Ubiquity Symposium

What is Computation?

Computation is Process

by Dennis J. Frailey

Editor's Introduction

Various authors define forms of computation as specialized types of processes. As the scope of computation widens, the range of such specialties increases. Dennis J. Frailey posits that the essence of computation can be found in any form of process, hence the title and the thesis of this paper in the Ubiquity symposium discussion what is computation.

Association for Computing Machinery
Advancing Computing as a Science & Profession

Ubiquity, an ACM publication
November, 2010

Ubiquity Symposium

What is Computation?

Computation is Process

by Dennis J. Frailey

The concept of *computation* is arguably the most dramatic advance in mathematical thinking of the past century. Denning [2010], in his opening statement, describes how *computation* was originally defined in the 1930s and how that definition has progressed through the ensuing decades. Church, Gödel, and Turing defined it in terms of mathematical functions, which they divided into the decidable (can be evaluated by algorithms) and the un-decidable. They were inclined to the view that only the algorithmic functions constituted computation. I'll call this the "mathematician's bias" because I believe it limits our thinking and prevent us from fully appreciating the power of computation. As Denning writes [italics mine], computation in that era was "defined as the execution sequences of *halting* Turing machines (or their equivalents)." Today, I believe we are breaking out of the era where only algorithmic processes are included in the term *computation*.

The 1960s and especially the 1970s, saw the concept of a process used in conjunction with the theory and practice of operating system design. It was such an elegant and powerful concept: the program is a description of the process, the computer is the enactor of the process, and the process is what happens when the computer (or, more correctly, the processor—since a computer may have multiple processors) carries out the program. So many concepts can easily be described from these simple constructs. For example, with the advent of time sharing and multiprogramming systems, we saw that a processor can enact several processes concurrently by such techniques as multitasking. A most elegant way to model such a system is in terms of the processes being carried out, not the number of computers involved or the mechanics of which processor might be carrying out which process at any given time. Much of the theory of operating systems was built upon these simple concepts, all centered on the idea of a process.

I recall a conversation in the early 1970s with several graduate students over the question of whether a computer program might be designed to run forever and, if so, how that would fit the prevailing notion that a computable function had to terminate. It was food for thought, leading to the notion that a process could run forever but such a process would be deemed *non-computable*. That aligned with the theory, but was it the right idea? Another conversation centered on the question of whether all computations had to be deterministic. Could stochastic (i.e., random) processes be included? Was this a moot point because no computer could have truly random behavior unless it was malfunctioning or being influenced by external, random events? Of course, in the emerging view, *computation* is not limited to what computers can do. Furthermore, many computing applications incorporate pseudo-randomness to great effect and the theory of stochastic processes is utilized to devise models and algorithms for such applications.

As Denning explains, there were more expansive concepts of *computation* being discussed in the 1960s as well. What about other, non-algorithmic phenomena associated with computers? Was *computation* limited to the actions of machines? Were "natural processes" forms of *computation*? As the years went by and especially by the 1990s, experts in various natural sciences, notably physics and biology, wrote about the similarities between the decidedly non-electro-mechanical processes they encountered in their work and the phenomena surrounding the computers they used as basic research tools. The essential point is that new discoveries in these fields were driven by the insights gained from understanding more and more about *computation*. How does DNA work? In many ways it's like a program. Perhaps biological systems carry out processes that do not quite fit our notions of algorithmic, but they exhibit properties that are more readily understood because of what we know about *computation*. As Denning goes on to say, the phenomena of interactive computing, natural information processes and continuous information processes began to broaden the definition of *computation*.

I observe a common denominator of almost all these expanded concepts of *computation*—they are defined as categories of ***processes***. One is inclined to ask this question: if every form of computation is a type of process, are there any types of process that are not forms of computation? Certainly there are if one insists that all computation must terminate or must be deterministic. But as we relax those stipulations, what other obstacles prevent any process from being a form of computation? My contention is none. If that is correct, why not just delete the modifier and state that computation is the same as process? This would admit to the possibility of non-terminating and non-deterministic computations/processes, but haven't we been grudgingly accepting that for quite some time?

