William M. McKeeman

Differential Testing for Software

Differential testing, a form of random testing, is a component of a mature testing technology for large software systems. It complements regression testing based on commercial test suites and tests locally developed during product development and deployment. Differential testing requires that two or more comparable systems be available to the tester. These systems are presented with an exhaustive series of mechanically generated test cases. If (we might say when) the results differ or one of the systems loops indefinitely or crashes, the tester has a candidate for a bug-exposing test. Implementing differential testing is an interesting technical problem. Getting it into use is an even more interesting social challenge. This paper is derived from experience in differential testing of compilers and run-time systems at DIGITAL over the last few years and recently at Compaq. A working prototype for testing C compilers is available on the web.

The Testing Problem

Successful commercial computer systems contain tens of millions of lines of handwritten software, all of which is subject to change as competitive pressures motivate the addition of new features in each release. As a practical matter, quality is not a question of correctness, but rather of how many bugs are fixed and how few are introduced in the ongoing development process. If the bug count is increasing, the software is deteriorating.

Quality

Testing is a major contributor to quality—it is the last chance for the development organization to reduce the number of bugs delivered to customers. Typically, developers build a suite of tests that the software must pass to advance to a new release. Three major sources of such tests are the development engineers, who know where to probe the weak points; commercial test suites, which are the arbiters of conformance; and customer complaints, which developers must address to win customer loyalty. All three types of test cases are relevant to customer satisfaction and therefore have value to the developers. The resultant test suite for the software under test becomes intellectual property, encapsulates the accumulated experience of problem fixes, and can contain more lines of code than the software itself.

Testing is always incomplete. The simplest measure of completeness is statement coverage. Instrumentation can be added to the software before it is tested. When a test is run, the instrumentation generates a report detailing which statements are actually executed. Obviously, code that is not executed was not tested. Random testing is a way to make testing more complete. One value of random testing is introducing the unexpected test—1,000 monkeys on the keyboard can produce some surprising and even amusing input! The traditional approach to acquiring such input is to let university students use the software.

Testing software is an active field of endeavor. Interesting starting points for gathering background information and references are the web site maintained by Software Research, Inc.[1] and the book *Software Testing and Quality Assurance*.[2]

Developer Distaste

A development team with a substantial bug backlog does not find it helpful to have an automatic bug finder continually increasing the backlog. The team priority is to address customer complaints before dealing with bugs detected by a robot. Engineers argue that the randomly produced tests do not uncover errors that are likely to bother customers. "Nobody would do *that,*" "That error is not important," and "Don't waste our time; we have plenty of *real* errors to fix" are typical developer retorts.

The complaints have a substantial basis. During a visit to our development group, Professor C. A. R. Hoare of Oxford University succinctly summarized one class of complaints: "You cannot fix an infinite number of bugs one at a time." Some software needs a stronger remedy than a stream of bug reports. Moreover, a stream of bug reports may consume the energy that could be applied in more general and productive ways.

The developer pushback just described indicates that a differential testing effort must be based on a perceived need for better testing from within the product development team. Performing the testing is pointless if the developers cannot or will not use the results.

Differential testing is most easily applicable to software whose quality is already under control, that is, software for which there are few known outstanding errors. Running a very large number of tests and expending team effort only when an error is found becomes an attractive alternative. Team members' morale increases when the software passes millions of hard tests and test coverage of their code expands.

The technology should be important for applications for which there is a high premium on correctness. In particular, product differentiation can be achieved for software that has few failures in comparison to the competition. Differential testing is designed to provide such comparisons.

The technology should also be important for applications for which there is a high premium on independently duplicating the behavior of some existing application. Identical behavior is important when old software is being retired in favor of a new implementation, or when the new software is challenging a dominant competitor.

