DEPARTMENT EDITOR: Colette Perold, colette.perold@colorado.edu

Mathematics, Logic, and Engineering in Computing

Peter J. Denning, *Naval Postgraduate School, Monterey, CA, USA*

Matti Tedre, *University of Eastern Finland, Joensuu, Finland*

From its beginning in the 1950s, noncomputing academics were skeptical about computer science because it seemed strong on technology and weak on theory. To answer the critics and shore up their case, computer scientists turned to a rich trove of computational methods from logic and mathematics. Computing from ancient times focused on methods of manipulating symbols that could be performed by people untrained in mathematics. Examples include ancient Babylonian algorithm-like step-by-step rules, Greek mathematical procedures like the Euclidean algorithm or the sieve of Eratosthenes, and al-Khwarizmi's algorithmic techniques. In the twentieth century, the mathematical logicians Turing, Gödel, Church, Kleene, and Post provided a solid foundational theory for the new field of computer science, showing what can and cannot be computed.

Much human computation is based on procedures of many steps, many of which depend on logic: decomposing a large task into a series of smaller ones, choosing between alternative tasks based on a condition, and repeating tasks until some condition was achieved. Popular history texts in computing cite the development of logic as a means for ultimately automating these choices. Boole's algebra for logic formulas (1854) gave a notation for conditions used in making the choices, which could be composed from simple true–false elements connected by AND, OR, and NOT. Shannon's insight (1938) made Boole's algebra the basis for describing electronic computer circuits. Frege's axiomatic predicate logic (1879) presented formal rules of inference and syntax, which, in the 1950s, came to be seen as the logical basis for programming languages. Like many others, Boole and Frege believed that logic was the foundation for rational human thought.

The notion that logic enabled computation opened the reverse possibility that computation could automate the logic of mathematical proofs. Beginning around 1900, prominent mathematicians and logicians sought to characterize the process of proof so precisely that an automatic procedure could decide whether any given proposition is provable in first-order logic. In 1928, the famous mathematician David Hilbert posed this Decision Problem as one of the fundamental challenges in mathematics [5].

To many, the Decision Problem looked eminently doable: a proof system consisted of given axioms and rules of inference, and a proof is a well-structured sequence of statements in which each statement is either an axiom or is constructed from previous statements by a rule. Hilbert and many others had believed for years that an algorithm for the Decision Problem existed, although they never could find it. Their hopes were permanently dashed when, in the 1930s, Gödel, Post, Turing, and Church proved that this was impossible. Their different systems of mechanization were all shown to be equivalent—any computation in one could be simulated in all the others. The famous Church–Turing thesis stated that all effective computations could be formalized as Turing machines. What an irony, that logic-inspired computation was incapable of answering the Decision Problem's question of whether logical proofs could be automated.

This irony supported the academic case for computer science. Not only did the work of Turing and the others provide a basic theory of computation, it led to the surprising conclusion that many important questions cannot be answered by computational algorithms. It made the case that a small set of logic principles governed the thought processes of designing algorithms.

For these reasons, when computing became an academic discipline in the 1950s, the popular disciplinary narratives of computing prominently featured mathematics and logic.

COMPUTING AS A BRANCH OF LOGIC

Many pioneers of the nascent computing field in the 1950s came from mathematics. They took it as a given

1058-6180 © 2022 IEEE
Digital Object Identifier 10.1109/MAHC.2021.3132110
Date of current version 7 February 2022.

that electronic automatic computers are governed by logic systems. Engineers agreed: logic was baked into computer architecture from the beginning. In his 1938 M.Sc. thesis, Claude Shannon showed that the functions of switching circuits, such as those found in telephone exchanges and motor control equipment, could be described by Boolean algebra. Through a series of developments into practical applications, Shannon's insight permeated the engineering world [1] and was eventually adopted for computer circuits. Electronic computer circuits came to be called logic circuits, and Boole's algebra of logic became the standard basis for computing. Logic's influence on circuit design continued in the 1950s and 1960s. For instance, an early design technique using Karnaugh maps (1953) enabled logic circuit designers to minimize the number of logic gates needed and avoid race hazards where a change of state could cause an output to flicker. Logic also influenced programming practice. The 1966 Böhm–Jacopini theorem restated the logic basis of programming languages: every computation could be constructed from simpler computations by joining them in sequences, if-then clauses, or iteration clauses. The theorem was taken as a support for "structured programming"—a programming practice that advocated a limited set of constructs to build programs.

