LAW OFFICE OF CEM KANER **Cem Kaner, Ph.D., J.D.** **kaner@kaner.com**
P.O. Box 1200 **408-244-7000 (Voice)**
Santa Clara, CA 95052 **408-244-2181 (FAX)**

The Impossibility of Complete Testing

Law of Software Quality Column, Software QA magazine

I'm writing this from a hotel room in Sacramento, California. The National Conference of Commissioners on Uniform State Laws (NCCUSL) is reviewing Article 2B of the Uniform Commercial Code (UCC)— a 340-page statute that will govern all contracts for the development, sale, licensing, and support of software. Next week, I'll meet with yet more lawyers about the same bill, at the annual meeting of the American Bar Association.

NCCUSL is the authoring organization for Uniform Laws (laws that are written to be identical in all states). Its expenses are mainly paid by the American state governments. It is effective at getting its bills passed. For example, over the past 12 months, 200 of NCCUSL's bills have been introduced in state legislatures and 95 have been passed so far.

This is a full meeting of NCCUSL. The 300 Commissioners (senior attorneys who serve as unpaid volunteers) are reviewing nine different draft Uniform Laws, involving such areas as child custody, guardianships, estate planning, sales of merchandise, and software. The discussion of 2B has been largely favorable. The expectation is that it will be approved for introduction in state legislatures in September, 1998.

For the next year, work on 2B will be done by a 16-member NCCUSL committee whose meetings are open to the public. A typical Article 2B meeting draws 40-80 people, mainly lawyers. The 2B Drafting Committee meetings have had more attendance and press coverage than any of NCCUSL's other Drafting Committees' meetings. I've been attending these meetings for the last 18 months and have been joined by other software quality advocates, including James Bach, Doug Hoffman, Brian Lawrence, Melora Svoboda, Clark Savage Turner, and Watts Humphrey (see Humphrey, 1997). The next meeting will be September 26-28 at the Sofitel Hotel in Minneapolis. The meeting after that will run November 21-23 in Memphis.

As I've said in more detail in previous columns, I think this is a dangerous law that will reduce software publishers' accountability to their customers. I've put together a website on this, www.badsoftware.com.

It's tempting to argue in these meetings that software products should be free of defects. Unfortunately, I don't think they can be, and therefore I don't think that the law should insist that they be. This complicates my position in these meetings, and so I spend a lot of time explaining simple things about software development to lawyers. One of those simple things is a lesson that we all know (or that I hope we all know) as testers—***it is impossible to fully test a program***.

Lawyers aren't the only people who don't understand this. People still sign contracts that promise delivery of a fully tested program. And I still hear executives and project managers tell testers that it is their responsibility to find all the bugs. But how can you find them all, if you can't fully test the program? You can't. As the testing expert in your company, you'll have to explain to people who might not be technically sophisticated.

This paper explores three themes:

1. I think that I've figured out how to explain the impossibility of complete testing to managers and lawyers, with examples that they can understand. These are my notes.
2. A peculiar breed of snake-oil sellers reassure listeners that you achieve complete testing by using their coverage monitors. Wrong. Complete line and branch coverage is not complete testing. It will miss significant classes of bugs.
3. If we can't do complete testing, what should we do? It seems to me that at the technical level and at the legal level, we should be thinking about "good enough testing," done with as part of a strategy for achieving "good enough software."

Tutorial on Impossibility

You probably know this material, but you might find these notes useful for explaining the problem to others.

Complete testing is impossible for several reasons:

- We can't test all the inputs to the program.
- We can't test all the combinations of inputs to the program.
- We can't test all the paths through the program.
- We can't test for all of the other potential failures, such as those caused by user interface design errors or incomplete requirements analyses.

Let's look at these in turn.

Can't test all inputs

The number of inputs you can feed to a program is typically infinite. We use rules of thumb (heuristics) to select a small number of test cases that we will use to represent the full space of possible tests. That small sample might or might not reveal all of the input-related bugs.

Valid Inputs: Consider a function that will accept any integer value between 1 and 1000. It is *possible* to run all 1000 values but it will take a long time. To save time, we normally test at boundaries, testing perhaps at 0, 1, 1000, and 1001. Boundary analysis provides us with good rules of thumb about where we can shortcut testing, but the rules aren't perfect.

One problem is that many programs contain local optimizations. Even though every integer between 1 and 1000 might be valid, the program might actually process these numbers differently, in different sub-ranges.

For example, consider the Chi-Square function, a function popular among statisticians. One of the inputs to Chi-Square is a parameter called "degrees of freedom" or DF, for short. There is a second input, which I'll call X. Let's write a program to calculate Chi-Square(X,DF). The shape of the Chi-Square function changes, depending on the value of DF.

- When DF = 1, the shape of Chi-Square (X,1) is exponential. Chi-Square(X,1) is large for tiny values of X, but drops rapidly toward 0 for larger values of X. A computer program can calculate the exact value of Chi-Square (X,1).
- When DF is larger than 1, its shape changes and we lose the ability to exactly calculate the value of Chi-Square. Abramowitz & Stegun's famous *Handbook of Mathematical Functions* (1964) provides a good formula for closely approximating values of Chi-Square.
- For DF of 100 or larger, Abramowitz and Stegun recommend using yet a third function that is faster and easier to compute.

If you don't know that the program uses three different algorithms to estimate the value of Chi-Square, you won't bother testing at intermediate values like 100.

There are lots of optimizations in code—special treatments for special cases or for ranges of special cases. Without detailed knowledge of the internals of the program, you won't know what to test.

Invalid Inputs: If the program will accept any integer between 1 and 1000, then to test it completely, you'd have to test every number below 1, above 1000, all the rational numbers, and all the non-numeric inputs.

Suppose you only tested –1, 0, and 1001 as your invalid numerical inputs. Some programs will fail if you enter 999999999999999—a number that has too many characters, rather than being too numerically large.

Edited Inputs: If the program accepts numbers from 1 to 1000, what about the following sequence of keystrokes: 1 <backspace> 1 <backspace> 1 <backspace> 1000 <Enter>? If you entered single digits and backspaced over them hundreds of times, could you overflow an input buffer in the program? There have been programs with such a bug. How many variations on editing would you have to test before you could be *absolutely certain* that you've caught all the editing-related bugs?

Timing Considerations: The program accepts any number between 1 and 9999999. You type 123, then pause for a minute, then type 4567. What result? If you were typing into a telephone, the system would have timed you out during your pause and would have discarded 4567. Timing-related issues are at the heart of the difference between traditional systems and client/server systems. How much testing do you have to do to check for timeouts and their effects? If timeouts do occur in your system, what happens with 1-pause-2-pause-3-pause where no one pause is quite long enough for a timeout, but the total pause time is? What if you type 1-pause-2 with a pause that is just barely enough to trigger a timeout? Can you confuse the system by typing the 2 just barely too late?

