Basic Science for Software Developers

David Lorge Parnas and Michael Soltys

July 18, 2006

1 Introduction

Every Engineer must understand the properties of the materials that they use. Whether it be concrete, steel, or electronic components, the materials available are limited in their capabilities and an Engineer cannot be sure that a product is "fit for use" unless those limitations are known and have been taken into consideration. The properties of physical products can be divided into two classes: (1) *technological properties*, such as rigidity, which apply to specific products and will change with new developments, (2) *fundamental properties*, such as Maxwell's laws or Newton's laws, which will not change with improved technology.

In many cases technological properties are expressed in terms of numerical parameters and the parameter values appear in product descriptions. This makes these limitations concrete and meaningful to pragmatic developers. It is the responsibility of engineering educators to make sure that students understand the technological properties, know how to express them, know how to determine them for any specific product, and know how to take them into account when designing or evaluating a product.

However, it is also the responsibility of educators to make sure that students understand the fundamental limitations of the materials that they use. It is for this reason, that accredited engineering programs are required to include a specified amount of basic science (see [9]). Explaining the relevance of basic science to Engineers is a difficult job; technological limitations are used to compare products; in contrast, fundamental limitations are never mentioned in comparisons because they apply to all competing products. As a result, the technological limitations seem more real and students do not perceive fundamental limitations as relevant.

For Software Engineers, the materials used for construction are computers and software. In this area too, the limitations can also be divided into two classes: (1) *technological limitations*, such as memory capacity, processor speed, word length, types of bus connections, precision obtained by a specific program, availability of specific software packages, etc., (2) *fundamental limitations*, such as limits on computability, complexity of problems, and the inevitability of noise in data.

Computer Scientists have developed a variety of useful models that allow us to classify problems and determine which problems can be solved by specific classes of computing devices.

The most limited class of machine is the finite state machine. Finite state machines can be enhanced by adding a "last-in-first-out" memory known as a stack. Adding an infinitely

extensible tape that can move both forwards and backwards through the reader/writer makes the machine more powerful (in an important sense) than any computer that can actually be built. Practicing software developers can use these models to determine how to approach a problem. For example, there are many problems that can be solved completely with the simplest model, but others must be restricted before they can be solved. Many people know these things in theory, but most do not understand how to use the theory in practice.

Like the students in other engineering disciplines, software engineering students must be able to understand and deal with technological limitations. Even the youngest have seen rapid improvements in technology and most of them easily understand the practical implications of those differences.

It is not useful to spend a lot of time on the technological limitations of specific current products. Much of what students learn about products will be irrelevant before they graduate. However, it is very important to teach the full meaning of technological parameters and how to determine which products will be appropriate for a given application.

Nonetheless, the fundamental properties of computers are very important because they affect what we can and cannot do. Sometimes, an understanding of these properties is necessary to find the best solution to a problem. In most cases, those who understand computing fundamentals can anticipate problems and adjust their goals so that they can get the real job done. Those who do not understand these limitations, may waste their time attempting something impossible or, even worse, produce a product with poorly understood or not clearly stated capabilities. Further, those that understand the fundamental limitations are better equipped to clearly state the capabilities and limitations of a product that they produce. Finally, an understanding of these limitations, and the way that they are proved, often reveals practical solutions to practical problems. Consequently, "basic science" should be a required component of any accredited Software Engineering program.

In the next section, we will give a few illustrations to make these points clearer.

2 A few anecdotes

2.1 What can be said with grammars

Many years ago, Robert Floyd encountered a graduate student who was trying to find a complete context-free grammar for Algol-60, one that specified that all variables must be declared before use. The student's plan was to use the grammar as input to a compiler generator. Floyd's understanding of CS fundamentals allowed him to prove that no such grammar could exist. The graduate student was saved months, perhaps years, of futile effort. With this information he understood that he would have to find another way to express those restrictions as input to his compiler generator. [6]

This anecdote makes it clear that it is very important to be able to decide whether or not a task is impossible. Some people spend their lives trying to solve impossible problems.

2.2 The meaning of computational complexity

Computer Scientists have developed ways to classify the complexity of algorithms and to classify problems in terms of the complexity-class of the best solution to those problems. This allows them to determine whether or not an algorithm is as good as it can get (optimal). However, strange as it may sound, sometimes an "optimal" algorithm is not the best choice for a practical application.

