EGREGION: A Branch Coverage Tool for APL [*]

Robert Bernecky
Snake Island Research Inc
18 Fifth Street, Ward's Island
Toronto, Ontario M5J 2B9
Canada
+1 416 203 0854
bernecky@acm.org

Abstract

This article describes our experience with test suites and automated branch coverage tools for APL software maintenance, based on our use of them to verify Y2K compliance of an APL-based database system. We introduce *egregion*, a simple, easy-to-use tool that assesses branch coverage in APL functions. The tool comprises a pair of APL functions that report detailed and summary function-level information about code coverage of test suites. The *egregion* tool provides a line-by-line analysis of statement coverage, labels not branched to, branches never taken, branches always taken, transfer of control via non-branches, and branches to non-labelled lines. Although we do not consider this groundbreaking work, we do believe that the coverage tool will be valuable to APL programmers who are engaged in the creation of large, reliable applications.
This article describes our experience with test suites and automated branch coverage tools for APL software maintenance, based on our use of them to verify Y2K compliance of an APL-based database system. We introduce *egregion*, a simple, easy-to-use tool that assesses branch coverage in APL functions. The tool comprises a pair of APL functions that report detailed and summary function-level information about code coverage of test suites. The *egregion* tool provides a line-by-line analysis of statement coverage, labels not branched to, branches never taken, branches always taken, transfer of control via non-branches, and branches to non-labelled lines. Although we do not consider this groundbreaking work, we do believe that the coverage tool will be valuable to APL programmers who are engaged in the creation of large, reliable applications.

[*]This paper originally appeared in the APL98 Conference Proceedings. [Ber98]

1 Introduction

For most programmers, testing is a poor cousin of the design and writing process. For APL programmers, the situation is even worse. Because APL facilitates rapid development and on-the-fly run-time error correction, it has encouraged a *code cowboy* approach to application design, in which formal testing is, at best, an afterthought. This unbridled, wild west, quick-draw approach to application development is finally dying a well-deserved death, due to the legal implications of application support for the millennium – the so-called Y2K problem – as well as the far more serious problems of computers in life-critical, real-time systems such as fly-by-wire aircraft, automated transit vehicles, and computer-controlled radiation therapy machines.

As corporations become sensitized to these issues and adopt standards such as ISO9000, they will insist on more formal criteria for design, implementation, test, acceptance, and audit of computer-based applications. This timely change is a step toward maturity for the computing world, much as the misfortunes of the early steam age led to formalization of engineering as a discipline and profession.

An application programmer's primary goal is to guarantee the correctness of an application. Issues of efficiency, maintainability, and elegance are irrelevant if the answers are wrong or the application crashes. Unfortunately, there is no known way to achieve this guarantee algorithmically or mathematically. Rather than throw up our hands and admit defeat, we turn to other engineering techniques to increase our confidence that a program is operating as desired.

Formal testing of applications, via program test scripts that evolve in step with them, is as proper a part of system development as the actual writing of code. Just as civil engineers perform independent tests of the concrete being poured for a bridge foundation, so programmers should encourage independent evaluation of their applications, to ensure that they meet design requirements and operate reliably. Such testing must, by its nature, be objective, repeatable, and quantifiable. Code-cowboy testing, in which a programmer enters a few simple expressions and declares an application successfully tested, is no longer an acceptable activity for a professional programmer.

Software engineers have a variety of tools available to them for verifying that programs are working as desired.[Pre97] One of the simplest of these tools is code coverage, in the form of statement coverage or branch coverage. Code coverage, which measures execution of statements or lines of a program, is often denigrated as being inadequate to find many types of program faults. Yet, in our experience, those who are most adamant about its faults are those who have never used it in practice. We believe code coverage to be a good first step in formalizing the testing of applications, because it is easy to understand, simple to implement, offers rapid feedback, and encourages programmers, once they have used it for a while and seen its benefits, to consider more sophisticated methods of testing.

In the remainder of this article, we introduce the concepts of basic blocks, control flow graphs, statement coverage, and branch coverage. Next, we show how code coverage can be measured with simple expressions and function monitoring primitives that are part of many commercial APL interpreters. Those coverage expressions form *egregion*, a pair of simple, easy-to-use APL functions that facilitate measurement of several aspects of program execution, including line-by-line analysis of statement coverage, labels not branched to, branches never taken, branches always taken, unexpected transfer of control via signals and interrupts, and branches to non-labelled lines. Finally, we show a brief example of *egregion* in action, taken from the database application that brought us to write this tool.

2 Basic Blocks and Control Flow

Basic blocks and control flow graphs are fundamental objects in the design of optimizing compilers. When combined with techniques such as Static Single Assignment (SSA), they also provide us with powerful analytical methods for making assertions about the properties of variables in a program.[CFR+89, Ber97a]

In compiler design circles, a *basic block* is a sequence of program instructions that has only one entry point and one exit point of control flow. The entry point may be a confluence of flows from other basic blocks; at the exit point, control flow may branch out to more than one other basic block. A *control flow graph* comprises a set of basic blocks as the vertices of a graph and a set of directed edges joining appropriate entry and exit points of the basic blocks. Since a program ultimately has a single entry point and single exit point (to and from the operating system), two distinguished vertices are added to the graph to denote these *start* and *stop* nodes.