Can't test all combinations of inputs

A few years ago, printers and video cards both came out in higher resolution models. On a few programs, you could run the printer at high resolution (600 dpi) with no problem and you could run video at high resolution, but if you tried a print preview, crash. In retrospect, it's easy to understand that your video driver might interact with your printer driver, but the bug was a big surprise to us at the time. You probably know of interactions between mouse and video drivers. How many versions of mouse drivers should we test in combination with how many video drivers with how many printer drivers?

Suppose a program lets you add two numbers. The program's design allows the first number to be 1 and 100 and the second number to be between 1 and 25. The total number of pairs you can test (not counting all of the pairs that use invalid inputs) is 100 x 25 (2500).

In general, if you have V variables, and N1 is the number of possible values of variable 1, N2 is the number of possible values of variable 2, and NV is the number of values of variable V, then the number of possible combinations of inputs is N1 x N2 x N3 x . . . x NV. (The number is smaller and the formulas are more complicated if the values available for one variable depend on the value chosen for a different variable.) It doesn't take many variables before the number of possible combinations is huge.

We can use heuristics (domain testing) to select a few "interesting" combinations of variables that will probably reveal most of the combination bugs, but we can't be sure that we'll catch all of them.

Can't test all the paths

A path through a program starts at the point that you enter the program, and includes all the steps you run through until you exit the program. There is a virtually infinite series of paths through any program that is even slightly complex.

I'll illustrate this point with an example from Glen Myers' (1979) excellent book. Some students of mine have had trouble with Myers' diagram, so Figure 1 present the example in a flowchart-like format.

INSERT FIGURE 1 HERE

The program starts at point A and ends at Exit. You get to Exit from X.

When you get to X, you can either exit or loop back to A. You can't loop back to A more than 19 times. The twentieth time that you reach X, you must exit.

There are five paths from A to X. You can go A to B to X (ABX). Or you can go ACDFX or ACDGX or ACEHX or ACEIX. There are thus 5 ways to get from A to the exit, if you only pass through X once.

If you hit X the first time, and loop back from X to A, then there are five ways (the same five) to get from A back to X. There are thus 5 x 5 ways to get from A to X and then A to X again. There are 25 paths through the program that will take you through X twice before you get to the exit.

There are 5^3 ways to get to the exit after passing through X three times, and 5^{20} ways to get to the exit after passing through X twenty times.

In total, there are $5 + 5^2 + 5^3 + 5^4 + \ldots + 5^{20} = 10^{14}$ (100 trillion) paths through this program. If you could write, execute, and verify a test case every five minutes, you'd be done testing in a billion years.

Sometimes when I show this example to a non-programming executive, he doesn't appreciate it because he doesn't realize how simple a program this is. This program has just one loop. Most programs have many loops. This program has only five decision points. (For example, there's a decision point at A because you can go either from A to B or from A to C.) Most programs have many decisions. Students in their first semester of their first programming class will probably write more complex programs than this one.

Another path example

Some people dismiss the path testing problem by saying you can run all the tests you need with *sub-path testing*. A sub-path is just a series of steps that you take to get from one part of the program to another. In the above example, A to C is a sub-path. So are ACD, ACDF, and ACDFX. Once you've tested ACDFX, why should you test this sub-path again? Under this viewpoint, you'd probably test ABX, ACDFX, ACDGX, ACEHX and ACEIX once each plus one test that would run you through X the full 20 times. Think realistically, these people say. The program is not going to be so weirdly designed that it will pass a sequence like ACDFX one time and then fail it the next time. Sometimes they're wrong.

If you go through a sub-path with one set of data one time, and a different set of data the next, you can easily get different results. And the values of the data often depend on what you've done recently in some other part of the program.

Here's the example that I use to illustrate the point that simple sub-path testing can miss devastating bugs. As one of the system's programmers, I helped create this bug and then helped troubleshoot it after we started getting calls from beta testers in the field. I've simplified the description and I've probably misremembered some minor details, but the essence of the bug and the circumstances are captured here.

The application was a voice/data PBX (private telephone system) that used a custom station set (telephone, but with a built-in central processor, memory, and display). The station set had an LCD display. The top line showed information, the second line showed choices. For example, when you were talking to someone, the second line would show choices like "Transfer" and "Conference." Press the button below "Transfer" to transfer the call. If you had calls on hold, you could press a button to display the hold queue. For each call, you'd see a person's name or phone number, or a telephone line (we showed the best information we had). You could have up to ten calls on hold at the same time. The display could show up to four calls at once, and you could scroll through the list. Press the button under the name to connect to the call.

Phone systems have conflicting design goals. For example, fast performance is very important. When you pick up a phone's handset, you expect to hear dial tone right away. But you also expect absolute reliability. Error checking and error recovery take time and can interfere with performance.

When a call was put on hold, we stored information about it on a stack. We could have up to ten calls on hold at once, so the stack had to be ten deep. We made it large enough to hold twenty calls, just in case we somehow forgot to take a call off of the stack when you connected to it or you or the caller hung up. We had no reason to expect stack problems, but the essence of fault tolerant programming is that you anticipate problems that the system *could* have and make sure that if those problems ever arise (perhaps when someone adds a new feature), they are going to be controlled.

INSERT FIGURE 2 HERE

We would have liked to check the integrity of every stack every time that a phone call changed state (such as going on or off of hold). Unfortunately, this took too much time. Instead, we defended against the possibility of an error by allowing stack space for 20 calls instead of 10. We also created a Stack Reset command. Any time your phone got to an idle state (no calls connected, on hold, or waiting), we knew that your stacks had to be empty. Therefore, we forced your stacks to *be* empty.

Take a look at the simplified state diagram, Figure 2.

Your station set starts in the Idle state. When someone calls you, the set moves into the “Ringing” state. Either you answer the phone (“Connected” state) or the caller gives up, hangs up, and you go back to Idle. While you’re connected, you or the caller can hang up (back to Idle in either case) or you can put the caller on hold. When the caller is on hold, you can hang up on him. (This lets you get rid of a caller who is behaving inappropriately, without making it obvious that you are intentionally hanging up.)

INSERT FIGURE 3 HERE

Unfortunately, we missed a state transition. Sometimes, when you put a person on hold, she hangs up before you get back to her. See Figure 3. In retrospect, and in the way that I describe the state transitions, this is obvious. In retrospect, it is astonishing that we could have missed this case, given the ways that we drew our diagrams and discussed/reviewed our approaches. But we did. Mistakes happen, even when bright, careful people try to prevent them.