Seeking an Oracle

The ugliest problem in testing is evaluating the result of a test. A regression harness can automatically check that a result has not changed, but this information serves no purpose unless the result is known to be correct. The very complexity of modern software that drives us to construct tests makes it impractical to provide a priori knowledge of the expected results. The problem is worse for randomly generated tests. There is not likely to be a higher level of reasoning that can be applied, which forces the tester to instead follow the tedious steps that the computer will carry out during the test run. An oracle is needed.

One class of results is easy to evaluate: program crashes. A crash is never the right answer. In the triage that drives a maintenance effort, crashes are assigned to the top priority category. Although this paper does not contain an in-depth discussion of crashes, all crashes caused by differential testing are reported and constitute a substantial portion of the discovered bugs.

Differential testing, which is covered in the following section, provides part of the solution to the problem of needing an oracle. The remainder of the solution is discussed in the section entitled Test Reduction.

Differential Testing

Differential testing addresses a specific problem—the cost of evaluating test results. Every test yields some result. If a single test is fed to several comparable programs (for example, several C compilers), and one program gives a different result, a bug may have been exposed. For usable software, very few generated tests will result in differences. Because it is feasible to generate millions of tests, even a few differences can result in a substantial stream of detected bugs. The trade-off is to use many computer cycles instead of human effort to design and evaluate tests. Particle physicists use the same paradigm: they examine millions of mostly boring events to find a few high-interest particle interactions.

Several issues must be addressed to make differential testing effective. The first issue concerns the quality of the test. Any random string fed to a C compiler yields some result—most likely a diagnostic. Feeding random strings to the compiler soon becomes unproductive, however, because these tests provide only shallow coverage of the compiler logic. Developers must devise tests that drive deep into the tested compiler. The second issue relates to false positives. The results of two tested programs may differ and yet still be correct, depending on the requirements. For example, a C compiler may freely choose among alternatives for unspecified, undefined, or implementation-defined constructs as detailed in the C Standard.[3] Similarly, even for required diagnostics, the form of the diagnostic is unspecified and therefore difficult to compare across systems. The third issue deals with the amount of noise in the generated test case. Given a successful random test, there is likely to be a much shorter test that exposes the same bug. The developer who is seeking to fix the bug strongly prefers to use the shorter test. The fourth issue concerns comparing programs that must run on different platforms. Differential testing is easily adapted to distributed testing.

Test Case Quality

Writing good tests requires a deep knowledge of the system under test. Writing a good test generator requires embedding that same knowledge in the generator. This section presents the testing of C compilers as an example.

Testing C Compilers

For a C compiler, we constructed sample C source files at several levels of increasing quality.

1. Sequence of ASCII characters
2. Sequence of words, separators, and white space
3. Syntactically correct C program
4. Type-correct C program
5. Statically conforming C program
6. Dynamically conforming C program
7. Model-conforming C program

Given a test case selected from any level, we constructed additional nearby test cases by randomly adding or deleting some character or word from the given test case. An altered test case is more likely to cause the compilers to issue a diagnostic or to crash. Both the selected and the altered test cases are valuable.

One of the more entertaining testing papers reports the results of feeding random noise to the C run-time library.[4] A typical library function crashed or hung on 30 percent of the test cases. C compilers should do better, but this hypothesis is worth checking. Only rarely would a tested compiler faced with level 1 input execute any code deeper than the lexer and its diagnostics. One test at this level caused the compiler to crash because an input line was too long for the compiler's buffer.

At level 2, given lexically correct text, parser error detection and diagnostics are tested, and at the same time the lexer is more thoroughly covered. The C Standard describes the form of C tokens and C "white-space" (blanks and comments). It is relatively easy to write a lexeme generator that will eventually produce every correct token and white-space. What surprised us was the kind of bugs that the testing revealed at this level. One compiler could not handle 0x000001 if there were too many leading zeros in the hexadecimal number. Another compiler crashed when faced with the floating-point constant 1E1000. Many compilers failed to properly process digraphs and trigraphs.

