From: EMPIRICAL FOUNDATIONS OF INFORMATION
AND SOFTWARE SCIENCE
Edited by Jagdish C. Agrawal and Pranas Zunde
(Plenum Publishing Corporation, 1985)

ON THE VALIDATION OF COMPUTER SCIENCE THEORIES

B. J. MacLennan

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Abstract: We address normatively the demarcation problem for Computer Science: How can Computer Science research be conducted scientifically? First we attempt to delimit the subject matter of Computer Science, and conclude that it is not computers but programs. Since programs are not physical objects, it is difficult to see how they can be studied empirically. The rest of the paper is devoted to an explanation of how this can be done. This method is first illustrated by a hypothesis of narrow scope, analogous to a physical law. Next it is illustrated by a theory of wide scope - the Turing Machine model of computers. The approach is summarized in the conclusions.

1. SUMMARY

We claim that the most important theories underlying Computer Science have never been empirically verified. As an example we consider Turing Machine Theory.[1]

It is well known that Turing Machines are used as models of real computers, and that theoretical results about Turing Machines, such as the impossibility of a decision procedure for the Halting Problem,[1,2] are considered valid as assertions about real computers. However, we know that Turing Machines differ from real computers in several significant ways. For example, Turing Machines have a potentially infinite (i.e., finite but unbounded) memory, whereas real computers have a finite, bounded memory. These differences do not, per se, mean that the Turing Machine is an inadequate model of real computers. For example, the potentially infinite memory of the Turing Machine could be considered an idealization or approximation of the large but finite memory of real computers. This leads us to ask a crucial question: Is the Turing Machine an adequate model of real computers?

Since this question asks about the correspondence between an abstract mathematical model, the Turing Machine, and a real-world phenomenon, computers, it can be answered only by an empirical procedure. Our goal is to show how empirical techniques can be used to confirm or refute the Turning Machine model of computers. In the process we demonstrate the application of empirical techniques to the validation of Computer Science theories in general.

The basic approach is an adaptation of the hypothetico-deductive method[3,4,5,6] commonly used in the sciences. That is, we deduce predictions from the hypothesis in question, and then investigate whether these predictions hold in fact. For example, a prediction made by Turing Machine Theory is the impossibility of a decision procedure for the Halting Problem. Does this prediction hold in fact, that is, for real computers.

There are several ways we can test this prediction. We might try to refute it by trying to find a decision procedure for the Halting Problem for real computers. Alternately, we might try to confirm it by showing that the existence of such a decision procedure would contradict other empirically validated laws, such as the conservation of mass-energy. In both cases the results of empirical investigation contribute to the confirmation or refutation of the Turing Machine model of computers. In this paper we explore both possibilities.

It is hoped that the systematic application of empirical techniques such as these will eliminate some of the sterility characteristic of much of the theory of Computer Science.

2. INTRODUCTION

It has been observed that discipines with the word "science" in their names usually aren't. This leads us to pose the demarcation problem[7,8] for Computer Science: Is Computer Science a science? Or is it something else, such as an art, engineering discipline, or pseudoscience? And further, if Computer Science is a science, then what sort of science is it? Is it more like the natural sciences or the mathematical sciences? Or perhaps the engineering sciences or the social sciences are a better model.

When we have answered these descriptive questions, what Computer Science is, we can turn to the normative questions, what Computer Science should be. In particular, should Computer Science be a science? Or should it be an art or engineering discipline? And if it should be a science, then should it be patterned after physics, or mathematics, or linguistics; or should it follow a pattern all its own?

In this paper we attempt to answer some of these questions. In particular, we aim to show the role of empirical methods in Computer Science. We do not claim any originality for the specific results presented herein; they are used solely to illustrate the methodology. However, we do claim that this methodology is distinctly different from that which is usual in Computer Science.

3. SUBJECT MATTER OF COMPUTER SCIENCE

Contrary to its name, the subject matter of Computer Science is not computers, per se; that is more the subject of Computer Engineering, a branch of Electrical Engineering. Computer Science is generally more concerned with software (i.e., programs) than with hardware.[9] That is, it is concerned with the logical rather than the physical properties of computers.

It might be (and has been) contended that Computer Science is an engineering discipline.[9] Certainly many computer scientists, especially in industry, are engaged in the production of hardware/software systems that are intended to satisfy a need. To answer this objection it is useful to distinguish an engineering discipline from a craft or art (in the Τεχνη sense). Crafts and arts tend to be based on informal, often intuitive, knowledge and experience, whereas engineering disciplines are based on an

underlying science. That is, an engineering discipline's knowledge is systematized, formalized, and often quantified. Its hypotheses have been more systematically validated. Of course, we rarely find a discipline that is purely craft or purely engineering - these are the extrema of a continuum, with most disciplines falling inbetween. Progress in a field is often measured by the distance moved from the craft extremum towards the engineering extremum.