The effect of the mistake was quite subtle. When the caller hangs up while she waits on hold, her call is dropped from the hold queue. The call no longer appears on your display and you can still put up to ten calls on hold. Other system resources, such as time slices reserved for the call and phone lines used by the call, are freed up. It is impossible for the system to try to reconnect to the call; the only error is that a bit of space on the stack is wasted for a short time. Because the stack depth is 20 calls, and the design limit is 10 calls on hold, the system can tolerate an accumulation of 10 such errors before there is any performance effect. Furthermore, as soon as the phone returns to the idle state, the stacks are cleared, and the wasted stack space is recovered.

If you did manage to run your phone so busy for so long that eleven of your callers hung up while on hold before you got the phone back to an idle state, and if you then tried to put a tenth call on hold, your phone would take itself out of service. It is bad for a telephone system to lose or drop calls, so calls on hold or waiting on the phone going out of service were automatically transferred to the hold and waiting queues of up to two other phones that you had designated as back-up systems. Once the calls were cleared off your phone, it rebooted itself, downloaded code and data from the main server (the PBX’s main computer) and was back in service within two minutes.

Suppose that the phones that your calls were transferred to were as busy as yours. Your transferred calls would fill their hold queue, people would wait longer to be answered, people would hang up while they were waiting, and soon those phones would crash too. And as they crashed, they would send their calls back to your phone. This is a one-phone-at-a-time example of a “rotating outage.”

We saw this problem at a beta site of stock brokers, the gymnasts of the telephone world. They loved our phone, except when it would mysteriously crash when a stockbroker was at her busiest. These crashes were expensive.

We were lucky that this was the worst consequence of the bug. Some communities’ 9-1-1 (emergency response) administrators were interested in beta testing our system. We were reluctant to send buggy code to 9-1-1 so, fortunately, we never did set up such a beta site. Back in those days, some 9-1-1 systems practiced triage. If you called 9-1-1 and the dispatcher decided that your call wasn’t urgent, then he would put you on hold and check the next waiting call. Once the genuine emergencies were handled, the nonemergency calls were taken off hold and dealt with. Under these circumstances, every phone's hold queue would often be full. Our bug would have been crashing 9-1-1 telephones, and that reduction in emergency response service would probably have resulted in unnecessary deaths. This was a serious bug.

Think about the paths through the program that you would have to test to find this bug in the laboratory. I'll define a "path" as something that starts when your phone is idle, goes through various steps, but ends up with the phone back at idle. A path might correspond to receiving an incoming call, answering it, receiving a second call, answering it (puts the first call on hold), switching between calls (putting each on hold), setting up a conference (all three talk together), receiving another call during the conference, putting the conference on hold, more switching back and forth, making an outgoing call with all of these people on hold, transferring the call, then reconnecting to the held calls and ending them, one at a time, until the phone was idle. That is one path.

As you design your set of tests, note that you have no expectation of finding stack bugs. Stack corruption issues had been cleared up about a year before. There were no known existing stack problems. Therefore you probably will not be taking special care to search for new stack failures. Furthermore, if you do run stack-related tests, you will discover that you can put 10 voice calls on hold, 10 data calls on hold, deal with 10 voice calls and 10 data calls waiting, and take calls on and off hold on any of those channels with no problem. If you hang calls up while they are holding, all visible indications are that that call is terminated correctly. And because the stack is fully reset as soon as the phone is allowed back to idle state (no calls connected, waiting. or on hold), you will not see a failure unless your test involves having 11 callers hang up while waiting on hold, and putting another 10 calls on hold together, before taking the phone back to idle. You can use a debugger to check the contents of the stack (we did glass box and black box testing), but if you run a simple test like putting the call on hold, letting the call disconnect itself, and then watching the phone reset its stack, the phone will appear to work correctly, unless you are very, very observant, because the system does a stack reset for your phone as soon as it hits idle, which happens in this test only a few instructions after the caller hangs up. So the stack will be clear, as it should be, after the simple caller-hung-up-on-hold test.

If you were not specifically looking for a problem of this kind, how huge a set of path tests would you have to design before you could be reasonably confident that your set of tests would include one that would reveal this bug? I think the number of paths (from idle state to idle state) that you would have to explore before you stumbled across this bug would be huge.

Lots of other tests

A system can fail in the field because the software is defective, the hardware is malfunctioning, or the humans who work with the system make a mistake. If the system is designed in a way that makes it likely that humans will make important mistakes, the system is defective. Therefore, to find all defects in a system, you have to test for usability issues. And hardware/software compatibility issues. And requirements conformance issues, timing issues, etc. There are lots and lots of additional tests.

Dick Bender publishes a program called SoftTest that analyzes a well-written external specification and produces an optimal set of tests using a method called *cause-effect graphing* (see Myers, 1979 or Kit, 1995). This method helps you select a relatively small set of combinations of data and sub-paths that covers a wide range of potential problems. As part of the process, SoftTest estimates the number of unique test cases that could be run through the program. I was at his office when they analyzed a complex program. The computer plodded along for a while, then printed its estimate: 10^{100} tests. (SoftTest then suggested a set of about 3000 tests that covered the situations that cause-effect graphing would tell you were interesting.) 10^{100} tests is such a big number that you might not realize how big it is. So here's a comparison. I've been told that current estimates of the number of molecules in the universe is 10^{90}. No matter how many testers and how much automation you use, you aren't going to run 10^{100} test cases in a commercially reasonable time.

Coverage Monitors

Some testing experts call coverage monitors state-of-the-art tools and tell us that we aren't doing adequate testing unless we can demonstrate 100% coverage using a coverage monitor. I don't think the situation is that straightforward (Kaner, 1999b), but there's room for argument. But other people go farther than this, saying that if you achieve 100% coverage as measured by a coverage monitor, then you have achieved "complete" coverage and you have done complete testing. This is baloney.

A coverage monitor is a program that will measure how many tests you have run of a certain type, out of a population total of possible tests of that type. You achieve complete coverage when you finish running all of the tests of that type. For example, the monitor might check how many statements (or lines of code) you have executed in the program. A more sophisticated monitor recognizes that you can branch in many ways from one statement to others, and counts how many branches you have executed. You achieve complete statement-plus-branch coverage when you've tested every statement and every branch. Most discussions of coverage that I see are either of statement-plus-branch coverage or of some technique that adds a bit more sophistication to the same underlying approach.

Coverage-based testing can miss data flow-related bugs

Statement and branch coverage look at software in the same way that you would if you were preparing a detailed flowchart. In 1967, in my first programming class, I learned that the key to good program design was to draw flowcharts. But the field has made progress since then. For example, it's often more important to look at the flow of data through the program, the ways that inputs are transformed into outputs, than to think about what instruction the processor will execute next.

More people know how to read flowcharts than data flow diagrams, so I'll present a data flow problem here in flowchart style. This is based on an example developed by Dick Bender in his course on Requirements-Based Testing. See Figure 4.

