DEBUGGING MYSELF

Brian Hayes

A couple of years ago, I had one of those annoying moments of self-discovery. Sitting at my computer, I clicked on an icon to launch a program. The software obediently uncoiled itself from the storage disk and loaded into memory—then silently folded up again and went away. I clicked on the icon again, and the program went through the same futile pantomime. I clicked a third time, and watched the performance once more. The moment of self-discovery came just in time to save me from clicking yet again. What I discovered was that I must have a fairly peculiar mental model of how a computer works if I could believe that a fourth attempt might be any different from the first three. How long would I have gone on clicking? Did I think I might outlast the machine—prove that I'm more stubborn? If I kept clicking long enough, would I wear down its will to resist? Did I think of the computer as an animal that can be trained by repetition and discipline? Or perhaps I saw it as a dull child who will learn, eventually, if you keep drumming away at the same lesson?

Ever since that incident, I have been keeping a diary of computer bugs and my reactions to them. Whenever something goes wrong with one of my computers, I make a note of it: what happened, what I was doing when it happened, what I did about it. The bug logs now run to 25 pages. One reason for keeping these records is to learn something about the nature of computer malfunctions, but what interests me more is the human response to the machine and to the little surprises it holds in store for us. In other words, the aim is not so much to debug the computer as to debug myself.

Sphexishness

If I had been starting a car instead of a computer program, making two or three attempts in a row would not seem so foolish. You don't get out and call the tow truck when the engine fails to catch the first time you turn the key. And if it were a lawn mower I were trying to start, several pulls on the cord would be the norm; a mower that roared to life on the first try might be disconcerting. Why should a computer be any different from these machines?

Brian Hayes is a former editor of American Scientist. *Address: 211 Dacian Avenue, Durham, NC 27701. Internet: bhayes@mercury.interpath.net.*

In point of fact, computers *are* rather different from lawn-mower engines. From a theoretical point of view, a digital computer is a deterministic machine, whose actions can be predicted in complete detail. Specifically, a computer can be considered a deterministic finite-state autom aton, or FSA. As the name suggests, a machine of this kind has only a finite number of possible states, or configurations, available to it. A snap-action light switch is an extremely simple FSA, with just two stable states—off and on. A pair of switches controlling a single lamp make up a slightly more elaborate FSA, with four states. The number of states in a digital computer is vastly larger: Roughly speaking, a computer than can store m bits of information has 2^m possible states. Values of m commonly exceed 100 million.

Ignoring some technicalities, a deterministic FSA works as follows. The machine starts out in some well-defined initial state; then, with each input received, it makes a transition to a new state, and possibly also produces some output. In the case of a computer, the initial state is the condition established when the power is first turned on; the inputs are events such as pressing a key or clicking the mouse; the outputs could include displaying information on a screen or sending it to a printer. At every instant the state of the machine is defined by the configuration of all the bits in memory as well as the bits in the registers of the central processing unit, and perhaps a few other bits also. (The bits stored on a magnetic disk may or may not be part of the machine state, depending on whether the disk is considered part of the computer or an auxiliary device.)

A finite-state automaton is deterministic if its next state and its next output depend only on its current state and current input. Suppose in state q the machine receives input α, making a transition to state p and emitting output β. Then we know with absolute certainty that if the machine is ever again in state q, input α will evoke exactly the same response, so that we will again see state p and output β. There is no room for chance or variation. The sequence of events that brought the machine to state q does not

matter. Events in the world outside the machine do not matter. Only the current state and current input matter.

Within this deterministic vision of computer operations, my attempts to deal with a computer as if it were a balky lawn mower look fairly ludicrous. I was in state q and making input α; but each time I tried, the machine went through a cycle of operations that immediately returned it to state q. Input α would then inevitably begin the cycle all over again. The program was no more going to be started by continuing to click on its icon than a burned-out light bulb will be lit by continuing to flip the wall switch.