In graph theory, the *cyclomatic number* of a graph `g` gives a rough measure of its complexity.[Pre97] This number is `(E g)+(C g)-V g`, where the functions `E,C,V` give the number of edges, number of components, and number of vertices, respectively.[1] It is generally accepted that the higher the cyclomatic number of a control flow graph, the harder the associated program is to comprehend and maintain. Hence, programs with low cyclomatic numbers are deemed to be superior to those with higher ones.

A branch-free APL function, such as the traditional one-liner, consists of exactly one basic block, giving the minimum theoretic cyclomatic number of zero.[2] The degenerate structure of non-looping APL functions significantly simplifies the comprehension and analysis of a program, by eliminating branches and effectively creating a single-assignment structure in which each name is given a value at exactly one place in the program.

Unfortunately, the real-world often forces nontrivial APL functions to contain loops. This may be due to the presence of one or more of the following factors:

- A limitation in workspace size may force an application to treat data in pieces. For example, the SHARP APL mainframe product restricts the maximum size of an active workspace to 16 megabytes. This makes it impossible to process large database feed files in parallel, because workspace full is a constant problem.

- Iteration may be forced by the environment. For example, a TCP/IP connection is inherently iterative, as are most applications involving files, regardless of whether they are native (operating system) files or APL component files. Similarly, a real-time data stream being received from a satellite or radio telescope must be analyzed as it arrives.

- Iteration may be forced by performance considerations, such as when a non-looping algorithm has exponential computational complexity. One class of problems that fall into this category are those that are amenable to dynamic programming solutions.[Ber95] Non-looping solutions to these problems are typically $O(2^N)$ in complexity, making such algorithms impractical for all but the smallest problems. By comparison, iterative dynamic programming solutions are far faster, being of polynomial complexity, typically taking time and space of order $O(N^2)$ or $O(N^3)$.

- Iteration may be forced by the use or creation of externally specified data structures over which the programmer has no design control. For example, compressed data files (ZIP, JPEG, MPEG) may not be amenable to parallel analysis or generation.

[1]The number of components is always one in a program that does not contain unreachable code.

[2]A matrix product written as three nested `:for` loops, by comparison, would have a cyclomatic number of four.

- Iteration may be forced by the algorithm itself, because there are no known non-looping solutions to it. For example, very large sparse systems of linear equations can not, in practice, be solved with dense arrays, so explicit sparse array methods are called for.[3] This eliminates the possibility of using APL primitives for solving such problems, because present-day interpreters do not provide support for sparse arrays nor for overloading primitives to let the user define such support.

Given that we are often forced to write iterative functions, the problem of validating that code to ensure that it is operating correctly becomes much harder. We turn to code coverage and unit testing of functions as a possible solution.

3 Code Coverage

This section introduces code coverage concepts and terminology, then discusses the pros and cons of code coverage as a tool for unit testing.

3.1 Code Coverage Terminology

Statement coverage is a measure of the number of statements in a program that were executed at least once by some program. Achieving 100% statement coverage implies that all statements in the program were executed at least once. In an APL function that contains no branches or event traps, any successful execution of the function gives 100% statement coverage.

[3]The sparse linear systems used in certain analyses of the space shuttle design were of order 250,000. Dense representations of such arrays occupy about 500 gigabytes and are impractical to store in present-day computers. Applying dense methods to their solution is out of the question.

Branch coverage is achieved when all edges of the control flow graph have been traversed by a program. That is, all statements have been executed, and all branch paths have been taken in each direction at least once.

Other, more comprehensive forms of code coverage are beyond the scope of this paper, as they tend to require complete flow control graphs and thus require the assistance of an APL compiler's front-end. Nonetheless, the interested reader is urged to look further into this topic.[Pre97, Sho83, Wie96, Ber97b]

We now turn to a brief discussion of the benefits and costs of code coverage.

3.2 Benefits of Code Coverage

Code coverage testing offers many advantages over code-cowboy methods of testing. Perhaps the most important of these is that it provides a quantifiable and objective basis for an assertion that the most egregious bugs have been removed from a program.[4]

Because code coverage encourages the use of formal test suites, tests are repeatable and objective. This makes them amenable to independent evaluation and reproduction by other parties, such as quality assurance teams. More importantly, they can be automated, allowing them to be run effortlessly whenever desired.

Other benefits of code coverage and formal test suites include the following:

- Once written, the incremental cost of maintaining and updating test suites is low, due to the fact that well-structured testbeds provide a quick and easy way for application developers to preserve the tests they normally create in the

[4]This is why we adopted `egregion` as the name of the branch coverage function.

process of enhancing or enhancing an application.

- Code coverage test suites increase the confidence developers have in their applications, and give them a feeling of accomplishment and a sense of knowing when a project is complete.

- The reliability of applications is improved, due to detection of a large class of common code faults. Execution of code coverage test suites turns up syntax errors, value errors, mismatched brackets and parentheses, and a few other egregious faults.

