CS 3100 – Models of Computation – Fall 2010

August 23, 2010

notes1, Handed out: August 23, 2011

Human society is, more than ever, reliant on computing systems operating correctly within automobile control systems, medical electronic devices, telephone networks, mobile robots, farms, nuclear power plants, etc. With so much entrusted to computers, how can we ensure that all of these computers are being built and operated in a manner responsible to all flora and fauna? How do we avoid potential disasters - such as mass attacks due to malevolent "viruses," or an avionic computer crashing and rebooting mid-flight? On a deeper, more philosophical note, what exactly is computation? Can we mathematically characterize those tasks that computers are capable of performing, and those they are incapable of performing? Studying *automata theory* and *mathematical logic* helps us obtain some of these answers. Automata theory and mathematical logic are the core topics in the *engineering mathematics* of computers, much like 'traditional engineers' apply differential and integral calculus to build better-engineered products such as bridges, automobiles, and airplanes.

What is 'Computation?'

One possible feeble answer is this: "if it involves a computer, a program running on a computer, and numbers going in and out, then computation is likely happening. Such an answer invites a barrage of additional questions such as "what is a computer?"; "what is a program?", etc. There are other tricky situations to deal with as well. Consider another place where computation seems to be happening: within our very body cells. Thanks to modern advances in genetics, we are now able to understand the mind-boggling amount of "string processing" that occurs within our cells - in the process of transcribing the genetic code (which resembles assembly code in a strange programming language), doing all of that wonderful string matching and error correction, and resulting in the synthesis of proteins. Is this also computation?

The short answer is that we *cannot* have either a comprehensive or a permanent definition of what 'computation' means. Unless we employ the *precise* language offered by mathematics, philosophical or emotionally charged discussions are bound to lead nowhere. One must build *formal models* that crystallize the properties observed in real-world computing systems, study these models, and then answer questions about computing and computation in terms of the models. The abstraction must also be at the *right level*. Otherwise, we will end up modeling a computer as a mindless electronic oscillator that hauls bits around.

Given all this, it is indeed remarkable that computer science has been able to capture the essence of computing in terms of a *single formal device*: the so called Turing machine. A Turing machine is a simple device that has finite-state control that interacts with an unbounded storage tape (or, equivalently, a finite-state control that interacts with two unbounded stacks, as we shall show very soon). In fact, several other formal devices - such as the Lambda calculus, Thue systems, etc. - were proposed around the same time as Turing machines. All these devices were also formally shown to be equivalent to Turing machines. This caused Alonzo Church to put

forth his (by now famous) thesis: "All effectively computable functions can be understood in terms of one of these models."

A Minimalist Approach

In a minimalist approach, models are created with the smallest possible set of mechanisms. In the case of computational models, it is a bit ironic that the first model proposed - namely Turing machines- was also the most powerful. However, with the increasing usage of computers,[1] two other models born out of practical necessity were proposed, roughly two decades after Turing machines were proposed: finite automata in the late 1950's, and push-down automata shortly thereafter. Rearranging computer history a bit, we will discuss finite automata first, push-down automata next, and finally Turing machines (see Figure 1). All these types of machines are meant to carry out *computational procedures* ("procedures" for short,) consisting of instructions. They differ primarily in the manner in which they record data ("state"). A procedure always begins at an initial state which is highlighted by an arrow impinging from nowhere, as in Figure 1. The "data input" to a procedure, if any, is provided through the data storage device of the machine. Each instruction, when executed, helps transform the current (data and control) state into the next state. An instruction may also read an input symbol (some view these inputs coming from a *read-only tape*). Also, at every state, one or more instructions may become eligible to execute. A deterministic machine is one that has at most one eligible instruction to execute at any time, while a nondeterministic machine can have more than one eligible instruction.

A procedure halts when it encounters one of the predetermined final states. It is possible for a procedure to never encounter one of its final states; it may loop forever. If a procedure is guaranteed to halt on all inputs, it is called an *algorithm*. Unfortunately, it is impossible to tell whether a given procedure is an algorithm.

