Advancing Scientific Computation by Improving Scientific Code Development: Symbolic Execution and Semantic Analysis

Mark E. M. Stewart
QSS Group Inc. at NASA Glenn Research Center
Brook Park, Ohio, USA 44135

Abstract. This paper presents an implementation and critical analysis of a technique for automated, rigorous scientific program comprehension and error detection. The procedure involves taking a user's existing code, adding semantic declarations for some primitive variables, symbolically executing the user's code, and recognizing semantic concepts from the symbolic expressions generated. This analysis provides high-level, semantic information and detects errors in a user's code. Since a user's code is executed symbolically, the analysis is general, replacing many test cases. Symbolic execution of a 5k line of code (LOC) scientific code demonstrates implementation of a practical symbolic execution / semantic analysis tool. Although this technique promises a powerful tool, several challenges remain.

1 Introduction

While computer hardware speed and cost have improved exponentially, software development has not experienced similar technology driven growth, and arguably, it remains a manual process. Symptomatically, software engineering techniques largely ignore code semantics—the what, how, and why of a computer code; these semantics are left to the code developer. As scientific code developers, we recognize that our scientific programs involve an organization of classic mathematical, logical, and physical concepts. Further, we recognize that programs from a wide range of scientific and engineering fields use and reuse these same fundamental concepts in different combinations.

These code semantics are important to all stages of the software development cycle. The planning, development, debugging, and verification of software involve correctly implementing semantics within the code. Understanding code during the maintenance phase or finding errors in code involves comprehending code semantics. For example finding the error in the second difference code (1) involves comprehending the semantics of difference equations.

```
FS(I,J) = DW(I+2,J) – 2.*DW(I,J) + DW(I-1,J)        (1)
```

The cost of the existing manual approach in terms of frustration, time, and effort is considerable, but it is still relatively small when compared with the cost of software failure—particularly for mission critical applications[1]. Consequently, what is necessary is a theory of code semantics.

This paper reports on an effort to formalize and automatically analyze these scientific code semantics. In particular, there are two interrelated elements of the current approach: symbolic execution of a user's code, and semantic analysis of the resulting symbolic expressions. The thesis of symbolic execution is that the semantics of a programming language's construct (the variables, data structures, and operators) can be faithfully represented symbolically. Further, this symbolic representation can be propagated during an execution of the code to provide a general and rigorous analysis. This approach is heavily dependent on the classical mathematical and logical concepts and notation that code developers and engineers are familiar with.

This symbolic execution, however, generates symbolic expressions; for example, symbolic execution of (2a, b) generates mathematical and logical expressions for A and N.

```
A = B * C                          2(a)
IF ( N .GT. 100 ) N = 100          (b)
```

As these and further expressions are generated and used they will grow exponentially—unless they are simplified. Here semantic analysis uses parsers to automatically recognize the use of mathematical and logical concepts and simplify these expressions. For example, (2b) could be simplified by bounding N by 100. Further, errors are detected when expressions cannot be recognized. For example, in Code 1, the first—but not the second—unit conversion is recognized; this process actually lead to the detection of this error in a code.

Code 1: *Analysis detects the unit conversion error*

```
pi = 4.*atan(1.)
degtor = 180./pi
…
angle = angle * degtor
…
angle = angle * degtor
```

This symbolic execution / semantic analysis procedure is appealing for three reasons. First, scientifically and intellectually speaking, it addresses fundamental code semantics which are the currency of program developers. Second, human programmers analyze code at approximately 0.5 LOC per minute; symbolic execution runs quickly—approximately 1000 times faster than a human—often faster than numerically executing the code itself! Third, the procedure uses the abstraction of symbols—not numbers—and the descriptive power of classical mathematical notation. This abstraction with symbolic notation is what makes this technique general and rigorous, that is, a single symbolic analysis can replace testing with a suite of conventional test cases. In Code 2, for example, if the search fails, a memory bounds error occurs. Symbolic execution detected this error, but numerical execution would require a specific set of inputs before this error occurred.

