Predicting Computer Behavior

Donald I. Good

Technical Report 20 May 1988

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work was sponsored in part by the Defense Advanced Research Projects Agency under ARPA Orders 6082 and 9151. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.

Abstract

The current practice of digital system engineering is not based on science and mathematics. As a result, computer systems in operation today exhibit unforeseen behaviors that cause loss of life, information, property and money. This paper identifies the major scientific and mathematical problems that need to be solved to advance the state of digital system engineering practice.

Chapter 1
INTRODUCTION

COMPASS 88. COMPuter ASSurance. Assurance of what?

- "A man with cancer is killed because a computer tells a radiation therapy machine to administer a lethal dose."
- "A rocket on the way to Venus has to be destroyed because a crucial line is left out of the computer software controlling it."
- "A bank is forced to borrow $23.6 billion overnight because of a computer, and the government-securities market narrowly escapes disaster."
- "Workers are killed by computer controlled industrial robots."

These are quotations from the call for papers for this conference. They get our attention because the behavior of these computer systems caused dramatic loss of life, property and money. What do we want assurance of? We want assurance that computer systems will not cause such harmful effects.

To gain that assurance, we must eliminate the cause of those effects. What is the common thread that ties these incidents together? What caused these effects? They were caused by a computer system that behaved in some unforeseen way. If any one of these four incidents had been foreseen, would they have been allowed to happen? I certainly hope not.

These are not isolated incidents. Unforeseen behavior in computer systems is common. In the sophisticated, high-tech vocabulary of our times, we call an instance of unexpected behavior a "bug", or if it’s benign, a "feature." If computing bugs were bacteria, someone would surely declare an epidemic. When these instances of unforeseen behavior cause serious effects, we hear about them. Those that do not cause serious effects, we ignore or tolerate as mere inconvenience. But, the ones that are potentially the most dangerous are the ones that we haven’t yet discovered.

The only way to be assured that dangerous bugs do not exist is to peer into the future behavior of a computer system. Past behavior is of no concern. The effects of that behavior already have been caused. We cannot undo them. The only thing we can do anything about is future behavior.

To get sound assurance that a computer system will not cause harmful effects, we must be able to predict accurately that it *will* behave within acceptable limits. Without that predictability, using a computer system is a gamble. The four incidents cited above are ones in which someone gambled, probably without even knowing it, and lost.

Chapter 2

SCIENCE, MATHEMATICS AND ENGINEERING

There are four key observations that lead to a rigorous, scientific basis for predicting accurately the behavior of a digital computer system.

The first observation is that the thing we need to make predictions about is a piece of computer hardware. What we need to predict is the electronic behavior of a small part of the physical universe. It is the electronic behavior of this piece of physical material that causes beneficial or harmful effects in the physical world.

The second observation is that there are established, well-developed scientific methods for predicting the behavior of the physical world. The most effective way is to identify a mathematical equation that describes the behavior of what we need to predict.

An example of such an equation is Ohm's Law for metallic conductors, $V=I*R$. It describes the mathematical relationship among the voltage, current and resistance of a metallic conductor. A process of current measurement transforms a physical observation about the conductor into a mathematical object, a number. Another process, resistance measurement, transforms a second physical observation into a second number. Then, strictly within the mathematical world, these two numbers are multiplied together to get a third number. By the miracle of mathematics, that third number accurately predicts the number that *will be* obtained by measuring the voltage of the conductor. This relationship among physical observation, measurement and mathematics is illustrated in Figure 2-1.

Figure 2-1: Science and Mathematics

```
    Physical       Measurement        Mathematical
Observation <----------------------> Analysis
```

The purpose of science is to discover the laws of physical behavior. Much of science can be viewed as the search for the mathematical equation that accurately describes some aspect of physical behavior. The mathematical application of those laws to physical systems is the basis of effective engineering. Electrical engineering, for example, is firmly based upon the mathematical application of Ohm's Law and other mathematical equations that accurately describe the physical behavior of electrical circuits.

The third observation, the one of science, is that there do exist mathematical equations that describe, with a very high degree of accuracy, the physical, electronic behavior of digital computers. The existence of such equations establishes a true scientific relationship between the observable physical behavior of a digital computer and mathematics. Once such equations are identified, the full power of mathematics can be used to analyze the physical behavior of a digital computer.

The final observation, the one of engineering, is that once such equations are identified, they can be used in the practice of engineering to build computing systems with physical behavior that is predictable with a very high degree of accuracy. In fact, one might well wonder how systems with predictable behavior can be engineered without applying them?

Chapter 3

MATHEMATICS FOR DIGITAL SYSTEMS

For the sake of concreteness, the presentation in this section is focused on the machine language of a fully synchronous digital system. However, what is said here applies to the behavior of any digital system that can be described accurately by a mathematical equation.

