Computability Pre-press (2021) 1–18; The final publication is available at IOS Press through http://dx.doi.org/10.3233/COM-180217
IOS Press

The Halting Problem and Security's Language-Theoretic Approach: Praise and Criticism from a Technical Historian

Edgar G. Daylight *
School of Media and Information, Siegen University, Herrengarten 3, 57072 Siegen,
Germany
egdaylight@dijkstrascry.com

Abstract. The term 'Halting Problem' arguably refers to computer science's most celebrated impossibility result and to the core notion underlying the language-theoretic approach to security. Computer professionals often ignore the Halting Problem however. In retrospect, this is not too surprising given that several advocates of computability theory implicitly follow Christopher Strachey's alleged 1965 proof of his Halting Problem (which is about executable — i.e., hackable — programs) rather than Martin Davis's correct 1958 version or his 1994 account (each of which is solely about mathematical objects). For the sake of conceptual clarity, particularly for researchers pursuing a coherent science of cybersecurity, I will scrutinize Strachey's 1965 line of reasoning — which is widespread today — both from a charitable, historical angle and from a critical, engineering perspective.

Keywords: undecidability, halting problem, security, computability theory, Internet of Things

1. Introduction

Wireless networks, sensors, and software are transforming our societies into an Internet of Things. We are starting to use Internet-connected drones, self-driving cars, and pacemakers designed to facilitate long-distance patient monitoring by doctors. These promising technologies also have negative impacts, some of which are unknown or downplayed by experts, companies, and marketers. When actual problems arise, pacemakers' software must be updated quickly to prevent malicious hackers from attacking such devices and taking over control of patients' lives. Drones and cars could become controlled remotely by malicious parties [1].

Identifying and anticipating such problems and preparing for risk mitigation is an urgent matter, in order to ensure human safety. Anticipation by responsible engineers is feasible in principle but lacking in practice. A possible reason for this omission is that software has become too intricate, even for experts, compared to, say, the `FORTRAN` programs used in the 1950s [2, Ch.1]. According to the present author this problem persists largely because computer science (broadly construed) lacks a revealing history of software and its mathematical underpinnings. Specifically, to develop Fred B. Schneider's much-wanted "science of cybersecurity" [3, p.47], researchers may want to build a technical past while contemplating the first principles of their new science.

The next part in this introduction conveys a flavor of technical contributions that can be expected from following a historical methodology (Section 1.1). Then I focus on security researchers' bread-and-butter distinction between models and modeled artefacts (Section 1.2). I reflect on the tendency to blur the distinction (Section 1.3) and zoom in on specific terminology (Section 1.4). In Section 2, I pay historical attention to the so-called "Turing Fix" in order to contextualize the language-theoretic approach to security. Coming to the body of this paper, Section 3 provides a technical, historical analysis of the Halting Problem pertaining to computer programs; that is, hackable technology. A potentially fruitful discussion follows in Section 4.

1.1. Merits of Historical Research

What Can We Learn from History? Historians of computing have researched the vulnerability of computer systems and the implications for worldwide human safety. Rebecca Slayton has examined the cold war history of missile defence systems and documented the birth of "software engineering" as a distinct research community [4]. Her writings have insightful messages for both researchers and policymakers, including the following observation:

*The author, also known as Karel Van Oudheusden, was financed by SFB 1187 "Medien der Kooperation" (Siegen University) and by ANR-17-CE38-003-01 "PROGRAMme" (Lille University). An online lecture on the contents of this article is available at: *https://dijkstrascry.com/lecture2*

> "By the time physicists began to note the limitations of software, the missile defense program was moving forward with a momentum all its own." [4, p.84]

In contrast to physicists, Slayton reports that software engineers raised red flags in a decades-old dispute about the viability of deploying effective missile systems. Taking the human out of the control loop (of a missile defence system) was perceived as feasible or infeasible by different communities of experts. To date, however, human-machine symbiosis remains key in real-time military systems [4].

Does History Repeat Itself? Historians assess past examples of change to improve our understanding of change in society today [5]. The previous (indented) remark about missile defence can be compared with a similar statement about the automated vehicles that are, or soon will be, using our roads:

> "Dozens of demonstrations by hackers and security researchers have proven it is entirely possible for criminals fifteen hundred miles away to seize control of your car when you are driving sixty-five miles per hour down the highway." [1, p.363]

These words — coming from a law enforcement expert — illuminate a gap between concerned specialists raising red flags [6, 7] and high-tech players that are developing and deploying highly automated cars due to the economic incentive: governments and industries simply do not want to "miss the bandwagon of tests, first deployments, and perhaps manufacturing too" [8].

Perhaps There Is No Technological Fix. Politicians, then as now, are willing to believe in a technological fix. In Slayton's narrative, the politicians are accompanied by physicists. Concerning self-driving cars (and other moving things, such as drones and pacemakers), politicians and several computer scientists are in the same camp: "A steady march toward the automated car is clearly under way," according to Stephen Casner et al. [9] in a flagship computer science journal, even though these authors raise profound technological difficulties that will be encountered when attempting to take humans out of the traffic-control loop.

Can History Help to Prevent Mistakes? Given that physicists pushed for automated missile systems with little insight into software's limitations, *perhaps computer scientists are building an Internet of Things without foundations*. Such developments pose risks. Historical research allows scholars to learn from past mistakes in order to identify these risks and to remedy them, and to avoid repeating the same mistakes. According to the present author, the elephant-sized mistake in the history of computer science is misappropriation of the Halting Problem, notably in the language-theoretic approach to security.

1.2. Conflating the Model and the Modeled Artefact

Slayton's narrative suggests that there is no technological fix for missile defence. Zooming in on the Reagan administration, Dave Parnas and other software engineers protested against the Strategic Defense Initiative (SDI) by alluding, in part, to an unjustified belief in a foolproof mapping of mathematical objects onto engineered artefacts — a central theme of the present article. Proving mathematical programs correct, Parnas explained, does not imply that the SDI software itself is correct [10, p.1334]. The mathematical program (= the model) should not be confused with the executable computer program (= that which is modeled). Parnas has argued against this kind of conflation throughout his career:

> "Most of the many published proofs of programs [in computer science] are actually proofs about models of programs, models that ignore the very properties of digital computers that cause many of the 'bugs' we are trying to eliminate." [11]

Here we see Parnas raise a red flag about a common conflation in today's science of computer programming. Similar complaints mainly come from engineers [12, 13], presumably because it is in software and security engineering — as opposed to computer science *pur sang* — that one is trained to repeatedly test the relationship between model and modeled artefact. Likewise, in the 1980s, Parnas emphasized the distinction between testing the missile defence software by means of simulation models and its actual use in a real, nuclear war [10, p.1328].

Model-modelee conflations are the nuts and bolts of the present article. A more famous example of such a conflation and its ramifications is the Y2K or millennium bug. As the previous century came to a close, an increasing

number of stakeholders became aware of the difference between a computer program (= an executable model) and the dates used in daily discourse (= modelee). Many people became worried about the limited regime of applicability of industrial computer programs [14, p.42], because those programs modeled dates (1970, 1900, and 2000) with only two digits (70, 00, and 00, respectively). One implication of this discrepancy between the model and the modelee was that a person born in 1900 turned zero years old in 2000, according to the computer program.

The model-modelee conflation underlying the Y2K bug led to financial repercussions in the 1990s, not to the loss of human lives. Unfortunately, the same can probably not be said indefinitely of modern pacemakers of which thousands have already had to be upgraded due to a security vulnerability [1, Ch.14][15]. Stakeholders who (implicitly) believe in a foolproof mapping — i.e., who conflate the mathematical model of the pacemaker's software with the deployed software itself (= modeled artefact) — will put too much weight on the "security proofs" obtained by researchers. Stakeholders include computer professionals, although, once again, security experts are fortunately among the skeptics [6, 16].