Code 2: *Analysis detects how the search failure results in a memory access error*

```
       Dimension array(100)
       …
       Do 10 I = 1, 100
          If ( test_value .le. array(i) ) goto 20
10   Continue
20   value = array(I)
```

Balancing these advantages are challenges. Symbolic execution is a difficult mathematical and computer science problem; a wealth of semantic concepts exist that can be organized in complex ways that must be recognized with high reliability.

The concept of symbolic execution was introduced by King[2] in 1976. In a review article, Coward[3] suggests symbolic execution has languished due to the difficulty of implementation, and cites four problems among the systems studied:

1) evaluating array references dependent on input (symbolic) values,
2) the evaluation of loops where the number of iterations is unknown,
3) checking the feasibility of paths: how to process branch conditions dependent on symbolic expressions,
4) how to process module calls: symbolically execute each call or execute once and abstract,

Code semantics have been the focus of some work[4,5,6] including the use of an ontology and parsers for natural language understanding[7]. The current work focuses on code analysis, in the belief that code synthesis[8,9] cannot be done completely without an understanding of code semantics. Petty[10] presents an impressive procedure where—during numerical execution—the units of variables and array elements are analyzed. The procedure can be easily applied to a user's code; however the numerical execution results in high wall-time and memory requirements.

In the following sections, the symbolic execution procedure is explained, key problems and solutions are presented, and results are demonstrated including results for a 5k LOC scientific code. Finally, the remaining challenges are discussed.

2 Symbolic Execution Procedure

In outline, the symbolic execution / semantic analysis procedure consists of three steps (Figure 1). First, the user adds semantic declarations to the program to be analyzed. Semantic declarations (3) provide the symbolic identity of primitive variables in the code; they replace any program inputs or input files.

A <= acceleration, m/s2; (3)
M <= mass, kg;

Second, a parser converts the user's code and semantic declarations into a tree representation in a language independent form. Third, symbolic execution / semantic analysis of the code occurs.