There are essentially three ways to organize the data (state) recording apparatus of a machine: (i) have *none* at all, (ii) employ one stack to record data, and (iii) employ two stacks to record data (in other words, employ zero, one, or two stacks)! A finite-state control device by itself (*i.e.*, without any additional history recording device) is called a *finite automaton* - either a deterministic finite automaton (DFA) or a nondeterminstic finite automaton (NFA). A finite automaton is surprisingly versatile. However, it is not as powerful as a machine with one stack, which, by the way, is called a *push-down automaton* (PDA). Again there are NPDA and DPDA - a distinction we shall study later.

A PDA is more powerful than a finite automaton. By employing an unbounded stack, a PDA is able to store an arbitrary amount of information in its state, and hence, is able to refer to data items stacked arbitrarily prior. However, a PDA is not as powerful as a machine with *two* stacks. This is because a PDA is not permitted to "peek" inside its stack to look at some state s held deep inside the stack, unless it is also willing to pop away all the items stacked since s was stacked. Since there could be arbitrarily many such stacked items, a PDA cannot hope to preserve all these items being popped and restore them later.

The moment a finite-state control device has access to two unbounded stacks, however, it will have the ability to pop items from one stack and push them into the other stack. This gives these two-stack machines the ability to peek inside the stacks; in effect, a finite-state control device with two stacks becomes equivalent to a Turing machine. We can show that adding further stacks does not increase its power! All computers, starting from the humble low-end computer of a Furbee doll or a digital wrist-watch, through all varieties of desktop and laptop computers, all the way to 'monstrously powerful' computers, (Say, the IBM Blue Gene/L [1] computer) can be modeled in terms of Turing machines.

There is an important point of confusion we wish to avoid early. Real computers only have a finite amount of memory. However, *models* of computers employ an *unbounded* amount of memory. This allows us to study

[1] Thomas J. Watson, Chairman of IBM in 1943, is said to have remarked, "I think there is a world market for maybe five computers." Well, there are more than five computers in a typical car today. Some cars carry hundreds, in fact!

the outcomes possible in *any* finite memory device, regardless of how much memory it has. So long as the finite memory device does not hit its storage limit, it can pretend that it is working with an infinite amount of memory.

How to Measure the Power of Computers?

In comparing computational machines ("computers," loosely), we cannot go by any subjective measure such as their absolute speed, as such "speed records" tend to become obsolete with the passage of time. For instance, the "supercomputers" of the 1950s did not even possess the computational power of many modern hand-held computers and calculators. Instead, we go by the *problem solving ability* of computational devices: we deem two machines M_1 and M_2 to be equivalent if they can solve the same class of problems, *ignoring the actual amount of time taken*. Problem solving, in turn, can be modeled in terms of *algorithmically deciding membership in languages*. Alas, these topics will have to be discussed in far greater detail than is possible now, since we have not set up any of the necessary formal definitions.

Complexity Theory

You may have already guessed this: in studying and comparing computing devices, we cannot ignore time or resources entirely. They do matter! But then one will ask, "how do we measure quantities such as time, space, and energy?" It is very easy to see that using metrics such as minutes and hours is unsatisfactory, as they are *non-robust* measures, being tied to factors with ephemeral significance, such as the clock speed of computers, the pipeline depth, etc. With advances in technology, computations that took hours a few years ago can nowadays be performed in seconds. Again, Turing machines come to our rescue! We define a unit of time to be *one step* of a Turing machine that is running the particular program or computation. We then define time in terms of the asymptotic worst-case complexity notation "$O()$" employed in any book on algorithm analysis (see for instance [2] or [3]). However, such a characterization is often not possible:

- There are many problems (called NP-complete problems) whose best known solutions are exponential time algorithms. It is unknown whether these problems have polynomial time algorithms.
- There are problems for which algorithms are known not to exist; for others, it is not known whether algorithms exist.

However, it is true that researchers engaged in studying even these problems employ Turing machines as one of their important formal models. This is because any result obtained for Turing machines can be translated into corresponding results for real computers.