All these examples, along with those forthcoming, convey an overarching theme:

> The history of science & technology is, besides a history of progress, one of conflations between models and modeled artefacts.

Eight more examples of conflations, and primarily of model-modelee conflations, follow in order to get the main theme across before zooming in on computability theory *per se*. First, in linguistics it is not uncommon to mistake the sentences deduced from a formal grammar for a natural language [17, Ch.3]. Second, in strong artificial intelligence one is trained to equate the computation of functions with cognition, rather than "merely" model the latter with the former [18, p.378]. Third, software scholars tend to fuse the categories of Turing machines and stored-program computers [19]. Fourth, engineers mistake a C computer program for a system realization [13, p.4839]. Fifth, and similar to Parnas's critique, computer scientists:

> "should not confuse mathematical models with reality and verification is nothing but a model of believability."
> — De Millo et al. [20, p.279]

Donald MacKenzie's history of formal methods in computer science provides extensive coverage regarding this fifth example [21]. Sixth, Willard Van Orman Quine claimed that Bertrand Russell conflated "propositional functions as notations and propositional functions as attributes and relations" [22, p152]. Seventh, Russell, in turn, had applauded Gottlob Frege for pointing out the common conflation of "all" and "any" in deductive reasoning [22, p.158]. Eighth, Julius König advocated a conceptual divide between "set" and "class" to avoid a now-famous conflation in set theory [23, p.149]. In sum, conflations abound in the intellectual history of computing.

1.3. The Practice of Conflating

The practice of conflating mathematical objects (models) and engineered artefacts (modelees) is powerful and troubling at the same time. It paves the way for:

(1) Mathematical rigor in the science of computer programming, which has led, among other breakthroughs, to powerful software analysis tools, e.g. [24]. Treating computer programs as mathematical objects allows for technical advancements in the software industry. In other words: some conflations are useful, if not both deliberate and useful.
(2) Ignorance of the model's limited regime of applicability in relation to the modelee [14, p.42].
    (a) As Parnas's critique presented previously indicates, it is not uncommon for computer scientists to confuse the mathematical results of the model in hand with the real-world properties of the modeled artefact (i.e., the deployed software in a missile defence system).
    (b) Likewise, ignorance of the modeling activity and too much devotion to the mathematical enterprise can lead to the decision to take the human entirely out of the control loop (of a real missile defence system or vehicle), which can easily result in more casualties than when humans and technology form a creative partnership, cf. [25].

The tendency to conflate models and modelees is omnipresent in mathematical ideals of engineering. Engineering is about making the modeled artefact similar to the model [14], and thus correlates knowledge and the making of artefacts. That is why engineering often starts from the assumption of an identification of a model and modelee, and then works its ways against the 'resistance of the material object' or 'the resistance of the modelee.' Thus, 'viewing the modelee as identical to the model' is both a source of insight (as in 1 above) and of error (as in 2 above), and making this distinction between 1 and 2 is a matter of testing and 'a posteriori' experience.[1]

1.4. Mathematical Objects versus Engineered Artefacts

Mathematical objects such as mathematical programs and computable functions are, in the present paper, consistently distinguished from physical objects, such as computer programs. The latter, also called engineered artefacts, consume energy and take up physical space (while the same cannot be said of mathematical objects).

The categorical distinction just made might be perceived as simplistic, and some readers will likely come across Platonic undertones in the sequel. In anticipation of criticism along these lines, I provide three responses. First, separating mathematical from physical objects can be accomplished without subscribing to Platonism *per se* [26]. Second, most if not all historical actors mentioned in the present article (implicitly) approved of the aforementioned categorical separation (and possibly even of Platonism itself). Third, and to the best of my knowledge, philosophers of computing have yet to thoroughly investigate physicalism, idealism, and other philosophical positions in connection with computer programming. For instance, it would be interesting in future work to scrutinize Strachey's 1965 proof from the perspective of mentalism.

Two further remarks can be made about the terminology used in this article.

- Strictly speaking a distinction can be made between a computer program (stored in, say, a laptop) and an executing computer program (in a laptop). Nonetheless, both objects consume energy and take up physical space; it is partly for this reason that I shall simply call both objects engineered artefacts.
- A mathematical program typically serves as a mathematical model of both a computer program and, especially, of an executing computer program. All objects introduced in the previous sentence (be they physical or mathematical) can be represented with a program text, which is, say, printed on paper.

For a different (yet seemingly compatible) analysis I refer to Ray Turner's philosophy [27] and the following two stipulations in particular:

- Turner distinguishes between symbolic programs and abstract mathematical objects. The former can be equated to the latter provided we incorporate the semantics of the containing language in our discourse. (I make no such distinction; instead, I use the umbrella term 'mathematical program.')
- Turner states that programs are not purely mathematical entities, for the former have to be physically manifested. (I use the term 'computer program' in this paper.)

Unlike me, Turner consistently uses the term 'technical artifact' to unify the symbolic and physical guises of programs [27, p.52]. Further research would be required in order to compare Turner's analysis with the ideas presented in this paper.

Additional philosophical grounding can be found in the textbooks of Timothy Colburn [28], William Rapaport [29], and Giuseppe Primiero [30], not in the present paper. The sequel is, after all, written with a historical and a technical hat, not with the hat of a philosopher. I let my historical actors speak for themselves, i.e., I use *their* words, and therefore often eschew definitions at the outset.

2. Language Theory and Security

Another way of expressing the tendency to conflate between the model and the modeled artefact is as follows: Many computer scientists implicitly believe in a foolproof mapping of computing models onto running machines. The belief in foolproof mapping is a point of discussion among technical historians and philosophers — including

[1] Thanks to Erhard Schüttpelz for sharing this bird's-eye view with me.

James Moor [31], Brian Cantwell Smith [32], James Fetzer [33, 34], Colburn [28], Donald MacKenzie [21], and Edgar Daylight [35] — yet it is often neglected among computer scientists. For the sake of being precise, as will be attempted next, provocations become difficult to completely avoid:

> It is not uncommon to observe computer scientists talk about their computing models as if these are the very computers and computer programs that they are analyzing, constructing, or executing. An awareness of their belief in a foolproof mapping is in various cases implicit, if present at all.

These words are paraphrased from other sources [31, 34, 35]. Examples of computing models are "Turing machines," named in honor of Alan Turing, the father of computer science [36]. To be even more precise, and in my own words:

> Computer scientists — and relatively few software and security engineers — believe in a Turing Fix in that they, often heedlessly, treat a laptop as a Turing machine or a computer program as a Turing machine program in their research, thereby downplaying the modeling activity in hand.

The genuine praise for engineers, expressed in the previous passage, does not imply that conceptual clarity pertaining to security research cannot be increased (from a Turing-Fix perspective). Examples are presented in Section 2.1 and Section 2.2. The present paper's focus on the Halting Problem is motivated and elaborated in Section 3.

2.1. Fred Cohen and Eric Filiol

The abstract of Fred Cohen's seminal 1987 paper, 'Computer Viruses: Theory and Experiments' [37], states:

> "This paper introduces 'computer viruses' and examines their potential for causing widespread damage to computer systems. Basic theoretical results are presented, and the infeasibility of viral defense in large classes of systems is shown. [. . .]" [37, p.22]

In his paper Cohen mathematically *models* computer-virus programs as Turing machines [37, p.25]. Subsequently, he uses the undecidability of the Halting Problem to make a direct claim about industrial practice [37, p.22], as the following quote from his paper indicates:

> "Protection from denial of services requires the detection of halting programs which is well known to be undecidable [38]." [37, p.22]

Strictly speaking — and only *strictly* speaking — this line of reasoning can be improved. (There is no reason to believe Cohen would disagree.) Protection from denial of services "is undecidable" with Cohen's Turing-complete modeling language, it is not impossible in an absolute, *practical* sense. Therefore, Cohen's abstract can be re-phrased like this:

> In this paper we model 'computer viruses' as Turing machines and subsequently use a classical undecidability result from computability theory to gain insights into the industrial problem of building viral defense systems.

