Editor: **Gerard J. Holzmann**
Nimble Research
gholzmann@acm.org

Randomly Right

Gerard J. Holzmann

IF YOU MAKE enough random predictions, some of them will likely be right. Similarly, if you check a software application in many randomly different ways, you'll likely find problems. This is, of course, not a deep insight. Fuzz testing has used this principle for quite a while to find bugs in applications that are too complex to test systematically with sufficient rigor. A key benefit of fuzzing is that it can be almost completely automated.

For random testing to work well, you must execute many tests. That could be time-consuming if it wasn't for the fact that all these tests are independent and can run in parallel. This makes randomized testing a natural fit for cloud computing. In an hour or so, a few million random test demons running in the cloud are likely to cover quite a bit more ground than humans running handcrafted test cases one by one. Perhaps a good analogy is that of a swarm of bees attacking an elephant. If the elephant has a weak spot, some of the bees are likely to find it quickly.

When a randomized test case locates a flaw, it generally isn't enough just to record which exact sequence of inputs led to the failure. For example, if the application is multithreaded, we might also have to know the thread-interleaving points along the execution path, and we'll have to know how the application interacted with its environment. Did it try to access a file or a network port, or did it try to communicate with a peripheral device in some way? All these details might be needed to let us reproduce and fix the problem.

So, it makes sense to ensure that we can control and randomize all aspects of an execution, including the process-scheduling decisions and the interactions with an application's environment. To randomize those interactions, we must be able to place the application in a fully controlled environment. This is where things get a little harder, and more interesting.

The FeaVer Experiment

In 1998 at Bell Labs, I started designing a new type of verification system that could solve these problems. The target for this system was the software for a new phone and data switch that Lucent Technologies was developing. I called the system FeaVer, as shorthand for the main problem it addressed: feature verification.[1–3]

The problem was to check whether the code that went into the switch complied with the regulatory requirements that were documented in a set of standards maintained at that time by Bellcore (today called Telcordia Technologies). The requirements applied to a range of phone services such as three-way calling, 911 calling, call blocking, call waiting, and call forwarding. Among many other things, we had to check that all the services and features couldn't interact in unforeseen ways. For example, it's reasonable to expect that 911 calls will always succeed no matter what other features are enabled or disabled on a phone.

This problem is called the *feature interaction problem*. With more than 100 possible features that aren't all equally

 0740-7459/17/$33.00

well understood or understandable and that can each be enabled or disabled on any phone, the problem is seriously complex. If the complexity of regular phone-switching software isn't enough to perplex the tester, adding the requirements for all possible call features makes testing even harder. So, this was the perfect target for an experiment to see whether we could improve the thoroughness of testing. With FeaVer, I aimed to find a way to use a logic model checker to randomize and parallelize the search for unexpected problems, by executing the call-processing code in a closed, fully controlled environment.

The Code Is the Model

At that time, model checkers worked only with handwritten abstract specifications—formal models carefully crafted to capture the essential behavior of the code. Constructing a suitable formal model for a complex software system could take months. That immediately put this approach at a disadvantage because the code for this application was not only complex but also in constant development. It simply wasn't an option to manually build and validate the accuracy of handcrafted models for each new release of the code.

This led to the bold decision to use the C code as its own formal model by teaching the Spin model checker to recognize it as an abstract state machine. Next, we had to find a way to isolate the code from its environment. All the inputs the application could receive from the outside world through its hardware peripherals, such as customer phone lines and links to other switches in the network, had to be stubbed out.

This last step was actually simpler than it might sound. It was no harder than building a standard software-only simulation environment for an application by replacing all hardware interfaces with software stubs. This method is common in the development of mission- or safety-critical software. For instance, this was done in the development of the millions of lines of code that controlled the Mars Science Laboratory mission that landed the Curiosity rover safely onto Mars in 2012. In our case, this step was actually a little easier because the stubs could be very simple. For example, consider a ring-tone circuit. We can emulate its behavior for the model checker as a simple automaton that can randomly claim to be in a free, busy, or failed state. The call-processing software should be able to handle each of these cases without problems.

One final thing I added to make verification more efficient was the capability to define source code transformations with a manually defined set of conversion rules. For instance, these rules stripped any code related to billing because billing wasn't the verification's immediate target. The rules basically defined simple code abstractions that let us focus on what mattered: proving the soundness of feature interactions.

Once trained on the code, the model checker also gave us full control of thread scheduling, which gave us the fully controlled environment needed for the testing process that followed.

The Needle and the Haystack

The real problem in this type of case is that the search space is incredibly large. There's no hope of exhaustively exploring that space in a traditional way, as model checkers typically try to do. Any search strategy can cover only a tiny slice of that space. The only reasonable strategy in these cases is to perform as many statistically independent randomized searches as possible.