- Specifications end up being enshrined in test suites, in the form of a set of typical inputs and specified outputs for the application. This may not seem to add value to existing sources of information, but when someone drops an archaic "Raiders of the Lost Spec" application in your lap for enhancement or repair, test suites often come to the rescue as a clear, executable specification of what the application is supposed to do and how it is intended to work. In the worst case, when the documentation and code have diverged due to poor maintenance procedures, they even serve to show you what the application really does!

- Function monitoring on live systems can provide you with CPU and elapsed time profiling information to track response time and predict bottlenecks before they become acute.

- Regression testing is trivial if the application already has a code coverage test suite, because a new regression test is usually just one more line in the test suite.

- With the assistance of a state-of-the-art APL compiler, such as APEX, all arguments, explicit and implicit, can be analyzed to detect all value errors, rank errors, and type mismatch (domain) errors.[Ber97a] A compiler can also report the type and rank of all function arguments, and sometimes provide other information above and beyond that. Other compiler-assisted methods can reduce the number of tests that are required and help to provide more complete testing at lower cost than competing methods.

- Support and QA personnel can use code coverage test suites as a pre-installation validation procedure for bug fixes and new releases of applications.

- Feedback from automated code coverage tools dramatically increases the speed of writing test suites. In the late 1980s, for instance, we applied code coverage techniques to the SHARP APL mainframe product as part of a performance-improvement initiative.[Ber89] By developing a tool that automatically merged hardware monitor information with program listings, we were able to write complete branch coverage test suites of a specific primitive within a few hours. As the interpreter was written in assembler code, this was a non-trivial task. The most interesting aspect of the study was the number of *system errors* – bugs extant within the interpreter – that came to light. Some of these were new, but others represented failures that had been reported in the past, but had never been corrected, due to our inability to reproduce them. Obviously, if we had made branch coverage test suites available in the years before, we could have been shipping a more robust product.

3.3 Objections to Code Coverage

Code coverage is not without its problems. As a method of unit test, it is not a panacea for producing reliable applications. Creating any test suite requires an outlay of programmer time; design for testability may complicate application design; construction of a formal test environment may be non-trivial. In addition, code coverage testing may not catch data-sensitive faults, such as divide by zero, and edge conditions, such as improper treatment of empty arrays.

This inability to detect a certain class of errors is often presented an as argument against all use of code coverage. A senior programmer with whom I worked for many years claimed that "…the serious bugs only show up at customer sites, so why should I waste my time testing?"[5] This inflexible attitude is difficult to change without strong support from senior management.

Clearly, egos form a strong basis of objections to the use of code coverage and to formal test suites in general. When I was managing the SHARP APL development department, I tried to introduce formal test suites as a normal part of our development procedures. I received the following reactions from several of our programmers:

- "I am a good, competent programmer. I don't need a babysitter looking over my shoulder."
- "Don't you trust my work?"
- "I'll test my work as I see fit."
- "Testing takes too long. I'll miss the shipment deadlines if I write test suites."
- "Testing is for wimps."

[5]He also was prone to annotating the "What tests were done?" section of the software change history log with "None, but what could possibly go wrong?" Subsequent trouble log entries usually answered this question.

A manager of a large Japanese software development group told me about his company's method of dealing with this attitude. Their management did not insist that programmers write code coverage test suites for their applications, but they did encourage it in this way: If the programmers did not supply a coverage test suite with their completed program, they were required to personally provide all product support for the application for the next five years!

The objections raised by developers reflects a degree of insecurity and arrogance in a group who are often, not without some cause, considered to be prima donnas.[6] They also reflect several management problems. The first is that imposing standards for product test without buy-in from developers produces resentment. Second, establishment of product ship dates without enough concern for the time required to create test suites from scratch is unrealistic, leading to shoddy products and Death March projects. A related problem is that, too often, testing protocols are established, but they fall by the wayside as deadlines and downsizing take their toll. Furthermore, upper management is not always sympathetic. Formal test procedures are seen as delaying product delivery and diverting developers from "useful work"

Obviously, writing test suites and maintaining them in step with the application has an up-front cost, but the incremental cost of maintaining test scripts in step with the application is fairly low, *if* they are, in fact, maintained in the same manner and with the same care as the application itself. In fact, if the scaffolding – the structure that is needed to support test suites – is built first, most or all testing during development can be performed in a manner that constructs the test suite as part of the development process, rather than as an add-on after development is complete. *Ad hoc* testing always requires some sort

[6]I reluctantly count myself as an alumnus of this group.

of test environment. All that we require is that developers stop throwing away their test suites after each development project.

It may also turn out that one set of test tools can serve many applications. Since that is our hope in writing this article, this is a good time to turn to a discussion of test principles and the code coverage tool itself.

4 Design for Test

Design for test is an approach to software design that is simply the application of standard engineering testing principles to computer programs. In the construction of buildings and airplanes, for example, design for test results in the inclusion of access ports and hatches in the artifact, to facilitate routine maintenance and regular inspection. Similarly, software should be designed from the outset to facilitate maintenance, testing, and inspection.

