorithmic analysis. The variables allowed are of basic type (integer, and floating point) or arrays. States, externals, inputs and signal variables are communicated in the model as global variables. Global variables names are composed by 8 characters (Variables which are only used in the model are local, their names do not have the first four characters. Example:

Liquid temperature of a drum coming from port 01.

Dm——————The Unit name, (i.e. Dm Drum).

01 ————— The port through which this variable is connected: two integers

Tm ——— The general property, (i.e. Tm Temperature)

L——— The specific condition of the property,(i.e. L liquid)

A—— The mathematical type: A *Algebraic state variables* D *Derivative* (cf fig 1.1)

2. Model Analysis Using the Computational Graph

Automatic model analysis can be carried out by the computational graph of the model. Iri [3] defined the *computational graph* as an acyclic graph of which a node corresponds to a basic operation, an input variable or a constant, and of which an arc corresponds to an operation. A compiler usually translates the sequence of arithmetic operations and procedure-calls of elementary functions into an internal representation, which is approximated by a composition of elementary arithmetic operations (monadic or dyadic), and possibly some functions (in one or two variables) that contains the selected branches and the loop unrolled as necessary. Then, the value of each entry depends only on the input variables or previously computed values. Given a set of input values for independent variables, intermediate variables are computed by elementary arithmetic operation or intrinsic functions, following the computational graph, to arrive to the final values.

Since the computational graph captures the precise sequence of operations and arguments implemented by a particular algorithm or function, it is possible to use this graph for a given analysis. Then for a set of specified values of all the variables in a model, the analysis $\Phi(x_l, x_m)$ of the variable x_m with respect to x_l is evaluated as:

$$\Phi(x_l, x_m) = \sum_{p \in P(x_l, x_m)} \bigcup_{i,j,k \in p} \Phi(x_i, x_k) \circ \Phi(x_k, x_j) \qquad (2)$$

Where, P is the set of feasible paths going from vertex x_l to vertex x_m, p is one of these paths, and i, j, k are nodes in this path. The elementary analysis of unary or

binary operations, and the concatenation (chaining) of operations is denoted by $\circ$. These operations are evaluated by overloading these operators. The summation is executed following the computational graph which accumulates the properties of the analysis.

To illustrate how the results of model analysis are propagated along the computational graph, two examples are presented; the first one shows the analysis of the computational graph

Example 2.1. *Computational graph.* Consider the equations for an induction engine [4]:

$$S = \frac{K_0 - \omega}{K_0} = 1 - \frac{\omega}{K_0} \tag{3}$$

$$\tau_E = \frac{V^2}{K_3 / S + K_2 * S + K_1}$$

$$\text{if } S > K_4 \text{ then } \tau_E = \tau_E * (K_5 + K_6 * S) \ .$$

$$K_7 \frac{d\omega}{dt} = \tau_E - \tau_M$$

$K_0, ..., K_7$ are parameters. S = Slip (which measures the distance from nominal speed, K_0), τ_E, τ_M =electrical and mechanical torque, V = voltage. If $S \le K_4$ the evaluation of the slip is given by the following intermediate variables:

$$\begin{aligned} v_1 &= V^2 \\ v_2 &= K_3/S \\ v_3 &= K_2 * S \\ v_4 &= v_2 + v_3 \\ v_5 &= v_4 + K_1 \\ \tau_E &= v_1/v_5 \end{aligned} \tag{4}$$

Observations regarding this evaluation:

- If $(S > K_4)$, the value of τ_E is "corrected", but the associated formulation is obscure, since the same identifier is reevaluated.
- For a given values of the parameters the derivative $\frac{d\omega}{dt}$ depends essentially on ω, V, τ_M

- If is expected that $S \approx 0$ is better to code eqn. (2.1) as $\tau_E = \dfrac{V^2 S}{K_3 + (K_2 * S + K_1) * S}$, which is more stable.

Figure (2.1) describes the evaluation graph of eqn. (2.1). The dotted line shows that exist two paths from S to τ_E. If $S > K_4$ then exists three paths from S to τ_E.

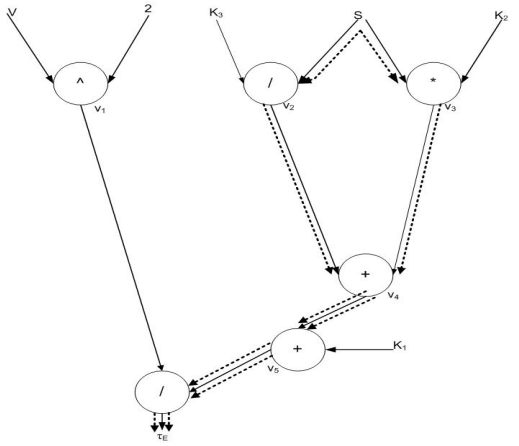

Fig. 1. Evaluation graph od eqn 3.

Example 2.2 *Operator overloading.* This example shows how operator overloading can be used in the analysis of a computer model.

