

Report on Automated Semantic Analysis of Scientific and Engineering Codes

Mark E. M. Stewart
Dynacs Engineering, Inc.
Cleveland, OH 44135, USA
Mark.E.Stewart@grc.nasa.gov

The loss of the Mars Climate Orbiter due to a software error [1] reveals what insiders know: software development is difficult and risky because, in part, current practices do not readily handle the complex details of software. Yet, for scientific software development the MCO mishap represents the tip of the iceberg; few errors are so public, and many errors are avoided with a combination of expertise, care, and testing during development and modification. Further, this effort consumes valuable time and resources even when hardware costs and execution time continually decrease. Software development could use better tools!

This lack of tools has motivated the semantic analysis work explained in this report. However, this work has a distinguishing emphasis; the tool focuses on automated recognition of the fundamental mathematical and physical meaning of scientific code. Further, its comprehension is measured by quantitatively evaluating overall recognition with practical codes. This emphasis is necessary if software errors—like the MCO error—are to be quickly and inexpensively avoided in the future.

This report evaluates the progress made with this problem. It presents recommendations, describes the approach, the tool's status, the challenges, related research, and a development strategy.

Recommendations

- From a technical perspective, the evidence indicates that a practical automated semantic analysis tool can be developed and thus work should continue.
- The tool should be developed in more than one stage:
 - 1) Complete a unit analysis and “wrapping” tool for general applications (estimate 1-3 person-years depending on requirements)
 - 2) Complete additional mathematical/physical properties—possibly as an open source project.

1. Background

Physical and mathematical formulae and concepts are fundamental elements of scientific and engineering software. These classical equations and methods are time tested, universally accepted, and relatively unambiguous.

Computer science understandably neglects these multi-disciplinary program details in favor of the general problem: analysis of code based on the programming language's semantics. Yet, much of the time and expense of scientific code development and maintenance derives from *manually* analyzing a code's physical and mathematical semantics. Further, the elegance of classical physical and mathematical methods suggests a method of automating scientific code semantic analysis. The work discussed here has explored the feasibility of semantic analysis.

To investigate code comprehension in this classical knowledge domain, a research prototype tool has been developed. The prototype incorporates scientific domain knowledge to recognize code properties (including unit formulae, physical and mathematical equations). Also, the procedure emulates program execution to propagate these symbolic properties through the code. Results of these analyses, including errors detected, are presented through a graphical user interface (GUI).

2. Tool description

From a user's perspective, this procedure involves taking a user's existing scientific or engineering code (1), adding semantic declarations, and viewing/querying the analysis results (Figure 1). Semantic declarations (distinguished by “C?”) identify primitive program variables using standardized technical terms (i.e., *mass*, *acceleration*).

```
C? MA == mass
C? ACC == acceleration          (1)
FF = MA*ACC
```

From the analysis perspective, this procedure involves three elements: a scientific semantics analysis procedure, a programming language emulator, and a graphical user interface; the following sections explain each element.

2.1 Scientific semantics analysis procedure

Classical mathematics emphasizes equations—lexical, sequential expressions that quantify physical or mathematical concepts. A parser is not only an effective way of representing a large set of these physical equations, but it can also efficiently recognize these equations in program expressions.

