Introduction to the Interactive Semantic Modelling System*

by

Richard E. Fairley
Department of Computer Science
University of Colorado
Boulder, Colorado

Report #CU-CS-038-74 January, 1974

Keywords:

Machine Aided Program Testing
Automated Program Testing
Software Certification
Software Reliability
Semantic Models

* This work sponsored by National Science Foundation Grant #GJ-40046

Introduction

The Interactive Semantic Modelling Systems (ISMS) is a software system designed to facilitate interactive program development, debugging, testing, and documentation. In addition, the system will permit familiarization of prospective users with new programs, and will serve as a pedagogical device for illustrating various programming concepts.

Programs are analyzed by the ISM System, and interactive displays of various program attributes are presented to the user. Programs can be executed statement by statement, either forward or backward in execution time. Names can be associated with values, and control flow with program text, under user control. Also, global summary information concerning program behavior can be displayed. Initially, the system will operate on ALGOL 60 programs. Later extensions to include PASCAL and FORTRAN are planned. The implementation language is PASCAL.

Emphasis is being placed on development of machine aids for interactive debugging and testing of programs. Debugging is the process of finding and correcting program bugs which produce known errors, while testing ascertains the degree to which a debugged program meets its specifications. Thus, program testing is the process of acquiring confidence that a functioning program correctly implements the algorithm(s) on which it is based; i.e., that the program conforms to its documentation. Testing and debugging are related activities in that program bugs are often discovered during the testing phase.

Exhaustive program testing is neither feasible nor desirable. Thus, in general, testing cannot demonstrate the correctness of a computer program. However, properly conducted tests can increase the level of confidence in a given program to the point that it can be certified.

Certification is an authoritative endorsement that the program meets certain standards of quality. According to Cowell and Fosdick [1], the process of certification should include examination of:

1. Completeness of program documentation
2. Performance of the program relative to its documentation
3. Comparison of the program with others of the same type
4. Adequacy of continuing maintenance and support

Thus, a certified program is one that has been thoroughly tested and is supported by the certifying agency.

Formal correctness proof [2], structured programming [3], and chief programmer teams [4] have all attracted interest as techniques to improve the quality of software. The adoption of these techniques will not invalidate the need for systematic program testing; testing will still be necessary to verify the behavior of programs in their operating environments.

The major goal of the ISM System is to provide machine aids to facilitate automated program testing and certification.

ISM System Concepts

Computer programs are characterized by a static syntactic structure and a dynamic runtime structure. It is thus natural to categorize the information content of a computer program as static information and dynamic information. Static information is extracted by performing a syntactic analysis of the program text, and dynamic information is collected by executing the program.

Static analysis can provide the following information:

-- mode and type of identifiers

-- usage of identifiers

-- statement types

-- basic blocks

-- control paths

-- input/output variables

-- graph structure

Dynamic analysis provides information collected during execution of the program. Dynamic information includes:

-- variable range summaries

-- statement execution counts

-- branch execution counts

-- control flow trace

-- variable trace

-- timing histograms

-- assertion checking

-- control flow traceback

-- variable dependency traceback

Dynamic information can be collected for a single execution, or accumulated over a number of program executions.

The ISM System is designed to collect, analyze, and display dynamic information. In addition, some static information is collected to support the dynamic analysis. Information is collected and placed into a data base prior to and during program execution. The user interacts with the ISM data base after the program has terminated, rather than interacting directly with the executing program.

The three major components of the ISM data base are:

-- an identifier table

-- a program model

-- one or more execution histories

In addition, the original source text, a compressed version of the source text, and a textual cross reference table are maintained.

The program being investigated is preprocessed prior to compilation and execution. During preprocessing, the identifier table and program model are constructed, and subroutine calls are automatically inserted into the program text. During execution, the invoked subroutines collect the execution history. The program is thus "self-metric" in that it collects information concerning its own behavior.