Stochastic Grammar

A vocabulary is a set of two kinds of symbols: terminal and nonterminal. The terminal symbols are what one can write down. The nonterminal symbols are names for higher level language structures. For example, the symbol "+" is a terminal symbol, and the symbol "additive-expression" is a nonterminal symbol of the C programming language. A grammar is a set of rules for describing a language. A rule has a left side and a right side. The left side is always a nonterminal symbol. The right side is a sequence of symbols. The rule gives one definition for the structure named by the left side. For example, the rule shown in Figure 1 defines the use of "+" for addition in C. This rule is recursive, defining additive-expression in terms of itself.

There is one special nonterminal symbol called the start symbol. At any time, a nonterminal symbol can be replaced by the right side of a rule for which it is the left side. Beginning with the start symbol, nonterminals can be replaced until there are no more nonterminal symbols. The result of many replacements is a sequence of terminal symbols. If the grammar describes C, the sequence of terminal symbols will form a syntactically correct C program. Randomly generated white-space can be inserted during or after generation.

A stochastic grammar associates a probability with each grammar rule.

For level 2, we wrote a stochastic grammar for lexemes and a Tcl script to interpret the grammar,[5,6] performing the replacements just described. Whenever a nonterminal is to be expanded, a new random number is compared with the fixed rule probabilities to direct the choice of right side.

In either case, at this level and at levels 3 through 7, setting the many fixed choice probabilities permits some control of the distribution of output values. Not all assignments of probabilities make sense. The probabilities for the right sides that define a specific nonterminal must add up to 1.0. The probability of expanding recursive rules must be weighted toward a nonrecursive alternative to avoid a recursion loop in the generator. A system of linear equations can be solved for the expected lengths of strings generated by

```
additive-expression additive-expression + multiplicative-expression
```

Figure 1
Rule That Defines the Use of "+" for Addition in C

each nonterminal. If, for some set of probabilities, all the expected lengths are finite and nonnegative, this set of probabilities ensures that the generator does not often run away.

Increasing Test Quality

At level 3, given syntactically correct text, one would expect to see declaration diagnostics while more thoroughly covering the code in the parser. At this level, the generator is unlikely to produce a test program that will compile. Nevertheless, compiler errors were detected. For example, one parser refused the expression `1==1==1`.

The syntax of C is given in the C Standard. Using the concept of stochastic grammar, it is easy to write a generator that will eventually produce every syntactically correct C translation-unit. In fact, we extended our Tcl lexer grammar to all of C.

At level 4, given a syntactically correct generated program in which every identifier is declared and all expressions are type correct, the lexer, the parser, and a good deal of the semantic logic of the compiler are covered. Some generated test programs compile and execute, giving the first interesting differential testing results. Achieving level 4 is not easy but is relatively straightforward for an experienced compiler writer. A symbol table must be built and the identifier use limited to those identifiers that are already declared. The requirements for combining arithmetic types in C (`int, short, char, float, double` with `long` and/or `unsigned`) were expressed grammatically. Grammar rules defining, for example, int-additive-expression replaced the rules defining additive-expression. The replacements were done systematically for all combinations of arithmetic types and operators. To avoid introducing typographical errors in the defining grammar, much of the grammar itself was generated by auxiliary Tcl programs. The Tcl grammar interpreter did not need to be changed to accommodate this more accurate and voluminous grammatical data. We extended the generator to implement declare-before-use and to provide the derived types of C (`struct, union, pointer`). These necessary improvements led to thousands of lines of tricky implementation detail in Tcl. At this point, Tcl, a nearly structureless language, was reaching its limits as an implementation language.

At level 5, where the static semantics of the C Standard have been factored into the generator, most generated programs compile and run.

Figure 2 contains a fragment of a generated C test program from level 5.