Consider the expressions described in figure 2, which describes the variation of area A, passing through a non return flow valve `dAdt`, according to the pressure drop, ΔP. In this figure: `auxm = (ΔP - ΔPmin)*A; auxs = auxm*k0; auxe = auxs – da/dt.`

```
/* evaluation of differential residual for valves area */
  if (ΔP < ΔPMin) {
   VcR[0] = - dAdt + KC*ΔP - ΔPMin)*A;      /* closing */
  } else {
   VcR[0] = - dAdt + KO*(ΔP - ΔPMin)*(1 - A);/* opening */
  }
```

Code 2.2 Expression which describe the opening of a non return valve

These instructions are evaluated simultaneously according to the standard arithmetic, and with the *selective algebra* for the analysis. The computational graph of VcR[0] is presented in fig 2, for the standard and the selective algebra. When ΔP < ΔPMin, the area of flow decreases according with `VcR[0] = - dA/dt + KC * (ΔP - ΔPMin)*A.` In this figure it is depicted how every operator is used to increment the number of references (e.g. abbreviated as A.r++), and the number of assignments (R[0].a++) in the '=' operator.

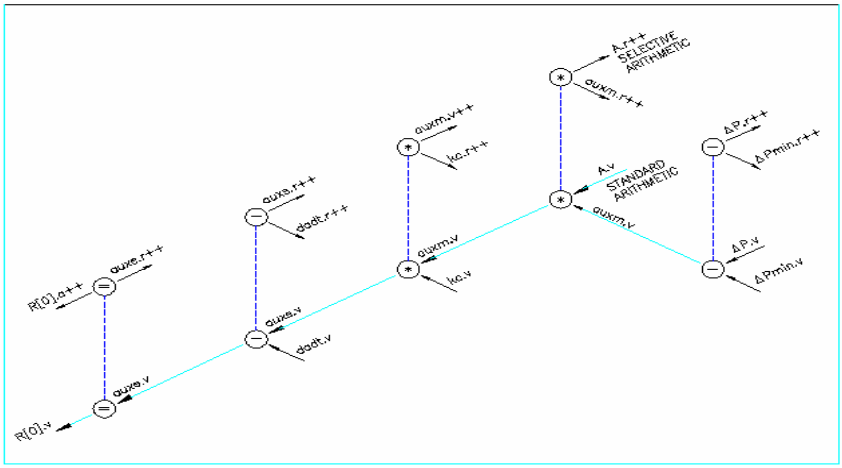

Fig. 2. Variation of area A, passing through a non return flow valve `dAdt`

2.1. Implementation of Automatic Model Analysis

To analyze a model, the computational graph is followed, this graph contains the sequence of instructions taken by the compiler to evaluate model outputs. The implementation was carried out by operator overloading. *Operator overloading* allows to change the arithmetic operators such as: +, *, and arithmetic functions like: `sin(), cos(),` into operations and functions for the selective analysis [5,6].

Thus, to analyze a model by operator overloading, it is necessary to:

- Specify the ***selective algebra*** for the analysis.
- Design a ***set of classes*** supporting this algebra of unary and binary operations through the overloaded arithmetic operators and elementary functions, and the relationship with other classes.
- Specify the ***expected behavior*** for global and local variables, when the execution is completed.

The composed analysis is evaluated automatically calling these functions in the order determined by the computational graph. Then the diagnoses of the analyzed terms required to compute the final result are assembled. To relate diagnoses with the variables, a list of variables of the different type of variables were build by a lexical analyzer.

This type of analysis produces values of properties of what is actually computed. If we want to cover all the feasible forms of behavior, a *branch testing* is needed. The elements of *branch testing* are [7]:

- *A **branch analyzer** to produce a branch condition.* The predicates of every conditional clause in a model are replaced by a logical flag. Every time the model is called with a set of flags, which produces a branch condition for every branch.
- *A **tester** to run the program against a sequence of test cases.* The model tester calls the model several times, every time with different set of test inputs. In every set of test inputs, it is expected that the model presents a different form of behavior.
- ***Probes** to analyze the condition of a branch when it is traversed.* Probes were implemented as a set of classes supporting the algebra for the analysis through the overloaded arithmetic operators and elementary functions.
- ***A report generator** to produce a table of local and global variables,* with their properties evaluated during the analysis at different branch conditions.
- ***An oracle**, to determine correctness of the program's output for some input.* The evaluated model properties are compared with the asserted properties for every variable by decision tables. Unexpected behavior are reported as diagnoses.

Since computational models are designed to run on a finite machine over finite input sets, it is possible to prove the correctness of any program by testing it over its whole input domain. Also, since every model represents only a process unit with only physical streams as external variables (with a discrete or continuous domain), the number of evaluations is modest.

Once the model has been successfully produced, it runs for simulation with basic variables of the C language, without the burden of operator overloading, so it can be efficiently executed.