Douglas Hofstadter of Indiana University coined the word *sphexish* to describe behavior like mine. Wasps of the genus *Sphex* go through a brief ritual before burying a paralyzed cricket as nourishment for their offspring. If you move the cricket during the ritual, the wasp has to start all over again. Keep moving the cricket, and the wasp will keep repeating the same sequence of actions. A naturalist once played this trick 40 times in a row on a single wasp. As Hofstadter points out, the naturalist was being just as sphexish as the wasp. A metanaturalist from another planet would have had a hard time telling which species was being manipulated and which was doing the manipulating. Likewise in my contest with the computer, we both had to cooperate to keep up the dance of sphexishness.

Microstates and Macrostates

How embarrassing to be caught exhibiting the mental habits of an insect! (I really do need debugging.) But on reading through my diaries, I am led to wonder whether my behavior was really so sphexish after all. There is a fair body of evidence that just trying again sometimes *does* work. It's even possible that if I had gone ahead and clicked a fourth time, the program would have launched itself successfully.

At one point in my years of diary-keeping, I was having trouble with a communications program. I could start the software, but when I asked it to dial the telephone, the program died. This happened on three occasions, all within a month or so. In each case, I simply restarted the program—repeating exactly the same sequence of actions I had followed the first time—and it worked fine. I never discovered the cause of these sporadic failures.

Another time, the very program that provoked this long introspective debugging session—the program that put me in the sphexish loop described above—quit without warning. In this instance, however, when I clicked to restart it, all was well again. Another program failed to install correctly when I first loaded it onto a hard disk, but going through the same series of operations a second time cleared up the trouble. The diaries are full of other problems that just go away on their own or fix themselves overnight. Evidently, the balky-lawn-mower approach to computer operation is sometimes effective.

The bug diaries suggest that most of my problems are recurrent, but they are *not* reproducible on demand. For example, the complaint that turns up most frequently (nine instances) concerns a glitch in my word-processing software. On occasion the program would dump a jumble of unprintable characters into a document, like a digital inkblot. The error occurred following a specific sequence of actions, but running through those actions would not reliably produce the error; it showed up only about one time out of 50.

The most infuriating bugs in my collection marred the operation of a small utility program whose function was to provide a hierarchical

