

Graph-Based Methods for Testing Computer Programs

A. Estrada ,J. M. Molina-Espinoza y D. Juárez-Romero *

Facultad de Ciencias Químicas e Ingeniería Centro de Investigación en Ingeniería y Ciencias Aplicadas
Universidad Autónoma del Estado de Morelos, Av. Universidad # 1001, Col. Chamilpa
Tel/Fax [52] (73) 297084, Cuernavaca 62210,
{aestrada,djuarez}@buzon.uaem.mx

Abstract. This work emphasizes testing methods useful to assess the quality of computer programs. In particular presents unit testing based on the computational graph. Once the algebra for the analysis is specified, the computational graph can be traced for a specific analysis using operator overloading, which is a feature available in object oriented languages. In this way, a specific property of a program can be obtained. Then, when the property is compared with the expected value, it is possible to assess correctness with the hope of eliminating potential problems. The scheme is illustrated with an analyzer to detect arithmetic errors.

Abstract. Este trabajo enfatiza los métodos de prueba útiles para evaluar la calidad de programas de computadora. En particular presenta métodos de unidades que utilizan el grafo computacional. Una vez que el algebra para el análisis ha sido especificada, el grafo computacional puede ser rastreado para un análisis específico mediante la sobrecarga de operadores, lo cual es una característica de los lenguajes orientados a objetos. De esta forma una propiedad específica de un programa puede ser obtenida. Así cuando la propiedad es comparada con el comportamiento esperado es posible evaluar que tan correcto es el programa, con la posibilidad de eliminar errores potenciales. El esquema propuesto es ilustrado con el análisis para detectar errores aritméticos.

Keywords: Program testing, algorithmic analysis, operator-overloading.

Introduction

The demand on reliable computer programs is constantly increasing, as a result of awareness of the effects of quality in applications like process control, and world wide transactions..

This work is derived from our experience in developing testing programs for dynamic simulation; and as designers and users of computer tools for computer program analysis [Estrada, 2003]. In particular attention is paid to the automatic algorithmic analysis.

Useful Definitions

A program *state* is the minimum information required to restart a program if we temporarily halt an execution.

A *precondition* is a predicate that must be true before a transition.

A *postcondition* is a predicate that must be true after a transition.

A *test driver* is a procedure used to call the testing procedure at different input conditions.

A *test stub* is an emulator of a procedure called by the test procedure that is more reliable than the actual implementation of the called procedure.

A *software component* is a semi-independent computer program that contains one or more procedures. A *path* is a unique sequence of branches from the function entry to the exit.

Type of Tests

A diligent tester will consider the valid inputs from each of these sources as well as invalid, unexpected inputs Whittaker, “What is software testing?, and why is it so Hard”. IEEE Software (1998).

We divide the type of tests according to the scope covered, according to the language elements analyzed, and according to the form of execution.

According to the Scope

Unit Testing. Tests individual software components.

Integration Testing. Tests collectively procedures previously tested individually. A recommended strategy is to start with a set of units with common objective or layer, then couple all units needed for that objective and after that, proceed to the next objective or layer. Finally, integrate all the layers with the units of the next layer up [Gonzalez, 1984]. To easy composition among units, they must be compatible, with low redundancy in features, and a simple interface, but strict to detect coupling errors. A desirable feature in the implementation language is that it does not modify calling arguments, for instance in Matlab (of Mathworks)

```
[r1,r2] = procedure(arg1, arg2);
```

If they are input-output arguments, they should appear in both sides of the equal sign.

System Testing. Tests all components that constitute a deliverable product. Usually, the entire domain must be considered to satisfy the criteria for a system test. Usually this type of tests selects a set of test scenarios, runs and evaluates them, measures test progress, and maintains a records of discrepancies with the expected results.

Functional Testing. It concentrates in what the program is supposed to do and how well it does it.

Boundary testing. Test the program in their boundaries. Transitions are often the more complex and error prone. Since loops introduce an unbounded number of paths, boundary testing considers only a limited number of looping possibilities.

