A NEW METHODOLOGY FOR GENERATING TEST CASES FOR A PROGRAMMING LANGUAGE COMPILER

by

Daniel M. Berry

Computer Science Department
University of California
Los Angeles, California 90024

THE PROBLEM

A compiler writer writes a compiler for a programming language *L*. He/she wants to test the compiler thoroughly *before* releasing it to the market, so that no customer of the compiler will find any bugs.

The goal of testing is to find *all* bugs that exist in the compiler. However, finding all bugs is an extremely hard thing to do if not impossible. Therefore, the compiler writer will be satisfied if testing finds all bugs that will be found by customers. In other words, if a bug that is not found by testing is never found by a customer, then the testing is considered satisfactory.

The point is that the only errors that will cause any problems are those that get caught by a customer after delivery of a supposedly debugged product. If there is an error in a combination of features that is never used, then this error will never be found. Since the resources, human, time, and money, to find errors are limited, these resources should be spent to find errors that will be found by a customer if they are not found during testing before release.

A bug or an *error* in a compiler is either

1. an incorrect handling of a correct program either at compile time or at run time or

2. a failure to detect an illegal program as illegal either at compile time or at run time as the case may be for the illegality.

Thus, an error in a compiler may be observable during its execution or during the execution of the code it generates.

Testing is done with a collection of test programs. A *test program* for a compiler for *L* is a possibly incorrect program of *L*, for which the correct response of the compiler and generated code is known and which is to exhibit at least one possible error of the compiler. An error is said to be *exhibited* by a test program if the test program causes the compiler to behave in the erroneous manner defined by the error.

The collection of test programs must be at least large enough to exhibit all the errors

that any user will find. On the other hand, too large a collection is not satisfactory either. For each test program, the compiler's and generated code's response must be determined *before* the test runs, and the results of the test run must be compared with the expected results. The larger the collection, the more such work is involved.

The ideal is to have exactly those test programs which serve to exhibit all errors of the compiler that will be found. Each error should be exhibited by no more than one program.

The *coverage* of a test program is the set of uses and misuses of features that are included in the program. Thus, one desires as small a set of test programs which nevertheless has sufficient coverage to exhibit all the errors of the compiler that will ever be found by a customer.

Since it is impossible to know ahead of time what are the errors of the compiler and what of these will be found by a customer, this ideal is impossible to achieve. In addition, it is extremely difficult to avoid having more than one test program exhibit the same error. Therefore, the usual strategy is to have a collection of test programs which thoroughly exercises the language, one which uses every feature of the language in all of its variations in the hopes that all of the errors or at least those that will be found are found by the programs of the collection.

THE CURRENT APPROACH TO TEST CASE GENERATION

Although there is a modest amount of literature on program testing, e.g, [Mye 79, Yeh77, IEEE 80, ACM 82], there appears to be a dearth of information on compiler testing. The most complete survey I have found [SC 80] does not go into too much detail about the generation of test cases. Thus the discussion below represents my best guess as to what seems to be done now.

The usual approach to generating a set of test programs begins by first identifying the individual features $f_1, \ldots, f_n$ in the language L. One then identifies for each such feature f_i,

1. typical cases,

2. boundary cases, and

3. misuses of the feature.

For example for the feature "integer variable", the typical case might be that

> the variable has the value 75893.

The boundary cases might be that

> the variable has the value 0,
> the variable has the value maxint,
> the variable has the value minint,
> the variable has the value 1, and
> the variable has the value -1.

The misuses might be that

the variable is not initialized,
the variable has overflowed, and
the variable has underflowed.

Thus for each feature f_i, one identifies some number m_i of cases, $c_{i,1}, \ldots, c_{i,m_i}$. Typically m_i will not be larger than 10.

One then generates a set of programs with coverage of all of the cases of all of the features. For each program, it is necessary to predict the compiler's response and the result of running the generated code if any. Care must be taken that cases involving fatal errors in the compiler or the generated code, i.e., for which the response is termination, are the last to be executed in their respective programs.

In order to predict the response of the compiler or generated code to a program, it is very nice if an existing stable compiler for the language L can be used. Then generation of responses is automatic and is not subject to human error.

