What is Theoretical Computer Science? (Preliminary Version)

Luca Aceto[*][†] Anna Ingolfsdottir[*][‡]

Abstract

This article presents a bird's eye view of Theoretical Computer Science aimed at a general scientific audience. It then focuses on a selected area within this general field, and discusses some of the applications of results from that area and of the fundamental theoretical questions that drive its development.

1 Introduction and Overview

Even for many scientifically cultured people, the mention of the expression "Theoretical Computer Science" (henceforth abbreviated to TCS) sounds like a contradiction in terms. Indeed, there seems to be little that is as practical as, and less theoretical than, Computer Science.

We all have some level of familiarity with computers, and our daily experience shows us how these machines have dramatically changed, and most likely will continue changing, many aspects of our lives. Moreover, embedded computing devices permeate our world to a degree that is sometimes not appreciated by the layman, since these systems and the software that increasingly provides their core functionalities are, as the name suggests, embedded in physical devices, and are therefore invisible to their users.

Computer Science artifacts are thus the heralds of a pervasive technology, and computing is perceived precisely as a *technology* that is responsible for groundbreaking advances in the way we handle information and interact with one another. This is all very well and is a great advertisement for our subject. However, these

[*]BRICS, Department of Computer Science, Aalborg University, 9220 Aalborg Ø, Denmark. Email: luca@cs.aau.dk (Luca Aceto) and annai@cs.aau.dk (Anna Ingolfsdottir).

[†]School of Computer Science, Reykjavík University, Iceland. Email: luca@ru.is.

[‡]Department of Computer Science, University of Iceland, Iceland. Email: annaing@hi.is (Anna Ingolfsdottir).

technological advances and the information technology revolution are hiding from the general educated public the role that the *scientific foundations* of Computer Science have played in making them possible. Moreover, it is one of our tenets that, like astronomy, biology, mathematics, physics and the other time-honoured areas of scientific endeavour, the *science* of computing has some basic foundational questions that are worth investigating in their own right, regardless of their potential technological impact. One of the great mysteries of basic science is that answers to these fundamental questions often turn out to have huge practical impact in the most unforeseen situations. As we shall see in this essay, TCS is no exception.

But, what is TCS, you may ask? As stated by Oded Goldreich and Avi Wigderson in their eloquent essay [16], TCS is the fundamental scientific discipline that aims at "understanding general properties of computing, be it natural, man-made, or imaginary." Most likely, you are using man-made computing devices daily—be they computers, PDAs, game consoles etc. However, it is becoming increasingly clear that Nature itself computes, and it is not uncommon these days to see, e.g., theoretical physicists discuss the information content of black holes or of the universe itself (see, e.g., [5]) or the feasibility and power of quantum computation or other computing principles from the physical world (see, e.g., [1, 40, 42]). Moreover, researchers in the developing subject of system biology use computational methods to understand the behaviour of living matter, and even a commercial software giant like Microsoft is contributing to these investigations [15]. As for imaginary computing devices, one should look no further than Alan Turing's analysis of the notion of algorithmic computation that led to the development of the so-called *Turing machines* and ultimately to the universal computer as we know it. (See Martin Davis' book [10] for a highly readable and riveting account of the historical development of the ideas that led to the universal computer, stressing the role that logic and logicians have played in it.)

In all of the aforementioned cases, a general pattern of scientific investigation that underlies all of the research in TCS emerges clearly. Research in TCS often starts from the desire to understand the properties of some notion of computation. In particular, we are interested in characterizing what algorithmic problems can be solved, in theory or in practice, using the chosen notion of computation. An *algorithmic problem* is described by specifying its collection of legal inputs, and its expected outputs as a function of the legal inputs. (In the main body of this essay, we shall see, however, that a large part of computation cannot be readily understood as producing outputs from inputs. See Section 2 to follow.) Having identified the type of computation to be studied, researchers in TCS abstract its essence in a *mathematical model of computation* that suppresses the low-level details of the object or artifact being studied, but still captures its essential features. This abstraction step is common to all forms of theoretical scientific investigation, and, as argued by Christos Padimitriou in his essay [30], is, in fact, *inevitable* in Computer Science because the high-level behaviour of the computer is its only aspect that we can hope to observe directly. Examples of classic models of computation that have had a long-lasting impact on the theory and practice of Computer Science are Turing machines, automata on finite and infinite words, (labelled) transition systems, parallel random access machines, rewriting systems and the lambda-calculus to mention but a few. (See, e.g., [3, 25, 34, 35, 36] for information on these models of computation.)