Automata Theory and Computing

In these notes, we approach the above ideas through automata theory. What is automata theory? Should its meaning be cast in concrete, or should it evolve with advances in computer science and computer applications? We will use it to refer to a comprehensive list of closely related topics, including:

- finite and infinite automata (together called "automata"),
- mathematical logic,
- computability and complexity (TR, co-TR, NP-complete, etc.),
- formal proofs,
- the use of automata to decide the truth of formal logic statements,

- automata on infinite words to model the behavior of reactive systems, and last but not least,
- applications of the above topics in formal verification of systems.

Automata theory is a 'living and breathing' branch of theory - not 'fossilized knowledge.' It finds day-to-day applications in numerous walks of life, in the *analysis* of computing system behavior. We illustrate the use of automata to describe many real-life finite-state systems, including: games, puzzles, mathematical logic statements, programming language syntax, error-correcting codes, and combinational and sequential digital circuits. We also illustrate tools to describe, compose, simplify, and transform automata.

Why "Mix-up" Automata and Mathematical Logic?

We believe that teaching automata theory hand-in-hand with mathematical logic allows us to not only cover concepts pertaining to formal languages and machines, but also illustrate the deep connections that these topics have to the process of *formalized reasoning* – proving properties about computing systems. Formalized reasoning about computing systems is escalating in importance because of the increasing use of computers in safety critical and life critical systems. The software and hardware used in life and resource critical applications of computers is becoming so incredibly complex that testing these devices for correct operation has become a major challenge. For instance, while performing 100 trillion tests sounds more than adequate for many systems, it is simply inadequate for most hardware/software systems to cover all possible behaviors.

Why Verify? Aren't Computers "Mostly Okay?"

Human society is crucially dependent on software for carrying out an increasing number of day-to-day activities. The presence of *bugs* in software is hardly noticed until, say, one's hand-held computer hangs, when one pokes its reset button and moves on, with some bewilderment. The same is the case with many other computers that we use; in general, the mantra seems to be, "reset and move on!" However, this surely cannot be a general design paradigm (imagine a machine getting stuck in an infinite reboot loop if the problem does not clear following a reset). In the modern context, one truly has to worry about the logical correctness of software and hardware because there have been many "close calls," some real disasters, and countless dollars have been wasted in verifying products before they are released. In 2004, the Mars Spirit Rover's computer malfunctioned when the number of allowed open files in flash memory were exceeded. This caused a shadow of uncertainty to hang over the project for a few days, with scientists wasting their time finding a cure for the problem. Recently, car companies have had recalls due to software bugs in their computerized engine control; the societal costs of such bugs ("software defects") are estimated to be very high [4, 5]. In 1996, Ariane-5, a rocket costing $2B, self-destructed following an arithmetic overflow error that initially caused the rocket nozzles to be pointed incorrectly [6].

A very important human lesson is contained in many software failures. In 1987, a radiation therapy machine called Therac-25 actually caused the death of several patients who came to get radiation therapy for cancer. At the time of the incident, the machine had recently been redesigned and its software was considered so reliable that many of its safety interlock mechanisms had been dispensed with. One of these interlocks was for monitoring the amount of radiation received by patients. To cut a long story short, the software was, after all, buggy, and ended up administering massive radiation overdoses to patients [7]. Every engineer shoulders the societal responsibility to adopt simple and reliable safety measures in the systems they design and deploy to avoid such catastrophes. The approach taken in each project must be: "when in doubt, play it safe, keeping the well being of lives and nature in mind."

The *root cause* of a large majority of bugs is the *ambiguous* and/or *inconsistent* specification of digital system components and subsystems. Software built based on such specifications is very likely to be flawed, hard to

debug, and impossible to systematically test. As an example from recent practice, the specification document of a widely used computer interconnect bus called the 'PCI' [8] was shown to be internally inconsistent [9]. Unfortunately, the design community had moved too far along to take advantage of these findings. Many of the disasters of computer science are not *directly* in the form of crashes of rockets or chips (although such disasters have happened[2]). In 2001, the United States Federal Bureau of Investigation (FBI) launched a project to overhaul their software to coordinate terrorist databases. After nearly four years and over $300 million dollars spent, the project has been declared to have an unrealizable set of software requirements, and hence abandoned.