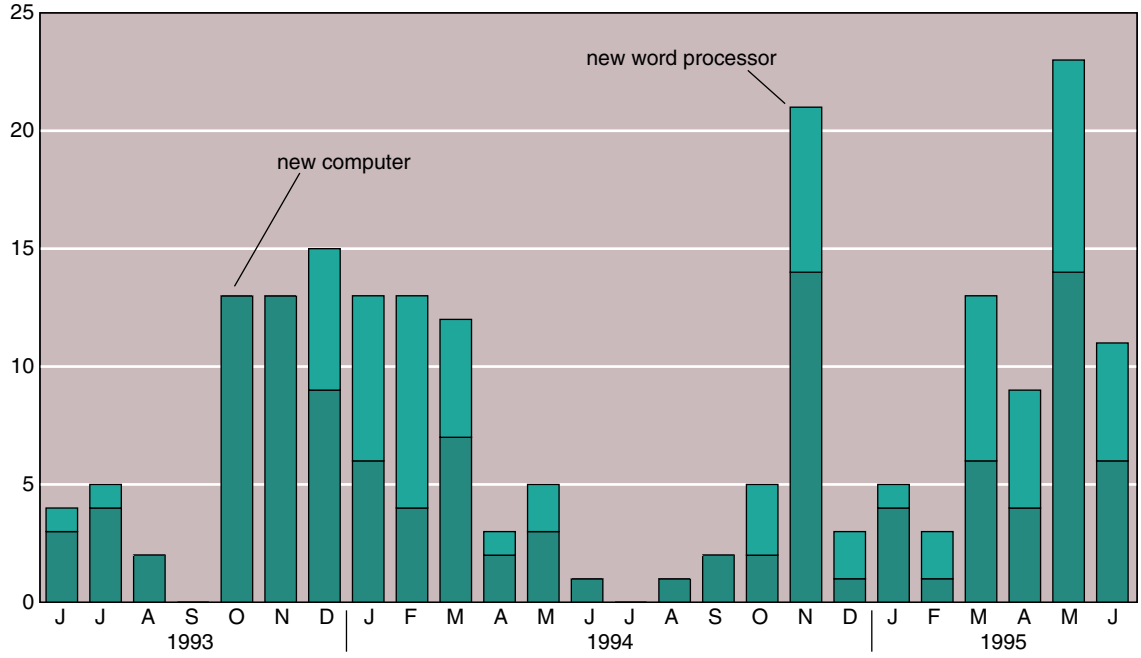

Figure 1. More than 200 bugs were recorded in two years. Darker bars tally only novel problems, ignoring recurrences.

display of menus and submenus. It was a handy tool—which now and then crashed the entire system. In particular, if I selected an item at some intermediate level in the nested structure of menus, the probability of a crash was about 0.1. Thus the crashes were just rare enough to tempt me to live dangerously.

How can these haphazard failures and fluky, random cures be reconciled with the view of a computer as a totally deterministic machine? One answer is that a real, physical computer isn't truly deterministic. The finite-state machine of computer science is an abstraction, or idealization—something like the dimensionless point of geometry or the frictionless gears of elementary physics. A real computer has to be built out of imperfect parts. Because of hardware failures or design errors, the machine could conceivably get stuck between two discrete states. Quantum fluctuations could cause the system to shift spontaneously from one state to another. Certain possible inputs—such as a cosmic ray passing through a memory chip, or a lightning strike on the power line, or an irate user's sledgehammer attack—could leave the machine in an indeterminate state. Any of these phenomena could disqualify the computer from the status of finite-state automaton.

A few events of this kind do show up in my bug diaries. Three times, as I sat down at the computer in dry winter months, a spark zipped from my finger to the mouse, with interesting consequences. But it is extremely unlikely that outside disturbances or arbitrary changes of state can explain the hundreds of other incidents in the diaries. If the inkblot bug in my word processor was caused by cosmic-ray strikes, how did the rays find just the right bit to clobber on nine occasions, and in three different computers? The problem with such explanations is that they explain too much; they can explain *anything*.

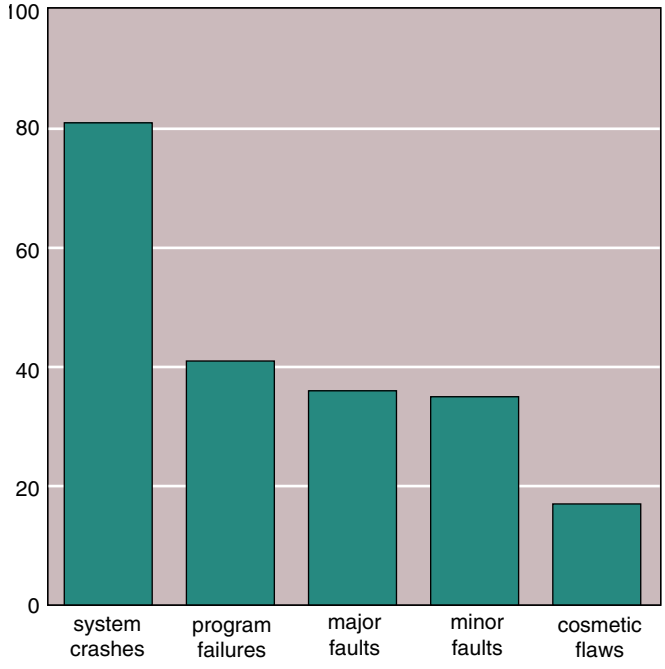

Figure 2. Spectrum of bugs classified according to their severity.

With very rare exceptions, computers *do* function as deterministic finite-state machines. If you put the machine in the same state and supply the same input, you will always get the same result. The catch is that putting the machine in the same state is no easy matter. A computer with $2^{100,000,000}$ possible states will necessarily have a great many states that superficially look alike but differ in their internal details.