Having identified a suitable abstract model for the computational phenomena under investigation, researchers in TCS will then use it to answer questions related to its computational power and its limitations. Possibly the earliest and most classic example of the effectiveness of this approach was given by Turing's negative solution to Hilbert's *Entscheidungsproblem* [38]. In the process of obtaining this groundbreaking result, Turing showed that the so-called HALTING PROBLEM, namely the problem of determining whether the computation of an input Turing machine M terminates on some input datum d, is algorithmically unsolvable. This celebrated theorem of Turing's was one of the first of a plethora of negative results that sharply define the power of the algorithmic method, and are one of the cornerstones of TCS. (See, e.g., David Harel's book [20] for a beautiful and very accessible exposition of results from the theory of computation pertaining to the inherent limitations of algorithmic problem solving.) Indeed, as argued by Papadimitriou in [30], together with its forefather Mathematical Logic, TCS is pretty much unique in its development of mathematical techniques for proving negative results, and we believe that this is one of the main intellectual achievements of our field.

Further examples of some of the most basic questions that researchers in TCS tackle in their work are:

- What is the nature of efficient computation? Does the notion of efficient computation depend on the model of computation?
- How many computational resources (e.g., time and/or space) does it take solve a specific algorithmic problem?
- Is it possible to exchange information so that eavesdroppers cannot access it?
- What are appropriate models of computation that are sufficiently expressive, and yet amenable to algorithmic analysis?

- When is a computing system correct? Is it possible to establish mathematically that a system does what it is supposed to?

This is a just a small, but hopefully representative, sample of the key questions that drive the developments in TCS. A more extensive overview of the many subfields of investigation that belong to TCS at large can be gleaned by looking at the contents of the highly influential two-volume *Handbook of Theoretical Computer Science* [26]. Volume A of that opus is devoted to areas of investigation that roughly belong to the general theme of *Algorithms and Complexity*. Examples are computational complexity classes like P and NP, Kolmogorov complexity, algorithms on graphs and strings, data structures, parallel algorithms and architectures, and cryptography. At the risk of over-generalizing, it seems fair to say that the flagship question in the "Volume A camp" of TCS, and possibly for TCS as a whole, is whether P equals NP, i.e., whether, as real-life experience seems to suggest, it is easier to verify the correctness of a purported solution to an algorithmic problem than to find the solution itself. At least, this is certainly the Computer Science problem that has the widest exposure in the scientific literature at large—see, e.g., [23]—, and, as argued by Stephen Cook in [9], the practical consequences of a positive solution to it would be stunning.

Volume B of the handbook is instead devoted to the general theme of *Formal Models and Semantics*. Sample topics that belong to this theme are models of computation like automata on finite and infinite strings, rewriting systems and the lambda-calculus, formal languages, type systems for programming languages, database theory, logic in Computer Science and models of concurrent and distributed computing. The emphasis here is on languages and models for describing computational phenomena and their semantics, with logic playing an extremely important unifying role in many of the investigations. (See, e.g., the excellent survey [18] for a discussion of the usefulness of logic in Computer Science.)

Since the publication of [26], areas like *Computational Learning Theory* [24] have become more and more important, and there has been a resurgence of interest in classic fields like coding and information theory that lie at the boundary between (discrete) mathematics and TCS. (See, e.g., [37] for a survey of the connections between coding theory and computational complexity.) Another key development has been the realization of the importance and of the power of *randomness* in computation. This has revolutionized the theory of algorithms, and TCS researchers have shown that, in many cases, probabilistic algorithms and protocols can achieve goals which are impossible deterministically. In other cases, randomized algorithms enable much more efficient solutions than deterministic ones. (See, e.g., [28] for a textbook presentation of the field of randomized algorithms.)