Thus, even if Computer Science were an engineering discipline (or even a craft), we would want to seek for an underlying scientific discipline. What we address in this paper is an empirical approach to this underlying science, that is, to the scientific study of software and of the logical properties of hardware.

4. DEMARCATION

Is Computer Science a science? Our interest in the demarcation problem[7,8] for Computer Science is not "name calling;" rather, it is constructive. If Computer Science is not a science, we want to determine how to make it one; if it is a science, we want to reinforce and extend its use of scientific method.

How can we determine if Computer Science is a science? First, we can ask whether its theories and laws are scientific in form, that is empirically validatable. There are many criteria for deciding this, such as empirical content, operational definability,[10,11,12] verifiability[13] and falsifiability.[7,8,14] Second, we can ask whether the theories and laws of Computer Science have in fact been empirically validated. We will attempt to answer these questions by investigating several theories and laws of Computer Science.

What would constitute a theory of Computer Science? When people talk about Computer Science theories they usually mean computability theory, formal language theory, automata theory, complexity theory, formal logic and proof theory, parsing theory, programming language semantics, numerical analysis, etc. Since these can all be considered mathematical theories, we are led to a common view: that Computer Science is a branch of mathematics.[9] The result of this view has been that the standard of validation for these theories has generally been that of mathematics, viz. deductive consistency. However, if we wish to use computer science as a foundation for software engineering, then this standard will not do. For this purpose we need to know that these theories apply to real computers and real software, that their assertions correspond to reality. Thus, the mathematical validity of these theories is not sufficient; we must also consider their empirical validity.

Have any Computer Science theories been validated empirically? Unfortunately, we have to answer in the negative. Of course, many of them have been validated informally, in the sense that practical systems based on them work. However, to the best of our knowledge, no one has actually attempted a systematic empirical validation of any of these theories.* In many cases this validation would not be too difficult, since the empirical data are already available. What is required is a demonstration of how these data confirm or refute the theories. In the following sections we outline two of these demonstrations.

*Some of these theories have come closer to empirical validation than others. For example, when a particular numerical algorithm behaves as predicted, it indirectly validates the numerical analysis upon which it is based.

5. EMPIRICAL VALIDATION OF COMPUTER SCIENCE HYPOTHESES

As an introduction to the empirical approach we consider the validation of a very specific hypothesis, a hypothesis that corresponds in scope to a physical law. Suppose our hypothesis is that a particular kind of sorting algorithm takes n log n time (where n is the number of items to be sorted). How can we validate this hypothesis?

A hypothesis of this sort is based on many approximations and idealizations which are conventionally assumed to be true. This hypothesis, and the assumptions on which it is based, can be tested by an experiment analogous to those performed by physical scientists. We will implement the algorithm and measure the time it takes to execute, while varying the conditions we believe to be relevant (essentially applying Mill's Method of Concomitant Variations).[15] In this case some of the potentially relevant conditions are:

- The number of items to be sorted (to determine the fixed overhead of the algorithm;
- The initial ordering of the items (to determine their sensitivity to the initial order, and form a basis for statistical analysis);
- The computer and/or programming language used for implementing the algorithm (to ensure that these aren't relevant factors as a result of, e.g., fixed overhead, special optimizations, memory collisions).

This approach has several benefits. First, it serves to validate the hypothesis. Second, it simultaneously helps to validate the assumptions upon which the analysis was based. Third, it gives us guidance in the practical application of these analytical techniques.

Computer Science hypotheses of this kind are sometimes validated in precisely this manner. However, more commonly computer scientists fall into one of two categories: practitioners, who make measurements without any underlying theory, and theoreticians, who don't test their theoretical analyses, because they have proved them mathematically. On one hand, the practitioners fail to develop useful laws and theories that would help them to predict the performance of future software. On the other hand, the theoreticians ignore at least two ways in which they could be wrong: (1) they might have made a mistake in their mathematical analysis; and (2) their analysis might not be applicable, that is, they might have ignored what was not negligible. As in the other sciences, experimentation guided by theory seems the more reliable method.