THE DRAWBACKS OF THE CURRENT METHOD

This method produces a large number of test programs containing a large number of test cases. However, this number is not unmanageable for the typical language with at most 100 features and 10 cases per feature. Such a language requires a collection of programs covering at most 10,000 cases.

Unfortunately, these test cases are not thorough enough. One should also test combinations of features. At this point, the number of test cases gets out of hand. Even if only the *pairs* of features are tested, for the above described language, there are at most 100,000,000 additional cases to deal with, i.e., to determine their effect on the compiler, to determine the results of their generated code, and to compare these effects and results with those of the actual tests. Even with all these additional cases, the tests may not be thorough enough. One should consider triples, quadruples, etc.

Thus, the problem with the current approach to compiler testing is that in order to have sufficient coverage to exhibit all errors that will be found, the number of test cases to consider gets out of hand in a combinatorial explosion.

A PROPOSED NEW METHOD

This paper offers a new approach to test case generation. The approach has the goal of generating cases with sufficient coverage to exhibit all errors that will ever be found while at the same time keeping the number of cases manageable. The approach borrows an old idea in the design of architectures for the interpretation of high level languages. As one is designing an interpreter for a high level language, one is trying to optimize the run time performance of the interpreter for those programs that do get presented to the interpreter. It does not matter how poor the architecture is for those programs that no one writes. Therefore, one collects statistics on the way the programming language is used. The usual kinds of statistics are frequencies of occurrences of each of the cases of each of the features in the language. Here, the frequencies help to identify the

cases. For example, if the integer 1 occurs frequently (as it does) then the integer 1 is a case of the integers. One collects both static and dynamic frequencies. The static frequencies tell how often each case appears in program texts and suggest which operation codes should be made shorter. The dynamic frequencies tell how often each case gets executed and suggest which instructions should be optimized for speed.*

It is also useful to identify sequences of instructions that appear together in the same sequence very often. Such a sequence is a candidate for merging into a single instruction that does the effect of the entire sequence with no more than one instruction fetch. For example, one very frequent sequence is that for the statement

$$x:=x+1.$$

This suggests that an increment-variable-by-1 instruction might be a useful instruction to have. Thus it is useful to collect data on frequencies of pairs, triples, quadruples, etc. of cases of features.

The proposed method for generating test cases is as follows:

1. Identify the features of the language and their cases as in the traditional method.

2. Form test programs from these cases in the traditional manner.

3. Collect statistics showing the frequencies of pairs and triples of the cases.

4. Take the pairs that constitute 99% of the occurrences among the pairs and take the triples that constitute 99% of the occurrences among the triples.

5. Form test programs from the taken pairs and triples of cases.

6. Determine the expected results for these test programs and run them through the compiler being tested.

WHY THE NEW METHOD SHOULD BE VIABLE

In order for the proposed method to be viable it must be that

1. it generates a set of test cases with sufficient coverage to exhibit all errors that will be found by a customer, and

2. it does not cause a combinatorial explosion in the number of test cases generated.

There is ample evidence that using statistics about the way the language L is used to generate test cases produces test cases that have these properties.

---

* Experience shows that the ranking of cases in the static and dynamic frequencies tend to be the same [Bry 80]. That is the code inside loops looks the same as that outside loops. Thus it usually suffices to collect the static data, which are also much cheaper and easier to obtain.

It has been observed by a number of authors [Bry 80, Erl 79, Mis 80] that language-use statistics are remarkably insensitive to variations in:

1. language. Measurements have been taken for ALGOL 60, PL/I, FORTRAN, SPL, SAL, XPL, and SIMULA 67 (See below for references).

2. expertise of programmer. Measured populations include beginning programming students, advanced computer science students, professional programmers both with and without training in structured programming, and physicists using the computer as a tool.

3. application of the program. Measured applications include educational computer science programming projects, commercial systems, and scientific research calculations.

Figures 1 through 6 reproduce tables given in [Erl 79] reflecting data from [Knu 71, Els 77, Wor 72, Wic 73, Tan 78, AW 75, Akk 67]. These data are over various languages, different programmer expertise, and different application areas. Some of these show static frequencies, others show dynamic frequencies, and others show both. The FORTRAN data by Knuth include professional scientific programmers as well as computer science students and faculty. Elshoff's PL/I data include professional commercial programmers before and after training in structured programming. Wortman's data are comprised of beginning programming students doing nonnumerical problems. Wichmann's ALGOL 60 data concern physicists doing scientific calculations. Tanenbaum's SAL data have advanced computer science students doing system programming. Alexander and Wortman's XPL data are about professional system programmers, and Akka's Atlas Autocoder data concern programs written by himself, a computer scientist.