The ability to write precise, as well as, *unambiguous* specifications is central to using them correctly and testing them reliably. Such *formal methods* for hardware and software have already been widely adopted. Organizations such as Intel, AMD, IBM, Microsoft, Sun, HP, JPL, NASA, and NSA employ hundreds of formal verification specialists.

Verifying Computing Systems Using Automaton Models

It turns out that in order to *prove* that even the simplest of computing systems operates correctly, one has to examine so many inputs and their corresponding outputs for correctness. In many cases, it will take *several thousands of centuries* to finish complete testing. For example, if such a computer has one megabit of internal storage, one has to check for correctness over 2^{10^6} states. This is a number of unfathomable magnitude. Modern symbolic state representations are employed to conquer this scale of complexity.

How does automata theory help ensure that safety critical systems are correct? First of all, it helps create *abstract models* for systems. Many systems are so complex that each vector consists of thousands, if not millions, of variables. If one considers going through all of the 2^{1000} assignments for the bits in the vector, the process will last thousands of millennia. If one runs such tests for several days, even on the fastest available computer, and employs a good randomization strategy in vector selection, one would still have covered only some *arbitrary* sets of behaviors of the system. Vast expanses of its state-space would be left unexamined. A much more effective way to approach this problem in practice is to *judiciously* leave out most of the bits from vectors, and examine the system behavior *exhaustively* over all the remaining behaviors. (In many cases, designers can decide which bits to leave out; in some cases, computer-based tools can perform this activity.) That is, by leaving out the right set of bits, we end up fully covering an abstract model. Experience has repeatedly shown that verifying systems at the level of abstract models can often find serious bugs quicker. The simulation of non-abstracted models is also a necessary part of verification in practice.

It turns out that abstracted models of most systems are *finite automata*. There are a number of techniques being developed that can represent, as well as manipulate, *very large* finite automata. These techniques help minimize the degree to which systems have to be abstracted before they can be exhaustively verified. This, in turn, means that the risk of accidentally missing an error due to overly heavy abstractions is also reduced.

Automaton/Logic Connection

Various branches of mathematical logic are employed to *precisely* and *unambiguously* describe both system *requirements* and system *behaviors*. In modern verification systems, automata theoretic techniques are often employed to process these logical specifications to check whether they are true or false. It is therefore important for students to see these topics treated in a cohesive manner.

The importance of precision and clarity in system descriptions cannot be overstated. Many system description documents rival Los Angeles telephone directories in their size, containing a very large number of subtle

[2]The Ariane rocket, worth $2B, was lost because of incorrect version of software running. Intel lost $.5B due to a floating-point bug in their Pentium II microprocessor.

assertions that tax the human mind. Each person reading such a document comes up with a different understanding. While engineers are incredibly smart and are able to correct their misunderstandings more often than not, they still waste their time poring over lengthy prose fragments that are ambiguous, and simply too complex to trust one's mind with. It has been widely shown that formal statements in mathematical logic can serve as very valuable augments to text-based documents, supplying the missing precision in the text.

One of the most serious deficiencies of an exclusively natural-language system description is that engineers *cannot mechanically calculate their consequences*. In other words, they cannot ask "what if" (putative) queries about scenarios that are not explicitly discussed in the document. An example from the I/O system world is, "what if I/O transaction x is allowed to overtake transaction y? Does it cause wrong answers to be returned? Does it cause system deadlocks?" The number of potentially interesting "what if" questions pertaining to any real system is extremely large. It is impossible or even counterproductive for a specification document to list all these questions and their answers. On the other hand, an approach where one is able to state the specification of complex systems (such as I/O buses) in a precise language based on mathematical logic, and is able to pose *putative* or *challenge* queries (also expressed in mathematical logic) is highly conducive to gaining a proper understanding of complex systems. Ideally, systems of this kind must either try to show that the posed conjecture or query is true, or provide a clear explanation of *why* it is false. Model checking based verification methods provide this ability in most formal verification based design approaches under development or in actual use today.