It is far easier and faster to design an application from the outset to facilitate testing and performance monitoring than it is to retrofit such a capability when construction is nearly complete. For example, if you use *ad hoc* function paging mechanisms, it may be difficult to enable monitoring when a performance problem arises and you want to find the bottleneck in the application.

5 Test Suite Design

Designing test suites is a fairly straightforward activity. However, bear in mind that are several pitfalls to avoid, if test suites are to help, rather than hinder, the rapid construction of reliable software. We now discuss those pitfalls, as well as some of the desirable properties that test suites should possess.

5.1 Self-containment

Whenever possible, test suites should not depend on the existence of external files, such as session logs captured in days of yore. Changes in output format and display technology makes these difficult to maintain or extend. Obviously, if you are testing an output driver, this recommendation is not applicable. Similarly, do not make your test suites depend on large database files that system administrators may decide to delete in the interest of saving disk space, or which may only be available on a production system, rather than on your development system.

5.2 Reproducible

All tests should be strictly reproducible. Do not use random numbers for testing your code unless you always set `⎕rl` to a known value first.[7] Using reproducible tests ensures that when something breaks during testing, it stays broken. Furthermore, you can tell when it has been fixed, because it is reproducible. This is not the case with random testing.

5.3 Effective testing

Use terse tests to validate the behavior of specific, sensitive parts of each function. A test suite with a vast number of test cases may, in fact, only exercise part of the application over and over again. As we shall see, a test script that had the appearance of having been generated by an outer product produced 138 distinct tests, but only exercised 51% of the function under test. By comparison, a test suite that was generated interactively in a few hours with *egregion* obtained more than 95% coverage with only 37 test cases. A few well-thought-out tests are always superior to many shot-in-the-dark tests.

[7]Setting `⎕rl←¯1↑⎕ts` is not reproducible. It will also fail sporadically, because zero is a bad seed for the random number generator.

5.4 Test error behavior

Make sure that the application validates its arguments properly and signals the appropriate error when an argument is, in fact, invalid. To do this, you should set a trap for the expected error, and ensure that, if the trap is not triggered, a fault message is generated. If the trap is triggered, ensure that the generated error is the one you were expecting.

5.5 Design for Extendability

Test suites should be built as open frameworks, so that it is obvious how to add additional tests, should they become necessary. Additional tests *will* become necessary, because programs are always falling apart due to new features or *code rot*.[8]

By designing for extendability, we give the test suite the ability to support testing of new features, as well as making it trivial for support personnel, quality assurance auditors, and developers to add tests that reproduce newly found code faults.

Amending a test suite to reproduce a fault *before* attempting to repair the code fault has several benefits. Specifically, it:

- ensures that you understand what causes the fault well enough to reproduce it;
- ensures that the repair does, in fact, fix the code fault;
- ensures that the repair does not cause existing uses of the code to break; and
- gives the QA and support teams a validation tool for use on a target system, so they can ensure that the fix has been packaged and installed properly, *before* the code is released for production use.

[8]*Code rot* is the mysterious phenomenon whereby any computer program that is not under scrutiny on a day-to-day basis, will fail in mysterious ways the next time you try to run it. Code rot, generally most virulent just before a major deadline, is a largely unstudied area of computer science. Like compost, it offers fertile ground for research.

5.6 Keep test suites with source code

Keep test suite source code in the same place that you keep the application source code, under the same versioning and release controls as the application code. There are two reasons for doing this. First, if test suites are not placed under version control, they will grow moldy, suffer code rot, and be out of date within a year or so, or will be deleted by a zealous system administrator seeking disk space. Second, test suites must evolve as the application evolves. If Release 2 of an application has new features over Release 1, then there must also be a Release 2 of the test suite, evolved in step with the application, to allow testing of those new features.

5.7 Assert all results

Ensure that a test case is working correctly, by executing an expression that compares the generated result to the required result. If this is not done, all that is known about the application is that it is executing quietly. It may or may not be executing correctly. This step may seem obvious, but people ignore it more than they practice it.

5.8 Do not set the fox to guard the chickens

Do not use the application's code as the method for asserting that the results are correct. You may only be getting the same wrong result twice! Note that this may effectively mean writing the application twice. Often, this is not such a bad idea. In our case, as it turned out, we wrote a few utility functions that, in conjunction, gave all the capability of the old monolithic function under test, but without the convoluted control flow of the old one. As time permits, we plan to replace the old code with the new functions. We certainly gained a better understanding of the application by writing the utilities.

Another approach to result validation is to use hard-coded results, perhaps obtained from a control run. However, this tends to be tedious to maintain, and it can sometimes be difficult to preserve captured results in a convenient form for comparison.

5.9 Check side effects

Checking side effects is particularly important when testing error behavior. If a function under test has the nasty habit of changing a variable that is global to itself, ensure that it does not change when an error is triggered by that function. This whole problem area can be neatly sidestepped by writing in a functional programming style, which inherently is free of side effects.

6 A Simple Code Coverage Tool

Most APL vendors offer function monitors with their products. A partial list of these is shown in Figure 1. Function monitors typically provide the developer with the number of times each function line was executed, the cpu time for execution of each line, and the elapsed time for execution of each line. The SHARP APL `⎕fm` system function also supplies times inclusive and exclusive of sub-function calls.