Later on in his paper, Cohen states that he has presented infeasibility results that "are not operating system or implementation specific, but are based on *the* fundamental properties of systems" [my emphasis]. Continuing, he writes that "they reflect realistic assumptions about systems currently in use" [37, p.34]. However, historians of computing will stress that the Turing-machine model of computation is not the only model that has been used throughout history in *attempts* to achieve these research goals. Indeed, many Turing incomplete mathematical languages are used in software industry.

Strictly speaking, Cohen conflates industrial problems such as the "detection" of computer viruses and theoretical notions such as "undecidability." For example,

> "Precise detection [of computer viruses] is undecidable, however, statistical methods may be used to limit undetected spreading either in time or in extent." [37, p.34]

It is in Cohen's model that precise detection of mathematically modeled computer viruses is an undecidable problem, not outside the model. Likewise,

"Several undecidable problems have been identified with respect to viruses and countermeasures." [37, p.34]

Cohen has identified several undecidable problems with regard to his model of the industrial problem in hand.

A similar and more recent account on computer viruses is Eric Filiol's 2005 book, *Computer Viruses: From Theory to Applications* [39]. Filiol begins the technical part of his exposition by putting "Turing machines" front and center yet without acknowledging that he is, in fact, selecting only one possible modeling technique. In Filiol's words:

> "The formalization of viral mechanisms makes heavy use of the concept of *Turing machines*. This is logical since computer viruses are nothing but computer programs with particular functionalities. [...] A Turing machine [...] is *the* abstract representation of what a computer is and of the programs that may be executed with it." [39, p.3, my emphasis]

Filiol's reasoning is only valid and effective if all researchers (and all hackers across the globe) accept that computer programs can be adequately modeled, and can only be adequately modeled, with Turing machines. But accepting either of these two premises would amount to ignoring a large part of the history of computer science and current-day practice. Alternative models of computation, such as linear bounded automata, have been introduced in the past precisely because the Turing-machine model was deemed inadequate [40]. In technical terms, neither the model fidelity of a Turing machine, nor that of a linear bounded automaton, is absolutely perfect with regard to the modeled artefacts in hand; i.e., ordinary programs, virus programs, computers, and other engineered artefacts (cf. Section 3).

> We should expect that a science of cybersecurity will not be built around a single model
> — Fred B. Schneider [3, p.51]

2.2. Len Sassaman et al.

On the praising one hand, historical observations concerning the importance of Turing incomplete modeling languages are backed up by security experts themselves and most notably by Len Sassaman et al., i.e., by advocates of the language-theoretic approach to security. Researchers in this niche develop techniques to strengthen the security of applications on a high level by using the properties of programming languages. They prevent vulnerabilities which, say, mainstream operating-system security is unable to handle.[2] Decidability matters, according to Sassaman et al., for "good protocol designers don't let their protocols grow up to be Turing-complete, because then the decision problem is UNDECIDABLE" [41, p.28]. On the other, critical hand, Sassaman et al. do at times fail to escape from the Turing Fix themselves. For example, they observe that "`HTML5` is Turing-complete, whereas `HTML4` was not" [41, p.29]. But, strictly speaking, Sassaman does not explicitly distinguish between the mathematical model(s) and the modeled artefact. Only a mathematical language can be Turing complete, not an engineered artefact. My re-phrasing leads to the following statement:

> Many people seem to prefer — and perhaps justifiably so — to mathematically model `HTML5` with a Turing-complete language L, whereas the same cannot be said of `HTML4`.

The bread-and-butter distinction in security research is, after all, that between models and modeled artefacts. So the extra nuance is hopefully appreciated by my readership. The following two more points are perhaps well-known too, yet deserve to be emphasized in the present context as well.

(1) Proving mathematical properties on one particular model (language L in the `HTML` example) of the engineered artefact under scrutiny (`HTML5`) is not the same as asserting properties of the artefact itself. Perhaps the following analogy is instructive. Consider the number five. My hand is a physical object that can serve as a representation of the number five. However, it is the number five, and not my hand, that has the mathematical property of being a prime number. Turing completeness or incompleteness can only hold for a mathematical modeling language of, say, the `HTML5` language, not for the industrial language itself.

[2] Paraphrased from the wikipedia entry on "language-based security." (Accessed on 11 October 2019.)

(2) The mathematical language L embodies conceptual knowledge about programming practices. This knowledge is "secondary and representational, hence, it is necessarily incomplete and partial, and it is error-prone (rightly evoking Cartesian doubt)" — a message that I have appropriated from Peter Brödner's insightful work [42] and which is also conveyed from a different angle in Cantwell Smith's celebrated 1985 paper 'The Limits of Correctness' [32]. Another source of inspiration comes from Edward Lee's plenary talk 'Verifying Real-Time Software is Not Reasonable (Today),' presented at the Haifa Verification Conference in 2012, where Lee described the *Kopetz Principle* — after Hermann Kopetz, from whom he learned it — with the following words:

> "Many (predictive) properties that we assert about systems (determinism, timeliness, reliability) are in fact not properties of an implemented system, but rather properties of a *model* of the system. We can make definitive statements about *models*, from which we can infer properties of system realizations. The validity of this inference depends on *model fidelity*, which is always *approximate*."

I refer to Lee's 2017 book *Plato and the Nerd* [14] for the bigger picture.

Philosophy Unplugged. The previous remarks are perhaps too simplistic for the professional philosopher. Yet they are on a par with the scholarly style employed by Phil Agre who argued that database specialists "shift repeatedly between treating entities as things in the world and treating entities as representations of things in the world." (The specialists whom Agre referred to are Michael Reingruber and William Gregory [43].) Agre expounded as follows:

> "Their choice of example facilitates the confusion, given that the word "play" refers to both the text and the performance, the representation and the thing represented. [...] These authors may have been misled by the practice, common but usually well defined in the literature, of using the word "entity" to refer both to categories of things (cars, people, companies) and to instances of those categories (my old 240Z, Jacques Martin, IBM)." [44]

Agre's narrative lends credence to the vexing claim, that, "the conflation of representation and reality is a common and significant feature of the whole computer science literature" [44].

3. Strachey's Halting Problem

Until recently, security research was deemed mostly orthogonal to language-theoretic considerations, which do prevail in compilation, programming language design, and other well-established branches of computer science. I borrow this observation from Sergey Bratus et al. [45, p.20]. As proponents of the language-theoretic approach to security, these authors seek a better understanding of the writings of Alonzo Church, Turing, et al. In their words:

> Computer security's core subjects of study—*trust* and *trustworthiness* in computing systems—involve practical questions such as "What execution paths can programs be trusted to not take under any circumstances, no matter what the inputs?" and "Which properties of inputs can a particular security system verify, and which are beyond its limits?" These ultimately lead to the principal questions of computer science since the times of Church and Turing: "What can a given machine compute?" and "What is computable?"
> — Quoted from Bratus et al. [45, p.16]

The advent of computability theory is indeed largely due to Church, Turing, Kurt Gödel, Emil Post, and other logicians. And the appropriation of the "Turing machine" concept by computer programmers after the second world war is a story that has been told, albeit only recently [36, 46, 47]. But the advent of undecidability results in programming has yet to be thoroughly analyzed and documented. The present author has made a preliminary contribution in this regard [2] and (hopefully) a more definite one in the following pages.