In Figure 4, we have a 7-statement program that uses one variable, X. This is a payroll program. It calculates your paycheck. X is the amount you are paid.

On line 1, the program initializes X. We'll give the program a bug, and say that line 1 initializes X to $1,000,000.

Line 2 calculates your base pay, setting a new value for X in line 3.

Line 4 determines whether the amount of your pay should be calculated using a different formula, perhaps because you are on vacation and so you get vacation pay instead of pay for hours worked. If so, X is reset in line 5.

At line 7, the program prints your check.

There are three first-order data flows in this program. Think of a data flow as involving a path from the place or event that sets a variable's value to a place where that value is used. The flows are 1-to-7 (line 1 sets a value for X, and the value is used in line 7), 3-to-7, and 5-to-7.

Now let's do some coverage testing.

For our first test, we'll go through lines 1 (sets X), 2, 3 (sets X), 4, 6, and 7 (prints X). This covers all the lines except 5 and it covers the branches from 2 to 3 and from 4 to 6. To achieve complete line and branch coverage, we need a test that includes line 6 and the branches from 2 to 4 and from 4 to 5. Therefore our next test runs through lines 1 (sets X), 2, 4, 5 (sets X) and 7 (prints X).

We now have complete line and branch coverage. Have we tested the program completely? Absolutely not.

This "complete" pair of tests only covers two of the program's data flows, 3-to-7 and 5-to-7. If you stopped here, no one would ever know that you'll get paid $1,000,000 if the program ever runs the path through lines 1 (sets X), 2, 4, 6, 7 (prints X).

Interrupt-related bugs

Here's another example of the holes in flowchart-driven (line/branch) testing. While a computer is executing a main program, it might receive a signal, called an interrupt, telling it to switch focus to another routine called the interrupt handler. The computer will save certain working variables in a temporary storage spot, then handle the interrupt, then recover its working variables and return to the main routine as if nothing had happened. You might jump to the interrupt handler from (almost) any line in the main program. Coverage monitors don't count these implicit jumps from every line to the interrupt handler, for every possible type of interrupt. A program that deals with an interrupt can affect the main routine in many ways. For example, it might reset a variable that's in use or take a device out of service. Here's a specific example.

In the main program, we'll input values for A and B, then check whether B is zero, and if not, divide A by B. Suppose that right after we check B, but before we divide, there's an interrupt. The effect of the interrupt is to reset B to 0. Now we come back to the main routine. It knows that B is not zero because it just tested B. Therefore it divides A by B (which has just been reset to 0) and crashes.

This situation might seem far fetched, but it is not unusual in event-driven systems, in which many different events can happen at any time. Traditional programmers often write routines that input A, input B, check B for zero, do a bunch of stuff, then divide A by B. In an event-driven world, programmers learn the hard way that the more intervening processing between the time you check B's value and the time you use it, the greater the chance that B's value has changed while you weren't looking. Therefore, real-time programmers will often check for B=0 occur *just before* B is actually used as a divisor, *each time* B is used as a divisor. This appears to make no sense in the mainline code, but it makes great sense when you remember that at any instant the program might switch to some other routine that can affect the values of the program's variables.

Coverage monitors miss issues like these.

Missing code

The problem in the telephone example was that we had forgotten to write about 10 lines of code to handle one state transition. All of the lines of code that we did have worked. We tested all of the relevant lines of code and didn't find this bug.

If you divide A by B, and forget to check for B=0, the program doesn't crash because a bad line is present. It crashes because the line that was supposed to test for B=0 is not there. The program is broken, even though every line and every branch work properly.

Coverage monitors tell you whether you have tested every line and every branch. They don't tell you that you need more lines and branches.

Requirements-related errors

Suppose you're writing a program to compute taxes, and the government changed its tax rules such that income earned before May 5, 1997 is taxed at one rate and income earned after May 5 is taxed at a different rate. The programmer, having never read the tax laws, calculates using the current (post-May) rate without realizing that a different formula is needed before May 5.

A coverage monitor will show that every line is tested and working, but it will not help you discover that a system requirement (calculate taxes for January to May correctly) has not been met and will not help you discover that another requirement (calculate taxes for May 5 itself) is ambiguously specified because the law refers to dates before and after May 5.

Compatibility/configuration and other errors

Coverage-based testing won't tell you that the program works with one printer but not with another allegedly compatible one. It won't tell you that the documentation is error-ridden, that the program is unusably complex, that the program fails only on specific values, that the program fails when too many processes are running or too many computers are demanding attention at once.

Coverage-based testing tells you that you have traced though the flowchart. No more, no less. Flowcharts are good, but bugs know how to hide in them. Coverage-based testing is not complete testing unless we broaden our definition of coverage dramatically, in which case we discover 10^{100} possible tests and never achieve complete coverage.

What Does This Mean For Software Quality?

I wouldn't spend so much of my time writing about testing and teaching people to be better testers if I wasn't absolutely confident of the value of testing. But if you can never come close to completeness in testing, then you would be unwise to think of testing as "quality assurance." Testing groups that name themselves "QA" are misleading themselves and the company they work in.

Some testing groups add a few metrics-measurement, inspection, and standards-compliance functions. Do this make them "quality assurance" groups. No.

Think back to the telephone system described above. An important aspect of the reliability of that system was the management of time-related risks. Suppose that you were running a QA group and the programmers made a lot of time-related bugs. Could you send the programmers out for retraining? Could you require them to adopt the new practices? Could you give bonuses to the ones who made the most progress? If you don't have the authority to manage and train the programming staff, then you don't have the power to assure quality. If you do have that power, your title is probably Vice-President of Product Development, not Manager of Quality Assurance. Testing groups and testing-plus-other-measurements groups are doing Quality Assistance not assurance.

Quality is multidimensional. A stakeholder in a product will justifiably consider a product to be low quality if it doesn't satisfy his reasonable needs. Those needs differ widely across stakeholders. For example, even if the English language version of a product is reliable, a localization manager can reasonably call it unacceptable if her team has to recompile it in order to translate and adapt the program for another country. A support manager can reasonably call a reliable program unacceptable if it is impossible for customers to understand. A program that isn't written in a maintainable style won't stay reliable for long. And a customer might be happier with a program that occasionally crashes but is easy to use and that does the tasks that he needs, rather than a harder, less complete program whose failures can never be blamed on coding errors.

Software companies have to balance these different quality-related demands. To do it, they have to make tradeoffs during design and development. A drive to eliminate all coding errors from a program might not be the most effective drive to improve its overall quality.