To achieve this, I designed the random searches to be small and fast. Because all the little randomized tests were independent, they could execute in parallel to boost coverage as much as possible.

In 1998, we didn't have access to cloud networks, but we could dedicate a small cluster of machines to the search process. I persuaded my management at Bell Labs to invest in 16 machines for this purpose, each running at the blazing speed of 500 MHz to run all the randomized checks. With this small cluster's help, I could check some 200 logic properties for the feature requirements in about 40 minutes of real time. That was fast enough to keep track of the evolving code base and to uncover problems shortly after each new version of the multithreaded code was checked in.

Of course, the whole effort would have been pointless if it didn't uncover real problems in the code. It did, and perhaps not surprisingly, it did so far more effectively than the human testers who exercised the same code base. The automated FeaVer system turned out to locate about 10 times more violations of the feature requirements than the human testers did.

In a typical run of the system, the first violations of the feature requirements would be reported in a few minutes of runtime, with the run steadily uncovering more problems until most had been found in about 40 minutes from start to finish. Out of curiosity, I repeated the process on a more recent desktop with 32 CPU cores running at 2 GHz each. As I expected, the runtime dropped to seconds, with the first problems

reported in just a fraction of a second. Although technically this isn't surprising, it does mean that we could build a virtually interactive verification system in this way, even for complex commercial code. What could we achieve if we expanded this approach to a true cloud-size system, deploying a few orders of magnitude more small, smart, and randomized model-checking test engines?

Let's Play a Game

If we cast software testing, or formal verification, as a search problem, we can see that it has much in common with games like chess or go. Finding a path to a win at the start of a chess game requires searching an astonishingly large set of possible moves and board positions. The search space of somewhere between 10^{80} and 10^{100} possible games is large enough that, just as in software verification, exhaustive exploration isn't even remotely an option. Still, quite a few chess programs today can defeat the best human players. A milestone event in the slow improvement of chess programs over the years was the May 1997 match in which IBM's Deep Blue program defeated the then-reigning world champion Garry Kasparov.

Even harder than chess, in terms of the number of possible games that can be played, is go. Yet here too, 20 years after Deep Blue's victory, the AlphaGo program from Google DeepMind defeated the top go player, Ke Jie, by winning three out of three games played under standard contest conditions.

In both cases, the search spaces are large enough that they aren't fully searchable. Reportedly, AlphaGo also tries to leverage randomization and parallelization to overcome some of the obstacles. To do this, it uses Monte Carlo tree search. In addition, it uses neural networks and machine learning to learn strategies from large databases of previous games with known outcomes.

This leads to the question of whether we can learn something from game-playing machines to improve software verification tools. Although there are many similarities, there are also some important differences that might well turn out to be spoilers. I'll mention three.

First, the rules of chess and go haven't changed for a very long time, and vast databases of past games are available for both humans and computer programs to learn from. In comparison, every new program can be thought of as defining a new game, with unknown possible "board positions" (reachable states) and "moves" (executions). Although many general databases of software bugs and security vulnerabilities exist, it's unclear how these databases could be used to devise "game strategies" for a software verification tool.

Second, the complexity isn't any less for software verification than it is for games like chess and go—it's greater. Even a single 64-bit integer can hold 2^{64} possible values (about 10^{19}), so only six of these integers will create more possible states than there are board positions in a game of go.

Third, in chess and in go, the opponent of the machine is typically a human, not another machine. This means that successful strategies can consist of simply confusing the opponent into making bad moves. In software verification, the "opponent" is perhaps the scheduler making unexpected decisions or the environment interfering unpredictably in our executions. Some of these actions might lead to failure, and software verification aims to find those cases. That is, in software verification, we can't win by confusing our opponent; we must assume that the opponent always plays a perfect game.

What I've just discussed might make the effort to improve software testing seem somewhat hopeless. However, keep in mind that the target isn't to produce an infallible system. The target for the time being can be just to build a verification system that's significantly more effective than current methods, including the randomized-test methods.

Could we learn from the successful game strategies used in programs such as Deep Blue and AlphaGo to improve and automate software verification? It's too easy to say that it could never be done. After all, isn't it embarrassing that we might have autonomously driving cars on the road before we have autonomous verification systems that can thoroughly check the software that drives those cars?

References

1. G.J. Holzmann and M.H. Smith, "Software Model Checking: Extracting Verification Models from Source Code," *Formal Methods for Protocol Engineering and Distributed Systems*, Springer, 1999, pp. 481–497.
2. G.J. Holzmann, "Software Verification at Bell Labs: One Line of Development," *Bell Labs Tech. J.*, vol. 5, no. 1, 2000, pp. 35–45.
3. "The FeaVer Feature Verification System"; spinroot.com/feaver.

GERARD J. HOLZMANN works on developing stronger methods for the design and analysis of safety-critical software as a consultant and researcher at Nimble Research. Contact him at gholzmann@acm.org.