Half a century ago Bob Floyd demonstrated that no algorithm can decide whether infinitely many arbitrary context-free grammars are ambiguous [48]. On the other hand, Floyd knew very well that a context-free grammar is merely *a* model of the programming language under scrutiny. Specifically, in 1962, Floyd (and fellow first-generation computer scientists) mathematically modeled the syntax of the `ALGOL 60` programming language (i.e., a predecessor of `C`) with a context-free grammar. Strictly speaking, then, Floyd's undecidability result only holds for a specific

mathematical tool for the description (context-free grammars) of a particular aspect of a programming language (i.e., its formal syntax before contextual constraints are taken into account). Again, in my words:

> The undecidability result in hand, not unlike mathematical results in adjacent engineering disciplines, holds for the mathematical models of the engineered artefacts, not for the engineered artefacts themselves.

(This distinction between models and artefacts can also be made for modern languages such as Standard `ML` where the formal syntax & semantics serve as *prescriptions* for the language implementers [34, p.259].)

Let us call the *fidelity* of a mathematical model the degree to which it emulates the engineered artefact, also called the *target* [14, p.41]. In various research communities, the model fidelity is always deemed to be approximate [13, 32, 49] and mindful researchers use the model only when the target is "operating within" the "regime of applicability of the model" [14, p.42]. For the case study discussed next — Christopher Strachey's 1965 Halting Problem — I will argue that the model fidelity is indeed imperfect. Specifically, Strachey ignored the fidelity altogether in his activity of modeling physical computations, in which he used computable partial functions (mathematical objects) as models for executable programs (engineered artefacts).

3.1. Strachey's 1965 Letter

Strachey's letter 'An impossible program' appeared in January 1965 in the Computer Journal with the following opening sentence: "A well-known piece of folklore among programmers holds" — folklore which I call *Strachey's Halting Problem* — that it is

> "impossible to write a program which can examine any other program and tell, in every case, if it will terminate or get into a closed loop when it is run." [50]

This modern, and now common, interpretation of the Halting Problem is about executable programs. I refer to Appendix A for another popular instance of the same, common interpretation.

Strachey's narrative stands in stark contrast to the purely mathematical expositions of the Halting Problem, provided by Stephen Kleene, Martin Davis, and other logicians in the 1950s (not to re-mention, of course, the writings of Church, Post, and Turing in earlier years). Consider, for instance, having a look at *Davis's Halting Problem* in his 1958 book *Computability & Unsolvability*, which is about Turing machines only, not technology [51, p.70]. To be more precise, Davis's proof concerns numerical codes of Turing machines, obtained via a Gödel-style coding. His proof rests on Kleene's T predicate, which is defined over natural numbers and used in the context of computable functions.

The folklore erroneously attributes the proof of the undecidability of the Halting Problem to Turing, as clarified by Jack Copeland [52, p.40]:

> "The halting problem was so named (and, it appears, first stated) by Martin Davis.(*) The proposition that the halting problem cannot be solved by computing machine is known as the 'halting theorem.' (It is often said that Turing stated and proved the halting theorem in 'On Computable Numbers', but strictly this is not true.)"
>
> (*) "See M. Davis, *Computability and Unsolvability* (New York: MacGraw Hill, 1958), 70. Davis thinks it likely that he first used the term 'halting problem' in a series of lectures that he gave at the Control Systems Laboratory at the University of Illinois inn 1952 (letter from Davis to Copeland, 12 Dec. 2001)."

The proof is also sometimes erroneously attributed to Stephen Kleene. The reason of this incorrect attribution could be explained by the fact that in Kleene [53, p.382] one finds the following statement (even if no proof of it is given):

> "So by Turing's thesis (or via the equivalence of general recursiveness and computability, by Church's thesis) there is no algorithm for deciding whether any given number x is the Gödel number of a machine which, when started scanning x in standard position with the tape elsewhere blank, eventually stops scanning x, 1 in standard position."

Suppose $T[R]$ is a Boolean function taking a routine (or program) R with no formal or free variables as its argument and that for all R, $T[R] =$ **True** if R terminates if run and that $T[R] =$ **False** if R does not terminate. Consider the routine P defined as follows

rec routine P
  $\natural L$: **if** $T[P]$ **go to** L
  Return $\natural$

If $T[P] =$ **True** the routine P will loop, and it will only terminate if $T[P] =$ **False**. In each case $T[P]$ has exactly the wrong value, and this contradiction shows that the function T cannot exist.

Figure 1. Strachey's alleged 1965 proof [50]

Returning now to Strachey 1965, the follow-up sentences in his letter suggest — and the Strachey archives in Oxford support my historical interpretation[3] — that his only source of inspiration with regard to the topic in hand was a decade old conversation:

> "I have never actually seen a proof of this in print, and though Alan Turing once gave me a verbal proof (in a railway carriage on the way to a Conference at the NPL in 1953), I unfortunately and promptly forgot the details. This left me with an uneasy feeling that the proof must be long or complicated, but in fact it is so short and simple that it may be of interest to casual readers. The version below uses CPL, but not in any essential way." [50]

The programming language `ALGOL 60` was a predecessor of `CPL` (= Combined Programming Language), which in turn was a predecessor of the `C` language [54, p.116]. `CPL` programs could be compiled and executed, as also the following comment from one of Strachey's readers indicates:

> I equate "program" with "program capable of being run".
> — H.G. ApSimon, 27 August 1965.

The rest of Strachey's letter is his alleged proof in Figure 1. I encourage the reader to scrutinize Figure 1 before reading on. It makes an excellent exam question.

Strachey's proof relies on the notion of recursion. In this regard I mention that historical accounts pertaining to `ALGOL 60` and the advent of recursive procedures (and corresponding stack frames) have been written in recent years [55–57].[4]

3.2. Three Kinds of Reactions to Strachey's Letter

Strachey's 1965 letter invoked three kinds of reactions in Volume 8 (issues 1, 3, and 4) of the Computer Journal. The first reaction is mostly of historical interest: some readers complained about the concept of 'proof by contradiction,' some were used to encountering it in geometry and other classical domains, not in computer programming.

The second kind of complaint is important but also forgiving from a historical perspective. Strictly, Figure 1 is incorrect for the following reason: $T[\ldots]$ should be taken to be a *computable* Boolean function, not merely a Boolean function. The conclusion of Strachey's proof should then amount to the rejection that T is computable. Strachey did not seem to be aware of the need for this correction, as his replies to comments made by ApSimon indicate. Nevertheless, let it be clear that Strachey was advancing computer science by appropriating ideas from modern logic [2].

[3] Although Strachey was appropriating ideas from the lambda calculus, his incentives to do so were mostly orthogonal to the topic of incomputability. Moreover, his close collaboration with the logician Dana Scott only began around 1969 [54, p.116].

[4] Moreover, anticipating the — in my eyes, unjustified — critique that my analysis might be (a bit) anachronistic, I emphasize that there is no reason to believe, given the current state of the art in the history of programming languages [58], why Strachey could not suppose that $T[P]$ is indeed a program taking as input a program P (and not a code of P). I thank Simone Martini for sharing this supportive viewpoint with me.

It's the third kind of reaction that still sticks today. Using my own terminology: Strachey was *modeling* engineered artefacts (CPL programs) with mathematical objects (computable partial functions) and he was not explicit about this. In fact, one correspondent had mathematically modeled imperative programs with finite state machines. Before reading Strachey's letter he had come to the opposite conclusion, that the Halting Problem *is* solvable:

> [Strachey's] letter was of particular interest to me because I had, several months ago, proved that it is indeed possible to write such a program [...]
> — W.D. Maurer, 27 August 1965

Appendix B fully captures Maurer's position, which is common in engineering.