The key scientific step in predicting the physical behavior of a digital system is to identify the right equation. The equation might define the gate level operation of a digital system, the micro code level, the machine language level, an operating system or a higher level programming language. Thus, the mathematics described here apply both to certain levels of hardware engineering and to all of software engineering. I will refer to all of these levels as "digital system engineering."

The discussion in this section should be viewed as taking place in the mathematical world shown on the right of Figure 2-1. The identification of the equation that accurately describes the physical behavior of a digital system is what allows us to "cross the measurement bridge" and have mathematical discussions about the behavior of physical systems.

3.1 Describing System Behavior

Consider the complete sequence of observable states produced by a fully synchronous digital computer. Include in its state every bit that is observable by a machine language program. This includes the entire address space of the machine (including its program), every observable register and even the system clock (if it is observable). The electronic behavior of such a machine can be described very accurately by a system of equations such as

```
M(s,t,n) =
 if halt(s) or n=0
 then {(s,t)}
 else {(s,t)} @ M(cycle(s),t+ticks(s),n-1)

halt(s)  = ...

cycle(s) = execute(fetch(s))

fetch(s) = ...

execute(c) = ...

ticks(s) = ...
```

For every triple *(s0,t0,n)*, *M(s0,t0,n)* produces a sequence of pairs *{ (s0,t0), ..., (sk,tk) }* where *sk* is the state of the machine after starting it on state *s0* at time *t0* and allowing it to run for either *n* steps or until it halts. If the machine halts, $k<=n$; otherwise, $k=n$. Each ti is the time at which state si is observable.[1]

The equations above are defined (by filling in the ...) so that *M(s,t,n)* defines every component of every state. No steps in the sequence are left out. There are no intermediate states in between successive elements of this sequence. *M(s,t,n)* defines every observable bit in every observable successive state of the machine and the time at which the state can be observed.

Table 3-1: Example Behavior

	s0	s1	s2	s3	s4	s5	s6
1	1	0	1	0	1	0	0
2	1	0	0	1	0	1	0
3	0	1	0	1	0	0	1
4	0	0	1	0	1	1	0
5	0	0	1	0	1	0	0
t	0	2	3	4	6	8	9

To illustrate in more detail, Table 3-1 shows what *M(s0,0,6)* might be if these equations define a simple system with just five state components. The five state components are numbered "1" through "5" down the left side of the table. The columns of the table show the values of these components in the successive states "s0" through "s6." The bottom row, labeled "t", is the time of observation of the state in that column. Each column represents one pair *(si,ti)* of the sequence *{ (s0,t0), ..., (s6,t6) }*.

3.1.1 Completeness

Any equation that describes accurately some of the components of some of the steps of the state sequence produced by a digital system can be used to predict some aspects of the physical behavior of the machine. For an equation to be useful in predicting that the behavior of a computer *does not cause any* harmful effect in its physical environment, the equation also must be *complete* in the following sense. It must describe

- *all* of the potentially observable state components,
- *all* of the potentially observable steps in the state sequence and
- the *time* at which each such step occurs.

If the equation does not describe all of these things, it is easy to imagine how harmful physical effects could be caused by some system behavior that was outside the scope of the equation. If just one single bit falls outside the scope of the equation, it could be the one that sends the wrong electrical signal or sends the right signal at the wrong time.

At this point, I'd like to state an opinion about the use of partial recursive functions to describe computer behavior. Traditionally, partial recursive functions have been used to describe the relationship between the initial and final state of a machine. Such a description accurately describes some aspects of the behavior of the physical system, but it is not complete because it may not describe the sequence of intermediate observable states. An incomplete description is not sufficient to show the absence of behaviors that cause harmful effects. (I doubt that the family of the person who died from excess radiation would have taken much comfort from the fact that when the program that controlled the

[1]We assume that the *ti* are monotonically increasing. This is a realistic assumption about a physical system, and it ensures that *m* is a function -- i.e, it maps every triple into a unique sequence.

radiation device stopped, it shut off the radiation source.) Also, the traditional use of partial recursive functions has omitted timing issues. Intermediate system states and timing are important, sometimes critical, aspects of the behavior of a physical system. Effective engineering cannot ignore them.

This is not to say that we should avoid using partial recursive functions to describe the physical behavior of digital systems. We should use whatever mathematics accurately describes the physical behavior of the system. I suspect, however, that the mathematical emphasis on partial recursive functions that map initial states into final states and on their termination has distracted us from some of the important things that we need to describe about physical systems -- namely, intermediate states and timing. Certainly, sometimes we do want to know that a computer will stop running. But sometimes, particularly when it's controlling a physical device, we want to know that it does *not* stop; and we want to know what it's doing while it's running and how fast it's doing it.