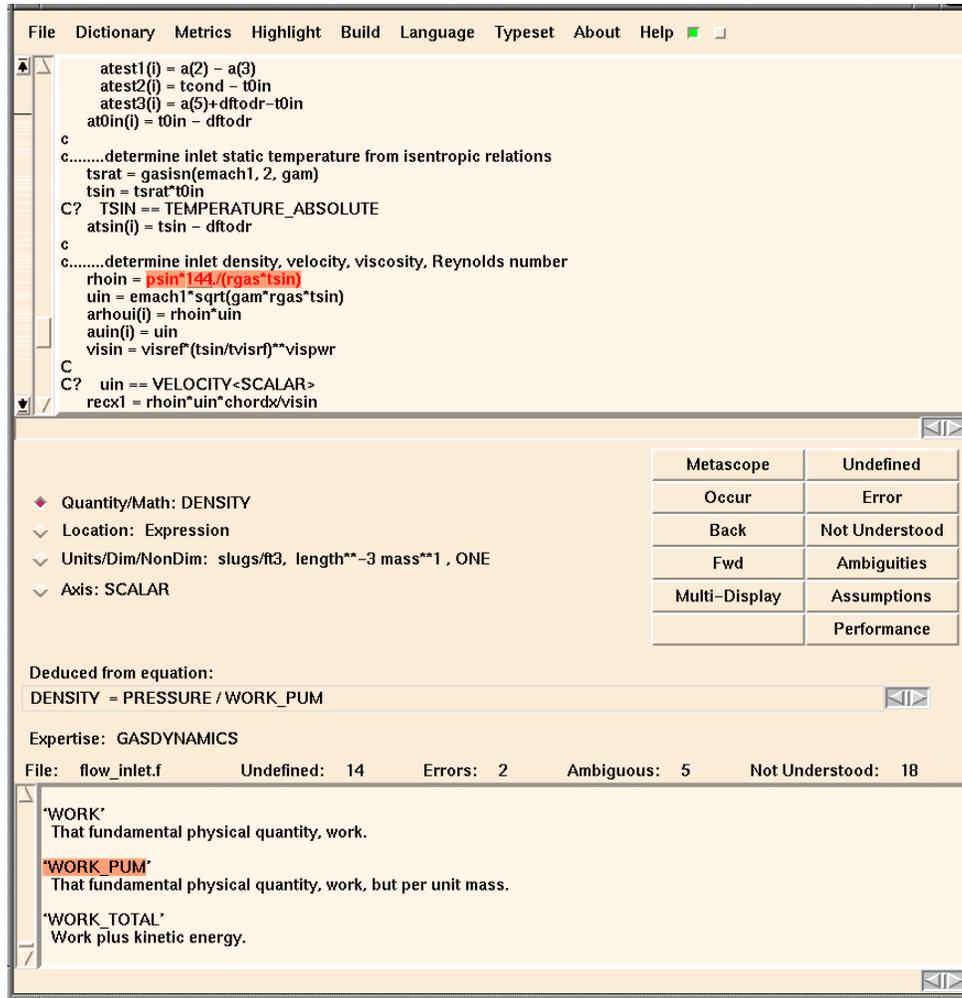


Figure 1: GUI display for the semantic analysis program. The top area displays the user’s code; the middle region explains selected text; the bottom area is a dictionary/lexicon.

In particular, (1) may be translated into expressions of code properties, including a physical quantity expression (2), and a physical dimensions expression (3).

$$\begin{aligned} \text{mass} * \text{acceleration} & \quad (2) \\ (M) * (L * T^{*-2}) & \quad (3) \end{aligned}$$

Parsers recognize formulae in these translated phrases. For example, a dynamics expert parser would include the rule (4), be able to recognize the phrase (2) as Newton’s law, and annotate the data structure.

$$\text{force} \Leftarrow \text{mass} * \text{acceleration} \quad (4)$$

The dimensions expert parser can reduce (3) and verify consistency. These properties (Table 1) reflect the different aspects of program statements that scientists and engineers analyze. Table 2 samples physical and mathematical rules.

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

Table 1: Scientific semantic properties analyzed by the procedure, including sample equations and number of parsers.