In the 70's Fred Brooks, working on visualization tools for chemists, announced that he wanted an optimal algorithm for a well defined problem. A very bright graduate student proposed such an algorithm and submitted a short paper proving that it was optimal. Brooks insisted that the algorithm be implemented and its performance compared with the performance of the method that they had been using; the performance of the "optimal" algorithm was worse than the old one. Computer Science complexity methods refer to asymptotic performance, that is, performance for very large problems. Algorithms that are not optimal may actually be faster than the "optimal" ones for certain values of the key parameters. Since a developer may find an "optimal" algorithm in a textbook, she must be aware of what "optimal" means and check to see that the performance is actually better in practice than other algorithms. Moreover, a developer who knows the asymptotically optimal algorithm can often modify it to produce an algorithm that will be fast for the application at hand.

Another such example is Linear Programming. The widely used Simplex algorithm is known to be exponential in the worst case. However, the Simplex has superb performance in practice (in fact, it's expected performance is provably polynomial). On the other hand, the ("worst-case") polynomial algorithm for Linear Programming, known as the Ellipsoid Algorithm, appears to be impractically slow. [10]

2.3 The practicality of a "bad" solution to the "Knapsack Problem"

In the "Knapsack Problem" the input is a set of weights $w_1, w_2, \ldots, w_d$, and the capacity, C, of the knapsack. We want to pack as much weight into the knapsack, without going over the capacity. The most obvious approach, starting with the largest weights, does not work, because if we have three weights $w_1 = 51, w_2 = 50, w_3 = 50$, and $C = 100$, and our strategy is to pack as much as possible at each step, we would put 51 in, and we would have no more space left for w_2 and w_3. The optimal solution in this case is of course $w_2 + w_3 = 100$.

The "Knapsack Problem" can be solved with Dynamic Programming, where we construct a table with dimensions $d \times C$ (d = number of weights, C = capacity), and fill it out using simple recursion. A classical worst-case running time analysis of the dynamic programming algorithm shows that it requires exponential time. The reason is that the algorithm builds a $d \times C$ table, so if C is given in binary, the size of the table is exponential in the size of the capacity (i.e., exponential in the size of the input). Therefore, the dynamic programming solution to the Knapsack Problem runs in exponential time in the size of the capacity of the knapsack, and hence it is asymptotically infeasible.

In fact, the dynamic programming solution to the "Knapsack Problem" is widely used in Computer Science. In applications such as Compilers and Optimization problems, equivalent

problems arise frequently, and they are solved using dynamic programming. The method is practical, even with many weights, for reasonable C.

One should not interpret this as meaning that the theoretical complexity is useless; *au contraire*, it demonstrates why even practitioners who think that they are not interested in "theory" should understand computational complexity when developing algorithms for difficult problems.

2.4 Maximum size for a halting problem

Two software developers were asked to produce a tool to check for termination of programs in a special purpose language used for building discrete event control systems. One refused the job claiming that it was impossible because we cannot solve the halting problem. A second, who understood the proof of the impossibility of the halting problem, realized that the language in question was so limited that a check would be possible if a few restrictions were added to the language. The resulting tool was very useful. Here again, an understanding of the nature of this "very theoretical" result was helpful in developing a practical tool with precisely defined limitations.

2.5 Can we prove that loops terminate

Dr. Robert Baber, a software engineering educator who has long advocated more rigorous software development [1, 2, 3, 4] was giving a seminar in which he stated that it was the responsibility of programmers to determine whether or not loops they have written will terminate. He was interrupted by a young faculty member who asserted that this was impossible, "the halting problem says that we cannot do that." In fact, the halting problem limits our ability to write a program that will test *all* programs for termination, not about our ability to check *a given* program for termination. This incident shows that a superficial understanding of computer science theory can lead people astray and cause them to be negligent.

Clearly, we must teach fundamentals in such a way that the student knows how to translate theoretical results into practical knowledge. For example, when teaching about the general undecidability of halting problems, one can accompany the proof with an assignment to determine the conditions under which a particular machine or program is sure to terminate. Comparing the general result with the specific example helps the student to understand the real meaning of the general result.

2.6 The implications of finite word length

In 1969, some software developers became enthusiastic about a plan to store 6 bytes in a 4 byte word. They proposed computing the product of 6 bytes and converting the result to a 4 byte floating-point number. Sadly, none of the programmers in the organization understood the impossibility of this scheme and they invested a lot of time discussing it. Luckily, an academic visitor[1] who did understand basic information theory, could convince them of its

[1] Dave Parnas.

applicability by providing a counter-example, i.e., an example where the same output would be obtained for two different inputs. It was quite possible that even extensive testing would not have revealed the error, but it would cause "bugs" in practice.