Software publishers have to balance these demands because they have economic constraints. By spending a few dollars to prevent software failures, you *can* avoid wasting a huge amount of money on the consequences of those failures. For example, at an average cost of about $23 per technical support call, you can save $1 million by avoiding the need for about 43,500 calls. Several mass-market products have such high call volumes that they can save a fortune by investing in call volume reduction. But how much should a company spend if it doesn't know how many copies of the product it will sell? We make guesses about product volumes, but in my experience, many of those guesses are wrong. At what point do the direct cost (staff salaries), the sales lost due to delay, and the opportunity costs (you want to be building your next product) outweigh the benefit to be gained by finding and fixing more bugs? Must we run every test, in order to search for every bug? Are there no bugs that can be left unfixed? Quality is *sometimes* free (over the long term). Quality/cost analysis is rooted in tradeoffs. (Kaner, 1996a).

Remember too that there are significant costs of delayed time to market. As the best known example, a product that comes first to market will often sell much better than a technically superior product that comes out a few months later. What is the proper tradeoff between investment in quality and cost of delay in the marketplace?

The quality/cost tradeoffs are difficult. I don't think that it is possible today to release an affordable product that is error free and that fully meets the reasonable quality-related objectives of all of the stakeholders (including the customers). The goal of what is recently being called the "Good Enough Software" approach is to make sure that we make our tradeoffs consciously, that we look carefully at the real underlying requirements for the product and ask whether the product is good enough to meet them. Many of these requirements are not expressed clearly, and our initial statements of them might overstate or understate the actual needs. Our understanding of requirements shifts as we learn more about the product and the stakeholders. (Lawrence, 1997; Lawrence & Johnson, 1997).

Among the tradeoffs that we make are decisions about how extensively to look for bugs—if we can't do all possible test cases, we have to do less, and we will miss some. Our goal is "Good Enough Testing." (Bach, 1997a).

Another piece of the "good enough" approach is explicit recognition of the fact that software companies choose to not fix some known bugs. I was surprised recently to read that the "good enough" approach defines itself in terms of not fixing all the bugs, and then to read that no one really deliberately ships with known bugs. If you haven't read that claim yet, don't bother looking for it. But if you did, let me say first that every software company that I have ever personally worked with or consulted to has made conscious, deliberate decisions to release software with known bugs. This is not new—I wrote about this practice of deferring bugs (we'll fix them later) as a normal practice in the industry in the first edition of my book (Kaner, 1988), extended the discussion of the bug deferral process in the second edition (Kaner, Falk & Nguyen, 1993), and no one has contacted me to tell me that this is unrealistic. I teach a course on testing at UC Berkeley Extension, at various hotels, at ASQC (now ASQ) San Francisco, and at many companies and the course includes extended discussion of risk analysis and persuasion techniques to minimize the number of inappropriately unfixed bugs. None of these students tell me that they fix all the bugs in their company. Many of them have shared with me the process they use for analyzing whether a particular bug must be fixed or not. As it applies to bug deferral decisions, which is only a small part of its scope, the goal of the Good Enough Software approach is to see that these bugs are analyzed thoughtfully and that decisions are made reasonably.

The Good Enough Software approach is difficult because we reject the easy formulations, like "Quality is Free" and "Test Everything." We live in a world of tradeoffs. We see imperfection as a fact of life. Our objective is to choose design, development, testing and deferral strategies that help us manage imperfection in the service of developing products that, despite their flaws, are excellent tools for their purposes. (Bach, 1997a,1997b).

Back to the Legal Stuff

What does it mean to accept that imperfections are inevitable, and that complete testing is impossible? Should we set a legal standard of perfection anyway and penalize manufacturers for all bugs, no matter how good the product is? I think not.

But there has to be a limit somewhere. It's one thing to release a product with a bug that you never found, despite reasonably good development (including testing) practices. It's a different matter when you release with a known bug that you didn't disclose to the customer. The customer uses the product in a normal way, runs into the bug and loses work and data, including hardware configuration data or data created by some other program. Recovering this will be expensive. If the software publisher knew of this problem, ran its cost/benefit analysis and chose to ship it anyway, why should the customer bear the cost of the bug?

And why should we protect publishers who choose to do virtually no testing? If they pass off their software as normal, finished product and it crashes all the time and does other damage, why should customers pay for the cost of that other damage?

The question of who pays for damage plays a big role in cost/benefit analyses (Kaner, 1996a; Kaner & Pels, 1997). If the publisher has minimal risk, it can afford to ship shoddier products.

Let me give you an example from the draft statute that is before NCCUSL. The draft addresses liability for viruses. The Reporter (senior author) of the draft wrote a cover memo to NCCUSL (Nimmer, 1997), in which he says that ***"Significant new consumer protections are proposed . . . The significant new protections include: creation of a non-disclaimable obligation of reasonable care to avoid viruses in the traditional mass market"*** (my italics). In this list of five significant new consumer protections, virus protection came first.

Look in the draft (NCCUSL, 1997) at Section 2B-313, Electronic Viruses. As advertised in the cover memo, 2B-313 does require software publishers to "exercise reasonable care to ensure that its performance or message when completed by it does not contain an undisclosed virus." This requirement applies to software products sold in stores, like Microsoft Works or a flight simulator game. For products sold over the internet or for products costing more than about $500 (there are other exceptions too), the publisher is allowed to disclaim liability for viruses by burying a statement that "we didn't check for viruses" in the fine print of a license that you won't see until after you buy the product.

The section defines reasonable care as searching for "known viruses using any commercially reasonable virus checking software." I don't know any software publisher today who only looks for known viruses, and very few mass market publishers use only one virus checker. According to the reviews that I see on the web (PC World and CNET cover this from time to time), no virus checkers catch 100% of the known viruses. Further, virus creators sometimes design their virus to be undetectable by the current version of a specific virus checker. (See the discussion of Microsoft AV in Cobb, 1996.) Many publishers (I suspect that this is the large majority) use at least three virus checkers, at least in the Windows and Windows 95 markets. The statute is thus defining "reasonable care" as something less than the minimum that I've seen done by any even-slightly-concerned software publisher. I don't think that this is a drafter's accident—I've explained virus-related issues to the Drafting Committee at length, at two separate meetings.

If the publisher exercises what the statute defines as "reasonable care," you can't sue it for a virus on the disk.

Section 2B-313 also requires ***customers*** to exercise reasonable care to check for viruses, and defines reasonable care in the same way for customers as for publishers. If you don't, then under Section 2B-313(d) "A party is not liable if . . . the party injured by the virus failed to exercise reasonable care to prevent or avoid loss."

Let's see what happens if you buy a program at a store, that has a virus that trashes your hard disk. You pay money to a technician to recover (some of) your files and to get your system working again.

If the publisher checked for viruses by using only one virus checker, you (the customer) can't recover for your expenses and lost data because the publisher has exercised "reasonable care."

If the publisher didn't check for viruses, but you (or your child) trustingly installed the program on your computer without checking for a virus, you can't recover because you didn't exercise "reasonable care."