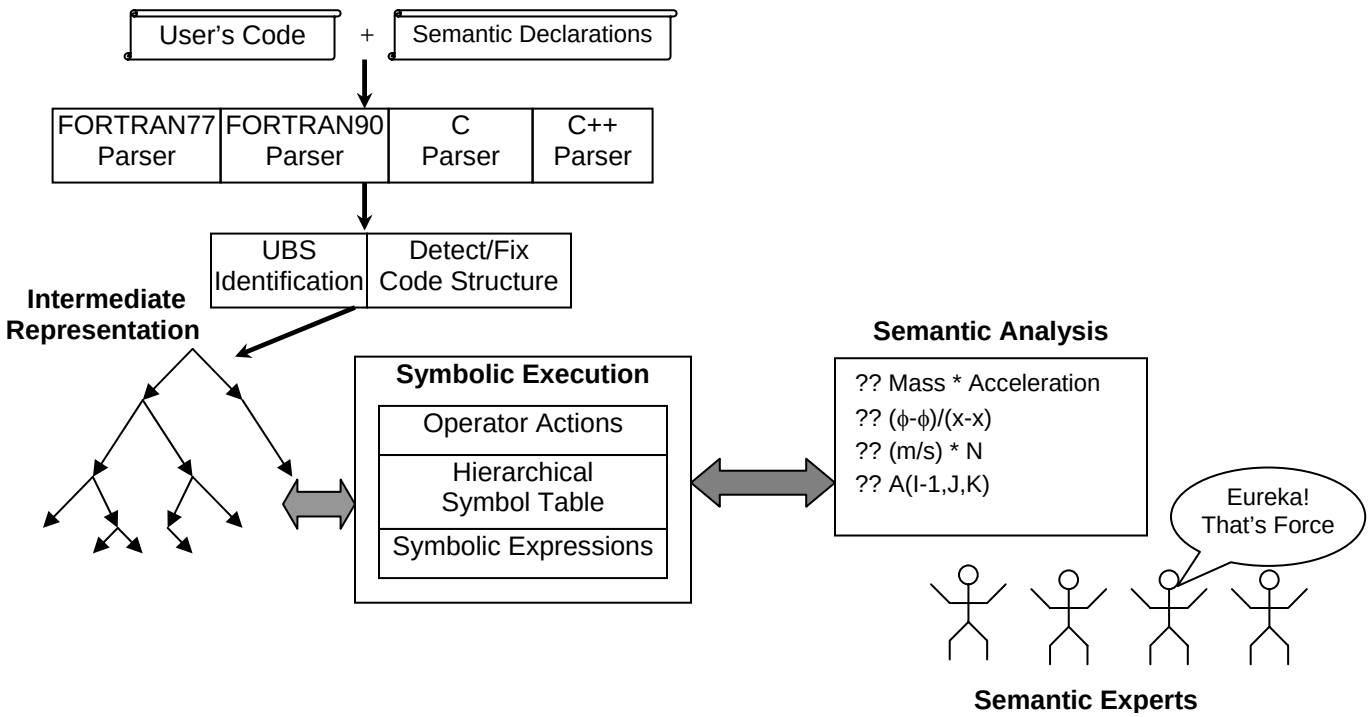

Figure 1: Overview of the symbolic execution / semantic analysis procedure. Expert parsers recognize formulae and concepts generated during symbolic execution.

2.1 Symbolic Execution

In symbolic execution, an emulator program executes statements from the user's program. Instead of loading into memory numerical values of input variables, the emulator uses symbolic values that describe the input variables; the emulator takes statements from the user's program (the common representation—a tree), performs the operations on these symbols, and generates symbolic expressions. Table 1 contrasts numerical and symbolic execution.

Table 1: Comparison of numerical and symbolic execution for code statements. For numerical execution, the input file contains "4 5"; the semantic declarations are (3).

Code Statement	Numerical Execution	Symbolic Execution/ Semantic Analysis
READ M, A	Put 4 into M, 5 into A	Attach Semantic Declaration to Instance of M and A
B = 10	Transfer Number Value	Transfer Symbolic Value
M * A	Calculate 4 * 5	Form "mass * acceleration", "kg * m/s2" and attempt simplification
If (A.eq.5) then B=5	A.eq.5 is True, so Transfer 5 to B	Form "A.eq.5 => 5 | B" and attempt simplification

Just as a compiler has an action (memory fetches, operations, and memory stores) for each statement, this symbolic execution process has a prescribed action or response to each statement encountered in a user's program. The difference is that the response is symbolic; the operators +,-, *, /, ** prompt the formation of a symbolic expression; an array reference prompts searching an array representation, and symbolically testing for relative position; a logical expression prompts symbolic testing of branches, and the exploration of potential paths.

2.2 Semantic Analysis

As statements are symbolically executed, the generated symbolic expressions become exponentially larger—unless they are simplified. The role of semantic analysis is to recognize and simplify the fundamental mathematical formulae used in these expressions.

These formulae are recognized and simplified using parsers[11,12]. Parsers' balance between efficiency[13] and recognition limitations make them a key element of compilers as well as a good choice for recognition in semantic analysis. From a carefully written set of rules with actions, YACC[11] generates a module that can recognize formulae and perform actions. Details of how formulae are recognized in parsers are given in [14].

Successful symbolic execution requires the semantic analysis of array references, array assignments, logical expressions, and other operations. But parsers can do more! Symbolic representations can be found for more semantic aspects of scientific and engineering code, including units, dimensions, vector analysis, and physical and mathematical equations. Table 2 provides a complete list.

The ultimate objective of this work is semantic analysis of all conceivable semantic properties which would perform extensive error checking. However, correct symbolic execution is a prerequisite for semantic analysis of all these properties. Consequently, the focus of this work has been to ensure correct symbolic execution and verify it with semantic analysis of units. As a result, semantic analysis of mathematical and logical expressions is relatively advanced when compared with analysis of physical formulae.

Table 2: Scientific semantic properties analyzed by the procedure, including sample equations and number of parsers.