Mathematical Quantity	Physical Quantity	Physical Quantity
$1 \Leftarrow q / q$	$F \Leftarrow m * A$	$^{\circ}C \Leftarrow ^{\circ}K - 273.15$
$0 \Leftarrow q_1 - q_2$	$p \Leftarrow F / \text{area}$	$^{\circ}F \Leftarrow 1.8 * ^{\circ}C + 32$
$\Delta q \Leftarrow q_1 - q_2$	$W \Leftarrow F * \text{length}$	$E \Leftarrow F / q$
$\Sigma q \Leftarrow q + q + \dots$	$E_k \Leftarrow \frac{1}{2} * m * U^2$	$V \Leftarrow I R$
$2q \Leftarrow q_1 + q_2$	$e_k \Leftarrow \frac{1}{2} * U^2$	$Pr \Leftarrow C_p \mu / k$
$\Delta^2 q \Leftarrow q - 2q + q$	$R_u \Leftarrow k * N_A$	$\text{Reynolds} \Leftarrow \rho * U * \text{length} / \mu$
$\partial q / \partial x \Leftarrow \Delta q / \Delta x$	$R \Leftarrow R_u / \text{Mol. wt.}$	$U_{\theta} \Leftarrow r \Omega$
$\partial^2 q / \partial x^2 \Leftarrow \Delta^2 q / \Delta x^2$	$R \Leftarrow C_p - C_v$	$\text{Circum} \Leftarrow 2 \pi r$
$\partial q / \partial y \Leftarrow \partial q / \partial x * \partial x / \partial y$	$\gamma \Leftarrow C_p / C_v$	$\text{vol} \Leftarrow \text{length} * \text{area}$
$\nabla \cdot \mathbf{q} \Leftarrow \text{expression}$	$c^2 \Leftarrow \gamma * p / \rho$	$\text{area} \Leftarrow \text{length} * \text{length}$
$\nabla \times \mathbf{q} \Leftarrow \text{expression}$	$p / \rho \Leftarrow R * T$	$M \Leftarrow U / c$
$\mathbf{q}_1 \times \mathbf{q}_2 \Leftarrow \text{expression}$	$h \Leftarrow e_1 + w$	$\partial m / \partial t \Leftarrow \rho * U * A$

Table 2: A sampling of mathematical and physical expert parser rules.

2.2 Programming language emulation

Semantic analysis of a code depends on more than scientific and mathematical properties; programming language semantics are important too. Consequently, code execution is simulated so that the analyzed properties (Table 1) are propagated faithfully from read statements through array storage and retrieval, through iterative and conditional statements, through external variables, and to and from subroutine calls—all in call tree order. The parser paradigm of section 2.1 does not readily apply here; specialized coding is required to handle each of these functions.

2.3 User Interface

The GUI gives the user control of the semantic analysis and displays the user's code as well as the analysis results (Figure 1). The analysis implements intermediate result storage, that is, when the user selects text from the displayed code, variable properties from *that* point in execution are displayed¹. This storage of expression and array properties for display increases memory requirements. A dictionary provides definitions of technical terms. Additionally, users may search for semantic concepts, navigation tools assist in discovering results, and equations are automatically typeset.

3. Current Status

This work has emphasized development of expert parsers and delayed program emulation improvements. Expert parsers were thought to be novel, progress limiting technology, while program emulation was thought to be mundane. The current status of the expert parsers and emulation functions reflects this strategy, as explained in

¹ For routines executed more than once, only a single execution copy may be retained.

the first two sections. In the last section, numerical recognition metrics gauge the tool's overall development.

3.1 Status of expert parsers

Using parsers to recognize equations has been successful. The parsers represent equations concisely and are computationally efficient, and recognition is robust. Parsers have some limitations [2, 3], but on balance they are successful.

The most mature experts are dimensional and vector analysis parsers since they involve a small set of semantic rules ($A \otimes B$ simplifies to C), and a small set of operands or symbols—($M, L, T, ^{\circ}T, \dots$) and ($x, y, z, r, \theta, \phi, \dots$) respectively. In comparison, the unit and grid location analysis experts involve a small set of semantic rules but a large set of operands—from inches, watts, and lumens to carats, chains, and cubits; conversion constants must also be known (i.e., 0.0254 m/in). Completion requires the addition of operands and conversion constants to tables.

The physical equation experts have a vast set of semantic rules (Table 2) and operands—approximately grouped into scientific disciplines; however, considerable experience and confidence has accumulated. Kinematics, dynamics, and gasdynamics expert parsers are well developed; chemistry and fluid equation parsers are partially developed; electromagnetic, nuclear, structural, geophysical, and astrophysical experts are undeveloped. Devising parsers for physical equations has been straightforward, and the world of physical equations may be captured best by community development in an open source project.

Recognizing mathematical equations is a harder problem (Table 2). For example, a difference, $\phi - \phi$ for any ϕ , may be zero or $\Delta\phi$; resolution depends on examining other properties of ϕ . These additional tests (5) make rule development difficult, and the resulting rules are sensitive.

$$\Delta\phi \Leftarrow \phi - \phi \quad (5)$$

{ Additional Tests }

These expert parsers are supported by data structures, adjectives, and Commutative, Distributive, and Associative transformations. This infrastructure is well developed.