Capacity testing. It finds the behavior of the system in the limits of its capacity.

Acceptance testing. It verifies readiness for use. It is similar to system testing except that it is handled informally by users external to the development process, even external to the entire company. The idea behind this testing is to get a critical evaluation of real-world use.

According to the Language Elements

After consulting some references of common errors, and record experiences on program development [Ghezzi et al, 1991],[Friedman, & Voas, 1995],[IBM],[Beizer, 1990] we considered that the most systematic form is to classify errors according to the programming rules are those presented by Hoare and coworkers (1987):

Table 1. Error classification

UNIT TEST	Detects	Verification Method
I/O	File access, attributes, and use	Relations of Inputs/Outputs
Variables	States Variable declaration, initialization, reference, assignation, range and precision. Memory, arrays and pointers	State coverage Orthogonality of states Domain coverage allocation/ deallocation of memory.
Expressions	Substituting a constant for a variable. Code: redundant or lacking Arithmetic: stability, overflow, underflow Assignation: incompatible assignation; Comparison of improper types Sequence of operations Incorrect Operand or Operator	Efficiency and accuracy of Operations. Computational Graph

Logic	Precondition-condition-postcondition.	Branch coverage
Flow control	Cycles: Jumps No terminations Error by one	Branch coverage Path coverage Edge coverage Boundary testing
Error handling	Label missing Errors without description	Assertion testing
Style	No use of standard language features, libraries of working group standards	Preprocessor
COUPLING TEST		
Interface Tests between procedures	Formal parameters are not consistent with the calling procedure. Returned value is not consistent with the expected value Invalid ending condition	Procedure state Parameter passing Returned values
Modules	Interaction among modules	State of Module
Events	Timing, serialization	Sequence-Event testing
Efficiency	Counting of operations. Memory required, number of procedure calling.	Execution profile Memory profile
Semantic	Verify module interaction Verify operator overloading	Equivalence testing (class coverage)
INTEGRATION TEST		
Composition	Composition (balance, complementarity's, contrast)	Preprocessor
Scenario	Scenario Tests	Modeling working environment. Testing execution progress.

Proposed method of analysis

“Dr Curtis enquired what was known about detecting logical errors in programs, He felt hi worst bugs were of this kind, and nothing he had heard would help him” Gentelman, Performance Evaluation of Numerical Software, Fosdick(ed.) IFIP, NordHolland Pub (1979)

An ideal program analyzer should give the level of detail as symbolic analysis, must trace the path followed by the program, and be easy to implement as a dynamic analyzer.

Algorithmic Analysis

Algorithmic Analysis use static analysis to generate tables of identifiers. Then, by following the computational graph it is possible to obtain properties related to the code. Graphs are a data-structure whose properties match the properties of a computer program, which contains conditional branches, loops and function calls. This computational graph can be obtained by overloading arithmetic (+, *, =,) and logic operators (&&, ||), and arithmetic functions used in the computer program. Then, as a by-product of every expression, information related to the states, unused or unassigned variables, or potential arithmetic errors (underflow, overflow) during execution can be obtained. The type of information obtained depends on how every operator is overloaded (see fig 3.1).

```

/* evaluation of a balance for a non-return valve */
if ( $\Delta P < \Delta P_{Min}$ ) {
    VcR[0] = -da/dt + KC*( $\Delta P - \Delta P_{Min}$ )*Ar;
} else {
    VcR[0] = -da/dt + KO*( $\Delta P - \Delta P_{Min}$ )*(1-Ar);
}
/* evaluation of algebraic residual for momentum */
if( $\Delta P > 0$ ){
    VcR[1] = -Wm + Kv*sqrt(+ $\Delta P$ )* Ar;
} else{
    VcR[1] = -Wm - Kv*sqrt(- $\Delta P$ )* Ar;
}

