# Demonic Nondeterminacy: A Tribute to Edsger Wybe Dijkstra

**Jayadev Misra**

August 25, 2003

Please permit me to talk about the paper "Guarded Commands, Nondeterminacy, and the Formal Derivation of Programs"[1], which Dijkstra published in 1975 in the Communications of the ACM. Specifically, I will talk about only one of the aspects, nondeterminacy.

By 1975, programming theory seemed to have addressed all reasonable features of programming languages. Scott's work on denotational semantics laid a foundation for mathematical treatment of language features. In particular, continuity of program composition operators —sequencing, conditionals, looping constructs and function composition— held the hope that a theory of correctness, based on manipulations of functions, is just around the corner. A very important, and highly overlooked piece of work, An Axiomatic Definition of the PASCAL programming language, by two of my co-panelists, showed that even sinful features, such as the GOTO, can meet redemption.

I was astounded, therefore, when I read Dijkstra's paper in 1975. I remember complaining to a friend that Dijkstra has needlessly complicated the programming problem, introducing a feature for which there is no demand. I first met Dijkstra in 1976, and by 1977 I had developed enough courage to ask him why he thought this feature was important. He sketched a little program on the blackboard, to compute the maximum of two numbers $x$ and $y$, in which the guards $x \leq y$ and $y \leq x$ are completely symmetric, and it is unspecified which guarded command is executed when $x = y$. I was unconvinced that the appropriate treatment of this trivial example deserved introduction of a new programming feature. Dijkstra probably wrote me off as being mentally deficient.

I became even more convinced in my position by misinterpreting two significant works: (1) the treatment of nondeterminacy in automata theory, and (2) the lack of continuity when fairness is added to nondeterminacy. Let me articulate the first concern, the second one is somewhat more technical.

Automata theory treats nondeterministic choice with clairvoyance. An execution with nondeterministic choices can be depicted as a tree, and a particular choice as a branch from a parent to a child. A nondeterministic machine chooses the appropriate branch at every point during execution so that a leaf node with the desired properties is reached eventually. Let us call this *angelic nondeterminacy*, for obvious reasons. To a large extent, Prolog has embraced angels.

What Dijkstra had proposed was to not only banish angels, but embrace demons. Not only was no clairvoyance allowed, but the programmer was asked to design the program assuming that the worst possible choice is taken at each stage. This was complete madness. The programmer's task was burdensome enough already. He had been doing a decent job of producing shoddy software with the existing programming language features, and really had no need for additional chaos.

The story would have ended happily there with my continued ignorance on this subject had I not attended a series of lectures on Communicating Sequential

Processes, given by Tony Hoare. Tony articulated the aspects of nondeterminacy which I could appreciate, that if a process is waiting to receive a message from one of several processes, its execution has to be described by some nondeterministic construct; nondeterminacy is not a luxury but a necessity! And the receiving process has to be coded so that independent of the message received it produces the correct output.

It became clear to me then that no clairvoyance is needed provided all leaf nodes in an execution tree have the desired property. The task of program design is not to implement angels, but to ensure that all possible outcomes meet the specification. This is particularly important when some of the choices may be outside the programmer's control, as is the case in concurrent computing or when operating in a failure-prone environment.

Later, during the mid 80s, my colleague Mani Chandy and I found another strong reason to embrace nondeterminacy. We observed that quite often the same algorithm is implemented very differently on different computing platforms. Fast Fourier Transform is coded differently on sequential machines, on parallel machines with shared memory and on machines which communicate through messages. The differences are significant enough that a correctness proof of the original algorithm is not sufficient to guarantee correctness of the clones. Adopting a high-level language to describe an algorithm for a specific platform made matters worse; the central algorithm receded to the background while the details of communication became the significant factor.

We realized that we could start with a highly nondeterministic algorithm and by limiting the nondeterministic choices differently, we would obtain programs suitable for different computing platforms. That is, demonic nondeterminacy allowed us to express a family of programs succinctly. Proving a single nondeterministic program resulted in proving the correctness of the entire family. Any restriction of nondeterministic choice yields a correct program provided a technical condition —fairness— is met.

Nondeterminacy has played an essential role in the works of Hoare and Milner, where the distinction between external and internal nondeterminacy became clearer. And, Hoare, Bird and de Moor, and others, have shown that relations may be as important as functions as the basis of our discipline. Nondeterminacy now permeates designs of programming languages, software systems, and, even, hardware systems. Dijkstra wrote a classic book[2] in which he demonstrated the importance of "keeping your options open as long as possible" by employing nondeterminacy. Yet, computing scientists, by and large, have not appreciated the radical novelty of this invention, and how it went against the grain when it was published in 1975.

## References

[1] Edsger W. Dijkstra. Guarded commands, nondeterminacy, and the formal derivation of programs. *Communications of the ACM*, 8:453–457, 1975.

[2] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.