A large percentage of level 5 programs terminate abnormally, typically on a divide-by-zero operation. A peculiarity of C is that many operators produce a Boolean value of 0 or 1. Consequently, a lot of expression results are 0, so it is likely for a division operation to have a zero denominator. Such tests are wasted. The number of wasted tests can be reduced somewhat by setting low probabilities for using divide, for creating Boolean values, or for using Boolean values as divisors.

Regarding level 6, dynamic standards violations cannot be avoided at generation time without a priori choosing not to generate some valid C, so instead we implement post-run analysis. For every discovered difference (potential bug), we regenerate the same test case, replacing each arithmetic operator with a function call, inside which there is a check for standards violations.

The following is a function that checks for "integer shift out of range." (If we were testing C++, we could have used overloading to avoid having to include the type signature in the name of the checking function.)

```
int
int_shl_int_int(int val, int amt) {
    assert(amt >= 0 && amt < sizeof(int)*8);
    return val << amt;
}
```

For example, the generated text

```
a << b
```

is replaced upon regeneration by the text

```
int_shl_int_int(a, b)
```

```
++ ul15 + -- ui8 * ++ ul16 - ( ui17 + ++ ui20 * ( sl21 & ( argc <<=
c14 ) ? ( us23 ) < ++ argc <= ++ sl22 : -- ( ( * & * & sl24 ) ) ==
0160030347u < ++ ( t5u7 ) . sit5m6 & 1731044438u * ++ ui25 * (
unsigned int ) ++ ( ld26 ) ) & ( ( ( 076l ) * 2137167721L * sl27 ?
ul28 & d12 * ++ d9 * DBL_EPSILON * 7e+4 * ++ d11 + ++ d10 * d12 * (
++ ld31 * .4L * 9.l - ld32 * ++ f33 - - .7392E-6L * ++ ld34 + 22.82L
+ 1.9l * -- ld35 >= ++ ld37 ) == 9.F + ( ++ f38 ) + ++ f39 *f40 > (
float ) ++ f41 * f42 >= c14 ++ : sc43 & ss44 ) ^ uc13 & .9309L - (
ui18 * 007101U * ui19 ? sc46 -- ? -- ld47 + ld48 : ++ ld49 - ld48 *
++ ld50 : ++ ld51 ) >= 239.61l ) ^ - ++ argc == ( int signed ) argc -
++ ui54 )- ++ ul57 >= ++ ul58 * argc - 9ul * ++ * & ul59 * ++ ul60 ;
```

Figure 2
Generated C Expression

If, on being rerun, the regenerated test case asserts a standards violation (for example, a shift of more than the word length), the test is discarded and testing continues with the next case.

Two problems with the generator remain: (1) obtaining enough output from the generated programs so that differences are visible and (2) ensuring that the generated programs resemble real-world programs so that the developers are interested in the test results. Solving these two problems brings the quality of test input to level 7. The trick here is to begin generating the program not from the C grammar nonterminal symbol translation-unit but rather from a model program described by a more elaborate string in which some of the program is already fully generated. As a simple example, suppose you want to generate a number of print statements at the end of the test program. The starting string of the generating grammar might be

```
# define P(v) printf(#v "=%x\\n", v)

int main() {
    declaration-list
    statement-list
    print-list
    exit(0);
}
```

where the grammatical definition of `print-list` is given by

```
print-list P ( identifier ) ;
print-list print-list P ( identifier ) ;
```

In the starting string above there are three nonterminals for the three lists instead of just one for the standard C start symbol translation-unit. Programs generated from this starting string will cause output just before exit. Because differences caused by rounding error were uninteresting to us, we modified this print macro for types `float` and `double` to print only a few significant digits. With a little more effort, the expansion of `print-list` can be forced to print each variable exactly once.

Alternatively, suppose a test designer receives a bug report from the field, analyzes the report, and fixes the bug. Instead of simply putting the bug-causing case in the regression suite, the test designer can generalize it in the manner just presented so that many similar test cases can be used to explore for other nearby bugs.