```

Fig 3.1 Expression to Describe the behaviour of a non return-valve

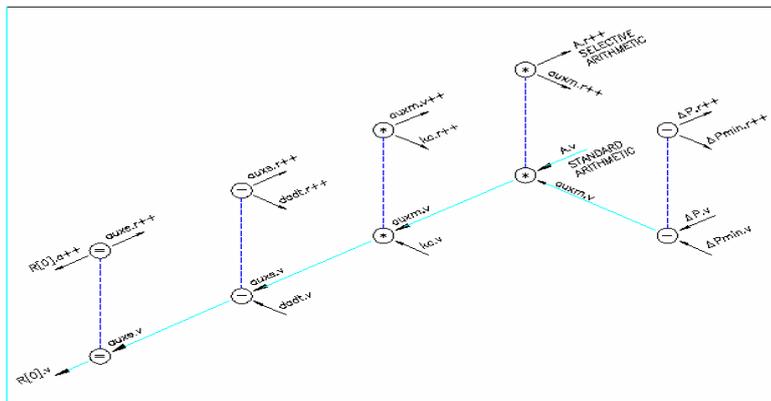


Figure 3.2 Computational graph of R[0] for the expression in fig 3.1.

The analysis is accomplished in a similar way as telephone poles hold telephone wires and cable television wires simultaneously. Then while the standard arithmetic is executed (lower layer in fig 3.2) the selective arithmetic is also executed (upper layer

in fig 3.2). The data structure $R[0]$ stores the arithmetic values, $R[0].v$, and the number of references $R[0].r$.

This work stems from the work on algorithmic differentiation [Griewank, 1996]. To analyze a model by algorithmic analysis, it is necessary to:

- Specify the *selective arithmetic* for the analysis.
- Design a *set of classes* supporting this algebra through the overloaded arithmetic operators and elementary functions, and
- Specify the *asserted behavior* for global and local variables, once the execution is completed.

The composed analysis is evaluated automatically calling these functions in the order determined by the computational graph, and assembling the results of the analyzed terms to compute the final result [Molina et al, 1999].

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.

Branch Testing to trace the Behaviour of the Computational graph

The elements for this type of testing are [Howden, 1987]:

- 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 *test generator* 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, it is expected that the model presents a different form of behavior.

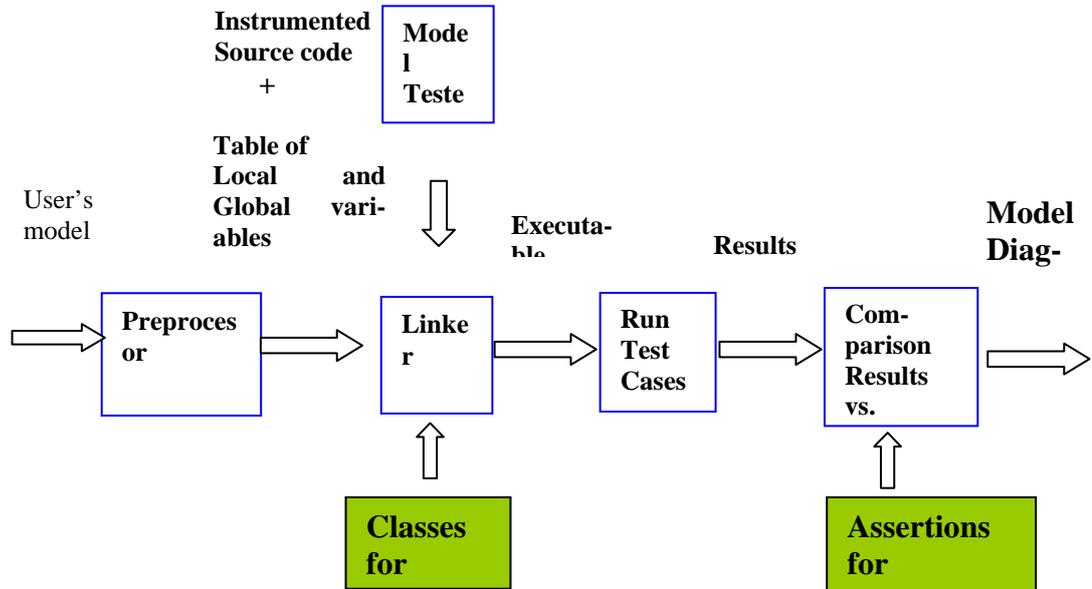


Fig 3.3 Tasks during Model Analy-

Probes to analyze the condition of a branch when it is traversed. A set of classes supporting the algebra for the analysis through the overloaded arithmetic operators and elementary functions are used as probes.