In all of these samples, the approximately 10 most frequent statement types or instructions constitute more than 85% of the occurrences both statically and dynamically. In addition, one notes that the most common statement type or set of instructions is the assignment or the set of instructions needed for the assignment. This statement or set of instructions constitutes about half of the portion of each sample statically or dynamically. An examination of the portion of each sample which constitutes the 85% most frequent of the occurrences statically or dynamically shows *nearly the same set of statements and instructions.*

In addition, Misherghi [Mis 80] has found that SIMULA 67 usage falls into the same patterns despite the fact that SIMULA 67 has some features, i.e., classes and quasi-parallelism, that appear in none of the above described languages. Amazingly, this is true even though Misherghi's sample includes many programs for a class in SIMULA 67 in which the students were required to use these features to solve problems designed to illustrate the use of these features. In other words, it may very well be that in an algorithmic language with special purpose features, the use of special features may be completely overshadowed by the use of the basic algorithmic language feature.

Thus, it seems clear that the coverage of the set of features generated by the proposed method should be sufficient to exhibit all those bugs that will be found by any customer, no matter his/her language, expertise, and application area.

All of the data of the previous section may be characterized by the fact that the same

few features or instructions form the vast majority of the features or instructions used. This fact is not surprising given Zipf's Law of Linguistics [Zip 49]. Zipf's Law suggests that in any human language, the frequency of an atom is inversely proportional to its frequency rank, as shown in figure 7.

In the language of letters appearing in a written text, the atoms are letters. In the language of words appearing in spoken or written text, the atoms are words. Thus, in a high level language, the atoms are the syntactic tokens, and in a machine language, the atoms are the op-codes and operands of instructions. With these latter interpretations of the concept of atom, the data of the above paragraphs clearly shows that algorithmic high level, intermediate, and machine languages follow Zipf's law. In addition the data of [Wor 72] shows that the language of consecutive pairs of instructions follows Zipf's law. Out of the approximately 1600 possible pairs of instructions, about 40, i.e., 2.5%, constitute 90% of the pairs that were actually used.

Thus there is ample reason to believe that a selection of test cases constituting 99% of the cases occurring in programs will yield a test case collection that is small enough to be manageable.

TESTING THE PROPOSAL

It is necessary to verify that the proposed methodology is feasible and effective, that is that the method of generating test cases yields a manageable set of test cases with sufficient coverage. It is desirable that the verification be done without risking a real on-going compiler development project. It is proposed to take a compiler construction project that was done in the past and for which accurate records of the nature of bugs found during testing and after delivery were kept. Statistics on the way the language is used must be collected. These statistics are then used to construct test cases in the proposed manner. The size of this set of test cases is compared with that of the test case set used in the actual compiler development. In addition, each error that was found during and after the actual compiler development is examined to determine if it would be exhibited in the statistically generated test case set. It is then possible to evaluate the effectiveness of the proposed methodology in producing manageable but covering test case sets.

REFERENCES

[ACM 82] *Computing Surveys* 14:2 (Junn 1982).

[Akk 69] Akka, D.S., *A Quantitative Comparison of Efficiencies of Compilers*, Master of Science Thesis, Department of Computer Science, Victoria University of Manchester (October 1967).

[AW 75] Anderson, W.G. and D.B. Wortman, "Static and Dynamic Characteristics of XPL Programs", *Computer* 8:11 (November 1975).

[Bry 80] Berry, D.M. "High Level Language Computer Architecture: An Overview and Some Principles", *Proceedings of International Seminar on Computer Science*, Santiago Chile (August 1980).

[Els 77] Elshoff, J.L., "The Influence of Structured Programming on PL/I Program Profiles", *IEEE Transactions on Software Engineering* SE-3:5 (September 1977).

[Erl 79] Erlinger, M.A. "Design and Measurement of Implementation Schemes for Retention Storage Management as Utilized in Block Structured Languages", Report UCLA-ENG-7972, Computer Science Dept., UCLA (1979).

