Vinton G. Cerf

DOI:10.1145/2631185

Responsible Programming

WELCOME TO “CERF’S UP!” I am grateful for Editor-in-Chief Moshe Vardi’s invitation to continue writing for *Communications*; this column succeeds the “From the President” column I penned during my service to ACM in that role.

Let me congratulate Alex Wolf, the newly elected ACM president. I know he will give exemplary service to our organization. Congratulations also go to Vicki Hanson and Erik Altman in their new roles as vice president and secretary/treasurer respectively. I know this team will provide first-rate leadership.

I also thank Alain Chenais, who ends his term as Past President and I begin mine. He has been a staunch, reliable, and active leader in ACM matters and I expect this will continue. There are many others elected to new positions or moving on as their terms in office end. I thank them all without enumeration, and commend them to your attention.

Lastly, allow me to note the enormous contributions of the ACM staff and, especially, the leadership of John White, CEO, and Pat Ryan, COO of ACM. They have accumulated a truly enviable record of steadfast leadership spanning the terms of many elected ACM officers.

Now to the substance of this column: responsible programming. What do I mean by that? In a nutshell, I think it means people who write software should have a clear sense of responsibility for its reliable operation and resistance to compromise and error. We do not seem to know how to write software that has no bugs...at least, not yet. But that, in a sense, is the very subject I want to explore.

My very good friend, Steve Crocker, drew me into a conversation about this topic a short while ago. As a graduate student, he had pursued a dissertation on provable correctness of programs. While

this is not a new topic, the objective continues to elude us. We have developed related tactics for trying to minimize errors. Model checking is one good example of a systematic effort to improve reliability for which ACM gave the Turing Award in 2007 to Edmund Clarke, Allen Emerson, and Joseph Sifakis. What is apparent, and emphasized by Crocker, is the tools available to programmers for validating assertions about program operation are complex, with user interfaces only a mother could love (my characterization). Formal proofs are difficult, especially for anything but the simplest sort of program. Just conceiving the appropriate conditional statements to characterize program correctness is a challenge.

Despite the Turing Halting Problem, it is still possible to establish lines of reasoning to show a particular program terminates or achieves a repeatable state under the right conditions. One can make other kinds of statements about I/O checking (for example, buffer overflows). Some unending programs, like email user agents, can still have characterizations of well-defined states. It is clear, however, it is not easy to develop succinct and potentially demonstrable statements about program behavior that show the likelihood the program will behave as desired. Yet harder may be demonstrating the program does not do something undesired.

While I have no ready solution to the problem, I believe better interactive tools are needed to test assertions about the program’s anticipated behavior while it is being written and to get some useful feedback from the composition and validation system that these assertions are somehow supportable. If not provable, then at least not disproved by counterexample perhaps. It seems fair to imagine that when a programmer is designing a program and actually writing the code, there is a model in the pro-

grammer’s head of what the program is supposed to be doing and, presumably things it is not supposed to do or should avoid. Whether this model is sufficiently clear and complete to allow provable or verifiable assertions to be made could be the subject of considerable debate.

One intriguing example of programming environments that is tangentially relevant comes from Bret Victor (<http://worrydream.com>) who has conceived and implemented a programming environment that allows one to see immediately the results of executing the current program. Obviously, the system can only do this when the programmer has reached a point where the program can be parsed and interpreted. Imagine an environment fashioned for continuous validation of a set of assertions, as the program is developed. One suspects heavy use of libraries could either help or hinder the process of verifying program correctness. If the library of subroutines is opaque to the verifying tools, bugs could be hidden. However, if a set of assertions that are invariant for the subroutine could be codified, the use of such a library might actually help the validation process. I am fairly certain a body of prior work exists that can be cited here, but my impression is such tools are not used regularly by professional programmers today.

It seems timely to suggest responsible programming calls for renewed efforts to verify proper operation of software many may depend upon heavily to work as advertised. To do this, we need much better tools and programming environments than seem to be available today. I await with great interest responses from ACM members more knowledgeable than I in this area.

Vinton G. Cerf is vice president and Chief Internet Evangelist at Google. He served as ACM president from 2012–2014.

Copyright held by author.