Before I take this further, I'd like to go back to the 1980s and explain how a somewhat different use of the term "process" gave me a new perspective on all of this. In the late 1980s, the software engineering community began to focus on the concept of a process as it relates to software development. What software developers do when they develop software is to carry out a process. [Hopefully it is a finite one!] These developers are the counterparts of the processors in the operating system model. What the software process community focused on was the *description* of the process—the “program” by which one develops software, so to speak. It was evident at the time that although processes were being followed by software developers, documentation of those processes was often non-existent or, at best, informal and ad-hoc. In that era, much of the literature on software development processes focused on notations for building process descriptions or models, but there was frequent confusion among the process itself, the description of the process, and the tools used to facilitate execution of the process. For example, Humphrey’s seminal book [1989] defined a process as consisting of “a set of tools, methods and practices used to produce a product.” Even today, papers on “software engineering process” are more likely to be discussing process descriptions or models than processes themselves.

Few authors writing on software development process seem to use the terminology in the same fashion prevalent in operating system theory. In my own software process work, I continued to rely on the definitions from my operating systems days. This helped me prepare clearer process models and more straightforward descriptions and analyses of what processes were actually taking place. For example, some in my circle asked whether two projects “using the same process” [actually the same process model] were really using the same process if their day-to-day work details differed. I would simply distinguish these as different instances of the same process model. Or consider the difference between a prescriptive model and a descriptive model of the same process. My OS-based terminology made this a trivial matter. This careful use of terminology was helpful in both the practice of software engineering and in teaching it, which I did in both academic and industrial settings. In [Frailey, *et al*., 1991], for example, we describe how more careful terminology cleared up confusion regarding the artifacts produced when a process is enacted. What I learned from these experiences is that imprecise use of process terminology tends to obscure the underlying principles.

Many of the more recent concepts of *computation* use the word “process” in an almost offhand manner as part of the definition. There’s an implicit assumption that the reader understands what a process is. So what is it? Here are some dictionary definitions [Merriam-Webster, 1985]:

> “Something going on”
>
> “A natural phenomenon marked by gradual changes that lead toward a particular result”
>
> “A series of actions or operations conducting to an end”

The first of the above strikes me as the most comprehensive, and the most akin to what we mean by *computation* in its most general sense. In the vernacular of the late 1960s, a process is “a happening.” *Computation* in its broadest sense, to me, is anything that happens [as opposed to things that are static]. If so, then the principles of *computation* are, in fact, the principles of processes.

Is this so? It certainly seems reasonable. For example, consider the halting problem [Turing, 1936]. There is an essentially identical problem for any process. Or consider the Blum speedup theorem [Blum, 1967], which states that any algorithm can be made arbitrarily faster provided there is enough memory. I suggest that the same theorem can be applied to any process. Of course to prove this we must do some theoretical homework. For example, we must determine how one evaluates the speed of a non-terminating process. But certainly anyone who has waited for internet delays will attest that there is such a thing as a faster version of a non-terminating process.

Why should we equate *process* and *computation*? If we did so we would overcome the "mathematician's bias" and deal with the complete scope of what *computation* has come to include. We could study ways to document processes, to automate them, to optimize them, and to predict their performance. We could develop a theory of processing analogous to our current theory of *computation*. Such a theory would be a better fit to the reality of what *computation* truly entails. Whether natural or artificial; mechanical, electrical or biological; described by a formal model or simply occurring naturally and in need of a formal description; processes should be the focus of our attention when we try to understand *computation*.

References

Blum, Manuel (1967), "A Machine-Independent Theory of the Complexity of Recursive Functions," *Journal of the ACM* 14: 322–336, doi:10.1145/321386.321395.

Denning, Peter J. (2010), "What is Computation? Opening Statement," *ACM Ubiquity* (October, 2010) doi: 10.1145/1880066.1880067.

Frailey, Dennis J., R. Bate, J. Crowley and S. Hills (1991), "Modeling Information in a Software Process," *Proceedings, 1st International Conference on Software Process*, IEEE Computer Society Press (ISBN 0-8186-2490-6; order no. 2490), (October, 1991), 113-121.

Humphrey, Watts (1989). *Managing the Software Process.* Reading, Mass.: Addison-Wesley Publishing Company. ISBN 0-201-18095-2.

Merriam Webster (1983), *Webster's Ninth New Collegiate Dictionary*. Springfield, Mass.: Merriam-Webster Publishers. ISBN 0-87779-508-8.

Turing, Alan (1936), "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society, Series 2*, 42 (1936), pp 230–265.

About the Author

Dennis J. Frailey (frailey@dfwair.net) is a recently retired Principal Fellow at Raytheon Company and an Adjunct Professor of Computer Science and Engineering at Southern Methodist University. He was a past vice-president of ACM and currently serves as vice-chair of the Educational Activities Board of the IEEE Computer Society.

DOI: 10.1145/1880066.1891341