Let us see how we can use the output of such a monitor to assist us in conducting a unit test of a function, `Da`. The function, edited for space reasons, is shown in Figure 2, along with the annotation provided by *egregion* as a result of running our test script. We executed a unit test script for the function under control of a function monitor. The monitor provided us a vector of numbers, `N`: the number of times each line in the function was executed. In the Figure, the numbers down the left margin, representing `N`, offer us several statistics about the execution of the program. Most obviously, the expression:

```
 N=0
```

tells us which lines of the function were never executed. For example, lines `22` and `30` were never executed. Such ines could contain mis-matched parentheses or brackets, value errors, syntax errors, and the like, but unless the test suite executes those lines, those errors will be found by the customer, not by us!

The remainder of this section highlights some expressions in the *egregion* function shown in Figure 2. If we locate the lines of the function that contain branch instructions, we can measure control flow over some paths of the control flow graph. A crude expression such as:

```
nb←~b←∨/'→'=fnt←⎕cr'Da'
```

is adequate for most purposes, although it is clear that a simple tokenizer and syntax analyzer such as is used in the front-end of APEX or Soliton's Logos source maintenance tool would perform better in the presence of quoted strings.[AGDH86, Ber97a]

An expression similar to that used to find branches marks the lines of the function that contain line labels:

```
 lab←'0123456789:' ⍝ Find labels
 lab←lab,'∆abc...xyz'
 lab←lab,'⍙ABC...XYZ'
 lab←∨/(∧\fnt∊lab)∧fnt=':'
```

Now we locate the various branches-in (`bi`) and branches-out of (`bo`) each line of the function:

```
 bi←N>¯1⌽N ⋄ bi[⎕io]←1
 bo←N>1⌽N ⋄ bo[⎕io]←1
```

We are now prepared to gather our statistics:

- *branch not taken*: Differences in line execution counts between a line and its successor give us control flow splits and joins. The predicates computed earlier give us branch statements (`b`) with no branch-out (`bo`):

```
b∧~bo
```

- *branch always taken*: Similarly, branch-out (`bo`) in a line with execution count of zero on the next line tells us that a branch was always taken.

  ```
  b∧bo∧1⌽N=0
  ```

- *label not branched to*: A labelled expression that is not a branch-in point reveals a branch target that was never hit:

  ```
  lab∧~bi
  ```

Similar expressions tell us which error-signal expressions were never exercised, which unlabelled lines were branched to, and which lines had unexpected control flow interruptions (*e.g.*, trapped domain error).

If we look at the output of *egregion* for the `Da` function, shown in Figure 2, we see line execution counts to the left of the function text, accompanied by highlighting characters. The `X` shows that lines 22 and 37 were never executed by the test script. Line 15 contains a label that was never branched to, although we deduce that control did fall into it from line 14, because the line execution counts are both 10. This may represent a missing test case or an unreferenced label. In this example, it is the former. Lines 3,25,32 contain branches that were never taken, reflecting missing test cases or dead paths in the control flow graph. Lines 21 and 29 contain branches that were always taken, again reflecting, in this particular function, missing test cases.

Examination of *egregion* output by the application developer is almost always enlightening, as it shows how limited is our knowledge of how our programs really work. The use of *egregion* leads to the creation of more robust applications through better test scripts.

6.1 Current *egregion* Limitations

Today's APL monitoring primitives record data on a line-by-line basis. This means that they are not able to provide you with direct information about control flow paths taken nor about execution of latter statements in a diamondized lines. Until more advanced monitoring tools are available, automated unit testing will be facilitated if you adopt a few stylistic coding conventions that let us deduce the missing information. If you use a source code control system, such as Logos, you can write generation scripts to automate most of this. Otherwise, you have to keep an eye on your programming style. Here are the conventions we adopted for use with *egregion*, together with the rationale for adoption of those conventions:

- If a line contains an APL GOTO (→), no diamond statement separators may follow the GOTO statement. This restriction arises because the *only* information available to the function monitor is the number of times the *line* was executed. Consider the 4-statement line:

  ```
  a ⋄ b ⋄ →c⍴label ⋄ d
  ```

  A function monitor might tell us that this *line* was executed five times, but We do not know how many times each *statement* in the line was executed. We do not have the information we need to determine if the statement `d` after the GOTO was executed or not. An undiamondizer function can be used to correct this problem in extant code by eliminating diamondized expressions from the code under test, or you could write a compilation filter for Logos-based code, as we did.

- If a line containing a GOTO or error signaller is immediately followed by a line with a statement label, an ambiguity arises, because the function monitors give us line execution counts, but no indication of control flow. In the following example, consider what would happen if the first line (containing a GOTO) was executed two times and always branched to the label `always`, and the line containing the label `target` was also coincidently branched to two times. The first column represents execution counts for the line in the second column:

  ```
   2  |  →cond⍴always
   2  | target: . . .
  ... | . . .
   2  | always: foo ⋄ →target
  ```

  Here, *egregion* would incorrectly claim, based on the identical line execution counts, that the GOTO was never taken, and that the line containing the label `target` was never the target of a branch. Without a fancier function monitor, we are unable to resolve this ambiguity directly. However, it can be easily resolved by inserting another line, such as a comment, between the two lines.[9] This task can be automated using Logos, so that you can generate such comments only when needed for purposes of code coverage.