In trying to reproduce a given state, you might run the same set of programs and invoke the same commands on the same data. These visible aspects of the computer's configuration can be termed its *macrostate*. For every macrostate there are a multitude of possible *microstates*, corresponding to the underlying bit patterns. Although the same programs are running in two macrostates, they may be loaded into different areas of memory, yielding very different microstates. Although the same commands are issued, they may be differently synchronized with the many unseen background processes that keep the computer going—processes for refreshing the display screen, blinking the cursor, reading the keyboard, and so on. Although a macrostate seems static, the microstate is changing with every cycle of the central processor chip, 10 or 50 or even 100 million times per second.

If the computer were making random transitions through its $2^{100,000,000}$ microstates, the probability of its ever stumbling on the same state twice would be negligible. (At 100 million states per second it would take $10^{15,000,000}$ years on average.) Of course the transitions are not at all random, and their true pattern presumably makes repetition somewhat more likely. On the other hand, included in the state of most computers is the state of a built-in clock and calendar, which counts off the milliseconds over a period of some decades. The computer will not revisit a state until the calendar "rolls over."

Because of the one-to-many mapping between macrostates and microstates, computers can remain rigidly deterministic in all their internal workings, but still seem wildly capricious to the poor sap at the keyboard. It's a nasty combination. It leaves me unsure whether the best approach to debugging is to correct the computer's logical errors or to heal its psychic wound.

The Phases of the Moon

A friend with experience of these matters once warned me: "Never let the computer know you're in a hurry." The hazard, of course, is that the machine will sense your impending deadline and choose the most inconvenient moment to crash. The more intelligent computers even seem to know that Federal Express closes earlier on Saturday than on weekdays.

It is easy to mock such notions, which seem to require us to believe in a mischievous spirit inhabiting the silicon, always on the lookout for

a chance to vex us. (Another friend advises me: "Don't anthropomorphize computers. They hate it.") But the idea of a computer reacting differently to a hurried user is not as implausible as it may seem. In the first place, urgency brings out a different spectrum of *human* bugs. I am capable of making mistakes either at leisure or in haste, but they tend to be different kinds of mistakes, and they probably expose different kinds of defects in the computer hardware and software.

Sometimes a speedy typist can create havoc even without making an error. One of the most devastating of all computer bugs was discovered about 10 years ago in the control software of a radiation-therapy machine called the Therac-25. The bug could lead to horrendous overdoses, and three patients died as a result. It turned out that one way of triggering the bug was to rapidly skip through a data-input screen, which allowed the radiation beam to be turned on before the computer had had time to adjust all the settings to their correct values. Typing the same sequence of keystrokes at a slower pace caused no trouble.

Bugs so erratic and mysterious that they seem to depend on the phase of the moon are an old programmers' joke, but at least one such bug really did exist. (The story is told in Eric Raymond's *New Hacker's Dictionary*.) A program written by Guy L. Steele, Jr., who was then at MIT, rejected its own data files if they had been written during certain phases of the moon. The explanation was not in the least supernatural. The data files included a time stamp, and Steele had playfully added the lunar phase to the usual date and time information. At certain phases of the moon, the time stamp exceeded an 80-character limit on line lengths, with the result that the file became unreadable.

Prolonged exposure to subtle and elusive bugs like these can lead people to approach the computer in an attitude of superstitious awe. They cling to whatever tricks or procedures worked the last time, without understanding the purpose of their own actions. They fear installing new versions of software, which might upset the delicate equilibrium of the entire system. The more severely afflicted practice cleansing rituals at the keyboard or offer sacrifices to propitiate the fickle gods of computation. I certainly don't endorse such behavior, which I see as another variety of sphexism. And yet I can't offer an alternative approach that can guarantee better results.

Every one of the malfunctions recorded in my diaries has a logical, rational explanation. I'm utterly certain of that. The fact remains, however, that I have been able to track down the logical, rational cause in only a handful of cases. Those few diagnostic successes concern problems in software I wrote myself or programs for which I had the direct assistance of the developer. Without access to the source code (the original program text), there is little hope of truly understanding a software fault. In that circumstance, superstition is as good a tool as any other for dealing with the problem.