- **A report generator**, which contains the complete picture of branch execution. A report generator produces a table of local and global variables with their properties extracted during the analysis at different branch conditions.
- **An oracle**, which determines correctness of the program's output for some input. The extracted model properties are compared with the expected properties for every variable. Discrepancies are reported as diagnoses

Since programs 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 [Howden, 1987]. Given the fact that 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 computational graph is obtained the following tasks are carried out (fig. 3.3)

- The dependence graph of inputs on outputs can be obtained, eliminating intermediate variables.
- Unused variables can be identified with the reachability analysis of output from inputs [Chen, 98].
- Feasible assignment of variables to equations can be detected with a bipartite graph [Westerberg, 1979].
- Dynamical models badly posed can be detected by with the structure of a linearized system [Soetjahjo, 1998].

Results

The analysis of the program's computational graph allows a good level of detail in the analysis. Object oriented languages facilitate the implementation of this type of analysis. We exemplify now this scheme with the analysis of efficiency and robustness of models for dynamic simulation.

Analysis of Efficiency & Robustness

The purpose of the analysis of efficiency and robustness is to detect domain values that causes under/overflow or an undefined value of arithmetic operations in a model. In real time simulation, it is also essential to detect the largest computing effort required in a model.

Algebra for efficiency and robustness:

- The result of a detected dangerous operation 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 only reports the operation count in variables, which are used to evaluate the final variables; thus, a variable, which is only used as a predicate of a conditional clause, is not taken into account.

Table 2 shows the arithmetic operators and functions whose operands are verified to be within its valid domain. If a potential error is detected, the analyzer issues a diagnostic flag, and execution continues using a *safe* arithmetic.

Table 2. arithmetic operators and functions verified

Arithmetic Operations and functions	Condition of risk	Safe arithmetic
$1/x$	$ x < \tau$	$1/(x + \varepsilon * \text{sign}(x))$
$\tan(x)$	$x < \tau$	$\tan(x + \varepsilon * \text{sign}(x))$
\sqrt{x}	$x < 0$	$\sqrt{ x }$
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

Attributes of a variable that is a class of this type:

- Its value.
- The number sums-subtractions $\{+, -\}$ required to calculate its value
- The number of multiplications-divisions $\{*, /\}$ required to calculate its value,
- The number of roots $\{\sqrt{\quad}\}$ required to calculate its value,
- The number of exponential and trigonometric functions $\{\exp(\quad), \log(\quad), \sin(\quad), \cos(\quad), \tan(\quad), \dots, \text{etc}\}$ required to calculate its value.
- A logical flag that is set `true` when the value of a variable of an intermediate variable is transferred to another variable by an assignment. Thus, the operation count is only taken once, faithfully evaluating the operations count.
- The number of faults in divisions, square roots, or trigonometric expressions.

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) produced by these inputs. The cost depends on how many test cases are used to cover the input domain.

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 for a model to fail.

For exhaustive testing in a reduced number of tests, this analysis requires the careful design of test cases. Offutt, Jin and Pan [Offut, 1999] 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 edges, can reduce the input domain.

Offutt, Jin and Pan [Offut, 1999] 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 edges, can reduce the input domain.

This section presents the results of the analysis of dynamic models. The characteristics of these models are displayed in Table 3. The first three models (SpEquipo, BbEquipo, WwEquipo) were adapted from a power plant simulator based in subsystems (steam-generation, water-feeding, etc). Model, MiEquipo, has been designed and improved by colleagues with detailed knowledge of this process unit. Model DmEquipo evaluates detailed physical properties for the liquid, steam and two-phase regions.