The effect of level 7 is to augment the probabilities in the stochastic grammar with more precise and direct means of control.

Forgotten Inputs

The elaborate command-line flags, config files, and environment variables that condition the behavior of programs are also input. Such input can also be generated using the same toolset that is used to generate the test programs. The very first test on the very first run with generated compiler directive flags revealed a bug in a compiler under test—it could not even compile its own header files.

Results

Table 1 indicates the kinds of bugs we discovered during the testing. Only those results that are exhibited by very short text are shown. Some of the results derive from hand generalization of a problem that originally surfaced through random testing.

There was a reason for each result. For example, the server crash occurred when the tested compiler got a stack overflow on a heavily loaded machine with a very large memory. The operating system attempted to dump a gigabyte of compiler stack, which caused all the other active users to thrash, and many of them also dumped for lack of memory. The many disk drives on the server began a dance of the lights that sopped up the remaining free resources, causing the operators to boot the server to recover. Excellent testing can make you unpopular with almost everyone.

Test Distribution

Each tested or comparison program must be executed where it is supported. This may mean different hardware, operating system, and even physical location.

There are numerous ways to utilize a network to distribute tests and then gather the results. One particularly simple way is to use continuously running watcher programs. Each watcher program periodically examines a common file system for the existence of some particular files upon which the program can act. If no files exist, the watcher program sleeps for a while and tries again. On most operating systems, watcher programs can be implemented as command scripts.

There is a test master and a number of test beds. The test master generates the test cases, assigns them to the test beds, and later analyzes the results. Each test bed runs its assigned tests. The test master and test beds share a file space, perhaps via a network. For each test bed there is a test input directory and a test output directory.

A watcher program called the test driver waits until all the (possibly remote) test input directories are empty. The test driver then writes its latest generated test case into each of the test input directories and returns to its watch-sleep cycle. For each test bed there is a test watcher program that waits until there is a file in its test input directory. When a test watcher finds a file to test, the test watcher runs the new test, puts the results in its test output directory, and returns to the watch-sleep cycle. Another watcher program called the test analyzer waits until all the test output directories contain results. Then the results, both input and

Table 1
Results of Testing C Compilers

Source Code	Resulting Problem
if (1.1)	Constant float expression evaluated false
1 ? 1 : 1/0	Several compiler crashes
0.0F/0.0F	Compiler crash
x != 0 ? x/x : 1	Incorrect answer
1 == 1 == 1	Spurious syntax error
-!0	Spurious type error
0x000000000000000	Spurious constant out of range message
0x80000000	Incorrect constant conversion
1E1000	Compiler crash
1 >> INT_MAX	Twenty-minute compile time
'ab'	Inconsistent byte order
int i=sizeof(i=1);	Compiler crash
LDBL_MAX	Incorrect value
(++n,0) ? -- n: 1	Operator ++ ignored
if (sizeof(char)+d) f(d)	Illegal instruction in code generator
i=(unsigned)-1.0F;	Random value
int f(register());	Compiler crash or spurious diagnostic
int (…(x)…);	Enough nested parentheses to kill the compiler
	Spurious diagnostic (10 parentheses)
	Compiler crash (100 parentheses)
	Server crash (10,000 parentheses)
digraphs (<: <% etc.)	Spurious error messages
a/b	The famous Pentium divide bug (we did not catch it but we could have)

output, are collected for analysis, and all the files are deleted from every test input and output directory, thus enabling another cycle to begin.

Using the file system for synchronization is adequate for computations on the scale of a compile-and-execute sequence. Because of the many sleep periods, this distribution system runs efficiently but not fast. If throughput becomes a problem, the test system designer can provide more sophisticated remote execution. The distribution solution as described is neither robust against crashes and loops nor easy to start. It is possible to elaborate the watcher programs to respond to a reasonable number of additional requirements.