The various components of the ISM data base are interfaced to permit the association of names with values, and control flow with program text. The entire history can be searched to collect global information. Thus, ranges of variables can be obtained, assertions about program behavior can be checked, variable and control flow traces can be accomplished, and statement execution counts can be obtained by interrogating the data base. Local information can be obtained by

aligning the history pointer to a particular position in the program model and examining the state of the computation. Information is recorded into the history to permit interpretation of the program model either forward or backward in execution time. Thus, execution time can be reversed in order to determine how a particular computational state was influenced by previous states.

A block diagram of the ISM System is presented in Figure 1, and the interrelationships among the various components of the ISM data base are illustrated in the following example.

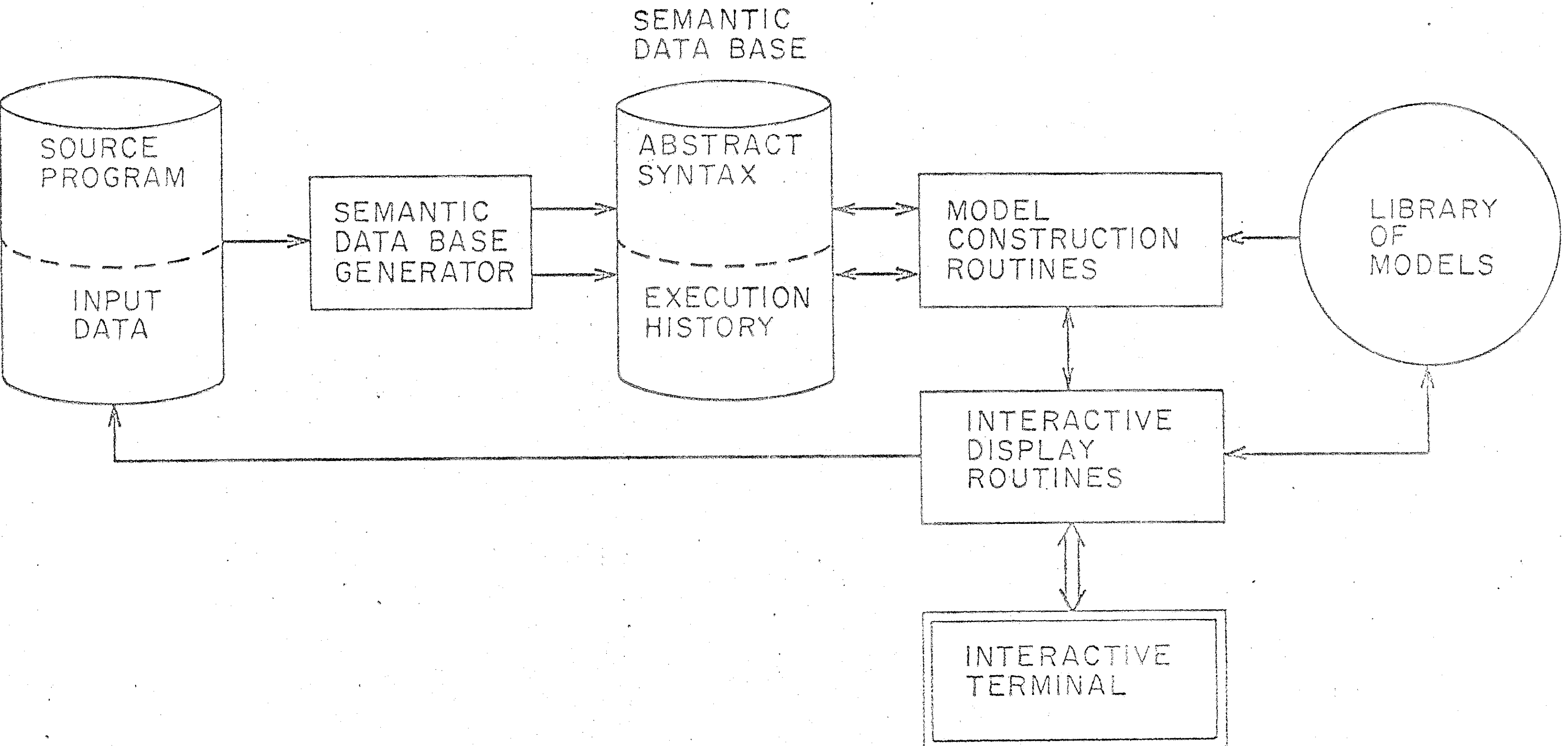

FIGURE 1. Schematic Overview of the Semantic Modelling System

An Example

Consider the following ALGOL 60 segment:

```
                                              line #
 L: A[K]:=K + 1;                                 1
    for I:=1 step 1 until N do                   2
        begin                                    3
        if A[I] = K then goto L;                 4
        K:=K + 1                                 5
        end                                      6
```

the identifier table contains the following information:

Symbol #	Name	Attributes
1	L	label
2	K	integer
3	I	integer
4	A	one dimensional integer array

the program model has the following form:

Model #	Entry type	Source pointer	Index	Remark
1	assignment	3	4	Symbol table # of A
2	ss-assgnmt	5	2	Symbol table # of K
3	for	12	9	Model # of end-for
4	iter-assgnmt	13	3	Symbol table # of I
5	if	22	8	Model # of end-then
6	goto	30	1	Model # of stmt L
7	end-if	32	5	Model # of if stmt
8	assignment	33	2	Symbol table # of K
9	end-for	38	3	Model # of for

The source pointer is an index into the compressed version of the program text, which has all non-significant blanks deleted. A cross reference table of the form (source pointer, line number) provides the mechanism for association of model entries with the original program text. The index field is a pointer either to an identifier table entry, or to another model entry. The remarks column describes the function of each index entry.

Assuming K = 1, and A[2] = 2 initially, the execution history will contain the following value and control flow information:

2, 1, outfor, infor, 1, false, 2, infor, 1, infor, 2

true, gotofrom 6, 3,2, outfor, infor, 1 -------

The history entries are used to interpret the model by matching the first history value with the first model entry and using the history entries to interpret the model as follows:

History entries	Remarks
2	variable #4 in identifier table updated, value = 2
1	variable #2 used as subscript, value = 3
outfor, infor	for loop entered
1	iteration variable (#3) initialized, value = 1
false	skip to end-then in model (entry #8)
2	variable #2 updated, value = 2
infor	counted at end of for loop
1	iteration variable value = 1
infor	return to start of loop (entry #3)
2	iteration variable (#3) updated, value = 2
true	continue with entry #6
gotofrom6	gotofrom6 used for backward execution, next model entry #1

History entries	Remarks
3	variable #4 updated, value = 3
2	variable #2 used as subscript, value = 2
outfor,infor	for loop entered
1	iteration variable (#3) initialized, value = 1

The outfor,infor combinations and the gotofrom entry permit backward interpretation of the program model (backward execution of the program). Starting with model entry #4 and the last history value (I:=1), backward interpretation proceeds as follows:

History entries	Remarks
1	iteration variable initialized, value = 1
infor,outfor	loop exited, next model entry is #2
2	variable #2 used as subscript, value = 2
3	variable #4 updated, value = 3
gotofrom6	next model entry is #6
true	next model entry is #4
2	iteration variable updated, value = 2
infor	control is at start of for loop
1	iteration variable, value = 1
infor	control is at end of for loop
2	variable #2 updated, value = 2
false	next model entry is #4
1	iteration variable updated, value = 1
infor,outfor	loop exited, next model entry is #2
1	variable #2 used as subscript, value = 1
2	variable #4 updated, value = 2

The instrumented source program which produces the execution history has the following form:

```
    L:  A[K]:=K + 1;
        assign(A[K]);
      ssassign(K);
        outfor;
        for I:=1 step 1 until N do
            begin
                  infor(I);
                  begin if test (A[I] = K) then
                            begin gotofrom(6);
                                  goto L
                            end
                     K:=K + 1
                  end
                  infor(I);
            end
        outfor;
```

The number of statements in the instrumented program will exceed the number in the original source program by a factor of 2 to 3.

Due to convenience and efficiency considerations, the internal representations of the various ISM data base components are somewhat different than portrayed in the preceding example. However, the external appearance of the data base is as described. A subsequent paper will describe the internal organization of, and method of access to, the ISM data base.