The ability to *decide* - provide an answer, without looping, for all possible putative queries - is, of course, a luxury enjoyed when we employ simpler mathematical logics. Unfortunately, simpler mathematical logics are often not as expressive, and we will then have to *deduce*, using partly manual steps and partly automatic (decidable) steps, the answer.

Avoid Attempting the Impossible

Automata theory and logic often help avoid pursuing the impossible. If one can *prove* that there cannot be a decider for some task, then *there is no point wasting everyone's time in pursuit of an algorithm* for that task. On the other hand, if one does prove that an algorithm exists, finding an *efficient* algorithm becomes a worthwhile pursuit.

As an example, the next time your boss asks you to produce a C-grammar equivalence checker that checks the equivalence between any two arbitrary C-grammar files (say, written in the language `Yacc` or `Bison`) and takes no more than "a second per grammar-file line," don't waste your time coding - simply prove that this task is impossible!

Solving One Implies Solving All

There is another sense in which automata theory helps avoid work. In many of these cases, researchers have found that while we cannot actually solve a given problem, we can *gang up* or "club together" thousands of problems such that the ability to solve *any one* of these problems gives us the ability to solve *all* of these problems. Often the solvability question will be whether it is tractable to solve the problems - i.e., solve in polynomial time. In many cases, we are simply interested in solvability, without worrying about the amount of time. In these cases also, repeated work is avoided by grouping a collection of problems into an equivalence class and looking for a solution to only *one* of these problems; this solution can be easily modified to solve thousands of practical problems. This is the motivation behind studying NP-completeness and related notions. We will also study the famous Halting problem and problems related to it by taking this approach of clubbing problems together through the powerful idea of *mapping reductions*.

Automata Theory Demands a Lot From You!

The study of automata theory is challenging. It exposes students to a variety of mathematical models and helps build confidence in them, thus encouraging them to be creative, to take chances, and to tread new ground while designing computing systems. Unfortunately, the notion of formally characterizing designs is not emphasized in traditional systems classes (architecture, operating systems, etc.), where the goal has, historically, been high performance and not high reliability. Therefore, it takes an extra bit of effort to put formal specification and verification into practice. Fortunately, the computer industry has embraced formal methods, and sees it as the main hope for managing the complexity and ensuring the reliability of future designs.

It is impossible to learn automata theory "in a hurry." While the subject is quite simple and intuitive in hindsight, to get to that stage takes patience. You must allow enough time for the problems to gestate in your minds. After repeatedly trying and failing, you will be able to carry the problems in your minds. In these notes, I attempt to present the subject matter through functional programming, choosing the popular scripting language Python. We shall be using Python3 for our work; the latest release at the time of writing of these notes is:

```
Python 3.2.1 (v3.2.1:ac1f7e5c0510, Jul  9 2011, 01:03:53)
```

A Brief Foray Into History

Let us take a historical perspective, to look back into how this subject was viewed by two of the originators of this subject - Michael Rabin and Dana Scott - in their 1959 paper (that, incidentally, is cited in their ACM Turing award citation [10]):

> Turing machines are widely considered to be the abstract prototype of digital computers; workers in the field, however, have felt more and more that the notion of a Turing machine is too general to serve as an accurate model of actual computers. It is well known that even for simple calculations, it is impossible to give a prior upper bound on the amount of tape a Turing machine will need for any given computation. It is precisely this feature that renders Turing's concept unrealistic.
>
> In the last few years, the idea of a finite automaton has appeared in the literature. These are machines having only a finite number of internal states that can be used for memory and computation. The restriction of finiteness appears to give a better approximation to the idea of a physical machine. [...]. Many equivalent forms of the idea of finite automata have been published. One of the first of these was the definition of "nerve-nets" given by McCulloch and Pitts. [...]