[IEEE 80] *Transactions on Software Engineering* SE-6:3 (May 1980).

[Knu 71] Knuth, D.E., "An Empirical Study of FORTRAN Programs", *Software Practice and Experience* 1:2 (April-June 1971).

[Mye79] Myers, G.J., *The Art of Software Testing*, Wiley, New York (1979).

[Mis 80] Misherghi, S.H., "An Investigation of the Architectural Requirements of SIMULA 67", Ph.D. Dissertation, Computer Science Dept., UCLA (1980).

[SC 80] Scowen, R.S. and Z.J. Ciechanowicz, "Compiler Validation — A Survey", National Physics Laboratory, UK (September 1980).

[Tan 78] Tanenbaum, A.S., "Implications of Structured Programming for Machine Architecture", *Communications of the ACM* 21:3 (March 1978).

[Wic 73] Wichmann, B.A., *ALGOL 60 Compilation and Assessment*, Academic Press, New York (1973).

[Wor 72] Wortman, D.B., "A Study of Language Directed Computer Design", Computer Systems Research Group, University of Toronto (December 1972).

[Yeh77] Yeh, R.T. (Ed.), *Current Trends in Programming Methodology*, Volume II, Program Validation, Prentice-Hall, Englewood Cliffs (1977).

[Zip 49] Zipf, G.K., *Human Behavior and the Principle of Least Effort*, Addison Wesley, Reading, MA (1949).

# OF PROGRAMS	440	25	120	34	959
# STMTS PER PGM	479.5	380	853	593	43.6
LANGUAGE	FORTRAN LOCKHEED	FORTRAN STANFORD	PL/1 NSP	PL/1 SP	SPL WORTMAN
STATEMENT					
ASSIGNMENT	41.0	51.0	41.2	33.7	63.9
IF	14.5	8.5	17.8	15.6	10.0
GOTO	13.0	8.0	11.7	2.8	.3
LOOP	4.0	5.0	7.2	9.5	14.8
CALL	8.0	4.0	2.0	8.2	2.7
PROC/SUBRTNE	1.0	1.0	.2	1.7	1.4
RETURN	2.0	2.0	.1	.1	.3
BEGIN/BLOCK			.1	.3	.2
END	1.0	1.0	7.5	11.6	
WRITE	4.0	5.0	2.6	1.1	
FORMAT	4.0	4.0			
DATA	2.0	.3			
COMMON	1.5	3.0			
CONTINUE	5.0	3.0			
DECLARATION	2.5	1.9	6.3	7.1	4.9
SUMMARY % OF STMTS SHOWN	103.5	97.7	96.7	91.7	98.5
% OF STMTS USED IN AVERAGES	80.5	76.5	79.2	69.8	91.7

STATIC SOURCE STATEMENT STATISTICS
Figure 1

LANGUAGE - WORTMAN = SPL

LANGUAGE - WICHMAN = ALGOL 60

STATEMENT	WORTMAN	WICHMAN
NAME	28.3	20.8
EVAL	23.7	25.4
LINE	8.7	
SUBS	6.2	6.3
CALL	4.3	3.9
SWAP	4.1	
POP	2.7	
BIT	2.5	1.5
STORE	2.3	4.9
CRET	1.7	
PARAM	1.6	3.1
DOTEST	1.4	1.5
ADD	1.2	3.1
DOINCR	1.2	
DOSTORE	.2	.2

COMPARISON OF WORTMAN AND WICHMAN
DYNAMIC
MACHINE LEVEL FRAGMENT PERCENTAGES

Figure 2

LANGUAGE = ATLAS AUTOCODER

# OF PGMS	66	52	44	38	
# OF STMTS/PGM	<1K	>1K <10K	>10K <100K	>100K	
STATEMENT	GROUP 1	GROUP 2	GROUP 3	GROUP 4	WEIGHTED AVERAGE
ASSIGNMENT	38.5	50.8	45.8	49.8	45.4
CONDITIONAL	14.9	17.0	20.3	21.7	17.9
GOTO	9.5	8.9	10.2	6.7	8.9
CALL	3.6	3.3	3.3	2.0	3.1
LOOP	3.1	1.9	2.2	1.9	2.3
DECLARE	1.9	3.3	1.6	2.2	2.0
BEGIN	3.3	.06	.01	.001	.12
# OF EXPR	133	140	139	120	133.6