I should add that superstition has often enough been my own refuge. Some years ago I instituted a campaign of "font hygiene" in the editorial offices of *American Scientist*, hoping to cure various murky computer ills. The problems were eventually dispelled, but I have no confidence that my elaborate precautions had anything to with the remedy.

The Spectrum of Bugs

I have sorted the error reports in my diaries into five categories, according to severity. The worst kind of event is a crash (also known as a freeze or a bomb), in which the entire computer system comes to a standstill. Getting out of this predicament often requires a "reboot." The next class of malfunction is a total program failure, where a single program stops working entirely but the rest of the system keeps running. The three lesser categories I call major program faults, minor program faults and cosmetic flaws.

What is the spectrum of bugs in these categories? Before beginning the diaries, I would have guessed that the distribution would be similar to that of earthquakes and forest fires and other natural disasters: There would be lots of little ones and only a few big ones. As Figure 2 shows, the actual distribution is just the opposite. Crashes are clearly the most common events, followed by total program failures, with the less severe problems trailing behind.

Reporting bias may have something to do with the shape of this spectrum. I was probably not as conscientious as I should have been about recording cosmetic flaws, and perhaps a fault that I consider major would be rated minor by more generous observers. But I think the main import of the spectrum can be trusted: When a computer fails, it usually fails big time.

The reason for this brittleness is no mystery. Most computer hardware and software cannot tolerate even the smallest malfunction. A single erroneous bit will derail an entire computation. In computers as in genetics—the analogy is a close one—most mutations are not merely harmful but lethal. Once a program gets onto the wrong track, there is almost no hope of recovery. Fragility is the price paid for the stabilizing effect of a digital architecture based on discrete states. The machine either functions perfectly or it does not function at all.

Cogniscienti will perceive that my statistics were gathered on microcomputers without a memory-protected multitasking operating system. On workstations and larger computers, system crashes are rare, because the operating system walls off each program in its own space; a program that runs amok can only destroy itself. The technology of memory protection will eventually reach smaller computers as well—it has been coming for 30 years now—with the result that many system crashes will be downgraded to program failures. That is an important gain.

Other oncoming developments are not so encouraging. As computers become more powerful, the size of both the microstate and the macrostate grows exponentially. This means there will be more pieces susceptible to failure, and quadratically more interactions among those pieces. The advent of parallel processing opens up a whole new dimension of potential errors. And the new software architecture known as componentware or document-centered computing could also make things worse. If you have half-a-dozen components all working on the same document, what happens when one of those components does something the others don't like?

Coping with these problems is going to require better tools. Up to now most tools for debugging have been intended for programmers and have been fully useful only with access to the source code. They present their information in terms of the microstate of the machine, which is difficult to relate to events in the user's world. What's needed is a tool that can diagnose faults in the macrostate, advising you, for example, that the system has crashed because two programs are locked in contention for the same hardware resource, or explaining that the program you have just tried to launch three times in a row needs more memory than is currently available. With information like that, computers might continue to be just as crazy, but people would be saner.

The best answer to the challenge of living on intimate terms with fallible computers is to somehow create more robust hardware and software, which doesn't go to pieces at the first sign of trouble. Ultimately we might aspire to build computers that work as well as coffeepots or toasters or even lawn mowers—devices that tend to fail gradually and gracefully, and also rarely. But building a computer as good as a coffeepot looks like a daunting challenge. It is a task surprisingly similar in some respects to the better-known quest for a computer as good as a brain. Success will be a long time in coming.

Bibliography

Hofstadter, Douglas R.. 1979. *Gödel, Escher, Bach: an Eternal Golden Braid*. New York: Basic Books. pp. 360–361.

Leveson, Nancy G., and Clark S. Turner. 1993. An investigation of the Therac-25 accidents. *Computer*, July, pp. 18–41.

Raymond, Eric (ed.) 1991. *The New Hacker's Dictionary*. Cambridge, Mass.: The MIT Press. pp. 280–281.