2.7 The limitations on push-down automata

Recently one of us had occasion to talk to some people who were familiar with the standard results about push-down automata, i.e., that the class of problems that they could solve was smaller than that for Turing machines. He reminded them that in today's market, one can buy an auxiliary disk and attach it to a laptop or other personal computer. He asked if this changed the fundamental properties of the machine (it does not). He then asked what would happen if we could buy an auxiliary push-down stack and attach it as a separate device on a push-down automata that already had one stack. All claimed that the result would still be a push-down automata, i.e., they did not recognize that having a second (independently accessible) stack changed the fundamental capabilities of the machine. The same group included many who did not realize that placing limits on the depth and item size of the stack in a push-down automaton made it no more powerful than any other finite state machine. This meant that they did not understand that there would be an upper limit in the number of nested parenthesis in an expression that would be parsed by any realizable push-down automaton, or that a twin-stack push-down automaton (with infinite stacks) was as powerful as a Turing machine and more powerful than any realizable computer.

2.8 The practical limitations of open questions

There are number of problems in computability and complexity theory that remain open. Many practitioners and students believe that these problems are of interest only to theoreticians. In fact, they have very practical implications. Probably the most dramatic of these is the "$\mathbf{P} = \mathbf{NP}$" question [5] for which a prize of $1,000,000 has been offered. This does not interest most students who realize that they will not win the prize. However, the question has very important implications in cryptography. Some very widely used encoding algorithms are only "safe" if the answer is that $\mathbf{P} \neq \mathbf{NP}$. If it is not, it might be possible to find ways to crack codes quickly (see [5, 7]). (The reason is that if $\mathbf{P} = \mathbf{NP}$, then that would imply the existence of a polytime algorithm for factoring, and such an algorithm would render the RSA encryption scheme insecure.)

3 A course in basic science for Software Engineers

McMaster Universities CEAB accredited Software Engineering Program includes a course designed to teach its students both the parts of "theoretical computer science" that they ought to know and how to use them. For a complete outline of the course see [12]; here we mention the main topics: (1) Finite Automata (finite number of states, and no memory), (2) Regular Expressions, (3) Context-Free Grammars, (4) Pushdown Automata (like finite automata, except they have a stack, with no limit on how much can be stored in the stack), (5) Turing

Machines (simplified model of a general computer, but equivalent to general computers), (6) Rudimentary Complexity.

For four years we used [8] as the textbook, but last year (2005-06) we used [11]. The former was perhaps better suited for engineers, but the latter has a better complexity section and de-emphasizes push-down automata which have lost ground in the last years as a theoretical constuction.

Deeper discussions of the basic subject matter can be found in [7, 8, 10]. However, these references do not discuss educational motivations as we do.

4 Conclusions

Established engineering accreditation rules require that each engineering student have a minimum exposure to basic science. As accredited software engineering programs are relatively new, there is no clear understanding what constitutes appropriate basic science. Although we believe that every Engineer should have been taught basic physical science, we believe that those who will specialize in software require a thorough exposure to the topics discussed above. This paper has illustrated why, a course on these topics should be required as part of the basic science component of a program for engineers specializing in software intensive products.

References

[1] R.L. Baber. *Software Reflected: the Socially Responsible Programming of Our Computers*. North-Holland Publishing Co., 1982. German translation: Softwarereflexionen: Ideen und Konzepte für die Praxis, Springer-Verlag, 1986.

[2] R.L. Baber. *The Spine of Software: Designing Provably Correct Software—Theory and Practice*. John Wiley & Sons, 1987.

[3] R.L. Baber. *Error Free Software: Know-How and Know-Why of Program Correctness*. John Wiley & Sons, 1991. German original: Fehlerfreie Programmierung für den Software-Zauberlehrling, R. Oldenbourg Verlag, München, 1990.

[4] R.L. Baber. *Praktische Anwendbarkeit mathematisch rigoroser Methoden zum Sicherstellen der Programmkorrektheit*. Walter de Gruyter, 1995.

[5] Stephen Cook. The P versus NP problem. `www.claymath.org/prizeproblems/p_vs_np.pdf`.

[6] Robert Floyd. On the nonexistence of a phrase structure grammar for ALGOL 60. *Communications of the ACM*, 5(9):483–484, 1962.

[7] Michael R. Garey and David S. Johnson. *Computers and Intractability*. Bell Telephone Laboratories, 1979.

[8] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2000. We used the 2nd edition, but the 3rd edition is now available.

[9] Canadian Council of Professional Engineers. Accreditation criteria and procedures, 2005. `http://www.ccpe.ca/e/files/report_ceab.pdf`.

[10] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[11] Michael Sipser. *Introduction to the Theory of Computation*. Thomson, 2006.

[12] Michael Soltys. `http://www.cas.mcmaster.ca/~soltys/se4i03-f02/course-outline.txt`.