2.2. Step by Step Model Testing

To analyze a model using the computational graph, by operator overloading, the following stages should be carried out by the user (see fig 2.3):

1) Codify the model with the name of identifiers as described in fig 1.2. Variable naming is used to determine the type of variable, its purpose and how to analyze it.

2) Apply Model Analysis with the model-name as argument.

The following tasks are executed by a batch file:

2.1 *Model Instrumentation*: Performs a lexical analysis on the source program to produce tokens, parsing the tokens to translate declarations of types into declaration of types required for the analysis (active or non-active variables). Thus an internal form of the model, compatible with ANSI C, is generated. In this internal form:

- Local and global declarations of types are translated in declarations of types (active or non-active variables) required for the analysis according with variable naming,
- Variables of procedure calls are translated in compatible abstract types.
- Branch predicates are substituted by logical flag which triggers the branching
- Sequential statements are tagged.
- An executable program is built to test the model.

This parser also a) builds a table of local and global variables, and b) constructs a *model tester*

2.2 *Model binding* with its *model tester* and their classes (which define the selective algebra) to produce an executable.

2.3Model Execution. The model is invoked several times with a different set of test inputs. A report is produced with the model output. This report presents a table of the local and global variables with their properties evaluated during the analysis at different branches.

2.4 *Model diagnosing*. Comparing the model properties obtained during the execution with the asserted behavior for every variable, a linked list is used to store and to display the diagnoses.

3. Correct the Model. The diagnoses (faults and warnings) generated by the analyzer serve the model-developer to correct the code. A model is considered acceptable if no severe errors are detected.

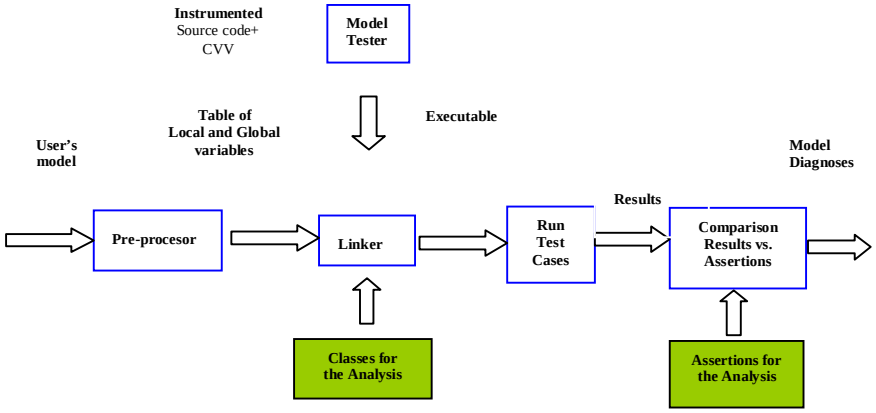

Figure 3 Sequences in the Analysis

2.4 Supporting tools

The classes for the state & structure analyzer were designed with Rational Rose [8. The code analyzer Codewizard [9]was used to review the implementation of classes. To build the lexical analyzers for model instrumentation we used the lexical analyzer Pclex [10].

The next section details the characteristics of three types of analyzers to evaluate properties of models of process units.

3. Characteristics of Model Analyzers

This section details the characteristics of analyzers developed.

3.1. Assignment & Reference Properties

The purpose of the analysis of Assignment and Reference is to detect the inappropriate use of variables. Reassignment of a variable obscures the code, which sometimes is due to careless code re-editing. Lauesen and Younessi [11] mention the following properties for visible quality in software: A variable should be assigned before it is used. A variable should have only one purpose in a procedure. A variable that is referenced (i.e. that appears in the RHS of an equation) without an assigned value can produce an undefined model behavior. Gani and Toneva [12] indicate that explicitly computed variables require an adequate precedence ordering to avoid referencing a variable without an assigned value.

Attributes of a variable that is a class of this type: Its value, Number of assignments (`Assg`), Number of references (`Ref`), List of line number(s) where a value was assigned.

Algebra for Assignment & Reference. A global variable has an initial value at the beginning of the execution. A local variable does not have an initial value. For this analyzer we define the algebra displayed in Example 3.1.