What if the publisher didn't check for a virus, but you did? Unfortunately, you missed it. You've exercised reasonable care and the publisher has not. Do you get reimbursed for recovery of your files and your system? Nope. After all, the publisher's duty is to check for a virus using one virus checker. Even though it didn't do that, you proved that if the publisher *had* checked for a virus using a commercially reasonably virus checker (yours), it wouldn't have found a virus. Therefore its failure to exercise reasonable care didn't cause the damage and therefore (this is how negligence law works, folks), the publisher is not liable.

OK, what if the publisher made it impossible to check its product for viruses until after you installed the program? For example, some manuals explicitly instruct you to turn off your virus checker during installation. You got the virus because you followed the instructions. *Now* do you get reimbursed for recovering your files and system? Nope. This is a consequential loss, and the publisher gets to disclaim all liability for consequential losses in the fine print of the you-can't-see-it-until-after-you-buy-the-product license.

When I was in law school, one of our professors defined the "blush test" by saying that an argument or a lawsuit is frivolous if you can't present it to the judge without blushing. I don't know how anyone could pass the blush test while saying that Section 313 provides a significant new consumer protection.

Section 313 will probably be revised—it's just too obvious. But over the coming 18 months, many other claims will be made about the fairness and customer protection features of Article 2B. When you hear them, think about Section 313 and ask your friends and state legislators to look carefully before they believe that Article 2B will promote the marketing of good enough software or even provide basic incentives for honesty and fair dealing in the industry.