  When we place a comment line between the two lines, *egregion* sees execution counts of `2 0 2` and correctly determines that the GOTO was always taken:

  ```
   2  |  →cond⍴always
   0  | ⍝
   2  | target: . . .
  ... | . . .
   2  | always: foo ⋄ →target
  ```

  The extra line allows the tool to provide the correct result, informing the test suite writer that another test is needed to exercise the branch-not-taken path or that there is a dead edge in the control flow graph.

- Code-cowboy branching – branching to absolute or relative line numbers, rather than to a label, is forbidden.

- At present, *egregion* is unable to detect the set of CASE-statement-like branch targets in an APL branch expression that selects one of a number of line labels. For example, this statement can branch three ways or fall through to the next statement, depending on the value and shape of `I`:

  ```
  →(lneg,lzero,lpos)[I]
  ```

  The absence of any syntactic comprehension of the program by the tool makes branch path accounting impossible without the assistance of Logos or the front-end of an APL compiler. All we can do is to ensure that all lines containing labels have been the target of a successful branch, and that fall-through paths have also been executed.

6.2 *egregion* in action

The synergy between test suites and tools unexpectedly surfaced recently in the APL application world when we had an application dropped in our lap with the assigned task of integrating a set of Y2K changes into the source code control system, validating the

[9]This method does not work for APL+Win, because `⎕mf` always gives zero as the number of times a comment line was executed. For this interpreter, you will have to delete all comments, and insert a dummy non-comment statement when testing.

application's Y2K compliance, and unleashing the application on a set of internal users, all within a few days. It all sounded very straightforward, since another programmer had already updated the code for Y2K support and created an extensive test suite to validate it – one single function had 137 different test cases written for it.

The reality of the situation was somewhat different. We integrated the changes, rebuilt the application, then ran the test suite. It apparently performed perfectly. Then, having built large systems before, we prudently dropped by the office of a database specialist who was a frequent user of the application, and asked him to try out the new code. In less than a minute, he had typed in three short expressions, each of which caused the new code to fail in a different way. We had run into a code-cowboy test suite! The huge test suite had not, in fact, tested many of the most common cases used in practice. This showed that a large number of tests had little or no correlation to effective testing of an application.

At this point in time, we had a problem. The application was supposed to "go live" on a next-millenium-dated test APL system the next day, and the code did not even work. It was tool time.

We built the *egregion* tool shown in Figure 3 and Figure 4 in an hour or so, made a minor change to the application to enable SHARP APL `⎕fm` function monitoring for all paged-in functions, then re-ran the failed test suite under control of *egregion*. It was immediately obvious that the test suite had failed to exercise half of the failing function, but had exercised other pieces of code over and over again, to no benefit. This explained the three failures created by our brown-thumbed colleague, but it was not going to help us to get the updated application out the door in timely fashion. It was test suite time.

The original test suite consisted of simple expressions that generated great volumes of output which were appended to a file. When the test suite completed, a second step compared the output file to a canned, hand-crafted version of the file, complaining if it saw differences between the two. We deemed this too fragile and labor-intensive, which is why we started over with a new approach, using assertions to validate correct operation.

We built a test suite driver that called a sub-function for each major part of the application. This approach allowed us to expand or change the test suite as needs developed and time permitted. We then wrote one driver sub-function at a time, with attention paid to the issues described earlier. Using separate sub-functions to test specific aspects of application performance facilitates rapid development and debugging, because a programmer need not test the entire application when working in an isolated part of the application.

We wrote a new test suite from scratch and ran it under *egregion*, using the tool to evaluate and improve branch coverage. In a half-day, including the time required to write the test script scaffolding, it showed up five distinct bugs in the above-cited function *after* it had passed all 137 developer-written tests! Our speed in creating the test script was due entirely to having immediate feedback on branch coverage. It told us what paths in the function had not been exercised. This made it trivial for us, even though we had almost no knowledge of the specification of the code we were testing, to add test cases that exercised all parts of the functions of interest.

The application went on to the millennium test system on schedule, as planned. While the dust was settling, we looked at the results of code coverage on the largest and problematic functions in the application. Figure 6 shows how the original test suite compares to the one we wrote with the assistance of *egregion*, for two typical functions, `Da` and `We`.

Looking at the number of test cases executed compared to the amount of code actually executed, it was very clear that the number of test cases had no relationship to the amount of code that was actually tested. With no feedback, the author of the original script was shooting in the dark.

The original test suite failed to traverse many edges in the control flow graph, as evidenced by the high *branch always taken*, *branch not taken*, and *no branch to label* counts, which would ideally all be zero.

In spite of having more than three times as many test cases, the original test suite missed testing half of the code and a large number of potential paths through it. The redundant tests in the original test suite served no useful purpose, and merely slowed the pace of testing. In contrast, *egregion* gave us concise test scripts and quantitative information about the quality of our test suite.