Similarly, early research programs in artificial intelligence were based on an idea that human intelligence (at least the rational part) is based on logic. This idea was celebrated in the monumental intellectual achievements of the early 1900s, notably Russell and Whitehead's *Principia Mathematica* and Wittgenstein's *Tractatus Logico-Philosophicus*. Logic was revered as a pinnacle of human intelligence. Not surprisingly, early AI focused on getting logic programs to perform intelligent actions. The logic theory of intelligence received a big boost in 1956, when the *Logic Theory Machine* of Newell, Simon, and Shaw proved 38 of the first 52 theorems of the *Principia*. Some saw that machine as an improvement to human intelligence—it produced in a few minutes proofs of several theorems that brilliant thinkers took years to prove. Logic came to be seen as a litmus test for machine intelligence.

This idea spawned a branch of AI devoted to logic programming, from which expert systems emerged in the early 1980s. The prototypes used new logic languages LISP and PROLOG. Engineers built special-purpose machines to run programs in these languages very efficiently. The Japanese Fifth Generation Project (1980s) was aimed at turbocharging expert systems by building supercomputers for massive logic operations, just as a numerical supercomputer could do with massive arithmetic operations. The United States responded with its Strategic Computing Initiative, more generally focused on supercomputers capable of solving "grand challenge problem" in science.

Logic pervaded other parts of computing as well. In 1970, Codd introduced the logic of relational databases, which became a major IBM project later in that decade and spawned a host of database companies. These systems are often queried and managed with the language SQL, which consists of logic expressions to select, join, and project records. Logic made the data management systems common in business simpler and more effective.

In the same time, there was an explosion of insights into the complexity of computations. It was well known that some problems are harder to solve than others—their algorithms take more time and memory. The NP-completeness theorems of Cook (1971) and Levin (1973) to characterize these problems are deeply rooted in logic. Many hard decision problems could be simulated with gigantic logic circuits; finding the answer amounted to finding an input to the logic network that produces the desired yes–no output. This is known in logic as the satisfiability (SAT) problem. Any fast algorithm for solving the SAT problem would be convertible to a fast algorithm for any of the numerous hard problems that could be simulated as a SAT problem. Our theory of complexity rests on logic.

CRACKS APPEAR

Yet, logic's hegemony had already started to show signs of cracking in the late 1960s. When software engineering was being born to address the software crisis, several pioneers from the logic-oriented community proposed that much of the unreliability of software would be eliminated if the software could be formally proved to meet its specifications, for then there would be no doubt that the software was error free once it was compiled. Formal verification, however, turned out to be a formidable challenge. It sparked heated debates that forever shaped the computing field.

It turned out that formal logic proofs could be carried out only for relatively small programs, but large systems were beyond their reach. Even if the program source code could be proved correct, there was the additional difficulty of proving that the compiled machine code as well as the hardware platform also met their specifications. In his 1983 Turing Award lecture, Ken Thompson reminded us that bugs were not a feature of program code alone but of the total software-hardware–human system. Logicians and engineers argued endlessly about the practicality of formal proof. Many engineers were concerned that recovery from defects and deterioration of hardware—such as a transistor failure or arrival of a signal corrupted by noise—could not be supported in the logic formalisms. The prospects for full and complete verification started to look very gloomy when many agreed that developing complete specifications representing the true intentions of the human stakeholders and users of systems could not be formalized—and therefore not amenable to the tools of standard logic.