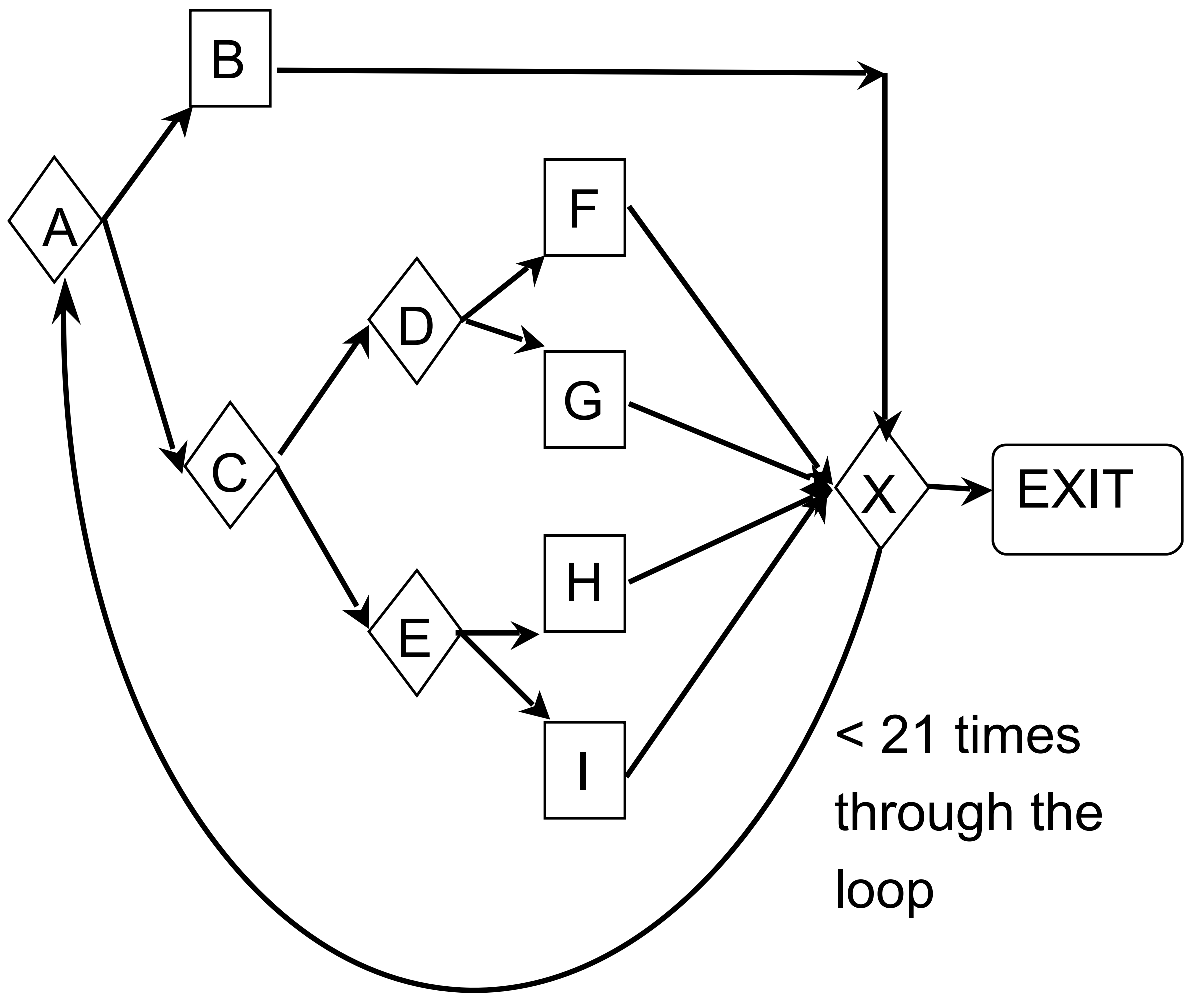


Figure 1. Flowchart of the paths example

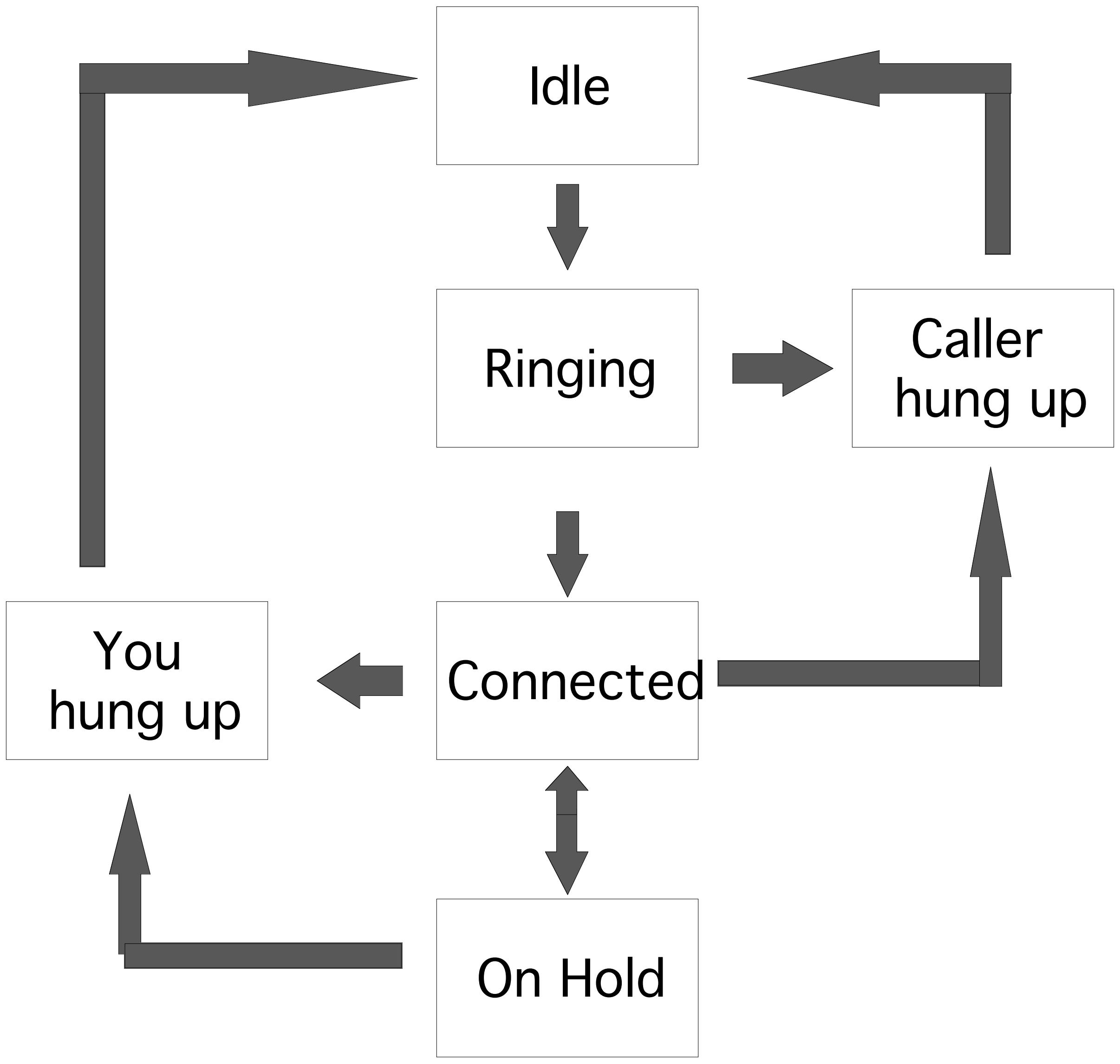


Figure 2. State diagram of the phone system

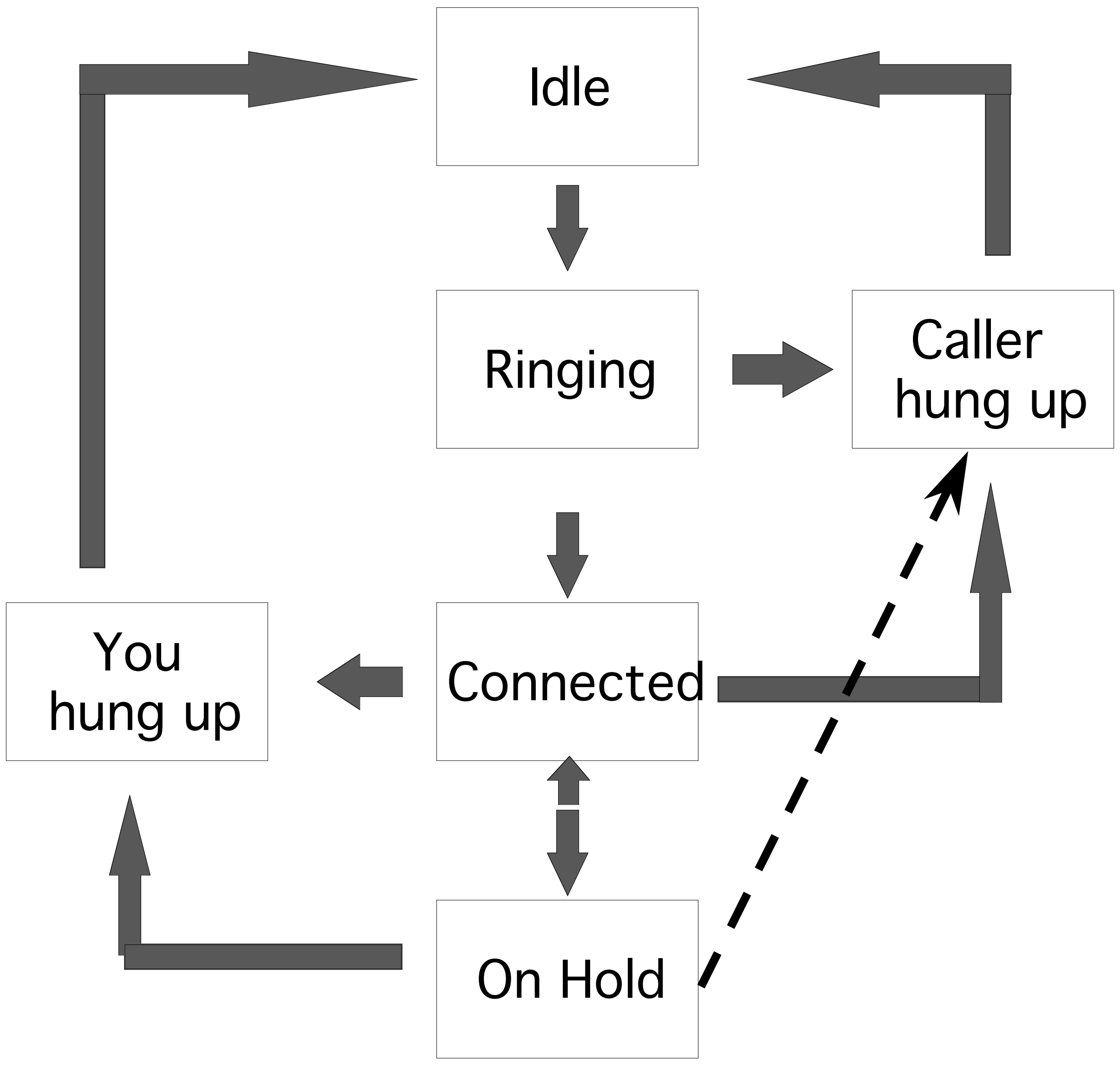
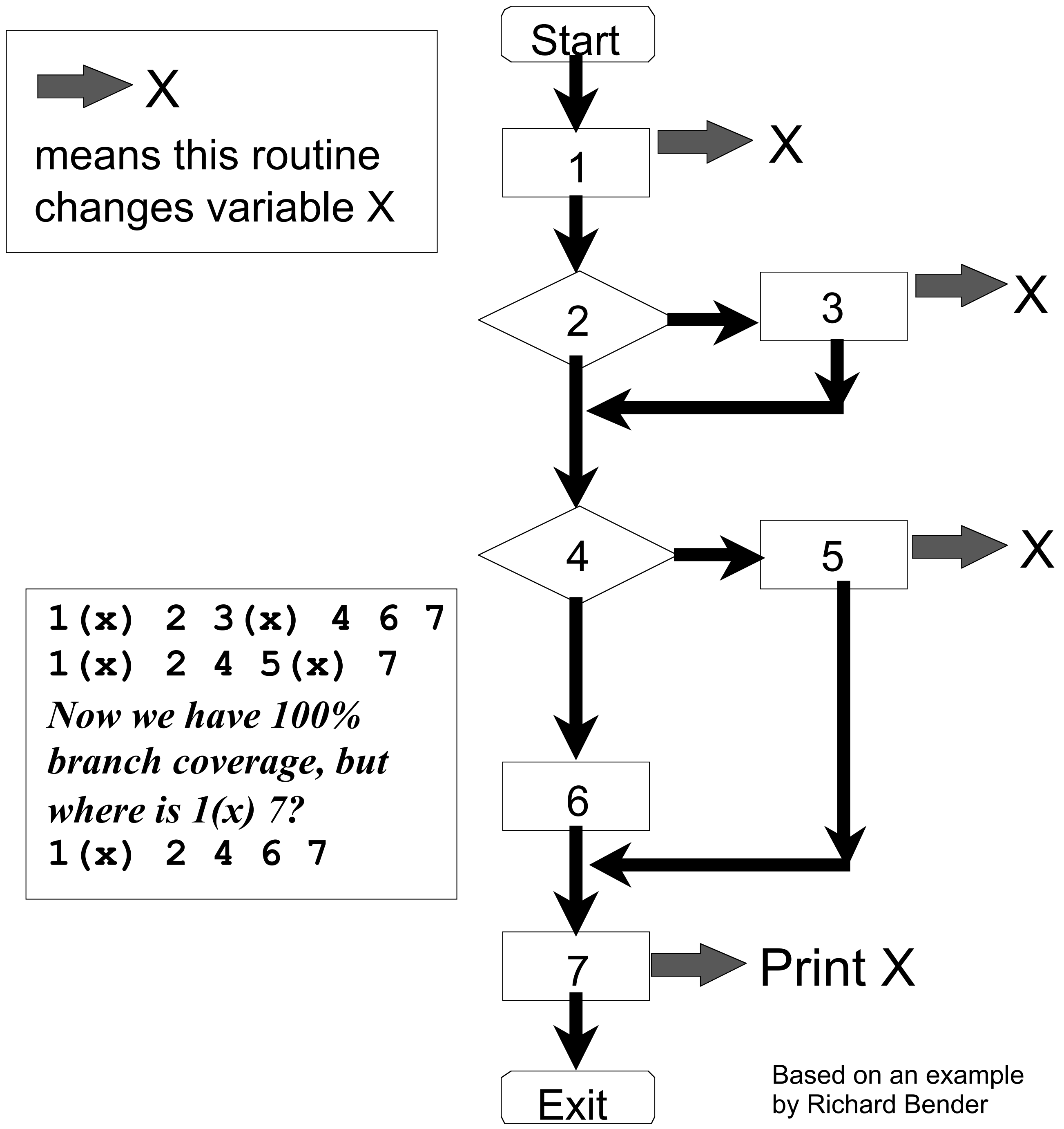