Features of ALGOL 60 which were not illustrated in the preceding example include:

-- block structure

-- scope of names

-- parameter passing

-- environment of procedure evaluation

-- recursive procedures

-- dynamic arrays

-- switch variables

-- multiple assignment statements

Techniques for instrumenting these and other ALGOL 60 constructs will be discussed in a forthcoming report.

The ISMS Preprocessor

The ISMS Preprocessor is used to read in and compress the source program, to produce the original source/compressed source cross reference table, and to generate the identifier table, the program model, and the instrumented source program. The identifier table, program model, and instrumented source program are prepared in three different passes over the source text, using three different parsers.

The parsers are generated by a parser generator program from a modified BNF grammar. Semantic actions can be invoked upon successful matching of a production rule with the input string, and output strings can be automatically generated. The parsers produced by the parser generator are top-down recursive descent parsers, in which syntactic backtracking is automatically handled.

The modified BNF notation for the grammars is described in BNF as follows:

(1) <MBNG> :: = <PRODUCTION> '.'

A modified BNF grammar consists of an arbitrary number of production rules (zero or more), terminated by the literal '.'.

(2) <PRODUCTION> :: = <LHS> '::=' [<METAELEMENT>]
[<FOLLOWING ALTERNATIVE>]
[<EOP>]

A production is of the form <Left Hand Side> ::= an arbitrary number of metaelements followed by an arbitrary number of alternatives, terminated by an End of Production.

(3) <LHS> :: = '<' ------ '>'

The Left Hand Side of a production is defined as the sequence of symbols appearing between the brackets.

(4) <EOP> :: = ';'

Productions are terminated by a ;

(5) <FOLLOWING ALTERNATIVE>::= <NEW ALTERNATIVE> [<METAELEMENT>]

(6) <NEW ALTERNATIVE> ::= '|'

Alternatives are delimited by a | .

(7) <METAELEMENT> ::= <TERMINAL STRING>|<NONTERMINAL NAME>|
<OUTPUT STRING>|<CODE STRING>|
<START REPETITION>[<METALEMENT>] <END REPETITION>

(8) <TERMINAL STRING> :: = ' ------ '

(9) <NONTERMINAL NAME> :: = '<' ------ '>'

(10) <CODE STRING> :: = '$' ------ '$'

(11) <OUTPUT STRING> :: = '+' ------ '+'

(12) <START REPETITION> :: = '['

(13) <END REPETITION> :: = ']'

Terminal strings are delimited by quotes, nonterminal names by brackets, code strings by $, and output strings by +'s. A code string is a sequence of statements that is executed upon successful matching of the associated alternative in a production rule. In this manner, semantic actions can be specified, or comment statements concerning semantic actions can be supplied in the grammar. An output string is a sequence of characters that becomes the argument of an output routine. A value is written into a prespecified file when the associated alternative in the production rule succeeds. The START REPETITION and END REPETITION brackets denote an arbitrary number of occurrences of the syntactic unit enclosed in the brackets.

The parser generator accepts sets of grammatical rules whose syntax conforms to the above description, and generates PASCAL code that can be used to subsequently parse and perform semantics on programs whose syntax conforms to those grammatical rules. In effect, the parser generator is very similar to a metacompiler.

The following examples illustrate various modified BNF rules, and the PASCAL code generated in response to those rules by the parser generator.

Example 1: <LHS> :: = <STR1>|<STR2>;

```
Line #

  1       PROCEDURE LHS;
  2       BEGIN IF MATCH THEN BEGIN NEWPHRASE;
  3             STR1;
  4             IF ¬ MATCH THEN BEGIN TRYALTERNATIVE;
  5             STR2;
  6       ENDPHRASE(nnn); END END END;
```

Lines 1 and 2 are generated in response to <LHS> :: = The nonterminal name on the left hand side of the production rule becomes the name of a corresponding PASCAL procedure. The nonterminals STR1 and STR2 are translated into procedure calls in lines 3 and 5. The alternation symbol, |, is translated into line 4. TRYALTERNATIVE is a procedure that sets MATCH := TRUE, and resets the parse string and stack pointers. MATCH is a global Boolean variable which tells whether the current alternative of a production rule has matched up to this point.