Property Analyzed	Sample Equation	Parsers
Physical Equation	force $\Leftarrow$ mass * accel	**3**
Math Equation	$\Delta\phi \Leftarrow \phi - \phi$	**5**
Logical Expression	$\phi \Leftarrow$ If (True) ϕ else θ	**2**
Value / Interval	[1,50] $\Leftarrow$ [0,49] + 1	**2**
Grid Location	$\phi_i \Leftarrow \phi_{i+1} + \phi_{i-1}$	**4**
Vector Analysis	$\boldsymbol{\phi}\cdot\boldsymbol{\phi} \Leftarrow \phi_x^2 + \phi_y^2 + \phi_z^2$	**1**
Non-Dimensional	$\phi/A \Leftarrow \chi/A + \varphi/A$	**1**
Dimensions	L $\Leftarrow$ (L/T) * T	**1**
Unit	m $\Leftarrow$ m/s * s	**1**
Object	fluid $\Leftarrow$ fluid * anything	**1**
Data Type	Real $\Leftarrow$ Real * Integer	**1**
Language Emulation	mass $\Leftarrow$ A(I,J,K) $\Leftarrow$ mass	**2**

3 Where Symbolic Execution Becomes Difficult: Five Key Pieces

Symbolic execution of mathematical operations (+, -, *, /, **), and assignment involve operator actions that are similar to numerical execution. However, the similarities end with array references, loops, and logical expressions! When these programming constructs involve variables with unknown values, the symbolic action code becomes more complex. While numerical execution of an array evaluation involves retrieving a value at a known index, symbolic execution must retrieve array elements within an index range. While numerical execution of a loop involves executing code a known number of times, symbolic execution must represent execution symbolically and group together symbolically equivalent operations within the loop range. While numerical execution of a logical expression involves using a known value to choose a logical branch, symbolic execution must pursue each possible branch.

Although the symbolic execution code is more complex than for numerical execution, advantages exist. The principle advantage is greater generality and rigorousness; the following three sections will pursue this issue for array representation, loops, and logical expressions.

3.1 Array Assignments and References

Array analysis is a major hurdle in symbolic execution; here this hurdle is overcome with an ontology for array indices that allows the grouping of symbolically identical array elements. After symbolic execution of Code 3,

Code 3: *Simple Loop shows how loops and array references are symbolically executed.*

```
          Integer  A(100)
          Read  N
          A(1) = 5
          Do 10 i=4,N
             A(i) = 1
10     continue
```

the array A is represented as in Figure 2 where the fourth through N^{th} (and 2nd,3rd and $N\text{-}1^{st}$ to 100th) array elements have been grouped together, while the first array element has not.

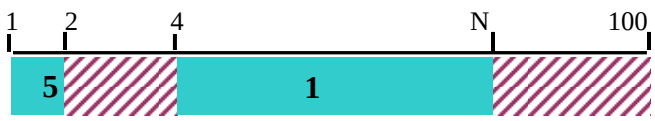

Figure 2: Symbolic representation of the array A() after execution of Code 3. Undefined values are diagonally shaded; array values are in bold; array indices are above.

This grouping of symbolically identical array elements is intuitively clear, yet it has implications. Grouping allows symbolic representation of arrays, and this array representation uses memory efficiently. More importantly, where loops apply an identical operation over large parts of an array—as is so common in scientific computing—the semantic analysis is reduced to one analysis of an array assignment or reference. With reductions like this—and others—symbolic execution and semantic analysis of a user's code is fast—possibly faster than numerical execution of the same code.

This grouping of array elements requires a particular ontology for array indices; the ontology entities are shown in Table 3. In Code 3, scientific code developers will easily recognize array variable "i" is a Counter, variable "N" is a Number; the remaining array index entities in Table 3 are more obscure and infrequently used. This ontology is not necessarily closed; as array index constructs with different semantics are encountered, this list and the accompanying rules can be extended.

Table 3: Entities in the Array Index Ontology. All are integer valued.