Table 1. Assumed Algebra for Arithmetic for assignment analysis

Model's code	**Evaluation of assignment & reference**	
	Assignment	**Reference**
	Start up of global variables	
`double AGlo, BGlo, CGlo, Dglo;`	`AGlo.Assg = 0`	`AGlo.Ref = 0`
`double EGlo[2], FGlo[2];`		
	`BGlo.Assg = 0`	`BGlo.Ref = 0`
	`CGlo.Assg = 0`	`CGlo.Ref = 0`
	`DGlo.Assg = 0`	`DGlo.Ref = 0`
	`EGlo[0].Assg = 0`	`EGlo[0].Ref=0`
	`EGlo[1].Assg = 0`	`EGlo[1].Ref=0`
	`FGlo[0].Assg = 0`	`FGlo[0].Ref=0`
	`FGlo[1].Assg = 0`	`FGlo[1].Ref=0`
`int Behave(double tiem, double EeR[2])`	**When procedure is Behave is invoked**	
`{`		
`double gLoc, hLoc;`	`gLoc.Assg = 0`	`gLoc.Ref= 0`
	`hLoc.Assg = 0`	`hLoc.Ref = 0`
	LHS	**RHS**
`CGlo = AGlo + BGlo;`	`CGlo.Assg++`	`AGlo.Ref++`
		`BGlo.Ref++`
`gLoc = 2.*BGlo;`	`gLoc.Assg++`	`BGlo.Ref++`
`if(Bglo > 10){`		`BGlo.Ref++`
`DGlo = gLoc;`	`DGlo.Assg++`	`gLoc.Ref++`

```
    } else {
      DGlo = hLoc;      DGlo.Assg++          hLoc.Ref++
    }
    for(I = 0; i>2;
i++){
      EGlo[i]      =    EGlo[i].Assg++       FGlo[i].Ref++
FGlo[i]*BGlo;
    }                                       BGlo.Ref++
    EeR[0]         =    EeR[0].Assg++        DGlo.Ref++
pow(DGlo, gLoc)                             gLoc.Ref++
    return(0);
  }
```

We observe in example 3.1 that since local variable `hLoc` does not have an initial assigned value, it could produce an assignment error when it is referenced in the second branch of the conditional expression, and propagate the error to the evaluation of EeR[0].

Assertions: Local Variables. A local variable should have an assigned value before it is referenced. A local variable can be recomputed only after it has been referenced at least once. The vector of residuals should be assigned exactly once. The other global variables that belong to a model should not be assigned, but they should be referenced in the model at least once.

Input Tests: A set of logical flags that trigger the conditional branches.

Use of Computational Resources. For *m* conditional clauses computation requirements grow as 2^m function evaluations, since the clauses are considered as independent.

Results: List of assignments and references of local and global variables at a given line number.

3.2. State & Structure Properties

The purpose of the analysis of states and structure is to obtain the dependence matrix. This matrix has the residuals in the rows and the independent variables in the columns. A nonzero element appears in position `[i, j]` of this matrix if a residual `i` depends on variable `j`. This matrix is useful to isolate errors, and to evaluate economically partial derivatives numerically [13]. It is also useful to detect sets of equations whose variables do not interact, thus they can be executed independently for parallel processing [14]. Since Shannon proved that the minimum set is limited by the maximum number of variables in an equation [15], the saving of computing effort in the numerical evaluation of partial derivatives is substantial in large systems of equations. The structural analysis displays the global variables used in an expression.

Attributes of a variable that is a class of this type: Its value, an integer value to identify the variable**,** a list of identifiers of global variables used to evaluate it (`Dep`).

Algebra for Structure. The preprocessor assigns a tag to every variable to identify it. For this analyzer, the algebra shown in example 3.2 is assumed:

Table 2. Assumed Algebra for structural analysis

```
double AGlo, BGlo, CGlo, DGlo;
double EGlo[2], FGlo[2];




  int   Behave(double   tiem,
double EeR[2])
{
    double gLoc, hLoc;
    CGlo = AGlo + BGlo;
    gLoc = 2.*BGlo;
    if(Bglo > 10){
      DGlo = gLoc;
    } else {
      DGlo = hLoc;
    }
    for(I = 0; i<2; i++){
      EGlo[i] = FGlo[i]*BGlo;

    }
    EeR[0] = pow(DGlo, gLoc)

    return(0);
  }
```

Results of State & Structure Start up

```
  AGlo.Dep = {},…,etc.
  EGlo[1].Dep = { }
  EGlo[2].Dep = { },…,etc.

  When procedure Behave is
invoked

  Dependence
  CGlo.Dep = {AGlo, BGlo}
  gLoc.Dep = {BGlo}

  DGlo.Dep = {BGlo}

  DGlo.Dep = {}


  EGlo[i].Dep  =  {FGlo[i],
BGlo}

  EeR[0].Dep    =    {DGlo,
BGlo}
```

When an intermediate variable is referenced, instead of using the dependence of intermediate variable itself, the dependence of this intermediate variable on the global variables is substituted. In the conditional clauses, the union of dependence of both conditional branches was assumed to detect all possible dependencies.

Assertion: A state variable that belongs to the analyzed model should appear at least in one residual equation.

Input Tests: A set of logical flags that trigger the conditional branches.

Use of Computational Resources: The effort required is similar to the Analysis of Assignment & Reference.

Results: The dependence matrix with respect to any of the mathematical types (fig 1.1).

For a single unit, the dependence matrix is rectangular. When enough variables are specified, the matrix becomes square.