3.1.2 Magnitude

The matrix in Table 3-1 makes clear some of the massive complexity that confronts digital system engineering. Suppose we ignore timing and consider just the progression of the system state through its successive transitions. Each such behavior of a system is represented accurately by one value of a matrix like the one in Table 3-1. How many such behaviors are there?

The matrix gives us a clear way to count them. For a binary computer system, there are just two possible values for each position in the matrix. Thus, the matrix has

$$2^{(m*n)} \tag{1}$$

possible values where *m* is its number of rows and *n* is its number of columns. Just the trivial matrix in Table 3-1 has 2^{35} possible values. Those two innocent looking numbers give more than $34 * 10^9$, over 34 billion possible behaviors for one absolutely trivial system.

For a real computer system, what are *m* and *n*? The number *m* is the total number of observable single bits in a system, and *n* is the total number of observable state transitions the computer makes over a given period of time. For modern computer systems, these numbers themselves are huge. How many bits are there in a 32-bit address space of 32 bit words? 2^{37}. This is over $137 * 10^9$. If a computer makes state transitions a rate of one every 100 nano-seconds, how many does it make in just one minute? $6 * 10^8$. How many different ways might such a machine behave in just one minute? More than $2^{(8 * 10^{19})}$. Think about that number. You can't. It's just too big. But is there any doubt that just one bit in just one state of just one of those behaviors could cause a harmful effect in the physical world? No. And, what are hardware manufacturers doing for us? Building bigger and faster computers *and* making them run in parallel!

3.1.3 Instability

Another major complexity that confronts the digital system engineer is the instability of digital systems. Just as a single bit in a single state can cause a harmful effect in the physical world, it also can cause the behavior of a digital system to progress in radically different ways.

If digital systems were stable, a small change in *s* would yield a small change in *M(s,t,n)*. Digital systems are about as far from being stable as one can imagine. The tiniest possible change in *s*, one single bit, can yield radically different results in *M(s,t,n)*. There is no continuity in *M(s,t,n)*. Computer programs in *s* must be expressed accurately to mega-bits of precision. Continuity is what many other engineering disciplines use to build safety factors into their systems. This safety net is not available to the digital system engineer.

3.2 Defining Acceptable Behavior

Once we have a mathematical equation that describes the physical behavior of a digital system, the full power of mathematics becomes available for us to use to manage this complexity. One of the first important things that becomes available to us is the ability to state a rigorous, mathematical definition of *acceptable* system behavior. We can state an acceptance test that distinguishes between acceptable and unacceptable behavior.

Suppose that we have a function, such as $M(s,t,n)$ above, that accurately describes system behavior. Once $M(s,t,n)$ is identified, we can choose an acceptance test function A that maps every system behavior into T if the behavior $M(s,t,n)$ is "acceptable" and into F if it is "unacceptable."

For predictability, it is important that the acceptance test be conclusive. That is, for every possible behavior, the test must tell us, "Yes, this behavior is acceptable" or "No, it is not." There is no room for a middle ground. To make a conclusive prediction, one must have a conclusive test. Therefore, an acceptance test must be chosen so that it produces with T or F for every possible behavior.

The full power of mathematics can be used to state an acceptance test. The test *may* contain mathematical objects and functions that do not have physical counterparts. For example, the acceptance test might use functions on real or imaginary numbers. The only practical requirement for predictability is that the test produce either T or F for every possible system behavior.[2]

Discovering a good acceptance test for various purposes may be difficult. In searching for acceptance tests that will demonstrate that a system has no behavior that causes a harmful effect, one must be especially careful. First, one must begin with a complete description of the system, so that one is certain that all possible behaviors are described. Then, one must find an acceptance test that definitely does not accept *any* harmful behavior. A test that rejects all harmful behaviors along with some others is sufficient, but it *must* reject *every one* of the harmful ones.

Once an acceptance test is chosen,

$$A(M(s,t,n)) = T \text{ for all behaviors} \qquad (2)$$

is a rigorous mathematical statement of acceptable system behavior. If this equation is true, then every behavior described by $M(s,t,n)$ is acceptable. Every possible behavior passes the acceptance test. None fail. The system has no unforeseen, unacceptable behavior. Equation 2 is a precise mathematical statement of "This system will behave acceptably."

I emphasize that the acceptance test is defined mathematically. It is a mathematical function. It is not a physical test. This is done quite purposefully for two reasons. The first is that it enables all of mathematics to be used in expressing acceptance tests. The second reason is that it avoids the necessity of physical measurements and the serious physical problems that may be associated with making them.