Table 3. Characteristics of Model Analyzed

Process Unit Name filename	Number of States $x(t), y(t)$	Number of Parameters p	Number of externals $u(t)$	Number of Residual Equations R	Max Nesting level of conditional Branches	Num of procedures
Electrical Pump BbEquipo.c 	8, 3 streams	36	2	4 energy, mass momentum	1	8
Water Walls WwEquipo.c 	8, 3 streams	1	5	4 Energy of Metal, Energy, momentum and mass of steam	0	8
Induction Engine MiEquipo.c 	2, 4 streams	22	2 1 logical	2 Torque Angular Speed	2	8
Drum with Single and double phase DmEquipo.c 	8, 4 streams	5	9	8 Mass, energy of liquid Mass, energy of steam Momentum Mass, energy Momentum of output steam	1	8 + 5 for physical properties

This analyzer checks if a model can handle the input streams. Results of the analysis from different test cases is shown in figures 4.1, 4.2, 4.3,4.4.

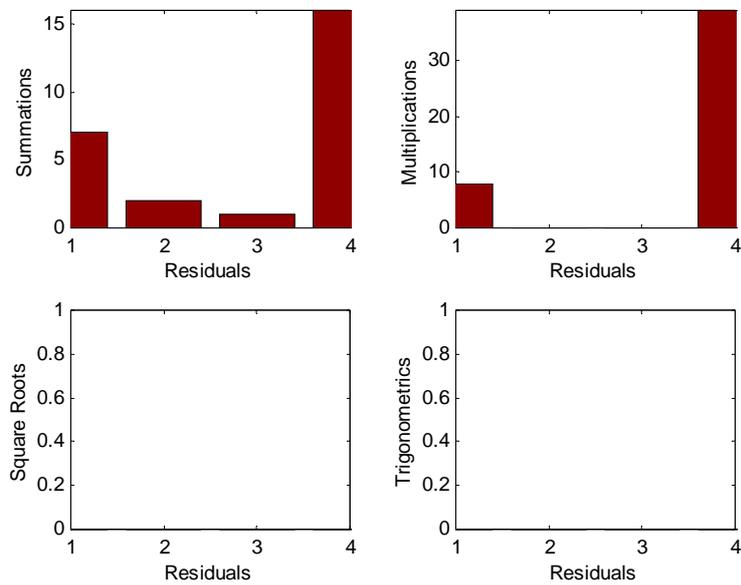
Notes:

Two different test cases are shown per residual.

Some test inputs caused models to incompletely finish computations.

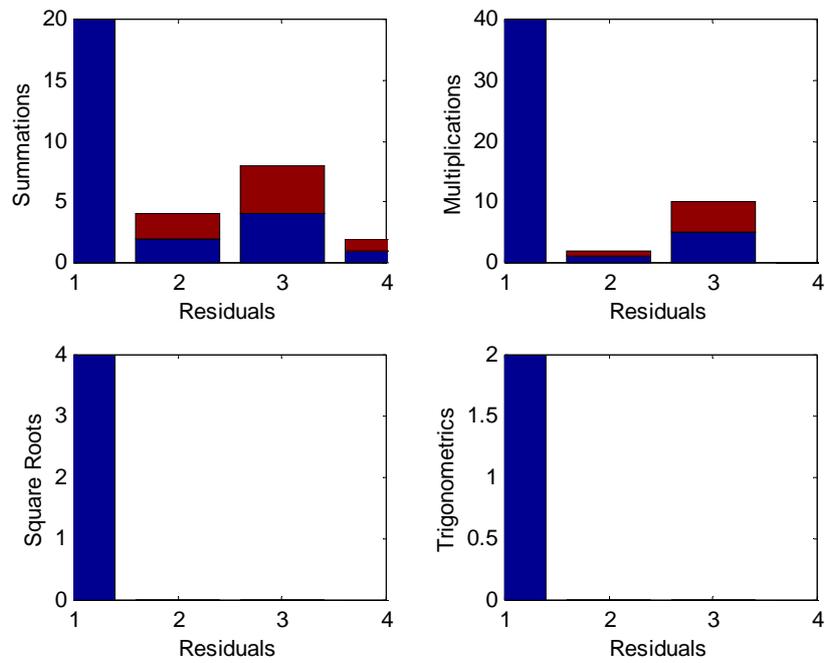
The residuals are the r.h.s. of algebraic or differential equations.