Line 6 is generated in response to the end of production marker ';'. ENDPHRASE and NEWPHRASE are procedures that manipulate stack pointers. Each procedure generated has a unique integer (nnn) that can be used to provide a trace of the parse, or can be used as an error diagnostic label.

Example 2: LHS ::= <STR1> ['STR2'];

Line #

```
1       PROCEDURE LHS;
2       BEGIN IF MATCH THEN BEGIN NEWPHRASE;
3             STR1;
4             IF MATCH THEN BEGIN REPEAT
5             TERMINAL('STR2');
6             UNTIL ¬ MATCH; MATCH := TRUE END;
7             ENDPHRASE(nnn); END END;
```

As before, lines 1 and 2 are generated in response to <LHS>::=. Line 3 is in response to the nonterminal name <STR1>. Line 4 is in response to the start of repetition bracket '['. Line 5 is in response to the terminal string 'STR2'. Line 6 is in response to the end of repetition bracket ']', and line 7 is in response to the end of production symbol ';'. TERMINAL is a procedure that matches its argument against the next nonblank input characters.

Example 3: <LHS> ::= 'STR1' <STR2> $ 'CODE' $ +OUTPUT+;

Line #

```
1       PROCEDURE LHS;
        BEGIN IF MATCH THEN BEGIN NEWPHRASE;
2             TERMINAL('STR1');
3             STR2;
4             'CODE'
5       PUTCHAR(OUTPUT);
6       ENDPHRASE(nnn); END END;
```

This example illustrates the use of the <CODE STRING> and <OUTPUT STRING> options in a production rule. 'CODE' is inserted into the generated procedure and provides a mechanism for specifying arbitrarily complex semantic actions. OUTPUT becomes the argument to an output routine. This provides a variety of capabilities; in particular subroutine calls can be automatically inserted into the source program to produce an instrumented source program.

Due to the recursive top-down nature of the parsers, left recursion is not permitted in the production rules of the grammars. Thus, the left recursive definition of an ALGOL 60 identifier as:

<identifier> ::= <letter>|<identifier><letter>|<identifier><digit>

is translated into the following equivalent form:

<identifier> ::= <letter>[<lord>]

<lord> ::= [<letter>] | [<digit>]

where the square brackets denoted arbitrary replication.

There are also certain restrictions on the ordering of production rules, and on the ordering of alternatives within production rules. The implementation of the parser generator requires that non-terminal symbols appear on the left hand side of a production rule before they are used on the right hand sides of production rules. Thus, the grammar is processed in "bottom-up" fashion. In certain case, efficiency considerations dictate a preferred ordering of alternatives within a production rule.

A detailed discussion of the parser generator and the ISM preprocessor will be provided in a subsequent report.

ISMS Program Analysis and Display

The ISM System has been designed to isolate preprocessing and execution of a user's program from the analysis and display of that program's behavior. This allows analysis of global program behavior, and permits backward execution of the program. Also, the analysis and display routines are designed to interface to the ISM database independent of the particular programming language being analyzed. This capability, when combined with the parser generator methodology utilized in the preprocessor should facilitate adaption of the ISM System to a number of different programming languages. A major goal of the project is to determine the flexibility and adaptability of the ISM design.

As illustrated in Figure 1, analysis and display is accomplished by interactive routines that access the data base and present information to the user in a meaningful format. A major aspect of the ISMS research project is to determine what information is useful for program testing, and how to display that information to the user of the system. The display formats are termed semantic models of program execution. Different semantic models will be required to model the many different attributes of computer programs.

A partial list of useful information that might be displayed includes:

-- variable range summaries

-- statement execution counts

-- branch execution counts

-- control flow trace

-- variable trace

-- timing histograms

-- assertion checking

-- control flow trace back

-- variable dependency trace back

-- identifier accessing environments

-- parameter passing

-- environments of procedure evaluations

-- recursive procedure evaluation

Variable range summaries, statement execution counts, branch execution counts, control flow traces, and variable traces can all be obtained by straightforward analysis of the ISM data base. Timing estimates can be collected by inserting subroutine calls into the instrumented source program to record a time reading before and after execution of the statement.