Strachey's line of reasoning in Figure 1 can thus be improved by formalizing his modeling activity, as I shall do in the following paragraphs.

3.3. Improving Strachey's Proof

Let R denote a CPL routine (or CPL program) with no formal or free variables as its argument. Let R^{model} denote Strachey's mathematical model of routine R. That is, R^{model} denotes a computable partial function, which:

(1) The reader can formalize in compliance with the solely mathematical — and correct — 1994 exposition of Davis et al. [59]. Intuitively, R^{model} captures the input/output behavior of routine R and discards all other details.
(2) Has an imperfect model fidelity.

Concerning the second point: a computable partial function is an *idealization* of a program executing on a physical machine. Three examples of idealization follow. First, the actual running time is abstracted away. One could, unlike Strachey, choose to model this aspect as well, but the point is that one will never be able to mathematically capture the real happening completely (cf. The Kopetz Principle in Section 2.2). Second, any program executing on any physical machine consumes a finite memory footprint. To improve — but, again, not perfectionize — the model fidelity, it would perhaps make more sense to resort to weaker models of computation. In fact, and as stressed before, at some point in history computer scientists started preferring linear bounded automata and finite state machines, perceiving them as less baroque than Turing machines.[5] Third, a perfect model fidelity would imply a one-to-one mapping between the computable partial functions R^{model} and the CPL routines R. Since Strachey's intended mapping is not one-to-one,[6] it follows that the model fidelity is not perfect.

Suppose $T[R^{model}]$ is a *computable* Boolean function taking R^{model} as its argument and that for all CPL routines R,

(1) $T[R^{model}] =$ **True** if CPL routine R terminates if run, and
(2) $T[R^{model}] =$ **False** if CPL routine R does not terminate if run.

We say that B *implements* A if and only if A models B. Now, suppose that a CPL program T_{prog} exists, which *implements* function T.

Consider the CPL routine P represented textually as follows:

```
rec routine P
  ♮L :    if T_prog[P] go to L
          Return ♮
```

Note, below, that T_{prog} has to be faithful enough to T for the proof to carry through. In general terms: the model fidelity has to be "good enough" if not perfect. Consider now the analysis:

---

[5] See Michael Mahoney [40, p.133]. Note also that I use the terms "Turing machine" and "computable partial function" interchangeably. This simplification would be unwarranted if Strachey and I were attempting to address the fundamental question *What is an algorithm?* [60], rather than: *What is an impossible program?* Another relevant question has recently been addressed by William Rapaport: *What is a computer?* [29].

[6] See Davis et al. [59] for the rigor. The crux is that computable partial functions R^{model} exist that have more than one target R. In layman's terms: multiple imperative programs R exist which are functionally equivalent (to R^{model}).

- If $T[P^{model}]$ = **True**, then — assuming a "good enough" model fidelity — $T_{prog}[P]$ evaluates to True at runtime. So then CPL routine P will loop. Assuming a "good enough" fidelity, this means that $T[P^{model}]$ = **False**.
- If $T[P^{model}]$ = **False**, then — under the same assumptions — $T_{prog}[P]$ evaluates to False at runtime. So then CPL routine P will terminate. Under the same assumptions this means that $T[P^{model}]$ = **True**.

In each case we have a contradiction. So, at least one assumption does not hold. Our assumptions include:

(1) Computable Boolean function T exists.
(2) T_{prog}, P, ... constitute valid CPL programs (e.g., they are compilable).
(3) The model fidelity is "good enough."

Strachey only took Assumption 1 into consideration and concluded, by *reductio ad absurdum,* that it did not hold. But who says engineered artefact T_{prog} exists and, especially, that the model fidelity is "good enough"? (Note, in passing, that a perfect model fidelity implies that Assumptions 2 and 3 hold.) At the very least, Strachey's philosophical stance on the relationship between computable partial functions and programming technology needs to be made explicit so that Assumption 2 and *possibly* also Assumption 3 can be safely ignored by charitable readers.[7]

3.4. A Philosophical Intermezzo

In retrospect, some critical readers will insist that Assumption 2 is a fact, not an assumption. Once one assumes that $T[R]$ in Figure 1 is a CPL routine, it follows that P *is* a valid CPL program. According to these critical readers, the only possible reading of Strachey's expression "Suppose $T[R]$ is a Boolean function taking ..." is "Suppose $T[R]$ is any CPL routine computing a Boolean function taking ...". If I then claim that Strachey did not seem to be aware of the need to specify that T has to be a *computable* Boolean function (as I have done in Section 3.2), my critical readers will insist that Strachey did not need to do so because T is computable by the very fact that it is supposed to be a "program."

There are several ways to respond to these insightful remarks, which in fact come from an anonymous critical reader for which I express my gratitude. First, suppose that I concur. Then the crux of my analysis remains intact: Assumption 3 is the main issue of the present article, not Assumption 2 (as my critical reader confirms). Second, the remarks of my critical reader were also expressed in Volume 8 of the Computer Journal by yet other scholars than those mentioned in the present article. No consensus on the matter was obtained in 1965, nor (apparently) up till this sentence in the present article, which is a point worth making explicit here. Third, strictly speaking, I disagree with the critical reader: there is a categorical difference between a computable function (a mathematical object) and a textual representation of that function, with the latter serving as a prescription for a physical computation on a real computer.[8] The critical reader is actually interpreting the word "program" as a mathematical object — For, how else can T be both a function and a program? — while the same word frequently refers to a compilable CPL program. This perfect ambiguity of the word "program" is precisely the core of my critique and is reminiscent of Parnas's remarks in Section 1.2.

4. Closing Remarks

The relation between basic computability theory, computer science, and computing practice, is something that is always taken for granted. At the very least, this paper has shown instead, with a simple but paradigmatic example, that this relation is subtle and could have several, distinct, and not necessarily consistent, interpretations.

To be more precise, I have provided a critical reading of a short letter by Christopher Strachey to the Computer Journal (1965), which presents a proof of the undecidability of the Halting Problem. The letter uses the (nowadays

[7] So far, I have avoided referring to a semi-related concept: "The Physical Church-Turing thesis." It should be stressed at least once, that, contrary to what one finds in computer science textbooks, there is no general consensus, let alone a mathematical justification, that this thesis holds [61].

[8] This categorical distinction is actually between abstract objects and the so-called "technical artefacts" of Ray Turner [62]. I have built on that distinction in my book [35] and prefer to avoid it here, for I aspire reaching out to a more general audience.

usual) diagonal technique, formulating it into a real programming language (Strachey's own CPL), and thus arguing that:

There are *no programs* in CPL that solve the Halting Problem. (*)

My critique in the present paper is that Strachey in his exposition mixed the layer of the actual program (which will be compiled and run on physical machinery) and the layer of the mathematical model of the program. By doing this, (*) is a "non sequitur" from the alleged proof. One needs an additional hypothesis of "fidelity" of the model with respect to actual programs, something that Strachey never observed, and even less cared to formulate.

Like in many papers in programming language theory, when an author says "this program does that" it is always tacitly assumed that the utterance means "this program, when executed on an abstract machine with unlimited memory, unbounded time, true unbounded integer arithmetic, and perfect fidelity for at least conditional and while statements, does that." In itself, it is a harmless way of simplifying an exposition, provided both the speaker and the listener agree on the convention. In this paper I have argued that this is not always the case.