AKKA'S DYNAMIC SOURCE STATEMENT STATISTICS [AKKA67]

Figure 3

INSTRUCTION	STATIC	DYNAMIC	SEMANTICS
NAME	30.3	28.3	Creates indirect address
EVAL	20.7	23.7	Fetches value to ds
LINE	10.0	8.7	Source program line number
SWAP	4.8	4.1	Interchange top 2 ds entries
POP	4.8	2.7	Delete top ds entry
PARAM	3.8	1.6	Check procedure parameter
STORE	3.4	2.3	Assign value to variables
CALL	3.1	4.3	Execute program segment
CAT	2.9	.8	String concatenation
SUBS	2.8	6.2	Subscript array variable
ENTER	2.3	1.0	Enter block/procedure
LFUNC	1.4	.2	Pseudo-variable assignment
SCOPEID	1.0	1.0	Provide block/procedure number
ADD	1.0	1.2	Addition
CRET	.9	1.7	Conditional return from segment
CYCLE	.9	1.5	Loop through segment
BIT	.7	2.5	Force to type bit (IF statement)
DOSTORE	.7	.2	Initialize DO loop variable
DOTEST	.6	1.4	Test for DO loop termination
DOINCR	.6	1.2	Increment DO loop variable
SUB	.5	.5	Subtraction
EQ	.3	1.4	Test for equality
END	.3	.2	End block/procedure execution
UNDEF	.3	.2	Create undefined value on ds
DIV	.3	.0	Divide
ALLOC	.2	.0	Allocate array variable
NE	.2	.6	Test for not equal
GT	.2	.5	Test for greater than
MUL	.2	.1	Multiplication
HALT	.2	.0	End program execution
NEG	.1	.3	Arithmetic negation
LT	.1	.4	Test for less than
PWR	.1	.0	Exponentiation
LE	.1	.2	Test for less than/equal
AND	.1	.4	Logical and
GE	.1	.3	Test for greater than/equal
OR	.1	.3	Logical or
GOTO	.0	.1	Branch to label
NOT	.0	.0	Logical not
FREE	.0	.0	Free array storage

WORTMAN MEASUREMENTS FOR SPL

Figure 4

LANGUAGE = FORTRAN

STATEMENT	STATIC	DYNAMIC
ASSIGNMENT	51	67
IF	10	11
GOTO	9	9
DO	9	3
CALL	5	3
WRITE	5	1
CONTINUE	4	7
RETURN	4	3
READ	2	0
STOP	1	0

PERCENT DISTRIBUTION OF SOURCE STATEMENTS
FROM
KNUTH'S FORTRAN STUDY [KNUT71]

Figure 5

LANGUAGE = SAL

STATEMENT	STATIC	DYNAMIC
ASSIGNMENT	46.5	41.9
CALL	24.6	12.4
IF	17.2	36.0
RETURN	4.2	2.6
FOR	3.4	2.1
EXITLOOP	1.4	1.6
WHILE	1.1	1.5
REPEAT	0.5	0.1
DO FOREVER	0.5	0.8
CASE	0.3	1.2
PRINT	0.3	<0.05

STATIC AND DYNAMIC STATEMENT PERCENTAGES
FROM
TANENBAUM [TANE78]

Figure 6

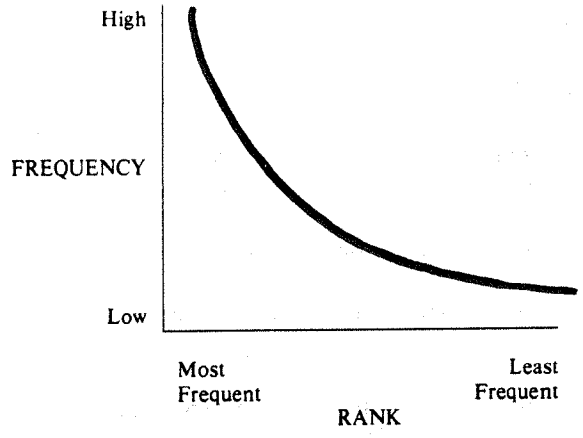

Figure 7