7 Summary and Future Work

Branch coverage tools are not new to the software engineering world, but they have not been used very widely in APL applications. The amount of attention being paid at present to correct application behavior is changing this. The *egregion* tool offers a simple method for obtaining quantitative, reproducible, objective evaluations of unit test suites in the APL environment. The tool also provides a structure for creation of a performance measurement tool. The speed with which applications can be inspected using test suite results evaluated by *egregion* should give application developers an incentive to use it on a regular basis, even if their code is already a paragon of correctness.

One large SHARP APL site is now looking into making automated execution and evaluation of test suites an integral part of the software development and release process.

8 Acknowledgements

Richard Levine made a number of constructive suggestions that improved the readability and presentation of this article.

Product	Monitor enable	Monitor disable	Monitor report
APL+Win	`1 ⎕mf 'foo'`	`0 ⎕mf 'foo'`	`⎕mf 'foo'`
Dyalog APL	`(⍳N) ⎕MONITOR 'foo'`	`'' ⎕MONITOR 'foo'`	`⎕MONITOR 'foo'`
SHARP APL	`63 ⎕fm 'foo'`	`0 ⎕fm 'foo'`	`¯62 ⎕fm 'foo'`

Figure 1: Function monitors in APL interpreters

References

[AGDH86] David B. Allen, Leslie H. Goldsmith, Mark R. Dempsey, and Kevin L. Harrell. LOGOS: An APL programming environment. In *APL86 Conference Proceedings*, volume 16, pages 314–325. ACM SIGAPL Quote Quad, July 1986.

[Ber89] Robert Bernecky. Profiling, performance, and perfection. In *ACM SIGAPL APL89 Session Tutorials*. ACM SIGAPL, August 1989. ISBN 0-89791-331-0.

[Ber95] Robert Bernecky. The role of dynamic programming and control structures in performance. In Marc Griffiths and Diane Whitehouse, editors, *APL95 Conference Proceedings*, volume 26, pages 1–5. ACM SIGAPL Quote Quad, June 1995.

[Ber97a] Robert Bernecky. APEX: The APL parallel executor. Master's thesis, University of Toronto, 1997.

[Ber97b] Robert Bernecky. An overview of the APEX compiler. Technical Report 305/97, Department of Computer Science, University of Toronto, 1997.

[Ber98] Robert Bernecky. EGREGION: A branch coverage tool for APL. In Sergio Picchi and Marco Micocci, editors, *APL98 Conference Proceedings*, pages 1–20. APL Italiana, July 1998.

[CFR$^+$89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method for computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, January 1989.

[Pre97] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1997.

[Sho83] Martin L. Shooman. *Software Engineering: Design, Reliability, and Management*. McGraw-Hill, 1983.

[Wie96] Karl E. Wiegers. *Creating a Software Engineering Culture*. Dorset House Publishing, 1996.

```
Legend:
X Line never executed
◇ Line contains diamonds and branches
→ Branch always taken
↓ Branch never taken
: Label never branched to
      33 [  0] Da;C;r;D;I;E;F;dtb;ts;b;i;txt;cr
      33 [  1] cr←⎕av[157]
      33 [  2] ts←⎕ts ◇ dtb← 0 100 ⍝ date radix; snap ts once
   ↓  33 [  3] →err×⍳0=⎕nc 'B'
      33 [  4] →l9×⍳(PER≥B)∨(1↑⍴data)<2
       1 [  5] txt←'error. upward timepacking only',cr
       1 [  6] txt←txt,' Use clear first if desired'
       1 [  7] txt ⎕signal 863
      32 [ 11] b←(⍴A)⍴ 0 1 ◇ b←(b∧A∊⍳B)∨(~b)∧A>1800
      32 [ 12] b[1]←A[1]∊ ¯99 ¯69 ¯68
      32 [ 13] →(∧/b)⍴l3
      10 [ 14] ⍝
    : 10 [ 15] err:txt←'error.use for example: 13 periods
      10 [ 19] txt ⎕signal 863
      22 [ 20] l3:⍎(0=4 ⎕ws 'Yend')/'Yend←Tot>0'
  →   22 [ 21] →l4×⍳F←(1↑⍴data)∊ 0 1
X      0 [ 22] r←Δyy/ 1 0 ↓data ◇ D←PER,Δyy/data[1;]
      22 [ 23] l4:PER←year←B
      22 [ 24] i← ¯99 ¯69 ⍳A[1]
   ↓  22 [ 25] →(i>2)⍴err ⍝ Invalid right argument
      22 [ 26] Atf←i×Atf≠3
      22 [ 27] →(A[1]=¯69)⍴l6
      16 [ 28] ⍝ <periods dated PPPP to PPPP>
 ◇→   16 [ 29] data←(1,⍴A)⍴A←Δdgen(1↓A),ts[1] ◇ →l8
       0 [ 30] ⍝
       6 [ 31] l6:⍎(Tot>0)/'Tot←1-Tot'
   ↓   6 [ 32] →err×⍳1≠2◇⍴A
       6 [ 33] A←((0.5×⍴A),2)⍴A←1↓A
       6 [ 34] data←(1,⍴A)⍴A←dtb⊥⍉⌽A ◇ Dow←⍳0
      22 [ 35] l8:⍎(0≠4 ⎕ws 'Yend')/'Yend←Tot>0'
  →   22 [ 36] →0×⍳F
X      0 [ 37] r←r[;I←D[1] Δind 1↓D]
```