3.3. Reliability & Efficiency

The purpose of the analysis of reliability and efficiency is to detect domain values that causes under/overflow or an undefined value of arithmetic operations in a model. Code analysis is relevant since two evaluating procedures representing a mathematical function may have widely varying stability and efficiency. The combined algebraic-differential equations (eqns. 1.2, 1.3) have more degrees of freedom to represent an expression, than in ordinary-differential equations since the algebraic-differential terms need only to equate to zero [16]. In real time simulation, it is also essential to detect the largest computing effort required in a model.

Attributes of a variable that is a class of this type: Its value, the number sums-subtractions (`Sum`) multiplications-divisions (`Mul`), the number of square roots, the number of exponential and trigonometric functions (`Art`): {`sqrt(), exp( ),log( ),sin( ), cos( ),tan( )`,.., etc} required to calculate its value. The number of faults in divisions, square roots, and trigonometric expressions.

Algebra for reliability and efficiency:

- The result of a detected dangerous operand is forward propagated from the intermediate variables to the final variables, the vector of residuals.
- The operations required to evaluate intermediate variables are counted only once.
- The analysis reports only the operation count in variables, which are used to evaluate the final variables (i.e. the algebraic or differential residuals); thus, a variable, which is only used as a predicate of a conditional clause, is not taken into account. This is shown in example 3.3.

Table 3. Algebra for analysis of Efficiency

`double AGlo, BGlo, CGlo, DGlo;`
`double EGlo[2], FGlo[2];`
	Results of State & Structure
	Start up
	
	`AGlo.Sum=0; AGlo.Mul=0; AGlo.Art=0;`
	`EGlo[1].Sum=0; EGlo[1].Mul=0 EGlo[1].Art=0`
	`EGlo[2].Sum=0,…,etc.`

`int Behave(double tiem, double EeR[2])` `{`	**When procedure Behave is invoked**

`double gLoc, hLoc;`
`CGlo = AGlo + BGlo;`
`gLoc = 2.*BGlo;`
`if(Bglo > 10){`
	Operation Count
	
	`CGlo.Sum=AGlo.Sum+BGlo.Sum+1;`
	`CGlo.Mul=AGlo.Mul+BGlo.Mul;`
	`gLoc.Sum=BGlo.Sum;`
	
	`gLoc.Mul=BGlo.Mul+1;`

```
      DGlo = gLoc;  |  DGlo.Sum=gLoc.Sum;
                    |DGlo.Mul=gLoc.Mul;

    } else {
      DGlo = hLoc;  |  DGlo.Sum=hLoc.Sum;
                    |DGlo.Mul=hLoc.Mul
    }
    for(I   =   0;
i<2; i++){
      EGlo[i]     =  |  EGlo[i].Sum=FGlo[i].Sum+BGlo.Sum;
FGlo[i]*BGlo;
    }               |  EGlo[i].Mul=FGlo[i].Mul+BGlo.Mul+1;
    EeR[0]        =  |  ErR[0].Sum = DGlo.Sum+gLoc.Sum
pow(DGlo, gLoc)
    return(0);      |  ErR[0].Sum = DGlo.Mul+gLoc.Mul

  }                 |  ErR[0].Art = DGlo.Art+gLoc.Art+1;
```

Table 4 shows the arithmetic operators and functions whose operands are verified to be within its valid domain. If a potential error is detected due to a *unsafe* arithmetic, the analyzer issues a diagnostic flag, and execution continues using a *safe* arithmetic.

Table 4. arithmetic operators and functions verified

Arithmetic Operations and functions	**Condition of risk**	**Safe arithmetic**
1/x	$\|x\| < \tau$	$1/(x + \varepsilon*sign(x))$
tan(x)	$x < \tau$	$\tan(x + \varepsilon*sign(x))$
√x	$x < 0$	$\sqrt{\|x\|}$
$\mathbf{x^y}$	$x < 0$	$\|x\|^y$
log(x)	$x < 0$	$\log(\|x\|)$
exp(x)	$x > x_{max}$	$\exp(x_{max})$

τ is a tolerance

ε is a related with machine precision

Copy-constructors were required to transfer the arithmetic statistics/diagnoses to another variable during its declaration.

Assertion: No diagnostics should appear in the arithmetic operations.

Input Tests: The external real variables.

Use of Computational Resources: This analysis only considers the paths (combination of branches covered) produced by these inputs. The cost depends on how many test cases per input variable are required to cover the domain.

For exhaustive testing of the continuous domain in a reduced number of tests, this analysis requires the careful design of test cases. Offutt, Jin and Pan [17] have developed a program for reducing testing input domain. Given a set of input variables

with domain u_{min}- u_{max}, symbolic analysis obtains the predicate of conditional clauses in terms of the input variables. If the predicate has linear dependency on the input variables, analyzing all the independent or nested conditional clauses, which can traverse all the arcs, can reduce the input domain.

4. Results

For every test case is produced: a list of operation counts required to evaluate the vector of residuals, and a list of diagnostics in the evaluation of these variables. With this information, the model-developer can select alternative forms of coding: factoring common sub-expressions, or using alternative efficient arithmetic (e.g., $\sqrt{x}$ instead of $x^{0.5}$). If any diagnostic appears, it is necessary to bound the value of dependent variables to a given domain, to regroup terms, or to use an equivalent arithmetic of operators to reduce the possibilities of an arithmetic exception.