Entity	Role of Entity
Integer Constant	A Known, Unchanging Value
Number	A Variable: Unspecified, Unchanging Value
Counter	A Variable: taking on all Integer values in a range
Compressed Counter	Scalar Representation of Counters for Multiple Array Indices
Enumeration	A Product of Integer Constant and Number expressions
Offset	Delineates Multiple Arrays Stored in 1-D Array
Offset Index	Offset plus Compressed Counter
Index Number	A Variable: Unspecified, Unchanging Value, in a range
Compressed Index Number	Scalar Representation of Index Numbers for Multiple Array Indices

During symbolic execution of a user's code (Figure 1), when the emulator encounters an array reference or assignment, operator action routines are initiated that locate the required elements in the array representation. These action routines compare the array indices—symbolically!—with the symbolic bounds of the groupings in the array representation. For example, to reference A(N+1) in Figure 2, the procedure compares N+1 with 1, 2, 4, N, and 100, and concludes A(N+1) is in the final grouping of array entries from N+1 to 100. Note that the existence of A(N+1) is subject to the condition that $N+1 \leq 100$—a condition imposed on number N.

3.2 Loop Evaluation

Loop evaluation is a further hurdle in symbolic execution. The issue is whether dependencies exist *between* loop iterations; if not, then straight line symbolic execution is possible. Here this hurdle is overcome by analyzing when variables are used and set within the loop code. Note that here the terms 'used' and 'set' are not used in a global sense; instead they are used in a local sense—within the loop code only.

The loop in Code 3 is the simplest loop type. Within the loop's code, all variables and array entries are Set-Before-Used (SBU); consequently no dependencies exist between loop iterations, and the loop can be executed in a single pass.

Code 4 contains a variable, ijlast, which is Used-Before-Set (UBS) within both loops. Loop code blocks with UBS variables have dependencies between loop iterations and require two symbolic execution passes—more if the loop block is nested. In the first symbolic execution pass, the procedure determines the variable's initial value and the change between iterations; by mathematical induction, an expert parser tries to recognize the variable's symbolic value at a general iteration. The second execution pass executes with this general form. Loops containing UBS variables are identified in a pre-processor step (Figure 1) so that this analysis is used only where needed.

Code 4: *Code for calculating an Offset Index, demonstrates inter-iteration dependencies for the variable ijlast—hence it is a UBS variable.*

```
      ijlast = imax(1) * jmax(1)
      ijx    = - ijlast
      ijq    = - 4 * ijlast
      do 20 ng = 1, ngrid
        ijx       = ijx + ijlast
        ilocx(ng) = ijx
        do 10 nb = 1, nblade(ng)
          ijq          = ijq + 4 * ijlast
          ilocq(nb,ng) = ijq
          ijlast       = imax(ng) * jmax(ng)
   10   continue
   20 continue
```

3.3 Conditional Expressions

Symbolic execution of conditional expressions is a challenging issue since symbolic values in the condition force the examination of each conditional branch. The current procedure symbolically executes the statements from each possible branch. A hierarchical symbol table allows an independent sibling symbol table for each branch (with inheritance of parental symbolic values). For each variable, the procedure forms a conditional expression that is valid following the conditional expression. The simplified expression is propagated through the following code.

3.4 Subroutine Calls

Calls to routines are not problematic in this symbolic execution procedure. The current procedure responds to routine calls by symbolically transferring call line parameters and global variables to a child symbol table, and then symbolically executing the routine. Upon completion of the routine, call line parameters and global variables can be updated in the parent symbol table.

In the test cases studied, repeat calls to routines were not excessive since routine calls within loops are executed once or a few times. In principle, symbolically identical routine calls need not be repeated, but this feature has not been implemented yet.

3.5 Speed of Symbolic Execution/Semantic Analysis

The wall time requirements for symbolic execution of a code are fundamentally different from the wall time requirements of numerical execution. Two opposing issues influence the wall time—and decide the economics—of symbolic execution.

First, symbolic execution is orders of magnitude slower than numerical execution on an operation by operation basis. Numerical execution of "A*B" includes memory accesses and a floating point operation—usually within optimized software and hardware; symbolic execution of "A*B" includes constructing a data structure representation of the expression, and its examination by several expert parsers (Figure 1). Further, array index evaluation can be enormously expensive, due to the symbolic search through the array representation.