4.1. Separation of Concerns

The presented analysis suggests that a good dissemination strategy, regarding undecidability and its practical relevance, should comply more with Martin Davis's aforementioned expositions rather than with Strachey's. The crux is to remain solely in the mathematical realm of computable partial functions or Turing machines when explaining undecidability to students and fellow researchers. A refined remark holds for undecidability results pertaining specifically to mathematically modeled computer viruses (cf. Section 2). A mathematical object such as a function cannot be executed nor hacked. A *separate* concern, then, is to discuss and debate how *that* mathematical impossibility result could — by means of a Turing complete *modeling* language of computation — have bearing on the engineered artefacts that are being modeled.

4.2. Historical Interpretation

My educated guess is that in 1965 — not to mention today — almost every engineer would have preferred not to model unbounded-memory computations when perusing the computational limits of programming technology. The engineer would explain that, in the interest of finding a useful "limit on technology," s/he would preferably resort to finite state machines instead of computable partial functions. However, even then the model fidelity is far from perfect; a computer is not a finite state machine, it can only be modeled as such.

What about programming language experts in the 1960s and 1970s who, due to their very research agenda, were abstracting away from their computing machinery? On the one hand, some programming language experts (such as Edsger W. Dijkstra) insisted on using finite state machines in their mathematical work [35, Ch.6]. On the other hand, and as discussed above, mathematicians such as Strachey preferred using *computable partial* functions. (Although the reader should keep in mind that the italicized adjectives in the previous sentence are of my choosing and that Strachey's 1965 letter is solely about "functions.") All this begs the following questions pertaining to Strachey:

(1) Why did Strachey rely on infinite memory in his analysis of computation? (Similarly, why did Cohen and Filiol do so as well in the context of modeling computer viruses? — See Section 2.)
(2) Why did Strachey present his alleged proof in the first place?

With regard to the first question: the short answer is that it allowed Strachey, in his programming language research in the 1960s, to proceed from the simplest mathematical case (infinite memory) to the more complex (man-made constraints) of intrinsically finite artefacts — paraphrasing J.W. Waite, Jr [35, p.216]. A longer answer, also in connection with Dijkstra's views, appears in Chapter 6 of *Turing Tales* [35]. Concerning the second question, the only justification I can find is intellectual pleasantry and based on Strachey's personal writings I don't think he would disagree with me either.

In sum, Strachey's 1965 letter needs drastic re-writing before it can be disseminated "as a proof" to fellow computer scientists today. As a university lecturer, I have been asked to teach Strachey's proof — and, likewise, to defend Hopcroft et al.'s faulty reasoning (presented in Appendix A) — "as such" to Master's students; that is, to fuse real systems and models of computation. Or, to use the words of one of Strachey's readers in 1965:

> . . . for determining whether or not a program gets into a closed loop is something programmers are doing every day. It would be very odd if some of the tests and intuition they use in doing this could not be turned into worthwhile compiler diagnostics. Writers of these in the world of practical application should not let Strachey's formidable piece of generality frighten them off!
> — P.J.H. King

Many computer professionals share King's intuition and some, if not many, ignore the Halting Problem altogether. Malicious hackers are not constrained by Strachey's impossibility result and hopefully benevolent academics aren't either. If the present article carries any practical weight it is due to my rectification of Strachey's 1965 line of reasoning — reasoning that, like a virus, has spread widely through various niches of computer science, with the niche of language-theoretic security being a very notable one indeed.

Acknowledgements

Thanks to Erhard Schüttpelz and Michiel Van Oudheusden for commenting on multiple drafts of the present paper. Gratitude to Simone Martini and Liesbeth De Mol for discussing Strachey's Halting Problem with me for several months. I also received detailed and useful comments from two anonymous referees of the annual conference *Computability in Europe (2018)*; one reviewer is the "anonymous critical reader" mentioned in Section 3.4. That person now turns out to be Simone Martini again. I have used his feedback extensively, especially at the beginning of Section 4. I also thank three anonymous referees of the present journal, *Computability*, for providing encouraging and extensive feedback in 2019. Three insightful, technical comments in Section 3.1 come verbatim from the second referee. He also aptly pointed out, in a 2020 review, that the findings of Len Adleman [63] and the more recent work of Jean-Yves Marion [64] could be discussed in connection with my scrutiny of 'modeling computer viruses.' Finally, I praise Karine Chemla's diligence as acting editor of this article.

Appendix A. Hopcroft, Motwani, and Ullman

Strachey's 1965 proof is widespread in computer science today, as the following two sections pertaining to Hopcroft et al. illustrate.[9]

A.1. Turing Machines and Computers

Turing machines are mathematical objects and computers are engineered artifacts. The former can serve as mathematical models of the latter. Likewise, the latter can serve as engineered models (i.e., implementations) of the former. John Hopcroft, Rajeev Motwani, and Jeffrey Ullman take a very similar position in their 2007 textbook [65]. However, in their Chapter 8, they also attempt to mathematically — albeit informally — demonstrate that a computer can simulate a Turing machine and that a Turing machine (TM) can simulate a computer. All this in order to come to the following dubious result:

> "Thus, we can be confident that something not doable by a TM cannot be done by a real computer." [65, p.372]

How can a Turing machine, just like a prime number, "do something" in the way that a real computer does something? I will argue that to make sense of this claim, we need to be explicit about our modeling activities. Specifically, we should distinguish between two persons:

(1) An engineer who models (i.e., implements) the Turing machine with a real computer.
(2) A scientist who mathematically models the real computer with a Turing machine.

For an elaborate distinction between an engineer and a scientist, see Lee's *Plato and the Nerd* [14].

Coming back to Hopcroft et al.'s Chapter 8, "simulating" a Turing machine with a computer (cf. 1.) is easy provided that we are allowed to:

[9] Extracted from my blog: *www.dijkstrascry.com/HopcroftUllman*

"accept that there is a potentially infinite supply of a removable storage device such as a disk" [65, p.372]

The authors then note that "this argument is questionable," since the "physical resources to make disks are not infinite." Subsequently, the authors ask their readership to be "realistic in practice" so that the aforementioned simulation carries through [65, p.372]. Fine, but then we are stepping away from a purely mathematical argument. That is, "simulation" acquires an engineering connotation.

Coming then to the simulation of a computer with a Turing machine (cf. 2.): to do this in polynomial time, the authors now introduce a *finiteness* constraint — which the skeptic could contrast with the aforementioned supposition that, practically speaking, there are an *unbounded* number of resources. Specifically, the authors do put "a limit on the number of bits that one computer word can hold" in order for the proof to work out [65, p.369]. There is no contradiction here; nevertheless, the choices made by Hopcroft et al. are clearly *modeling* choices so that their overall argument works out in the first place.

In a word, the three authors are actually mathematically proving an equivalence between the following two very similar mathematical models:

- a Turing machine
- a carefully crafted digital abstraction (i.e., a mathematical model) of a real computer

Hopcroft et al. are merely perceiving real computers in a specific way, i.e., in compliance with classical computability theory. They are implicitly working with a particular mathematical model of a real computer, not with a real computer itself. The authors are thus definitely *not* backing up their following two claims [65, p.362]:

(1) A computer can simulate a Turing machine.
(2) A Turing machine can simulate a computer.

If the authors want to keep the verb "to simulate," then they should replace "computer" with "a mathematical model of a computer." (Moreover, the mathematical modeling language has to be Turing complete.) Alternatively, if the authors prefer to keep the word "computer," then they should replace the verb "to simulate" with the verb "to model." In the latter case, we again get something like this:

(1) A computer can model (i.e., implement) a Turing machine.
(2) A Turing machine can mathematically model a computer.