We have found convenient first to follow this steps 1)to analyze the models for adequate Assignment & Reference; then to correct the code as necessary, 2) to analyze the States & Structures (which will not be upset by unassigned variables), and finally 3)to analyze Reliability & Efficiency, with the available relation of equations and variables (to define the test inputs).

4.1. Results of Model Analysis

The analysis was applied to typical models required in energy cycles. The characteristics of these models are displayed in Table 5. The first three models (**Sp, Bb, Ww**) were adapted from a power plant simulator based in subsystems (like steam-generation, water-feeding). The last two models were built using modeling guidelines, proposed in this work, and tested during its development with these analyzers. These models were written in the C programming language.

This section present the results when they analyzers are applied to the modes of Table 5.

Sp: Gas-Steam Superheater
Bb: Electrical Pump
Ww: WaterWalls
Mi: Induction Engine
Dm :Drum with Single and double phase

These results obtained were validated by code inspection.

Table 5. Characteristics of Model Analyzed

Process Unit Name filename	Num. Para-meters P	Num. States x(t), y(t)	Num. externals u(t)	Num. Residual Equations R	Max. Nesting level of conditional Branches	Num. Proce-dures
Sp	1	15, 4 streams	2	5	0	8
Bb	36	8, 3 streams	2	4	1	8
Ww	1	8, 3 streams	5	4	0	8
Mi	22	2, 4 streams	2 1 logical	2	2	8
Dm	5	8, 4 streams	9	8	1	13 (5 for physical properties)

.4.2. Results of The Analysis of Assignment & Reference Properties

The results of this analysis are summarized in table 6.

Table 6. Assignment & Reference Properties of Model

	Local Variables		Global Variables	
Model	**A declaration not referenced**	**An assignment not referenced**	**No Referenced**	**Invalid assignment**
Sp	1	--	2	4
Bb	16	2	--	--
Ww	--	--	--	--
Mi	--	--	--	--
Dm	12	573	--	--

Sp Model**.** The analyzer detected 2 global variables without reference, it also detected that for some conditional branches there are 4 invalid assignments. As more

detailed knowledge was required to correct the first error, the analysis of this model was stopped at this analysis.

Bb Model**.** The analyzer diagnosed 2 unnecessarily declaration and assignments, these are minor faults.

Ww and **Mi** Models did not present diagnoses.

Model **Dm**. The analyzer diagnosed unnecessarily declarations and assignments. The assigned variables are elements of arrays of coefficients used to evaluate physical properties through polynomial approximation.

4.3. Results of The Analysis of State & Structure Properties

Here, the dependence matrices with respect to algebraic **x,** and differential states **y,** are shown in tables 7-8 Nonzero elements are represented by ⊗.

Table 7. Dependence matrices for models for mode Bb

Res	**x**	**x**	**x**	**x**	**X**	**y**
	x[0]	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[0]**
BbR[0]				⊗	⊗		
BbR[1]		⊗			⊗		
BbR[2]			⊗			⊗	
BbR[3]							⊗

Bb Model has a variable x[0] that does not appear in the residual equations. This detection avoids a structural singularity during its numerical solution.

Table 8. Dependence matrices for models for mode Ww

Res	**x[0]**	**x[1]**	**x[2]**	**x[3]**	**x[4]**	**x[5]**	**x[6]**	**u[0]**	**u[1]**	**u[2]**	**u[3]**	**u[4]**	**y[0]**	**y[1]**
WwR[0]				⊗			⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗
WwR[1]	⊗			⊗		⊗	⊗						⊗	
WwR[2]		⊗			⊗	⊗								
WwR[3]			⊗			⊗								

Table 9. Dependence matrices for models for mode Mi

Res	**x[0]**	**u[0]**	**u[1]**	**u[2]**	**y[0]**
MiR[0]	⊗	⊗	⊗		⊗
MiR[1]	⊗			⊗	

`Ww, Mi` models do not present any diagnoses, but the structure obtained automatically is useful in the solution of the model.

Table 10. Dependence matrices for models for mode Dm

Results	x[0]	x[1]	x[2]	x[3]	u[0]	u[2]	u[3]	u[4]	u[6]	u[9]	y[0]	y[1]	y[2]	y[3]
Dm R[0]				⊗		⊗			⊗	⊗				
Dm R[1]	⊗	⊗		⊗	⊗	⊗		⊗	⊗	⊗	⊗	⊗	⊗	⊗
Dm R[2]	⊗	⊗		⊗	⊗	⊗		⊗	⊗	⊗	⊗	⊗	⊗	⊗
Dm R[3]	⊗													⊗
Dm R[4]											⊗	⊗	⊗	⊗
Dm R[5]	⊗	⊗												
Dm R[6]				⊗			⊗							⊗
Dm R[7]			⊗											⊗

In **Dm** Model although external variables `u[1], u[5], u[7] and u[8]` are not used to evaluate the residuals, its model structure is adequate, (unlike **Bb** model), since these external variables could be used elsewhere.