3.2 Status of program emulation

Program emulation involves functions that imitate program execution, in particular, transferring properties as the programming language semantics dictate. These functions include assigning a result to a scalar or array variable, resolving an array reference, analyzing loops and their contained statements, analyzing conditional expressions, using external variables, transferring parameters and execution between routines, and executing routines in call tree order.

All these functions, except conditional analysis, have been implemented. Development of conditional analysis has been delayed to concentrate on other issues². Within loops, iterative statements, $A = A \otimes B$, and inductive statements, $A_{i+1} = F(A_i)$, are under-developed.

A difficult problem in program emulation is assigning results to arrays and resolving array references.

```

DO 10 I = 2, NI-1
  A(3,I) = velox
10 CONTINUE
...
var1 = A(1)
var1 = A(3,ii)
var1 = A(3+NI*ii)

```

(6)

On assignment, the array assignment function must deduce the array's structure and store all the entry's properties in the array representation. To resolve an array reference, the array reference function uses the indices to search the array representation. Memory limits force the elements of an array to be represented with only a few entries. The existing functions implement a good algorithm, but it is not mature.

3.3 Performance measurement

The success of this tool has been evaluated by calculating and monitoring several metrics. Execution speed is better than analyzing one thousand lines of code (K loc) per min; observed memory usage is at most 15 Mb / K loc. These results are evidence of reasonable performance in the future. Further, as the number of semantic rules grows, execution time should grow linearly. This is because, theoretically and experimentally, the tool's execution time is linear in the number of parsers. Further, each parser's memory usage is quadratic in the number of rules.

However, the key metric measures program comprehension. In this metric, the program representation is searched for each operation, $A \otimes B$ where $\otimes \in \{+, -, *, /, **\}$; function reference, $f(a)$; and array reference, $A(i,j,k)$. The

² The approach will be to construct, simplify, and propagate truth tables. Memory requirements may increase dramatically.

recognition rate is the fraction of these operations where the property (units, mathematical/physical quantity) is understood.

This recognition fraction is measured for a set of blind test cases and development test cases. The twelve test cases include one-, two-, and three-dimensional CFD codes for turbomachinery problems, an experimental data reduction code, and a chemically reacting fluid flow code. These are typical, practical, scientific and engineering codes. Figure 2 shows recognition over time for unit analysis. For clarity, only development test cases are shown. Physical/mathematical recognition measurements are not presented here because of the recent emphasis on unit analysis and property propagation.

4. Challenges and strategy

The short-term challenges are to improve propagation of properties during simulated execution and to deliver a useful tool. The long-term challenges include improving the physical/mathematical recognition rate and extending the tool's capabilities.

4.1 Short-term challenges

High recognition rates are necessary for a useful tool, and efforts have focused on this problem. Originally, rule omissions and errors in the expert parsers were believed to be limiting recognition. However, unit analysis is mature and should handle all expressions; but unit recognition is low for larger codes (Figure 2). This result implies that program emulation functions are not propagating units and other properties during execution. Additional work has confirmed this observation, and the current strategy is to improve property propagation as reflected in unit recognition rates.

No single error or misconception is causing this propagation problem. Usually, an array reference or array assignment is the point of failure, but the problem originates elsewhere since the array analysis functions are very dependent upon the expert parsers for their analyses. The ultimate cause varies from an outright error, to a rule omission, to an unanticipated language feature like compressed indices (i.e., $A(i+N_i*(j-1))$ instead of $A(i,j)$).

Once a propagation error occurs, the inference sequence is broken and the error cascades through any subsequent code. Consequently, the recognition fraction can be substantially reduced for *all* analyzed properties (Table 1). Resolving this problem should improve all property recognition rates.

In summary, the short-term challenges are:

- improve program emulation,
- implement conditional analysis,
- improve and polish other property analyses to deliver a basic tool.