6. EMPIRICAL VALIDATION OF TURING MACHINE THEORY

The performance of a particular algorithm, such as a sorting algorithm, is a very narrow kind of hypothesis. Are there Computer Science hypotheses of wider scope, comparable to the laws and theories of physics and chemistry? Many of the important theoretical results of Computer Science, such as the computability and uncomputability results, are based on idealized models of computation such as the Turing Machine[1] and the lambda calculus.[16] The applicability of these theoretical results to real computers depends on the extent to which these idealized models are accurate descriptions of real computers. We can state this hypothesis* more formally:

*This hypothesis is related to Church's Thesis (really, Hypothesis),[16] which states that Turing computability is a good model of effective computability.

The Computer = Turing Machine Hypothesis

The Turing machine is a good model of real computers.

The statement of this hypothesis is intentionally vague, since research is required to determine its limitations and the extent of its applicability. That is, we want to know to what degree and in what ways the Turing machine is a good model of real computers. Since many of the theoretical results of Computer Science are based on the Turing machine model, much of the applicability of this theory to practical problems depends on the truth and limitations of the above hypothesis. Thus, the investigation and validation of this hypothesis should be an important problem for the computer scientist.

How can we confirm or refute this hypothesis? At first inspection it would seem that this hypothesis could not possibly be true. The Turing machine is defined to have a finite but unbounded (i.e., potentially infinite) memory - something possessed by no real computer. However, we can consider the unboundedness of the Turing machine's memory to be an idealization (i.e., approximation) of the large memory capacity of real computers. It is analogous to the physicist's use in analysis of an infinitely long, infinitely thin wire, or the chemist's use of an infinitely divisible gas. The question is, Is this a good approximation? One way to answer this question (proceeding hypothetico-deductively)[3,4,5,6] is to consider various predictions made by the Turing machine model, and to ask whether they are true of real computers.

One of the most important predictions of the Turing machine model is the Halting Theorem,[1] which we now explain. The Halting Problem for a particular Turing machine and a particular input tape is the problem of deciding whether that Turing machine halts (i.e., produces an answer) when given that tape as input. A decision procedure for the Halting Problem is a Turing machine that will decide the Halting Problem for any given Turing machine/input tape pair. The Halting Theorem states that there is no decision procedure for the Halting Problem. That is, there is no Turing machine that will decide, for an arbitrarily given Turing machine and input tape, whether the given Turing machine will halt when run on the given input tape. This result has considerable practical importance when applied to real computers, for it says that we can never write a program that will decide whether another given program will produce an output when run on a given input. An extension of the theorem says that most interesting properties of programs are algorithmically undecidable.

The proof of the Halting Theory proceeds in much the same way as Gödel's proof[17] of his famous incompleteness theory: we assume the decision procedure exists and use it to construct a paradoxical self-referential Turing machine, which leads to a contradiction.[2] The contradiction forces us to reject our assumption of the existence of a decision procedure for the Halting Problem. Indeed, the same technique works to show the nonexistence of a decision procedure for most any property of interest of Turing machines. Thus, the applicability of the Turing machine model is a crucial question. The nonexistence of these decision procedures are analogous in importance to the physical results that assert the nonexistence of perpetual motion machines.

There are several methods for validating empirically a hypothesis that asserts nonexistence. One method is to conduct a scientific search[18] for the thing in question, in this case, a decision procedure for the Halting Problem. Such searches, which are common in scientific investigation, are effective to the extent that we can enumerate the possible "places" where the sought object could be found, and then explore those "places". Since every computer scientist "knows" that a decision procedure for the Halting

Problem is impossible, few have ever looked for one. Thus this approach to validation of the Turing machine model has not been seriously pursued. It would probably not be very fruitful, anyway, due to the large, irregular, multidimensional space that would have to be searched.

Another method for validating a nonexistence hypothesis is to show that the existence of the thing in question would contradict other empirically validated hypotheses. That is, we can show that either we must accept the nonexistence of the thing in question, or we must give up other (presumably more strongly held) beliefs. That is, we accept the nonexistence results because to deny it would require us to reject other hypotheses, and in turn find new explanations for the evidence by which those other hypotheses had been validated. This, of course, is a common process in science. It seems a more promising approach to validating the Halting Theorem. Our problem is to show that the existence of a decision procedure would contradict other known laws.

One of the most obvious respects in which the Turing machine model differs from real computers is that the Turing machine has a potentially infinite memory, whereas real computers don't. Is this a significant difference? We can find out in exactly the same way a natural scientist would: alter the property in question, i.e., apply Mill's Method of Differences.[15] In this case the property we are altering is the finiteness of the Turing machine's memory. We can then ask whether properties of interest, such as the Halting Theorem, still hold under the conditions of a very large but finite memory. If they do, then we have justified our use of the Turing machine approximation of real computers. However, if these properties are sensitive to this alteration of property, then we must consider the possibility that the Turing machine model neglects significant characteristics of real computers.