Figure 2: Code coverage detailed report sample output

```
    ∇ r←egregionSummary fm;i;j;codes;f;fn
[1]    ⍝ Summarize code coverage report from egregion
[2]    ⍝ fm is character matrix output from egregion
[3]    i←+/∧\fm[;1]≠' ' ⍝ Find end of legend
[4]    codes←1 0↓(i,¯1↑⍴fm)↑fm ⍝ Legend less "Legend"
[5]    codes←((1↑⍴codes),(+/∨\⌽' '∨.≠codes))↑codes
[6]    f←(i,0)↓fm ⍝ Delete boring stuff
[7]    ⍝ Count number of violations
[8]    r←+⌿f[;⍳1↑⍴codes]=(1↑⍴f)⌿⍉codes[;,1]
[9]    r←⍕((1↑⍴r),1)⍴r
[10]   fn←f[2;]
[11]   fn←(fn⍳']')↓(¯1+fn⍳';')↑fn ⍝ Get function name
[12]   fn←'Code Coverage Summary for: ',fn
[13]   r←r,'=',codes ⍝ Simple report
[14]   i←(¯1↑⍴r)⌈¯1↑⍴fn
[15]   r←' ',[⎕io](i↑fn),[⎕io] ((1↑⍴r),i)↑ r
    ∇
```

Figure 3: Code coverage summary report function

```
    ∇ r←fnt egregion n;bi;bo;dab;br;dia;cnt;nb;lab;s
[1]    ⍝ Produce code coverage report for function
[2]    ⍝ fnt is ⎕cr of function. n is vector of ilne counts
[3]    lab←'0123456789:' ⍝ Find statement labels
[4]    lab←lab,'∆abcdefghijklmnopqrstuvwxyz'
[5]    lab←lab,'⍙ABCDEFGHIJKLMNOPQRSTUVWXYZ'
[6]    lab←∨/(∧\fnt∊lab)∧fnt=':'
[7]    nb←fnt[;1]='⍝' ⍝ Find comments
[8]    fnt←('<[>,I3,<] >' ⎕fmt ¯1+⍳1↑⍴fnt),fnt ⍝ Append line #s
[9]    r←' X'[1+(~nb)∧n=0] ⍝  non-executed lines
[10]   r←((⍴r),1)⍴r
[11]   br←fnt='→' ⍝ Branches
[12]   dia←fnt='⋄' ⍝ Diamonds
[13]   dab←(∨/br)∧∨/dia ⍝ Lines with diamonds and branches
[14]   r←r,' ⋄'[1+dab]
[15]   bi←n>¯1⌽n
[16]   bi[⎕io]←1 ⍝ Branch in
[17]   bo←n>1⌽n
[18]   bo[⎕io]←1 ⍝ Branch out
[19]   ⍝ Branch always taken
[20]   r←r,' →'[1+(~nb)∧(∨/br)∧bo∧1⌽n=0]
[21]   ⍝ Branch never taken
[22]   r←r,' ↓'[1+(~nb)∧(∨/br)∧~bo]
[23]   ⍝ All labels should be branched to
[24]   r←r,' :'[1+lab∧(~bi)∧n≠0]
[25]   r←r,' ',(⍕((1↑⍴n),1)⍴n),' ',fnt
[26]   nb←1 40⍴'Legend:'  ⍝ Build legend
[27]   nb←nb,[⎕io]40↑'X Line never executed'
[28]   nb←nb,[⎕io],40↑'⋄ Line contains diamonds and branches'
[29]   nb←nb,[⎕io],40↑'→ Branch always taken'
[30]   nb←nb,[⎕io],40↑'↓ Branch never taken'
[31]   nb←nb,[⎕io],40↑': Label never branched to'
[32]   nb←nb,[⎕io],40↑' '
[33]   s←(¯1↑⍴r)⌈¯1↑⍴nb
[34]   nb←((1↑⍴nb),s)↑nb
[35]   r←nb,[⎕io] ((1↑⍴r),s)↑r
    ∇
```

Figure 4: Code coverage detailed report function

```
Code Coverage Summary for: We
 68=X Line never executed
  1=◊ Line contains diamonds and branches
  8=→ Branch always taken
 11=↓ Branch never taken
  2=: Label never branched to
```

Figure 5: Code coverage summary report sample output

Test metric (desired value)	Da		We	
	Original	*egregion*	Original	*egregion*
Number of test cases (0)	138	37	131	38
Lines of code tested (100%)	51%	90%	40%	97%
Branch always taken (0%)	33%	10%	29%	11%
Branch not taken (0%)	11%	3%	39%	0%
No branch to label (0%)	9%	4%	13%	9%

Figure 6: Comparison of testing methods