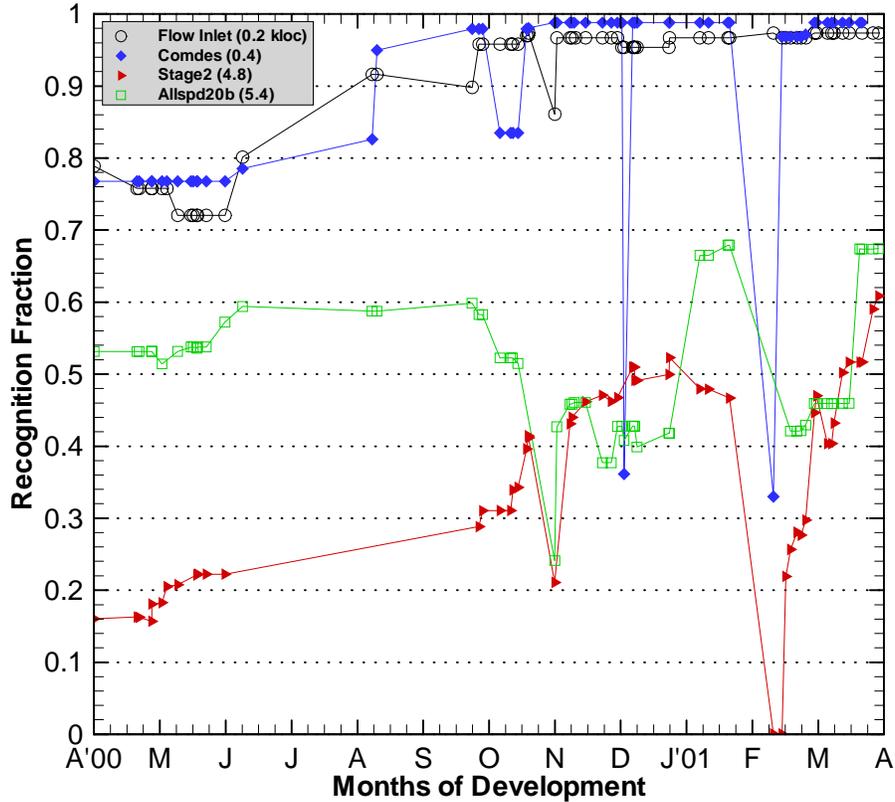


Figure 2: The variation in unit recognition fraction during recent development for the four development test cases. Unit recognition measures propagation of properties during emulation of a code’s execution. The smaller codes, Comdes and Flow Inlet, easily reach high unit recognition. Recent work has focused on improving emulation in the larger, array intensive STAGE2 code, which demonstrates improvement. The Allspd20b test case has remained untouched except for brief visits reflected in sharp upward jumps. The deep valleys in the graph correspond to the development and introduction of new semantic analysis code, including subroutine call tree ordering. Other volatility in the graphs reflects smaller code modifications. For clarity, the eight blind test cases are not shown; they show gradual improvement and range from recognition fractions of 0.20 to 0.55.

4.2 Long-term challenges

Beyond the short-term problem of improving program emulation and unit recognition, there are several important issues. One is improving the physical/mathematical recognition rate. As program emulation and propagation problems are resolved, this metric will certainly increase, but additional work will be necessary. Other potential long-term extensions of the tool include:

- Simulate/analyze inter-code file/message communications
- Analyze C and FORTRAN 90 code,
- Summarize routines as interfaces to reduce simulation,
- Analyze code for performance, programming language information, and parallelization,
- Extend physical rules to other scientific domains,
- Extend to applications other than those of science and engineering,
- Recognize algorithms, not just equations.

5. Similar tools and research

Tools and research already support software development and maintenance. However, there is apparently little overlap with the automated semantic analysis work described here.

Tools exist that assist in software development, including programming language syntax/semantics checking [4], visual program debugging, and formal verification of concurrent systems [5] and concurrent elements of software [6]; but, each is distinctly different from semantic analysis. Techniques also exist that assist in software development, including object oriented software development [7] and verification and validation techniques.

Software synthesis tools exist [8, 9], yet they are again distinct from this work because they concern software synthesis based on high level and not fundamental semantics. Analysis and modification of code parse trees is time-

honored in compiler optimization and is being explored in automated parallelization [10].