Hence we will formulate a bounded memory version of the Halting Problem. Consider the class of all program-input pairs for some particular computer with a finite, bounded memory. Since both the program and its input must fit in this bounded computer, the size of these program-input pairs is bounded. Hence, the program-input pairs are expressed in a finite, bounded alphabet (usually '0' and '1'), the number of program-input pairs is a fixed, finite number; call it N. We are seeking a program H for a bounded computer that will decide the Halting Problem for each of these program-input pairs. That is, for each such program-input pair, H will decide whether that program will halt when given that input. Since H runs on a finite, bounded computer, it has a finite, bounded amount of memory available for its computations. Does such an H exist?

The answer is clearly "yes": an N-entry table can be used to decide the Halting Problem by looking up a given program-input pair. For each program-input pair we have a one-bit entry in the table, the bit indicating that the pair does or doesn't halt. In addition we have a small fixed number P of bits containing the program to do the lookup. Hence the size of H is N + P bits.

This solution to the Halting Problem for finite, bounded computers is not practical, as we can see by considering the size of H. Suppose our program-input pairs are all bounded by L (i.e., are at most L bit long). Then the number of these pairs is:

$$N = 2^L + 2^{L-1} + 2^{L-2} + \ldots + 2^1 + 2^0 = 2^{L+1} - 1$$

The program H requires N bits in its table. Hence, the size of H is an exponential function of the maximum size of the program-input pairs that it decides. For example, to decide the program-input pairs that would fit in a

16 kilobyte $\simeq 10^5$ bit personal computer would take a table of size

$$2^{(10^5)} = 10^{10^5 \log 2} \approx 10^{30103} \text{ bits}$$

This is more than the estimated number of particles in the universe. Hence, the solution of the Halting Problem by this method is impossible for even relatively small programs.

This leads to an obvious question: Is there a more efficient decision procedure for the finite, bounded Halting Problem? Thus we must seek lower bounds on the size of the decision procedure. It is easy to see that a lower bound on the length of the decision procedure must be close to L. For, if the length of H is even a little less than L, then we can write a program Q shorter than L that "halts if and only if it doesn't halt" (in the same way that this is done in Gödel's and Turing's proofs).[2] Since this is a contradiction, we must conclude either:

1. The computer lacks the necessary instructions to build Q from H; or

2. The program Q is longer than L, and hence won't fit in the computer for which we have a decision procedure.

Now we appeal to observation: all real computers* have the instructions necessary to build Q from H (they are very simple). Hence, (1) is empirically refuted. Therefore, we must conclude (2): the length of Q is greater than L. But, it can also be shown empirically that Q is just a little longer than H, so we can conclude that a lower bound on the length of H is a number nearly as big as L. That is, H is almost too big to fit on the computer in question. Thus we have two (very loose) bounds on the size of the decision procedure. They can probably be tightened, but we do not know if this has been done.

What do these results say about the empirical validity of the Halting Theorem? We have found the following results by combining theoretical analysis with observation:

- Certain finite, bounded computers can decide the Halting Problem for other finite, bounded computers.

- If the decision procedure can fit in the object computer at all, then it occupies most of the object computer's memory.

Tighter bounds on the size of the decision procedure probably exist, but we are not aware of them. In any case, they are not relevant to our purposes here, since we are concerned with the method, not the particular results. Thus we consider the conclusions we would draw in each of two circumstances:

1. Suppose it were found that the lower bound on the size of H is close to an exponential function of L. That is, it takes a very large computer to decide the Halting Program for a much smaller computer.

   In this case we would be justified in saying that the Turing machine model is a good approximation to real computers (at least with regard to the Halting Theorem). This is because, although

*As is often the case when we formalize previously informal concepts, there is an apparent circularity in the definition of the concept. In this case, we would not call a device a computer if it did not contain the requisite instructions.

the Halting Problem is decidable in a nonempirical sense, it is not decidable in fact, that is, in the real world. More precisely, it is not decidable for any but the smallest computers.*

2. Suppose it were found that to solve the Halting Problem for a given object computer it takes a computer of comparable size to the object computer. For example, a size 2L computer might be adequate to solve the Halting Problem for a size L computer.

   In this case, since the Halting Problem can be solved with an only moderately larger computer, the Turing machine model will have been called into question. This reason is that the Turing machine model ignores the very characteristic of real computers that is relevant - their finite memory.