Particularly pleasing for TCS buffs has also been the increasing number of applications of models and ideas from the theory of computation to other sciences. Indeed, the ideas of TCS have been, and are being, exported to fields as disparate as statistical physics, economics, biology and quantum physics. As a single example, let us mention the way in which models and ideas from TCS have been used by Leslie Valiant in his book [39] to propose a neuroidal model of some brain activities like memorization and learning that seems to explain how our minds go about achieving some of their basic tasks, and whose predictions are being confirmed by the neurosciences.

Our goal in the remainder of this article is to focus the above general discussion by presenting a selected sub-field within TCS, some of the applications of results from that field and some of the fundamental theoretical questions that drive its development. In particular, we shall offer a bird's eye view of some aspects of concurrency theory—the branch of TCS that deals with the development of models and specification languages that can be used to describe, and reason about, systems consisting of a collection of interacting processes. We state at the outset that this is a biased choice of sample topic for an introductory essay on TCS, and cannot be considered canonical in any way. It is just a reflection of some of our own research interests, and gives us the opportunity to discuss how beautiful theory developed within the TCS community can and does have practical impact.

We hope that this piece will entice our readers to explore TCS as a fascinating area of scientific endeavour. The effort we have put in writing it will be amply re-payed if our readers will think of computing not just as an amazing technology, but also as a worthwhile scientific endeavour on a par with the classic sciences.

2 Concurrency Theory

As mentioned in the previous section, the "standard" view of computing systems is that, at a high level of abstraction, these may be considered as black boxes that take inputs and provide appropriate outputs. This view agrees with the description of algorithmic problems. Recall that an *algorithmic problem* is specified by giving its collection of legal inputs, and, for each legal input, its expected output. An abstract view of a computing system may therefore be given by describing how it transforms an initial input to a final output.

In this view of computing systems, non-termination is a highly undesirable phenomenon. An algorithm that fails to terminate on some inputs is not one the users of a computing system would expect to have to use. A moment of reflection, however, should make us realize that we already use many computing systems whose behaviour cannot be readily described as a transformation from inputs to outputs—not least because, at some level of abstraction, these systems are inherently meant to be non-terminating. Examples of such computing systems are:

- operating systems,
- communication protocols,
- control programs and
- software running in embedded system devices like mobile telephones.

At a high level of abstraction, the behaviour of a control program can be seen to be governed by the following pseudo-code algorithm skeleton

```
loop
   read the sensors' values at regular intervals
   depending on the sensors' values trigger the relevant actuators
forever
```

The aforementioned examples, and many others, are examples of computing systems that interact with their environment by exchanging information with it. Like the neurons in a human brain, these systems react to stimuli from their computing environment (in the example control program above these are variations in the values of the sensors) by possibly changing their state or mode of computation, and in turn influence their environment by sending back some signals to it, or initiating some operations whose effect it is to affect the computing environment (this is the role played by the actuators in the example control program). David Harel and Amir Pnueli coined the term *reactive system* in [19] to describe a system that, like the aforementioned ones, computes by reacting to stimuli from its environment.

As the above examples and discussion indicate, reactive systems are inherently parallel systems, and a key role in their behaviour is played by communication and interaction with their computing environment. A "standard" computing system can also be viewed as a reactive system in which interaction with the environment only takes place at the beginning of the computation (when inputs are fed to the computing device) and at the end (when the output is received). On the other hand, all the example systems given before maintain a continuous interaction with their environment, and we may think of both the computing system and its environment as parallel processes that communicate one with the other. (A process is "something going on; a series of actions or operations conducing to an end" [29].) In addition, unlike with "standard" computing systems, as again nicely exemplified by the skeleton of a control program given above, non-termination is a *desirable* feature of some reactive systems. We certainly do *not* expect the operating systems running on our computers or the control program monitoring a nuclear reactor to terminate!