Test Analysis

The test analyzer can compare the output in various ways. The goal is to discover likely bugs in the compiler under test. The initial step is to distinguish the test results by failure category, using corresponding directories to hold the results. If the compiler under test crashes, the test analyzer writes the test data to the crash directory. If the compiler under test enters an endless loop, the test analyzer writes the test data to the loop directory. If one of the comparison compilers crashes or enters an endless loop, the test analyzer discards the test, since reporting the bugs of a comparison compiler is not a testing objective. If some, but not all, of the test case executions terminate abnormally, the test case is written to the abend directory. If all the test cases run to completion but the output differs, the case is written to the test diff directory. Otherwise, the test case is discarded.

Test Reduction

A tester must examine each filed test case to determine if it exposes a fault in the compiler under test. The first step is to reduce the test to the shortest version that qualifies for examination.

A watcher called the crash analyzer examines the crash directory for files and moves found files to a working directory. The crash analyzer then applies a shortening transformation to the source of the test case and reruns the test. If the compiler under test still crashes, the original test case is replaced by the shortened test case. Otherwise, the change is backed out and a new transformation is tried. We used 23 heuristic transformations, including

- Remove a statement
- Remove a declaration
- Change a constant to 1
- Change an identifier to 1
- Delete a pair of matching braces
- Delete an if clause

When all the transformations have been systematically tried once, the process is started over again. The process is repeated until a whole cycle leaves the source of the test unchanged. A similar process is used for the loop, abend, and diff directories.

The typical result of the test reduction process is to reduce generated C test programs of 500 to 600 lines to equally useful C programs of only a few lines. It is not unusual to use 10,000 or more compile operations during test reduction. The trade-off is using many computer cycles instead of human effort to analyze the ugly generated test case.

Test Presentation

After the shortest form of the test case is ready, the test analyzer wraps it in a command script that

1. Reports environmental information (compiler version, compiler flags, name of the test platform, time of test, etc.)
2. Reports the test output or crash information
3. Reruns the test (the test input is embedded in the script)

The test analyzer writes the command scripts to a results directory.

Test Evaluation and Report

The person who is managing the differential testing setup periodically runs scripts that have accumulated in the results directory to determine which ones expose a problem of interest to the development team. One problem peculiar to random testing is that once a bug is found, it will be found again and again until it is fixed. This argues the case for giving high priority to the bugs exposed by differential testing. Uninteresting and duplicate tests are manually discarded, and the rest are entered into the development team bug queue.

Summary and Directions

Differential testing, suitably tuned to the tested program, complements traditional software testing processes. It finds faults that would otherwise remain undetected. It is cost-effective. It is applicable to a wide range of large software. It has proven unpopular with the developers of the tested software.

This technology exposed new bugs in C compilers each day during its use at DIGITAL. Most of the bugs were in the comparison compilers, but a significant number of bugs in DIGITAL code were found and corrected.

Numerous special-purpose differential testing harnesses were put into use at DIGITAL, each testing some small part of a large program. For example, the C preprocessor, multidimensional Fortran arrays, optimizer constant folding, and a new `printf` function each were tested by ad hoc differential testers.

The Java API (run-time library) is a large body of relatively new code that runs on a wide variety of platforms. Since "Write once, run anywhere" is the Java motto, the standard for conformance is high; however, experience has shown that the standard is difficult to achieve. Differential testing should help. What needs to be done is to generate a sequence of calls into the API on various Java platforms, comparing the results and reporting differences. Technically, this procedure is much simpler than testing C compilers. Chris Rohrs, an MIT intern at DIGITAL, wrote a system entirely in Java, gathering method signature information directly out of the binary class files. This API tester may be used when the quality of the Java API reaches the point where the implementors are not buried in bug reports and when there are more independent implementations of the Java run time.