Note that the modeling in 1. and the modeling in 2. are very different. Moreover, modeling implies idealizing (cf. the Kopetz Principle in Section 2.2). In this regard, the authors incorrectly draw the following conclusion:

"Thus, we can be confident that something not doable by a TM cannot be done by a real computer." [65, p.372]

The previous statement only holds if the authors have demonstrated an isomorphism between Turing machines on the one hand and real computers on the other hand. But they haven't. The isomorphism they consider only holds between Turing machines and their carefully crafted abstractions of real computers.

After all, is it not possible, that, if we look inside a real computer and refrain from mapping our observations onto our favorite mathematical objects, that the computer is, in some sense, doing something for us that Turing machines do not do? Only if we look at real computers with our traditional spectacles — in which computable partial functions are *the* preferred objects — can we equate the Turing machine with the computer in a traditional and rather weak sense.

A.2. Turing Machines and Computer Programs

There is more. Hopcroft et al. also want to equate Turing machines to computer programs. My contention, in contrast, is to view a Turing machine as one possible mathematical model of a computer program. A finite state machine is another mathematical model of a computer program. And a linear bounded automaton is yet another. Hopcroft et al. claim that the Turing machine is better than the finite state machine (see below). I, however, view neither model to be better, for it all depends on the engineering task in hand.

Hopcroft et al. clearly view computer programs as Turing machines:

> "[I]t is undecidable whether a program prints *hello, world*, whether it halts, whether it calls a particular function, rings the console bell, ..." [65, p.413]

However, every now and then Hopcroft et al. seem to be aware that computer programs are not Turing machines after all:

> "Programs are sufficiently like Turing machines that the [above] observations [...] are unsurprising." [65, p.413]

Furthermore, Hopcroft et al. are aware that computer programs can also be perceived as finite state machines (or finite automata), but they consider a finite state machine to be a poor choice. In their own words:

> "[T]reating computers as finite automata [...] is unproductive." [65, p.322]

(It is not always unproductive, it all depends on the engineering task in hand.) The authors stick to the Turing machine model and motivate their choice by stipulating that computer memory can always be extended in practice:

> "If we run out of memory, the program can print a request for a human to dismount its disk, store it, and replace it by an empty disk." [65, p.322]

So, to champion the Turing machine model, the authors have decided to model a composite system: a real computer with a human being. No-nonsense engineers, by contrast, will prefer to use a weaker model of computation and stick to the system in hand: a real computer and nothing else. Hopcroft et al. eschew linear bounded automata and other alternative models of computation. Yet, based on their motivation to dismiss finite state machines, I would opt for a linear bounded automaton and not a Turing machine. But, of course, if I do *that* then the much-desired undecidability result vanishes, for linear bounded automata have a trivially decidable halting problem. That, in short, explains why theoreticians defend the Turing machine as the "best" model of computation in an average computability course.

Appendix B. Maurer

One lengthy response to Strachey's 1965 letter came from Ward Douglas Maurer on 27 August 1965, presented below. Maurer demonstrated a common line of reasoning in engineering which is based on a *finite* abstraction, not an infinite abstraction, of computer memory. Maurer observed that with his approach Strachey's diagonal argument fails.

> Sir,
>
> I have just come across Strachey's letter (*The Computer Journal*, Jan. 1965, reprinted in *Computing Reviews*, July 1965) on the impossibility of writing a program which "can examine any other program and tell, in every case, whether it will terminate or get into a closed loop when it is run." The letter was of particular interest to me because I had, several months ago, proved that it is indeed possible to write such a program, at least in the case of finite memory. It may be of interest to compare my approach with Strachey's (and Prof. Turing's) to observe why the results are not in fact contradictory.
>
> A computer with finite memory has a finite number of states b^m, where b is the number of values which each memory element can take (two, for a binary computer) and m is the number of memory elements. Let us say that a routine terminates if and only if it comes to an instruction which transfers to itself; i.e., does not change the state of the computer. Then a program terminates if and only if the computer eventually reaches a state such that it is the same as the next state. Specifically, let M be the memory of the computer, which is a finite set (including all registers and the location counter); let B be the set of values which each memory element can take ($B = [0, 1]$ for a binary computer), let S be the set of all maps $S : M \to B$, that is, all states (or instantaneous descriptions) of the computer, and let $I : S \to S$ be the map which determines, for each state of the computer (including the value in the location counter, of course) the next state of the computer.
>
> [blank line inserted for readability]

A program is now a particular state S of the computer. (A program may, of course, be represented by various states S, each of which has the same values in that subset of M in which the program is stored; but this point is not essential to the argument.) To determine whether the program S terminates, one simply calculates $I(S)$, $I^2(S)$, ..., until a power $I^{i+j}(S)$ is found which is equal to $I^i(S)$. The program S terminates if and only if $j = 1$. The various states $I^k(S)$ may be kept in a finite memory M' which is disjoint from M; the process will always terminate, since S is finite, and since each $I^k(S)$ has a finite representation, the memory M' may likewise be taken as finite. Thus the theorem is proved.

It is interesting to note that Strachey's disproof does not seem to involve memory; it is applicable to programs running in finite memory, and itself uses a finite procedure which does not use recursion or pushdown storage. The difficulty seems to be that what was actually proved above is the following: Given any program in a finite memory M, there exists a program in a finite memory M' (whose cardinality depends on that of M) which will determine whether the original program terminates or not. Strachey's arguments do not contradict this fact. If Strachey's program P is imbedded in M, and his $T(R)$ (which determines whether R, and in particular P, terminates or not) is imbedded in M', then P calls T, so that P is in fact imbedded in $U \cup M'$ [i.e., $M \cup M'$], and thus the conditions of the statement are violated. In general, M' must be much larger than M.

Sincerely yours,
W.D. Maurer

References

[1] M. Goodman, *Future Crimes: Inside the Digital Underground and the Battle for our Connected World*, Corgi Books, 2016.

[2] E.G. Daylight, *The Dawn of Software Engineering: from Turing to Dijkstra*, Lonely Scholar, 2012.

[3] F.B. Schneider, Blueprint for a science of cybersecurity, *The Next Wave* **19**(2) (2012), 47–57.

[4] R. Slayton, *Arguments that Count: Physics, Computing, and Missile Defense, 1949-2012*, MIT Press, 2013.

[5] P.N. Stearns, Why Study History?, American Historical Association, 1998. `Positionstatement: historians.org/about-aha-and-membership/aha-history-and-archives/historical-archives/why-study-history-(1998).`

[6] U. Lindqvist and P.G. Neumann, The Future of the Internet of Things, *Communications of the ACM* **60**(2) (2017), 26–30.

[7] J. Somers, The Coming Software Apocalypse, *The Atlantic* (2017), See: theatlantic.com/technology/archive/2017/09/saving-the-world-from-code/540393/.

[8] K.D. Grave, Networking self-driving cars, a comment on Edgar Daylight's blog (dijkstrascry.com) from a specialist in the automotive industry, 6 August 2016. `dijkstrascry.com/comment/2232{#}comment-2232`.

[9] S.M. Casner, E.L. Hutchins and D. Norman, The Challenges of Partially Automated Driving, *Communications of the ACM* **59**(5) (2016), 70–77.

[10] D.L. Parnas, Software Aspects of Strategic Defense Systems, *Communications of the ACM* **28**(12) (1985), 1326–1335.

[11] D.L. Parnas, The use of mathematics in software quality assurance, *Frontiers of Computer Science in China* **6**(1) (2012), 3–16.

[12] S.W. Golomb, Mathematical Models: Uses and Limitations, *IEEE Transactions on Reliability* **20**(3) (1971), 130–131.

[13] E.A. Lee, The Past, Present and Future of Cyber-Physical Systems: A Focus on Models, *Sensors* **15** (2015), 4837–4869.

[14] E.A. Lee, *Plato and the Nerd: The Creative Partnership of Humans and Technology*, MIT Press, 2017.