Other anomalies about the completeness of logic as the basis of computing appeared. In the mid-1960s, Lotfi Zadeh argued that engineers are frequently faced with ambiguous situations where a condition can be partially true and partially false at the same time. He proposed "fuzzy logic" as an extension of Boolean logic that allows truth values to be represented by numbers between 0 and 1. Fuzzy logic proved useful in many physical devices, but did not gain much foothold in the AI and logic community. Still another anomaly in the field of AI was found with expert systems, which were expected to perform as a human expert by acquiring enough deductive rules and facts. Dreyfus (1972) challenged this idea on the grounds that much expert behavior does not follow known rules; expert systems might become competent, he argued, but not expert [4]. Many other anomalies between the expectations of what AI could achieve and what AI actually achieved arose from the presumption that intelligence is founded in formal logic.

In 2013, Moshe Vardi lamented about the accumulation of gloomy conclusions about logic two decades before. He recalled how he and his colleagues experienced a feeling that large-scale program verification may indeed be hopeless: "First-order logic is undecidable, the decidable fragments are either too weak or too intractable, even Boolean logic is intractable [6]." Moreover, as computers invaded many new areas such as entertainment, cyber-physical control systems, office tools, art, transportation, medicine, and more, skillful development relied progressively less on logic and more on design acumen, human communication, aesthetics, social savvy, and other nonformal skills. Continuing progress in important technologies such relational databases, Boolean reasoning, and model checking did not stave off the growing feeling that computing could not be reducible to logic.

NO COMPUTING WITHOUT ENGINEERING

Over the 1980s, there was a growing consensus that the logic view does not cover many engineering, science, and technology aspects of computing. We were being pulled back to the historic notion that the roots of computer science are a complex mixture combining mathematics, science, and engineering. Logic did not cover everything in computing. The 1989 *Computing as a Discipline* report crystallized this growing feeling among people in the computing field [3]. Herewith a few examples.

Start with logic circuits. The logic formulas describing circuits assume that the signals are 0 and 1. But these are abstractions. The 0 and 1 represent states of the circuits, such as voltage low or high. Because a physical circuit can be in transient states that are neither 0 nor 1, the logic of the abstraction is unable to deal with some physical behaviors. The "half signal problem" asks what happens when one part of a circuit tries to read another part that has not settled into a definite 0 or 1 state. The "arbitration problem" asks what happens if a logic signal and clock signal arrive at the input of a flip-flop circuit at the same time. This condition can trigger the flip-flop into a metastable state that is neither 0 nor 1 and can crash the CPU when it persists for many clock ticks. These problems have physical solutions that cannot be derived in logic. Physical circuits display important stochastic behaviors that cannot be addressed by logic alone.

Next, consider the architecture of computers. In 1945, John von Neumann published the ideas of a team of pioneers from the early computing projects who defined a better architecture that would be more reliable and faster than their previous machines. What emerged is now known as the von Neumann architecture. It separated the computer into CPU, memory, and input–output, and defined the CPU cycle that fetches and executes programs stored as instructions in the main memory. While this architecture has often been held as an example of logical and analytic thinking in computing, it was actually the product of engineering improvements for efficiency and reliability [2].

One of the innovations of that architecture was to fetch instructions from main memory rather than paper tapes or cards. This engineering innovation greatly speeded up program execution. However, folklore developed that the stored program idea was the implementation of Turing's universal machine. This is not so. Historians find evidence to the contrary that the architecture was not influenced by Turing's model, nor did Turing have in mind an architecture of the same type. Other folklore held that the new architecture would be easier to build than its predecessors. In some ways, it surely was, but at the same time it also created new challenges. For instance, Maurice Wilkes, who led the EDSAC project at the University of Cambridge, said that one of the many engineering challenges was finding a technology that could support a main memory large enough to hold all the instructions of the program. Wilkes found that a mercury delay line did the job better than other available technologies. With this and other engineering innovations, he got the EDSAC machine working a year earlier than its U.S. version (EDVAC). Wilkes steadfastly maintained that there are important aspects of computing that logic cannot address.