Second, the computationally demanding parts of scientific codes are the iterations of code within loops. Yet, in symbolic execution of a loop, symbolically equivalent (and numerically different) iterations can be grouped together and analyzed once. This can be a massive saving! In addition to routine calls within loops being analyzed once, repeated calls to routines can be analyzed once when the calls are symbolically equivalent.

These two issues interact to produce faster symbolic than numerical execution for loop intensive code. Conversely, codes with fewer loops can execute more slowly symbolically than numerically.

4 Demonstration of Results

This symbolic execution / semantic analysis procedure has been developed and tested with three codes.

COMDES is a 1-dimensional aerodynamic design code written in FORTRAN77 with extensive use of aerodynamic formulae, relatively less use of mathematical formulae, and minimal use of subroutines. Symbolic execution completes successfully with 100% semantic analysis of units (Table4).

STAGE2 is a 5k LOC, 2-dimensional computational fluid dynamics (CFD) code that solves turbulent, aerodynamic flow over compressor blade sections. Written in FORTRAN77, it is aggressively coded and makes extensive use of mathematical formulae, loops, array references and assignments (compacting multi-dimensional arrays into a 1D array, and multiple blocks of data into a 1D array), conditional expressions, and routine calls. The symbolic execution and semantic analysis of units completes almost completely. Details are shown in Table 4. For realistic grids and number of iterations, the resulting loop sizes make symbolic execution much faster than conventional numerical execution.

RVCQ3D is a 3k LOC, 2-dimensional CFD code that solves steady, aerodynamic flow over compressor blade sections. Written in FORTRAN77, it uses a moderate coding style and makes extensive use of mathematical and logical formulae. Currently, this code is a development test case where the current procedure is being extended to achieve complete symbolic execution.

Code	Lines (k loc)	Semantic Declarations	Symb Exec Wall Time (s)	Symb Exec Wall Time/ k Lines (s)	Unit Recognition Rate (%)	Statements Executed (k)	Max Memory (MBytes)
COMDES	0.4	42	15.1	37.7	100.		2.8
STAGE2	4.9	87	199.4	40.7	93.9		65.3
RVCQ3D	3.3	49	-	-	-		-

Table 4: Current performance results for the semantic analysis program's analysis of three test cases. Max. memory is the gross memory required to represent and retain all local and global semantic information during the semantic analysis; the executable size is 5.0 MByte. Calculations performed on a PC with a Pentium 4 2.2 GHz processor with 512 MByte of RAM. The analysis results reflect the semantic analysis code's quality and not the quality or ability of the tested codes.

5 Discussion of Challenging Issues

In trying to achieve complete symbolic execution of the STAGE2 code, several challenging issues became apparent. In the following sections, the roles of semantic complexity, inference chains, and problem closure are discussed.

5.1 Semantic Complexity

Refinement of this procedure (so that STAGE2 symbolically executed successfully) involved finding errors, generalizing rules and actions, and handling programming language constructs. However, progressively more complex concepts were encountered, particularly where aggressive programming techniques were used. For array references, for example, indices have been encountered that store a multi-dimensional array in a 1D array, and store multiple blocks of data in a 1D array. A conceptually challenging array index is shown in Code 5. Further, some codes have been encountered where the conditional branching is so complex that it cannot be understood by an experienced programmer.

These experiences raise issues including, whether the population of semantic concepts used in code is limited or bounded. Will programmers use increasingly aggressive and obscure concepts? Does clear, well written code use only a bounded set of basic semantic concepts? Further, human programmers have limits to their knowledge and comprehension capabilities. It is believed that the answers to these questions will be revealed, in part, by additional work on symbolic execution and semantic analysis.