Figure 3. Corrected state diagram of the phone system.


X
means this routine changes variable X
Start
1
X
2
3
X
4
5
X
6
7
Print X
Exit
Based on an example by Richard Bender

```
1(x) 2 3(x) 4 6 7
1(x) 2 4 5(x) 7
```
Now we have 100% branch coverage, but where is 1(x) 7?
```
1(x) 2 4 6 7
```

Figure 4. Complete coverage misses a dataflow bug.

References

Abramowitz, M. & Stegun, I.E. (1964) *Handbook of Mathematical Functions.* Dover Publications.

Bach, J.S. (1997a) "Good enough testing for good enough software." *Proceedings of STAR 97 (Sixth International Conference on Software Testing, Analysis, and Review,* San Jose, CA., May 7, 1997, p. 659.

Bach, J.S. (1997b) *Is the Product Good Enough?* Unpublished manuscript, probably available at www.stlabs.com.

Cobb, S. (1996) *The NCSA Guide to PC and LAN Security.* McGraw-Hill.

Humphrey, W.S. (1997) *Comments on Software Quality*. Distributed to the National Conference of Commissioners on Uniform State Laws for their Annual Meeting, July 25 – August 1, 1997, Sacramento, CA. Available at several websites, including www.badsoftware.com.

Kaner, C. (1988) *Testing Computer Software.* TAB Professional & Reference Books.

Kaner, C., Falk, J., & Nguyen, H.Q. (1993) *Testing Computer Software*. 2nd Ed., International Thomson Computer Press.

Kaner, C. (1996a) "Quality cost analysis: Benefits and risks." *Software QA*, vol. 3, #1, p. 23.

Kaner, C. (1996b) "Software negligence and testing coverage." *Proceedings of STAR 96 (Fifth International Conference on Software Testing, Analysis, and Review,* Orlando, FL, May 16, 1996, p. 313. An earlier version of this paper appeared in *Software QA Quarterly*, Volume 2, #2, 1995, p. 18.

Kaner, C. (1996c) "Negotiating testing resources: A collaborative approach." Presented at *Software Quality Week*, San Francisco, CA, May, 1996.

Kaner, C. & Pels, D. (1997) "Software customer dissatisfaction." *Software QA,* vol. 4, #3, p. 24.

Kit, E. (1995) *Software Testing in the Real World*. ACM Press: Addison-Wesley.

Lawrence, B. (1997, April) "Requirements happen." *American Programmer*, vol. 10, #4, p. 3.

Marick, B. (1997) "Classic testing mistakes." *Proceedings of STAR 97 (Sixth International Conference on Software Testing, Analysis, and Review,* San Jose, CA., May 7, 1997, p. 677.

Myers, G.J. (1979) *The Art of Software Testing.* Wiley.

National Conference of Commissioners on Uniform State Laws (1997) *Draft: Uniform Commercial Code Article 2B – Licenses: With Prefatory Note and Comments.* Available at www.law.upenn.edu/library/ulc/ulc.htm in the Article 2B section, under the name *1997 Annual Meeting.*

Nimmer, R. (1997) *Issues Paper: UCC Article 2B – Licenses.* Distributed to the National Conference of Commissioners on Uniform State Laws for their Annual Meeting, July 25 – August 1, 1997, Sacramento, CA.