In short, Rabin and Scott observe that the theory of Turing machines, while all encompassing, is "too heavy-weight" for day-to-day studies of computations. They argue that perhaps finite automata are the right model of most real-world computations. From a historical perspective, the more complex machine form (Turing machine) was proposed much before the much simpler machine form (finite automata). All this is clearly not meant to say that Turing machines are unimportant—far from it, in fact! Rather, the message is that a more balanced view of the topics studied under the heading of automata will help one better appreciate how this area came about, and how the priorities will shift over time.

Disappearing Formal Methods

In the article "disappearing formal methods [11]," John Rushby points out how throughout the history of engineering, various new technologies had been [over] advertised, until they became widely accepted and taken

for granted. For instance, many olden-day radios had a digit ('6' or '8') boldly written on their aluminum cases, advertising the fact that they actually employed 6 or 8 (as the case may be) transistors. Centuries ago, differential and integral calculus were widely advertised as the "magic" behind studying planetary motions, as well as building everyday objects, such as bridges. Today, nobody hangs a sign-board on a bridge proclaiming, "this bridge has been designed using differential and integral calculus!" In the same vein, the term *formal methods* is nowadays used as a "slogan" to call attention to the fact that we are really using ideas based on logic and automata theory to design products, as opposed to previously, when we used no such disciplined approach. While the explicit advertisement of the technology behind modern developments is inevitable, in the long run when such applications are well understood, we would no longer be pointing out explicitly that we are actually designing, say, floating-point units, using higher order logic.

References

[1] Adiga, N. R., et al. "An Overview of the BlueGene/L Supercomputer". In *Conference on High Performance Networking and Computing: SC2002*, page 60, 2002.

[2] Cormen, Thomas H., Leiserson, Charles E., and Rivest, Ronald L. *Introduction to Algorithms*. McGraw-Hill Company, 1990.

[3] Aho, Alfred V., Hopcroft, John E., and Ullman, Jeffrey D. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[4] The Changing World of Software.
http://www.sei.cmu.edu/publications/articles/watts-humphrey/ changing-world-sw.html.

[5] RTI. "The Economic Impacts of Inadequate Infrastructure for Software Testing," Final Report, National Institute of Standards and Technology, May 2002. Research Triangle Institute (RTI) Project Number 7007.001. Accessed at http://spinroot.com/spin/Doc/course/NISTreport02-3.pdf (February 2006).

[6] Meyer, Bertrand. "Design by Contract: The Lessons of Ariane". *IEEE Computer*, 30(2):129–130, January 1997.

[7] Nancy G. Leveson. *SAFEWARE: System Safety and Computers*. Addison-Wesley, 1995.

[8] PCI Special Interest Group–PCI Local Bus Specification, Revision 2.1, June 1995.

[9] Corella, F., Shaw, R., and Zhang, C. "A formal proof of absence of deadlock for any acyclic network of PCI buses". In *Hardware Description Languages and their Applications*, pages 134–156. Chapman Hall, 1997.

[10] Rabin, Michael O. and Scott, Dana S. "Finite Automata and their Decision Problems". *IBM J. Res. Dev.*, (3):114–125, 1959.

[11] Rushby, John, M. "Disappearing Formal Methods". In *High-Assurance Systems Engineering Symposium*, pages 95–96, Albuquerque, NM, nov 2000. Association for Computing Machinery.

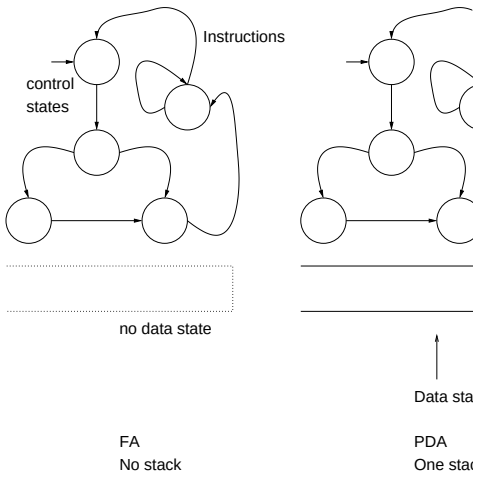

Figure 1: The power of various machines, and how to realize them