[15] Duizenden pacemakers kwetsbaar voor hacking, *De Standaard, 2 September 2017*.

[16] M. Vanhoef and F. Piessens, Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2, in: *ACM SIGSAC Conference on Computer and Communications Security 2017*, ACM, 2017, pp. 1313–1328.

[17] E. Schüttpelz, *Figuren der Rede: Zur Theorie der rhetorischen Figur*, Eric Schmidt Verlag GmbH & Co., 1996.

[18] J.H. Fetzer, People are not computers: (most) thought processes are not computational procedures, *Journal of Experimental & Theoretical Artificial Intelligence* **10**(4) (1998), 371–391.

[19] E.G. Daylight, A Turing Tale, *Communications of the ACM* **57**(10) (2014), 36–38.

[20] R.A. De Millo, R.J. Lipton and A.J. Perlis, Social Processes and Proofs of Theorems and Programs, *Communications of the ACM* **22**(5) (1979), 271–280.

[21] D. MacKenzie, *Mechanizing Proof: Computing, Risk, and Trust*, MIT Press, 2004.

[22] B. Russell, Mathematical logic as based on the theory of types, in: *From Frege to Gödel: A source book in mathematical logic, 1879-1931*, Harvard University Press, 1981.

[23] J. König, On the foundations of set theory and the continuum problem, in: *From Frege to Gödel: A source book in mathematical logic, 1879-1931*, Harvard University Press, 1981.

[24] B. Cook, A. Podelski and A. Rybalchenko, Proving Program Termination, *Communications of the ACM* **54**(5) (2011), 88–98.

[25] L. Mearian, Here's Why Self-Driving Cars May Never Really Be Self-Driving, *Computerworld*, See: computerworld.com/article/3171160/car-tech/heres-why-self-driving-cars-may-never-really-be-self-driving.html Accessed on 27th January 2018..

[26] S. Shapiro, *Thinking about mathematics: The philosophy of mathematics*, Oxford University Press, 2000.

[27] R. Turner, *Computational Artifacts: Towards a Philosophy of Computer Science*, Springer, 2018.

[28] T.R. Colburn, *Philosophy and Computer Science*, M.E. Sharpe, 2000.

[29] W.J. Rapaport, What is a Computer? A Survey, *Minds and Machines* (2018), Published online: 25 May 2018.

[30] G. Primiero, *On the Foundations of Computing*, Oxford University Press, 2020.

[31] J.H. Moor, Three Myths of Computer Science, *British Journal for the Philosophy of Science* **29**(3) (1978), 213–222.

[32] B.C. Smith, The Limits of Correctness, *ACM SIGCAS Computers and Society* **14,15** (1985), 18–26.

[33] J.H. Fetzer, Program Verification: The Very Idea, *Communications of the ACM* **31**(9) (1988), 1048–1063.

[34] J.H. Fetzer, Philosophy and Computer Science: Reflections on the Program Verification Debate, in: *The Digital Phoenix: How Computers are Changing Philosophy*, T.W. Bynum and J.H. Moor, eds, Blackwell, 1998, pp. 253–273.

[35] E.G. Daylight, *Turing Tales*, Lonely Scholar, 2016.

[36] E.G. Daylight, Towards a Historical Notion of 'Turing — the Father of Computer Science', *History and Philosophy of Logic* **36**(3) (2015), 205–228.

[37] F. Cohen, Computer Viruses: Theory and Experiments, *Computers and Security* **6**(1) (1987), 22–35.

[38] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979.

[39] E. Filiol, *Computer Viruses: from theory to applications*, Springer, 2005.

[40] M.S. Mahoney, *Histories of Computing*, Harvard University Press, Cambridge, Massachusetts/London, England, 2011.

[41] L. Sassaman, M.L. Patterson, S. Bratus and A. Shubina, The Halting Problem of Network Stack Insecurity, *USENIX; login:* **36**(6) (2011), 22–32.

[42] P. Brödner, Coping with Descartes' error in information systems, *AI & Society* **Published online: 17 January 2018**.

[43] M.C. Reingruber and W.W. Gregory, *The Data Modeling Handbook: A Best-Practice Approach to Building Quality Data Models*, Wiley, 1994.

[44] P.E. Agre, Beyond the Mirror World: Privacy and the Representational Practice of Computing, in: *Technology and Privacy: The New Landscape*, MIT Press, 1997.

[45] S. Bratus, M.E. Locasto, M.L. Patterson, L. Sassaman and A. Shubina, Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation, *USENIX; login:* **36**(6) (2011), 13–21.

[46] M. Bullynck, E.G. Daylight and L.D. Mol, Why Did Computer Science Make a Hero out of Turing?, *Communications of the ACM* **58**(3) (2015), 37–39.
[47] L. De Mol, Turing Machines, *The Stanford Encyclopedia of Philosophy* **2018 Edition** (2018), plato.stanford.edu/entries/turing-machine/.
[48] R.W. Floyd, On ambiguity in phrase structure languages, *Communications of the ACM* **5** (1962), 526, 534–.
[49] L.A. Suchman, *Human-Machine Reconfigurations: Plans and Situated Actions*, 2nd edn, Cambridge University Press, 2007.
[50] C. Strachey, An impossible program, *The Computer Journal* **7**(4) (1965), 313.
[51] M. Davis, *Computability and Unsolvability*, McGraw-Hill, New York, USA, 1958.
[52] B.J. Copeland (ed.), *The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life plus The Secrets of Enigma*, Oxford: Clarendon Press, 2004.
[53] S.C. Kleene, *Introduction to Metamathematics*, Van Nostrand, Princeton, New Jersey, USA, 1952.
[54] J. Stoy, Christopher Strachey and Fundamental Concepts, *Higher-Order and Symbolic Computation* **13** (2000), 115–117.
[55] E.G. Daylight, Dijkstra's Rallying Cry for Generalization: the Advent of the Recursive Procedure, late 1950s – early 1960s, *The Computer Journal* **54**(11) (2011), 1756–1772.
[56] G. Alberts and E.G. Daylight, Universality versus Locality: the Amsterdam Style of ALGOL Implementation, *IEEE Annals of the History of Computing* (2014), 52–63.
[57] G. van den Hove, On the Origin of Recursive Procedures, *The Computer Journal* **58**(11) (2015), 2892–2899.
[58] C. Strachey, Fundamental Concepts in Programming Languages, *Higher-Order and Symbolic Computation* **13**(1–2) (2000), 11–49.
[59] M. Davis, R. Sigal and E.J. Weyuker, *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*, 2nd edn, Morgan Kaufmann, 1994.
[60] W. Dean, Algorithms and the Mathematical Foundations of Computer Science, in: *Gödel's Disjunction*, 1st edn, L. Horsten and P. Welch, eds, Oxford University Press, 2016.
[61] O. Shagrir, Effective Computation by Humans and Machines, *Minds and Machines* **12** (2002), 221–240.
[62] R. Turner, Programming Languages as Technical Artefacts, *Philosophy and Technology* **27**(3) (2014), 377–397, First online: 13 February 2013.
[63] L. Adleman, An abstract theory of computer viruses, in: *Advances in cryptology – CRYPTO '88*, S. Goldwasser, ed., Springer, 1990, pp. 354–374.
[64] J.-Y. Marion, From Turing machines to computer viruses, *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* **370** (2012), 3319–3339.
[65] J.E. Hopcroft, R. Motwani and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley / Pearson Education, 2007.

"I am delighted to find that some useful theorems are at last emerging in the field on programming language theory."

— Christopher Strachey in 1971

"We are captured by a historic tradition that sees programs as mathematical functions and programming as the central practice for translating these functions into working systems."

— Peter J. Denning in 2004