Concurrency theory is the branch of TCS whose aim is to develop a general purpose theory that can be used to describe, and reason about, *any* collection of interacting processes. The theory of concurrency offers

- mathematical models for the description of the behaviour of collections of interacting processes that may compute independently and/or communicate with one another and

- formal languages for expressing their intended behaviour.

These ingredients give the foundations for the development of (semi-)automatic verification tools for reactive systems that support various formal methods for validation and verification that can be applied to the analysis of highly non-trivial computing systems. The development of these tools requires in turn advances in algorithmics, and via the increasing complexity of the analyzed designs feeds back to the theory development phase by suggesting the invention of new languages and models for the description of reactive systems.

As recent advances in algorithmic verification and applications of model checking [7] have shown, the tools and ideas of concurrency theory can be used to analyze designs of considerable complexity that, until a few years ago, were thought to be intractable using formal analysis and modelling tools. (Indeed, companies such as AT&T, Cadence, Fujitsu, HP, IBM, Intel, Motorola, NEC, Siemens and Sun are using these tools increasingly on their own designs to reduce time to market and ensure product quality.) A discussion of the details of these non-trivial applications of concurrency theory is beyond the scope of this survey. However, we should like to mention the general principles underlying modelling and verification of reactive systems, and to give a few, undoubtedly biased, pointers to some of their most interesting applications to date.

In Computer Science we build artifacts—implemented in hardware, software or, as is the case in the fast-growing area of embedded and interactive systems, using a combination of both—that are supposed to offer some well defined services to their users. Since these computing systems are deployed in very large numbers, and often control crucial, if not safety critical, industrial processes, it is vital that they correctly implement the specification of their intended behaviour. The problem of ascertaining whether a computing system does indeed offer the behaviour described by its specification is called the *correctness problem*, and is one of the most fundamental problems in Computer Science. The field of Computer Science that studies languages for the description of (models of) computer systems and their specifications, and (possibly automated) methods for establishing the correctness of systems with respect to their specifications is called *algorithmic verification*.

In *algorithmic verification* one abstracts the significant details of the design to be analyzed using a mathematical model. Typical models used in concurrency theory are based on variations on the classic notion of automaton. An *automaton* describes a system by listing its collection of possible *states of computation*—abstract representations of the relevant effect that the computation carried out so far has had on the system—together with the way that states change as the system performs basic computational actions.

In one approach to the correctness problem, automata are used not only to describe actual systems, but also their intended behaviour. An automaton IMP correctly implements the behaviour described by another automaton SPEC if the two automata describe essentially the same behaviour, but at different levels of abstraction or refinement. In this approach, the correctness problem is formalized by the mathematical problem of checking whether two automata are, in some well defined sense, "equivalent" or whether one is "a suitable approximation" of the other. Efficient algorithms and proof methodologies are often available to settle this question with the help of computer support. For instance, this approach has been used in [14] by Wan Fokkink, Jan Friso Groote and their research groups to analyze a well known sliding window protocol.

Another, very successful approach to the aforementioned correctness problem is *model checking*. In this approach, a computing system is again modelled as a collection of interacting automata—describing the states of the system's components and the effect that computation steps have on them—, whereas (un)desired properties of the system are expressed using some form of logic [11, 31]. Typical logics that are used to describe properties of systems allow one to express constraints like

"*It is always the case that each sent message is eventually delivered*,"

or

"*The lift system never ends up in a state where it cannot perform any action.*"

Given a collection of interacting automata and a logical formula, the model checking problem asks whether the system described by the automata is a model of the formula—that is, whether the system affords the property described formally by the logical formula. Moreover, if the answer to this question is negative, it is useful for debugging purposes to generate some diagnostic information expressing why the system does *not* enjoy the desired property.

Regardless of the logic one uses to describe correctness criteria for systems modelled as interacting automata, the key to the popularity and wide applicability of model checking is that algorithms for model checking are now implemented in efficient, fully automatic verification tools that can be used to check the correctness of very complex designs. Tools like COSPAN [13], nuSMV (see http://nusmv.irst.itc.it/)—a reimplementation and extension of SMV [27], the first model checker based on Binary Decision Diagrams [6]—, SPIN [22] (see http://spinroot.com/spin/whatispin.html) and UPPAAL [4] (see http://www.uppaal.com/) are examples of industrial strength model checkers.