4.4. Results of The Analysis of Reliability & Efficiency

This analyzer checks if a model can handle the variables in the input streams. Some test inputs caused models to incompletely finish computations. Results of the analysis at different test cases are shown in figures 3-6. It counts the number of every type of operation according with the procedures called and the branches taken. **Note:** Only two different test cases are shown per residual, they are presented in different color.

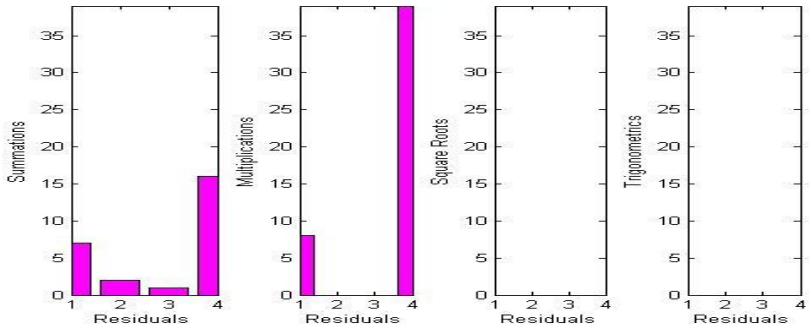

Fig. 3. Number of Arithmetic Operations for Bb model.

For some runs, **Bb** model does not compute any residuals, since a flag of invalid domain was detected internally, then the model returned without completing calculations, thus protecting the model.

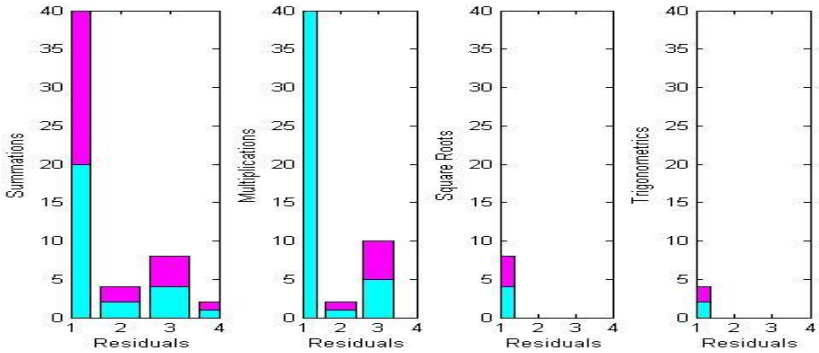

Fig. 4. Number of Arithmetic Operations for Ww model at two different conditions

Model **Ww** presents four divisions by zero, these divisions are due an arithmetic term which evaluates the inverse of the sum of fractions with small denominators: $(1/x1 + 1/x2)^{-1}$. To overcome divisions-by-zero this term can be refactorized as x1* x2/(x1 + x2). In one test case shown, the evaluation of the first residual dominates the whole computing effort.

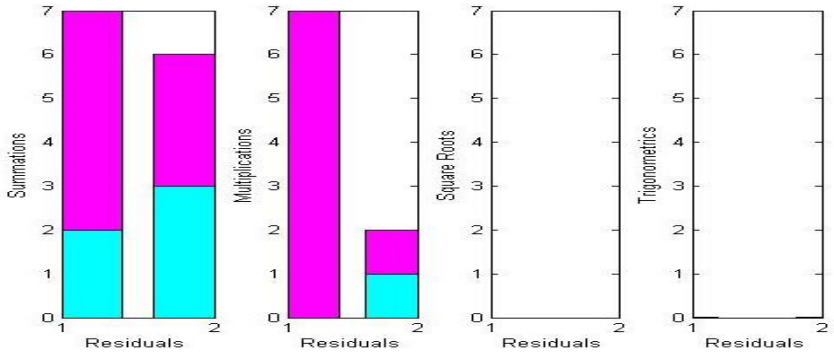

Fig. 5. Number of Arithmetic Operations for Mi model at two different conditions

The computations in this model of an induction engine vary according to its operating region: linear, or non linear. In the linear region only requires summations and multiplications to evaluate the residuals.

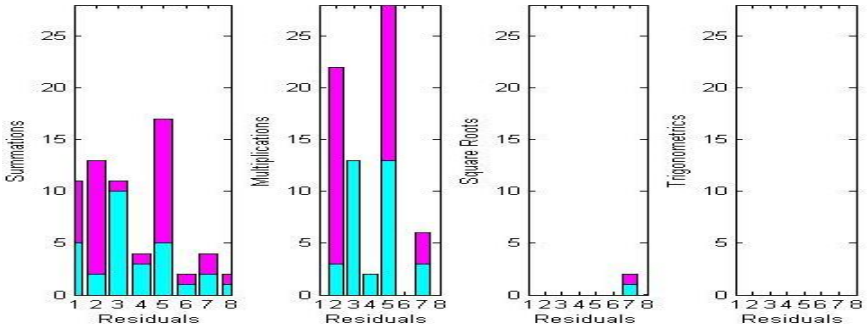

Fig. 6. Number of Arithmetic Operations for Dm model at two different conditions

In **Dm** Model, the computations in this model vary according to its operating region: in the single phase region (liquid), the model evaluates only summations and multiplications; in the saturated region, the model evaluates properties in both liquid and steam, which require a term with square roots.