Differential testing can be used to increase test coverage. Using the coverage data taken from running the standard regression suite as a baseline, the developers can run random tests to see if coverage can be increased. Developers can freely add coverage-increasing tests to the test suite using the test output as an initial oracle. No harm is done because even if the recorded result is wrong, the compiler is no worse off for it. If at a later time a regression is observed on the generated test, either the new or the old version was wrong. The developers are alerted and can react. John Parks and John Hale applied this technology to DIGITAL's C compilers.

The problem of retiring an old compiler in favor of a new one requires the new one to duplicate old behavior so as not to upset the installed base. Differential testing can compare the old and the new, flagging all new results (correct or not) that disagree with the old results.

Differential testing can be used to measure quality. Supposing that the majority rules, a million tests can be run on a set of competing compilers. The metric is failed tests per million runs. The authors of the failed compilers can either fix the bugs or prove the majority wrong. In any case, quality improves.

At Compaq, differential testing opportunities arise regularly and are often satisfied by testing systems that are less elaborate than the original C testing system, which has been retired.

Acknowledgments

This work was begun in the Digital Cambridge Research Laboratory by Andy Payne based on his earlier experience in testing DIGITAL Alpha hardware. The author and August Reinig continued the development as an advanced development project in the compiler product group in Nashua, New Hampshire. Steve Rogers and Christine Gregowske contributed to the work, and Steve eventually placed a free working prototype on the web.[7] Bruce Foster managed and encouraged the project, giving the implementors ideas faster than they could be used.

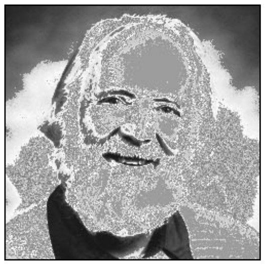
References and Notes

1. Information on testing is available at http://www.testworks.com/Institute/HotList/.
2. B. Beizer, *Software Testing and Quality Assurance* (New York: Van Nostrand Reinhold, 1984).
3. *ISO/IEC 9899: 1990, Programming Languages— C,* 1st ed. (Geneva, Switzerland: International Organization for Standardization, 1990).
4. B. Miller, "An Empirical Study of Reliability," *CACM,* vol. 33, no. 12 (December 1990): 32–44.
5. Information on Tcl/Tk is available at http://sunscript.sun.com/.
6. J. Ousterhout, *Tcl and the Tk Toolkit* (Reading, Mass.: Addison-Wesley, 1994).
7. Information on DDT distribution is available at http://steve-rogers.com/projects/ddt/.

General Reference

W. McKeeman, A. Reinig, and A. Payne, "Method and Apparatus for Software Testing Using a Differential Testing Technique to Test Compilers," U.S. Patent 5,754,860 (May 1998).

Biography



William M. McKeeman
William McKeeman develops system software for Compaq Computer Corporation. He is a senior consulting engineer in the Core Technology Group. His work encompasses fast-turnaround compilers, unit testing, differential testing, physics simulation, and the Java compiler. Bill came to DIGITAL in 1988 after more than 20 years in academia and research. Most recently, he was a research professor at the Aiken Computation Laboratory of Harvard University, visiting from the Wang Institute Masters in Software Engineering program, where he served as Professor and Chair of the Faculty. He has served on the faculties of the University of California at Santa Cruz and Stanford University and on various state and university computer advisory committees. In addition, he has been an ACM and IEEE National Lecturer and chairman of the 4th Annual Workshop in Microprogramming and is a member of the IFIP Working Group 2.3 on Programming Methodology. Bill founded the Summer Institute in Computer Science programs at Santa Cruz and Stanford and was technical advisor to Boston University for the Wang Institute 1988 Summer Institute. He received a Ph.D. in computer science from Stanford University, an M.A. in mathematics from The George Washington University, a B.A. in mathematics from University of California at Berkeley, and pilot wings from the U.S. Navy. Bill has coauthored 16 patents, 3 books, and numerous published papers in the areas of compilers, programming language design, and programming methodology.