A paradigmatic example that highlights the usefulness of model checking in discovering errors in system designs that are due to rarely occurring, or very complex, interactions is the analysis of a Bang & Olufsen audio/video protocol carried out in [21]. The protocol was developed by Bang & Olufsen to transmit messages between audio/video components over a single bus. Though it was known to be faulty, the error was not found using conventional testing methods. Using the model checker UPPAAL to analyze a model of the protocol, Havelund et al. were able to synthesize automatically a sequence of interactions between the protocol components that revealed the error. Based on this debugging information, they were then able to suggest a modification of the original protocol design, and prove it correct using UPPAAL.

Another example with a similar flavour is presented in [17]. There the authors formally modelled and analyzed a complex distributed system for lifting trucks. As a result of the formal analysis, four errors were found in the original design. Based upon the knowledge obtained during analysis, the authors proposed solutions for these problems and showed, by means of model checking, that the modified system meets the requirements.

A different application of model checking techniques is presented in [41], where the authors applied the file system model checking tool FiSC to three widely-used, heavily-tested file systems, namely ext3, JFS, and ReiserFS. Possibly surprisingly, the model checker found serious bugs in all of them. For each file system, FiSC found demonstrable events leading to the unrecoverable destruction of entire directories, including the file system root directory!

The latter example is paradigmatic of the current trend to use model checking and other techniques originating from concurrency theory and algorithmic verification to the analysis of systems other than communication protocols, which were the first application area for results from those fields. We expect that such “extroverted” applications will become increasingly prominent in the future. For instance, as witnessed by the developments in [12, 32], model checking tools are finding application in resource optimal scheduling and planning. Together with other formal methods from, amongst others, concurrency theory, model checkers are also currently employed by NASA to produce reliable software systems for use in their deep-space missions [33].

It is not our aim to offer a complete list of model checking tools and their applications. However, we hope that the aforementioned examples of real-life systems that have been analyzed using them will spur our readers to explore their use, and to appreciate their applicability in actual verification and validation tasks. The interested reader will find an informative, albeit somewhat dated, discussion of applications of formal methods in [8]. A gentle introduction to some of the mathematical models and logics used in concurrency theory may be found in the draft textbook [2].

Acknowledgments The first author dedicates this essay to the memory of his father, Trento Aceto (26 September 1915–14 April 2005), who passed away while this piece was being finalized.

We thank Björn Thor Jonsson and Kari Ragnarsson for carefully reading a draft of this article, and offering thoughtful suggestions that led to improvements in the presentation. Any remaining infelicity is solely our responsibility.

References

[1] Scott Aaronson. NP-complete problems and physical reality. *ACM SIGACT News*, 36(1):30–52, March 2005. Available from `http://www.arxiv.org/abs/quant-ph/0502072`.

[2] Luca Aceto, Anna Ingolfsdottir, and Kim Guldstrand Larsen. An introduction to Milner's CCS, 2005. Draft textbook. Available from `http://www.cs.aau.dk/~luca/SV/intro2ccs.pdf`.

[3] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, revised edition, 1984.

[4] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[5] Jacob D. Bekenstein. Information in the holographic universe. *Scientific American*, 289(2):48–55, August 2003.

[6] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[7] Ed Clarke, Orna Gruemberg, and Doron Peled. *Model Checking*. MIT Press, December 1999.

[8] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[9] Stephen Cook. The importance of the P versus NP question. *Journal of the ACM*, 50(1):27–29, 2003.

[10] Martin Davis. *Engines of Logic: Mathematicians and the Origin of the Computer*. W.W. Norton & Company Ltd, September 2001.

[11] E. Allen Emerson. Temporal and modal logic. In *Handbook of theoretical computer science, Vol. B*, pages 995–1072. Elsevier, Amsterdam, 1990.

[12] Ansgar Fehnker. Scheduling a steel plant with timed automata. In *RTCSA*, pages 280–286. IEEE Computer Society, 1999.