Similar work [11, 12, 13] conceptualizes software as a hierarchy of “cliches” or common implementation patterns. Cliches from a library may be automatically recognized from unannotated source code. Implementations of this concept are apparently limited; classical mathematical notation, scientific properties (Table 1), and semantic declarations are not mentioned.

6. Conclusions

The conclusions are threefold: this is a useful tool, work should continue, and the development plan begins with a unit analysis tool.

6.1 A useful tool

The Mars Climate Orbiter’s fatal software error was a unit error [1]—the orbiter transmitted an impulse value in English units while ground-based software expected metric units. This error compromised trajectory modeling, and the navigation team did not recognize the telltale signs. One can realistically expect that this tool could find this error—when mature with extensions for inter-code communications.

However, the MCO software error is the tip of the iceberg. Few errors are so public; most errors are avoided with extensive testing and care, which involve expensive and time-consuming manual effort. Yet, this tool promises to automatically analyze more than units, and all these properties are important for a wide range of scientific codes.

Lastly, this tool is also a discovery tool; it documents the code minutia that is critical for maintenance. Most programmers devote many hours to analyzing unfamiliar legacy code to ensure their changes have only the desired effect. This tool would alleviate this problem.

6.2 Evidence supporting continued effort

There are three technical reasons supporting continued development of this semantic analysis tool. First, expert parsers are a good approach to equation recognition. They robustly and efficiently recognize equations used in practical scientific codes. Second, the tool integrates expert parsers, program emulation, and other techniques, which demonstrate measurable progress in analyzing practical scientific codes. Third, there is apparently no other work being done on this particular problem.

6.3 Development plan

Development of a unit analysis tool would be a useful initial product that can be realized in the short term with limited manpower. The resulting tool could discover

program details needed for maintenance or “wrapping”. Further, this objective focuses effort on lynchpin issues, including program emulation.

Lastly, a unit analysis tool would be a good foundation for an extended product. Existing analyses could be improved, additional analyses could be added, and extensions could be investigated.

7. Acknowledgements

This work was supported by the NASA High Performance Computing and Communications program through the Computing and Interdisciplinary Systems Office (contract NAS3-98008) at NASA Glenn Research Center. Greg Follen was the monitor.

8. Bibliography

- [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] M.E.M. Stewart, S. Townsend, “An Experiment in Automated, Scientific-Code Semantic Analysis,” AIAA-99-3276, June 1999, (<http://www.grc.nasa.gov/WWW/GDS/SAM/paper.pdf>).
- [3] M.E.M. Stewart, “An Experiment in Scientific Program Understanding,” 15th IEEE Conference on Automated Software Engineering, Sept. 2000, (http://www.grc.nasa.gov/WWW/GDS/SAM/paper_ase00.ps).
- [4] R. K. Moniot, “*fnchek*” (<http://www.dsm.fordham.edu/~fnchek>), (Fordham University, New York, 1989).
- [5] G. Holzmann, “The Model Checker SPIN,” *IEEE Transactions of Software Engineering*, 23(5):279, May 1997.
- [6] W. Visser, et. al. “Model Checking Programs,” 15th IEEE Conference on Automated Software Engineering, IEEE Computer Society, Sept. 2000.
- [7] W. Y. Crutchfield, M. L. Welcome, “Object Oriented Implementation of Adaptive Mesh Refinement Algorithms,” *Scientific Programming* 2 (1993) 2, 145-156.
- [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] B. Di Martino, C. W. Keßler, “Two Program Comprehension Tools for Automatic Parallelization,” *IEEE Concurrency*, Jan.-March 2000.
- [11] L. M. Wills, “Automated Program Recognition: A Feasibility Demonstration,” *Artificial Intelligence* 45 (1-2): 113-172 (1990).
- [12] A. Quilici, “A Memory-Based Approach to Recognizing Programming Plans,” *Comm. of the ACM*, 37(5):84 (1994).
- [13] C. Rich, R. Waters, “The Programmer’s Apprentice Project: A Research Overview,” Memo 1004 (MIT AI Lab 1987).