In either case our empirical investigation has had two important benefits: (1) It has given us more confidence in the applicability of our theory to the real world; and (2) It has given us greater insights into the reasons that an important result (i.e., the Halting Theorem) holds.

7. CONCLUSIONS

Although Computer Science has a rich and well developed theory, there have been few attempts to show empirically that this theory applies to real computers and real programs. This lack can perhaps be attributed to the gap that separates Computer Science Theory from Computer Science practice.

We have attempted to show how empirical methods can be used to validate both specific laws (e.g., the performance of a particular algorithm) and general theories (e.g., Turing machine theory). As in the other sciences, a primary method is the testing of predictions by actual measurements. A more important method, at least at this stage of development of Computer Science, is conceptual validation, that is, the validation of scientific formalizations (explications) of information ideas.[19,20] To illustrate this we have investigated the validation of Turing machines as formal models of real computers. We have shown that observations can be used to confirm or refute this model.

We hope that wider practice of this approach will make moot the question of whether Computer Science is a science.

ACKNOWLEDGMENTS

The work reported herein was supported by the Office of Naval Research under contract number N00014-84-WR-24087.

REFERENCES

1. Alan M. Turing, On computable numbers, with an application to the

*The reader might question the usefulness and precision of a model of computers that doesn't apply to "small" computers. How big does a computer have to be for the model to apply? This situation is not unusual in science. For example, Charles's and Boyle's Laws do not apply to "very small" volumes of gases; statistical mechanics does not apply to "small" numbers of particles.

Entscheidungsproblem, Proc. London Math. Soc., 1936-7, 42 (2), pp. 230-265; 1937 43 (2), pp. 544-546.

2. Bruce J. MacLennan, A Computer Science Version of Gödel's Theory, Naval Postgraduate School Computer Science Department Technical Report NPS52-83-010, 1983.
3. R. B. Braithwaite, Scientific Explanation, Cambridge Univ. Press, Cambridge, Mass., 1953.
4. Pierre Duhem, The Aim and Structure of Physical Theory, P. P. Wiener, trans., Princeton, 1954.
5. Carl G. Hempel and Paul Oppenheim, Studies in the logic of explanation, Phil. of Science, 1948, 15, pp. 135-175; also in reference 15.
6. Carl G. Hempel, Aspects of Scientific Explanation, The Free Press, New York, 1965.
7. Karl R. Popper, Conjectures and Refutations, Harper & Row, New York, 1963.
8. Karl R. Popper, The Logic of Scientific Discovery, Harper & Row, New York, 1968.
9. S. Amarel, Computer science, Encyc. Computer Science, First Edition, A. Ralston and C. L. Meel, eds., Petrocelli/Charter, New York, 1976, p. 316.
10. Carl G. Hempel, A logical appraisal of operationism, The Validation of Scientific Theories, Philipp G. Frank, ed., Beacon Press, Boston, 1956.
11. Henry Margenau, Interpretations and misinterpretations in operationalism, The Validation of Scientific Theories, Philipp G. Frank, ed., Beacon Press, Boston, 1956.
12. Percy W. Bridgman, The Logic of Modern Physics, Second Edition, New York, 1948.
13. Karl Pearson, The Grammar of Science, Walter Scott, London; and Charles Scribner's Sons, New York, 1892.
14. Imre Lakatos, The problem of appraising scientific theories: three approaches, Mathematics, Science and Epistemology, J. Worrall and G. Currie, eds., Cambridge Univ. Press, Cambridge, 1978.
15. John S. Mill, A System of Logic, Ratiocinative and Inductive, Being a Connected View of the Principles of Evidence and the Methods of Scientific Investigation, Eighth Edition, Longmans, Green, and Co., London, 1843.
16. Alonzo Church, An unsolvable oroblem of elementary number theory, Amer. J. Math., 1936, 58, pp. 345-363.
17. Kurt Godel, On formally undecidable propositions of principia mathematica and related systems I, The Undecidable, Martin Davis, ed., E. Mendelson, Trans., Raven Press, Hewlett, New York, 1965.
18. E. Bright Wilson Jr., An Introduction to Scientific Research, McGraw-Hill, New York, 1957.
19. Bruce J. MacLennan, A Commentary on Mill's Logic Book I: Of Names and Propositions, Naval Postgraduate School Computer Science Department Technical Report NPS52-83-013, 1983.
20. Rudolf Carnap, Logical Foundations of Probability, Second Edition, Univ. of Chicago Press, Chicago, 1962, pp. 1-18.