5. Conclusions

A computer framework for evaluating specific algorithmic properties of computer models for dynamic simulation was presented. This framework is based on the definition of the algebra for the analysis, the implementation of a set of classes supporting this selective algebra by operator overloading, and the specification of the asserted behavior for the analysis. This framework was applied to the design and implementation of software tools: to evaluate Assignment & Reference, States & Structure, and Reliability & Efficiency of simulation models. Every analyzer diagnosed specific characteristics of the tested models. With these diagnoses, the models can be corrected to improve their quality, thus becoming correct, efficient and resilient for steady state or dynamic tests.

5.1. Limitations of this Type of Analysis

- No static variables are allowed in the model, since static variables store the previously computed values of a variable.

- Concatenated loops (disjoint), nested loops (one inside another) can be analyzed, but not knotted loops (branch in/out in the middle of a loop). But this type of loops is avoided with the use of structured programming.
- Macros are not permitted in the model source code, since they obstruct model instrumentation.

5.2. Further Work

More attention is required to analyze models with cycles, thus increasing the range of application of the available model analyzer.

6. Acknowledgements

The Automatic Differentiation Code ADOL-C developed by A. Grienwank *et al* [18] enlightened us about analysis of algorithms. We acknowledge the technical assistance given by P. Conley on the use of the Abraxas lexical processor. B Sandoval created a static analyzer to obtain the assignment and references in a model, which showed us the level of complexity involved with a purely lexical based analysis. Y. Mendoza, G. Ruiz, G. Porras, and J. A. González tested the performance of the analyzers developed.

7. References

1. Gonzalez S, Méndez E., Kuhlman F., Castelazo I.:, "Modularization Guidelines in the Development of Large-Scale System Models for Simulation" , IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-15, No 5, Sep/Oct pp 665-669. (1985)
2. Barrero L., Canales R., Juárez-Romero D., Mendoza Y., Ruiz G. and Santoyo-Gutiérrez S.: "Systematic Model Building-Testing of Simulation Models for Training Simulator", Computers and Chem. Eng. v 27 pp 1421-1430, (2003).
3. Iri M..History of Automatic Differentiation and Rounding Error Esimation", in Griewank A. & Corliss F.: Automatic Differentiation of Algorithms theory, Implementation and applications" SIAM (1992)
4. Guru B.S. And Hiziroglu H.R.:Electric Machinery and Transformers, Oxford Series(2000)
5. Griewank A. "Evaluating Derivatives Principles and Techniques of Algorithmic Differentiation", SIAM. (2000)
6. Bücker H.M. "Special section: Automatic differentiation and its applications" Future Generation Computer Systems, v 21 pp 1322-1323 (2005)
7. Howden W. E.Functional Program Testing and Analysis Mc Graw Hill (1987)
8. Rational Rose, IBM http:// www.ibm.com/software/rational/
9. Codewizard, Parasoft http://www.parasoft.com/jsp/products.jsp
10. PcLex, Abraxas: http://www.abxsoft.com/
11. Lauesen S., Younessi H "Is Software Quality visible in the Code", IEEE Software Jul pp 69-73. (1994)
12. Gani R. and Toneva R.: Simultaneous Steady State and Dynamic Simulation of Chemical Processes,Computers and Chemical Engineering, 13 563-570, (1989)

13.Curtis A. R., Powell M. J. D. and Reid J. K. "On the Estimation of Sparse Jacobian Matrices" J Inst Math Appl. v 13 pp 117-119. (1974)
14 Juárez-Romero D., and Pantelides C. C., Multiprocessor Solution of Nonlinear Equations for Chemical Process Simulation, Transputer Mailshot pp 43 - 52, Sep(1990)
15 Sloane N. J. A., Dwyner A., Claude Elwood Shannon Collected Papers, IEEE Press pp 584-587 (1993)
16 Campbell S. L., Hollenbeck R. "Automatic Differentiation and Implicit Differential Equations" in M. Berz, Bishof C., Corliss G., Griewank A: Computational Differentiation Techniques, Applications and Tools, SIAM pp 215-227. (1996)
17 Offut A., Jin Z , Pan J.. "The Dynamic Domain Reduction Procedure for Testing Data Generation", Software Practice and Experience v 29 (2) 167-193 . (1999)
18 ADOL-C,. http://www.math.tu-dresden.de/wir/project/adolc/ Sep. (2005)