Next, consider large multiuser networked systems. The Multics project at MIT (1965) dreamt of a "computer utility" that would dispense cheap computing power over a network to the masses. No one knew how to organize large operating systems that would coordinate hundreds of users. The logic-inspired view at the time was to carefully construct the operating system as a set of modules that interacted by well-defined interfaces. But these systems had great difficulties with coordination and were prone to many errors and crashes. Operating systems designers at MIT, IBM, and elsewhere invented a new idea, the process, as the basic entity demanding service from the system, and they organized the system as a "society of cooperating processes." This led rapidly to successful operating systems and a new theory of concurrent process coordination. Although logic helped to make coordination theories more precise, neither the gestation of the process idea nor its development was in logic—the process arose in the pragmatics of coordinating activities in an operating system. Over the years, the engineering understanding of these systems led to very small operating system kernels that could be formally verified by the methods pioneered in the 1970s. Today the sel4 secure operating system kernel illustrates how an engineered system can progress to the point of being a commercially viable, fully verified kernel.

Computational science, which grew up in the 1980s, is based on the idea that many physical processes can be viewed as information processes that can be simulated on a computer. Formal logic does not capture the simulation and modeling prevalent in computational science.

Finally, consider the performance of computer systems. As users, we want computers to get our jobs done as fast as possible within the constraints of processors and memory. Complexity theory gave order-of-magnitude estimates of running times of algorithms on a single CPU. But it was not able to predict the response time when multiple jobs were competing for the CPU. The solution to this was again found by engineers who recognized that queueing theory could answer the question. Computer scientists plunged into the performance analysis and prediction problem and discovered very fast algorithms to compute throughput and response times for operating systems (and the Internet) built as networks of servers. This led eventually to a thriving performance-evaluation industry. But queueing theory is not a product of logic, and performance evaluation is an empirical, not logical, matter.

In all these systems, engineering was oriented toward finding what works and what does not work. This is often accomplished with lots of trial and error, tinkering, and experimenting. There is often no theory or science available to understand what is going on; understanding is developed by trying things out. For instance, designers of early time-sharing systems found no theory that could predict their response time. Once time-sharing systems started to show promise, a rich body of theory emerged to accurately predict response time, guide the design of systems, and evaluate their performance.

The modern computational thinking movement for K-12 education has embraced the idea that computational thinking is founded on logical thinking. The movement has defined curricula that teach computing principles using generic logic puzzles and games. This has been controversial because generic logic does not demonstrate the unique aspects of computing and because it omits study of engineering and design in computing systems.

In truth, computer science is built on a complex framework of understandings from logic, mathematics, science, and engineering. Combined together, these different modes of thinking produced the amazing progress we have seen in computing. Computing is bigger than logic, and logic is less foundational than many people believe: designing, building, experimenting, and are at least equally foundational for the field.

REFERENCES/ENDNOTES

[1] L. De Mol, M. Bullynck, and E. G. Daylight, "Less is more in the fifties: Encounters between logical minimalism and computer design during the 1950s," *IEEE Ann. Hist. Comput.*, vol. 40, no. 1, pp. 19–45, Jan.–Mar. 2018.

[2] L. De Mol and M. Bullynck, "Roots of 'Program' revisited," *Commun. ACM*, vol. 64, no. 4, pp. 35–37, 2021.

[3] P. J. Denning *et al.*, "Computing as a discipline," *Commun. ACM*, vol. 32, no. 1, pp. 9–23, 1989.

[4] H. Dreyfus, *What Computers Still Can't Do.* Cambridge, MA, USA: MIT Press, 1992. Originally published as "What computers can't do" in 1972.

[5] D. Hilbert and W. Ackermann, *Principles of Mathematical Logic*, 2nd translated ed. Providence, RI, USA: AMS Chelsea Pub., 1958, p. 122.

[6] M. Y. Vardi, "A logical revolution (slides for keynote)," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, 2013, Art. no. 1.