Assertions can be local or global. A local assertion is verified at a particular point in the program, and global assertions are true throughout the entire execution history. A local assertion might be of the form: I < J after statement #10. Global assertions can be used to verify range and monoticity of variables, execution paths, initial and final values of variables, number of loop traversals, etc. Assertions are checked by comparison with the execution history.

Variable dependency and control flow trace backs are provided by backward execution of the program. A particularly valuable technique is the flowback analysis capability described by Balzer [5]:

> This analysis appears in the form of an inverted tree, with the bottom node corresponding to the value for which the flowback analysis was desired. Each node consists of the source-language assignment statement that produced the value, the value itself, and links to nodes at the next level. These nodes correspond to the non-constant values in the assignment statement displayed in the node that links with these nodes. These nodes have the same format as the original and are linked to nodes for all non-constant values used in the particular assignment statement producing their value.

Figure 2 illustrates the use of flowback analysis. This technique will be particularly valuable when used in conjuction with other displays to determine information such as the range of input data required to force execution down a particular path in the program graph.

Parameter passing and execution environments can be displayed as information structure models that evolve with time. The execution history is regarded as a sequence of snapshots, and the displays will consist of these sequences. One of the most appealing information structure models is the Contour Model [6]. The Contour Model is an intuitive, pictorial representation of block structured processes (e.g., ALGOL 60 programs) which emphasizes identifier accessing environments and transfer of control mechanisms.

Another interesting candidate for an information structure model is the Vienna Definition Language representation of a computation, which models the computation in terms of control, environment, denotation, unique name, and dump components [7]. Variations of these models will be investigated as semantic models of flow of control mechanisms in computer programs.

In addition to the models described, structural models of the source text will provide formatted displays of the syntactic structure of the program. The user will be able to request or suppress source text at a certain level of program structure to obtain a structural overview of the program. For instance, selective suppression of the bodies of procedures, IF--THEN--ELSE statements, compound statements, iterative loops, and recursive procedures are possibilities for structural displays.

An important computational model which is <u>not</u> within the scope of the present investigation is the Data Sensitivity Model. A data sensitivity model would

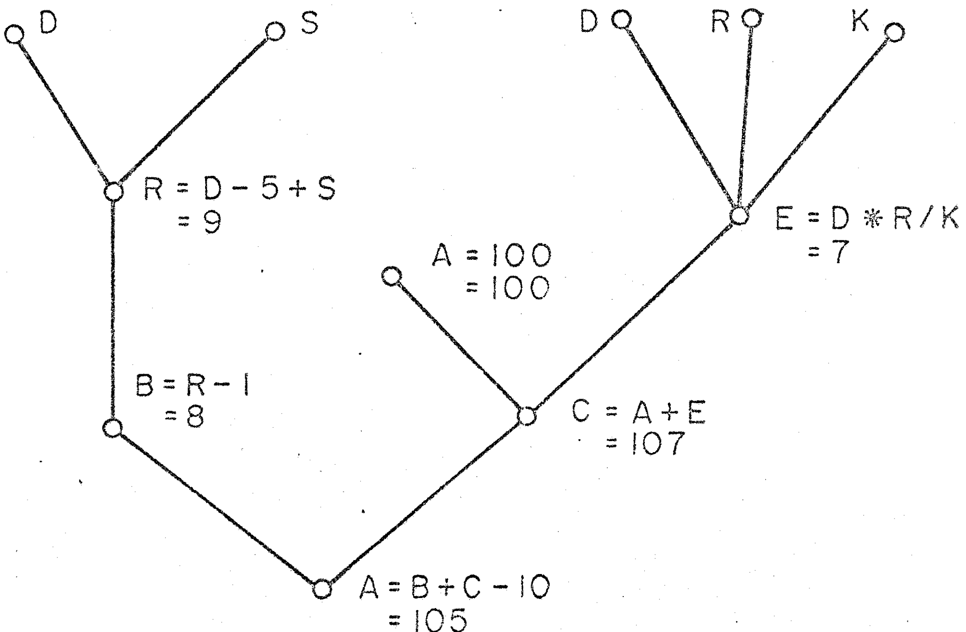

FIGURE 2. <u>An Example of Flowback Analysis</u>

(from Reference 5)

display the effects of input data inaccuracies and finite word length on the computation being performed by tracing the numerical significance of the computation. A system of this type has been described by Bright, Calhoun, and Mallory [8]:

> A FORTRAN program processed by the system is executed in an artificial arithmetic, in which every arithmetic step produces, in addition to the numerical result, an estimate of the number of significant digits in that result . . . . The user may specify initial accuracy of data; may select any part or parts of a program for execution in the error-indicating mode; and may request accuracy of any variable quantity at any point during program execution.

A similar technique could be integrated into the ISM System by saving additional information in the data base. A display of source statement, values computed, and errors accumulated in the computation could then be displayed as an aid in determining the numerical properties of the program. Combining this capability with the flowback analysis technique would provide a powerful and versatile program testing tool.

Summary

To summarize, the Interactive Semantic Modelling System is a batch-interactive system for collecting, constructing, and displaying semantic models of program execution. A variety of software tools for program testing, documentation verification, and pedagogical purposes are being developed.

Program modelling and history-gathering occur during preprocessing and execution of the program in the batch mode, and result in creation of a data base that contains a model of the program and a history of its execution. Modelling tools are used to access the data base to construct models of the program, and to present displays of these models to the user at an interactive console. The primary advantages of the data base approach are:

1. The program is executed in the actual usage environment.
2. Experimentation with a variety of modelling aids is facilitated.
3. Batch efficiency and expertise are combined with interactive flexibility.
4. The modelling tools are language independent.

The primary disadvantage is that the user cannot directly interact with his executing program. However, we envision a short response time for re-execution of the same program with new input data, based on a previous system of similar design [9].

The user of the ISM System will use a command language to selectively collect or suppress history information. Details of the command language will be provided in a separate report.

Major goals of this project are to consolidate existing modelling techniques, and to investigate new modelling and testing aids which cannot be envisioned

until the system is constructed and actual experimentation begins. Thus, a flexible, generalized data base approach has been utilized to facilitate implementation of existing models, and experimentation with new models.

In addition to investigating the utility of various program models, attention is being given to the human factors involved in man-machine system design. A major feature of the system will be the ability to alternate between models in a natural and straightforward manner. This is essential if the user is to gain new viewpoints and insights into the behavior of his program.

Applications arising from the ISMS project may include:

1. Development of procedures and software tools for systematic machine aided program testing.
2. Development of a programming concepts and program testing laboratory for student use.
3. Development of a program testing compiler which would provide user requested information concerning program behavior.
4. A prepackaged ISM System to aid in testing of computer programs in the environment of a large, shared resource computing network, such as the ARPA Network.
5. Design specifications for a programming language to facilitate program validation.

Bibliography

1. Cowell, Wayne R. Ed., "Proceedings of the Software Certification Workshop", August, 1972.

2. Floyd, Robert W., "Assigning Meanings to Programs", Proceedings of Symposia in Applied Mathematics, Vol. 19, American Mathematical Society, 1967.

3. Dijkstra, Edsger W., "Notes on Structured Programming", T. H. Report 70-WSK-03, Technological University Eindhaven, April, 1970.

4. Baker, F. T., "Chief Programmer Team Management of Production Programming", IBM Systems Journal, Vol. 11, No. 1, 1972.

5. Balzer, R. M., "EXDAMS -- Extendable Debugging and Monitoring Systems", RAND Memorandum RM-5772- ARPA, April, 1969.

6. Johnston, J. B., "The Contour Model of Block Structured Processes", ACM SIGPLAN Notices, Vol. 6, No. 2, February, 1971.

7. Wegner, Peter, "The Vienna Definition Language", ACM Computing Surveys, Vol. 4, No. 1, March, 1972.

8. Bright, H. S., and others, "A Software System for Tracing Numerical Significance During Program Execution", AFIPS SJCC Proceedings, Volume 38, 1971.

9. Fairley, R. E., "A Batch-Compatible Interactive System for Dynamic Modelling", Ph.D. Dissertation, UCLA, 1971.