Fig. 4.1 Number of Arithmetic Operations for model Bb at two different runs



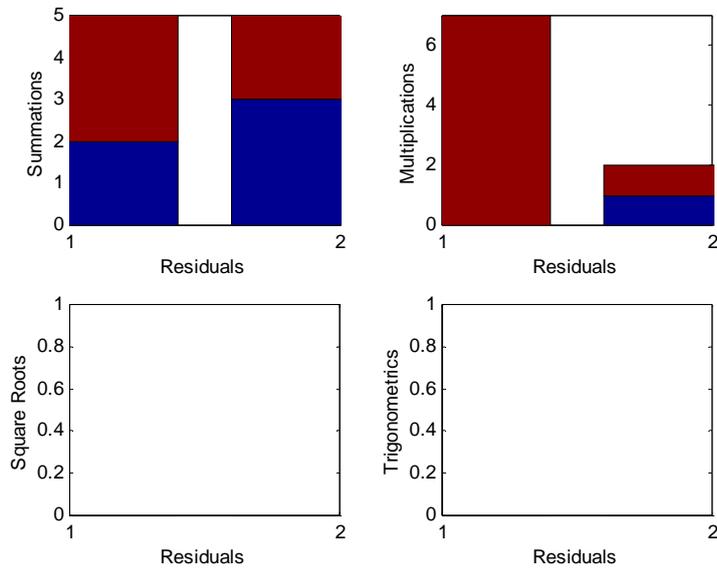
For some runs model **Bb** does not compute any residuals, since an invalid domain was detected internally, then the model returned without completing calculations.

Fig. 4.2 Number of Arithmetic Operations for model **Ww** at two different runs



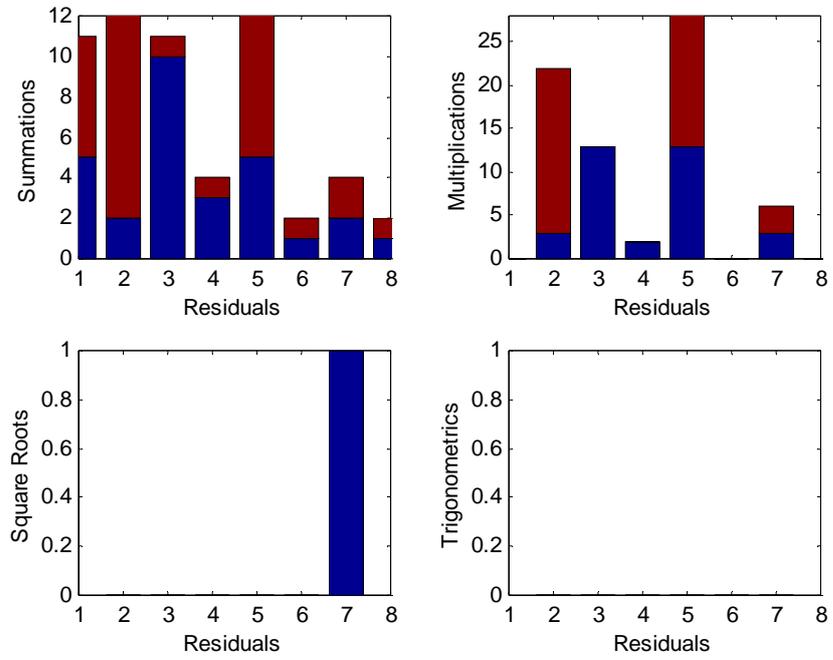
Model **Ww** presents four divisions by zero, which are due an arithmetic term which evaluates the inverse of the sum of fractions with small denominators : $(1/x_1 + 1/x_2)^{-1}$. To overcome divisions-by-zero this term can be refactored.

Fig. 4.3 Number of Arithmetic Operations for model Mi at two different runs



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 to evaluate the residuals.