[13] Kathi Fisler and Robert P. Kurshan. Verifying VHDL designs with COSPAN. In Thomas Kropf, editor, *Formal Hardware Verification*, volume 1287 of *Lecture Notes in Computer Science*, pages 206–247. Springer, 1997.

[14] Wan Fokkink, Jan Friso Groote, Jun Pang, Bahareh Badban, and Jaco van de Pol. Verifying a sliding window protocol in μCRL. In Charles Rattray, Savi Maharaj, and Carron Shankland, editors, *AMAST*, volume 3116 of *Lecture Notes in Computer Science*, pages 148–163. Springer, 2004.

[15] Microsoft Research Centre for Computational and Systems Biology. `http://dit.unitn.it/~bioinfo/index.php?option=com_content&task=view&id=22&Itemid=51`.

[16] Oded Goldreich and Avi Wigderson. Theory of computation: A scientific perspective, 2001. Available from `http://www.wisdom.weizmann.ac.il/~oded/toc-sp2.html`.

[17] Jan Friso Groote, Jun Pang, and Arno G. Wouters. Analysis of a distributed system for lifting trucks. *J. Log. Algebr. Program.*, 55(1-2):21–56, 2003.

[18] Joseph Y. Halpern, Robert Harper, Neil Immerman, Phokion G. Kolaitis, Moshe Y. Vardi, and Victor Vianu. On the unusual effectiveness of logic in computer science. *Bull. Symbolic Logic*, 7(2):213–236, 2001.

[19] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems (La Colle-sur-Loup, 1984)*, volume 13 of *NATO Adv. Sci. Inst. Ser. F Comput. Systems Sci.*, pages 477–498. Springer-Verlag, Berlin, 1985.

[20] David Harel. *Computers Ltd.: What They Really Can't Do*. Oxford University Press, 2000.

[21] Klaus Havelund, Arne Skou, Kim Guldstrand Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol: an industrial case study using UPPAAL. In *IEEE Real-Time Systems Symposium*, pages 2–13. IEEE Computer Society, 1997.

[22] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.

[23] Clay Mathematical Institute. The millennium problems: P vs NP problem. http://www.claymath.org/millennium/P_vs_NP/.

[24] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, 1994.

[25] R.M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[26] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science*. Elsevier and MIT Press, 1990. Volume A: Algorithms and Complexity. Volume B: Formal Models and Semantics.

[27] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.

[28] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, August 1995.

[29] Merriam-Webster OnLine. http://www.webster.com/.

[30] Christos H. Papadimitriou. Database metatheory: Asking the big queries. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California*, pages 1–10. ACM Press, 1995.

[31] Amir Pnueli. The temporal logic of programs. In *Proceedings 18^{th} Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[32] Jacob Illum Rasmussen, Kim Guldstrand Larsen, and K. Subramani. Resource-optimal scheduling using priced timed automata. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 220–235. Springer, 2004.

[33] Patrick Regan and Scott Hamilton. NASA's mission reliable. *Computer*, 2004. Available from http://www.computer.org/computer/homepage/0104/Regan/.

[34] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[35] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 2003.

[36] Wolfgang Thomas. Automata on infinite objects. In *Handbook of theoretical computer science, Vol. B*, pages 133–191. Elsevier, Amsterdam, 1990.

[37] Luca Trevisan. Some applications of coding theory in computational complexity. *Electronic Colloquium on Computational Complexity (ECCC)*, 043, 2004.

[38] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. Lond. Math. Soc.*, 42(2):230–265, 1937. Available at http://www.abelard.org/turpap2/tp2-ie.asp.

[39] Leslie G. Valiant. *Circuits of the Mind*. Oxford University Press Inc, USA, January 2001. Paperback 254 pages.

[40] Leslie G. Valiant. Three problems in computer science. *Journal of the ACM*, 50(1):96–99, 2003.

[41] Junfeng Yang, Paul Twohey, Dawson R. Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *OSDI*, pages 273–288, 2004.

[42] Andrew Chi-Chih Yao. Classical physics and the Church-Turing thesis. *Journal of the ACM*, 50(1):100–105, 2003.