Code 5: *Array index, i, demonstrates increasing semantic complexity.*

```
c                        spline evaluation
      iend = 1
      Do 400 j=1,n-1
        ibeg = iend
        Search for iend .st. x[j+1] ≤ xnew[iend]
        Do 500 i=ibeg,iend-1
         ... ACTION …
 500    continue
 400  continue
```

5.2 Inference Chains and Recognition Reliability

Another challenge of symbolic execution is reliability. Recognizing and simplifying one expression depends on all the previous recognition results—the inference chain—and failing to recognize one result usually prevent any further recognition. For example, in Code 3, a failure to locate and assign to A(1) compromises the remainder of the analysis.

A code's inference chains can be exceedingly long. In COMDES, chains as long as 140 inferences have been measured, and the longest chains in STAGE2 are probably at least an order of magnitude greater. As code size and number of inferences increase, the chance of a recognition failure also increases and reliability decreases.

This reliability challenge for symbolic execution contrasts with the human programming process. It is believed that human programmers do not analyze code in this manner (from the start of execution); instead they look at smaller sections of code, assume the identity of variables and array layouts, make simplifying assumptions, and look for local inconsistencies.

5.3 Problem Closure

Complete symbolic execution of the STAGE2 code involved previous work[14] plus a large extension effort that was completed (part-time) over 3 years. This effort involved the conception, development, and refinement of several techniques, including techniques for array references and assignments, loop execution, conditional expressions, and routine calls.

Given this effort for one code, the crucial feasibility questions are "How much development is required for full symbolic execution of the next user's code?" and "When does symbolic execution of a successive code become routine?". Refinement of symbolic execution by development and testing with the code RVCQ3D is a key test of the procedure's state of development. The expectation is that previous work will apply so that development time drops for each successive user's code until closure is reached or fundamental limitations are discovered.

6 Conclusions

When this work was started, it was not clear that a 5k LOC scientific code could be automatically, rigorously, symbolically analyzed. It was not clear that the semantics could be formalized, or that this formalization could be automated, or that sufficient reliability could be achieved.

Now, what is not clear is how the population of semantic concepts will grow, how obscure concepts will be recognized, and how complex expressions will be simplified in yet larger codes.

However, what has always been clear is that semantics are inseparable from scientific code development, and that semantics must be formalized and automatically recognizable so that we better understand and test our scientific codes.

References

[1] "Mars Climate Orbiter Mishap Investigation Board Phase I Report," Nov. 10, 1999, (ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf).

[2] King, J. C., "Symbolic Execution and Program Testing," Communications of ACM, 1976, **19**, (7), pp. 385-394

[3] Coward, P. D., "Symbolic Execution Systems-A Review," Software Engineering Journal, Nov. 1988. P 229-239

[4] L. M. Wills, "Automated Program Recognition: A Feasibility Demonstration," *Artificial Intelligence* 45 (1-2): 113-172 (1990).

[5] A. Quilici, "A Memory-Based Approach to Recognizing Programming Plans," *Comm. of the ACM*, 37(5):84 (1994).

[6] B. Di Martino, C. W. Keβler, "Two Program Comprehension Tools for Automatic Parallelization," *IEEE Concurrency*, Jan.-March 2000.

[7] J. Allen, Natural Language Understanding (Benjamin/Cummings, Menlo Park, 1987)

[8] E. Kant, "Synthesis of Mathematical Modeling Software," IEEE Software, May 1993.

[9] M. Lowry, et. al. "Amphion: Automatic Programming for Scientific Subroutine Libraries", in Proc. 8th Intl. Symp. on Methodologies for Intelligent Systems, Charlotte, North Carolina, Oct. 1994, pp. 326-335. Springer-Verlag Lecture Notes in Computer Science, Vol. 869.

[10] Petty, G. W., "Automated computation and consistency checking of physical dimensions and units in scientific programs," Software—Practice and Experience, 2001; **31** pp.1067-1076

[11] S. C. Johnson, "Yacc--Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32. (AT&T Bell Laboratories, Murray Hill, 1977).

[12] A.V. Aho, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools (Addison-Wesley, Reading, 1986).

[13] C. L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence* **19** (1982).

[14] M.E.M. Stewart, S. Townsend, "An Experiment in Automated, Scientific-Code Semantic Analysis," AIAA-99-3276, June 1999