Fig. 4.4 Number of Arithmetic Operations Dm at two runs



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.

Conclusions And Further Work

Testing plays a key role in both assessing and achieving quality (Friedman, Voas, Software Assessment: reliability safety, testability, 1995)

Approaches to testing are becoming more systematic. In particular, algorithmic analysis allows us to extract code properties that can be used to verify output correctness. The computational graph matches adequately the algorithmic properties of common computer programs, and since it can be obtained with the help of operator overloading, then graph-based testing methods are effective to extract relevant features of computer programs.

Derived from this work we can also emphasize the relevance of an adequate testing plan combined with available mathematical techniques for program testing to improve computer programs.

Offut, Liu, Abdurazik and Ammann [Offutt, 2003] presented criteria for generating test inputs from state-based specifications. The test inputs include tests at transition predicates, pairs of transitions and sequences of transitions that contain inputs necessary to modify the software into the appropriate state for the test values. This work has some useful similarities with control theory which advocates to analyze model behavior by analyzing its states.

Acknowledgements

The program for improvement of higher education, ProMEP provided funds to acquire and use the commercial tools discussed here, and to develop the graph based testing methods.

References

- Estrada A. "Sistematización del desarrollo para aseguramiento de Calidad de Programas de Computadora", MsC Thesis, FCQI, UAEM, 2003.
- Gonzalez S., E. Méndez, F. Kuhlman, and I Castelazo, "Large Scale Power Plant Model, development, Part I: Modularization" pp 359-366.in Proc. of Intl. Conf. on Power Plant Simulation, R Fernandez-Del-Busto, D.L. Hetric, E. Mendez Eds. Mor., Mex Nov 19-21, (1984).
- Ghezzi C., M. Jazayeri, D. Mandroli, *Fundamentals of Software Engineering* Prentice Hall, (1991).
- Friedman M. & J Voas, Software Assessment: reliability safety, testability, J Wiley & Sons, (1995)
- IBM, "Orthogonal defect Classification", <http://www.research.ibm.com/softeng/ODC/ODC.HTM>
- Beizer B., Software Testing Techniques, 2nd edition, New York: Van Nostrand Reinhold, (1990.)
- Hoare C.A. R., I. J. Hayes, H.E. Jifeng, C.C. Morgan, A.W. Roscoe, J. W. Sanders, I.H. Sorensen, J. M. Sivey, and B. A. Sufrin "Laws of of Programming", Communications of the ACM v 30 no pp 672-682, (1987).
- Griewank A., D Juedes, J Srinivasan, and C Tyner, "ADOL-C a package for the automatic differentiation of algorithms written in C/C++", ACM Trans. Math Software, 22 pp 131-167, (1996).
- Molina J. M., J. R. Zamora, Y. Mendoza, D Juárez "Analysis of Computational Models by Operator Overloading, Congreso Internacional de Computación", IPN, D.F Nov, pp 202-206, (1999).
- Howden W. E. Functional Program Testing and Analysis Mc Graw Hill, (1987).
- Chen Y., E. R. Ganser, E Koutsofios, "A C++ Data Model supporting Reach ability analysis and Dead Code Detection". IEEE Trans. Soft Engng, v 24, No 2 pp 682- 693, (1998).
- Westerberg A. W., H.P. Hutchinson, R.L. Motard and P. Winter Process Flowsheeting, Cambridge Univ Press, Cambridge, England, 1979.
- Soetjahjo J., Y.G. Go, O.H. Bosgra "Structural diagnose tool for interconnection assignment in model building and re-use", Computers.Chem. Engng. V 22 suppl, pp s933-s936, 1998.
- Offut J., Z Jin , J Pan "The Dynamic Domain Reduction Procedure for Testing Data Generation", Comp Practice and Experience v 29 (2) 167-193, 1999.

Offutt J., S. Liu, A. Abdurazik, P. Ammann “Generating test data from state-based specifications”
Software Testing, verification and Reliability , v 13, No 1, pp25-53 ,2003.