The first problem is that one must have a physical system to measure. With the acceptance test defined strictly mathematically, one can rigorously analyze the behavior of a digital system *before* it is physically constructed. This enables an engineer to evaluate alternative system structures objectively before actually

---

[2]These considerations have implications for formal specification languages. A specification language should provide a means of stating i) the equations that describe system behavior and ii) acceptance tests. For predictability, the language should require that all acceptance tests be conclusive, -- i.e., produce either T or F. Most current specification languages provide only a part of mathematics. Therefore, they limit the acceptance tests that can be stated. A notable exception is Z [Hayes 87] which is a notation based on elementary set theory.

building them. This has strong economic implications in both hardware and software engineering.

A second problem is that physical measurement must measure the right thing. One must know that the measurement instrument is making the right transformation from physical phenomena into mathematical objects. This, too, has strong implications for both hardware and software engineering. Measuring a machine state is not a simple task.

A third problem is that sometimes measurement perturbs the physical phenomena being measured. This is an important issue for acceptance tests that involve timing. How can one be sure that the instrumentation that measures the timing has no effect on timing?

A fourth problem is the repeatability of system behavior. A physical experiment is of no value in predicting future behavior unless the behavior is repeatable in time. For a physical measurement of the acceptance test to be useful as a predictor, there must be some amount of time u such that $A(M(s,t,n)) = A(M(s,t+u,n))$ where u is enough time to make and evaluate the physical measurement and to restart the system. There are a great many systems for which this is not true.

3.3 Demonstrating Acceptable Behavior

Equation 2 is the mathematical statement of acceptable system behavior. Given this mathematical formulation, the full power of mathematical analysis can be used to demonstrate that this equation is true.

3.3.1 Enumeration

Normally, one would not begin a discussion about proving a mathematical equation by discussing proof by enumeration. But that's what I'm going to do. The reason for this is that what is said in this section also applies to the physical testing of systems, and physical testing is the current state of engineering practice.

In principle, one can prove Equation 2 by evaluating it for all possible cases of system behavior. The practical problem with this is the overwhelming number of cases of possible system behavior, as was illustrated earlier.

As another example, suppose we were going to demonstrate the truth of Equation 2 for a 32 bit adder by physically testing its addition of all possible pairs of 32 bit numbers. This would require 2^64 (more than 1.84*10^19) additions. At a rate of, say 100 nano-seconds per test, this would take over 58,000 years. The adder won't last that long, and neither will the person waiting for the results! One might also wonder how those results are going to be checked and how long that might take? What then, of testing a system with more than a 64-bit state?

These calculations about the number of possible system behaviors tell us something important about the effectiveness of using enumeration to prove Equation 2. Enumeration can be effective *only if* the total number of possible system behaviors is small.[3] If the number is not small, some kind of additional mathematical analysis is required in addition to, or instead of, enumeration. We can be more precise about this, as follows.

[3]Strictly speaking, Dijkstra's often quoted comment "Program testing can be used to show the presence of bugs, but never to show their absence" [Dahl 72] is not true. The statement was made in the context of an adder example similar to the one given here. In that context, it is quite true.

Suppose we select some set of behaviors and use them as a "test pattern" by evaluating Equation 2 on them, and suppose that the results of those evaluations tell us whether the equation is true or false for some specific number of behaviors *k*. A measure of the effectiveness of this test pattern is *k* divided by the total number of possible system behaviors. If we ignore timing, the effectiveness of a test pattern is

$$k / 2^{(m*n)} \qquad (3)$$

where *m* is the number of observable state components and *n* is the number of observable state transitions.[4]

If we choose a test pattern of *k* distinct behaviors, and all it tells us is the value of Equation 2 for those *k* behaviors, then Equation 3 shows the basic futility of proof by enumeration as a means of demonstrating the truth of Equation 2. The numerator increases only linearly with the size of the test pattern, but the denominator increases exponentially with the product of the number of observable state components times the number of observable state transitions. In most practical situations, the effectiveness measure has to be incredibly small; proof by enumeration is just not effective.

Equation 3 also shows what is necessary to achieve effective proof by enumeration. Essentially, we have two choices; increase the numerator or decrease the denominator.

To increase the numerator, we must have a test pattern of *k* behaviors that determines the value of Equation 2 for many more than *k* behaviors. That only can be done in the presence of some additional kind of mathematical analysis of Equation 2. For proof by enumeration to be effective (in the presence of a large denominator in Equation 3), it *must* be done in conjunction with some additional kind of mathematical analysis.

There are instances of "powerful" test patterns in mathematics. For example, given the proof of an induction hypothesis, *H(n)->H(n+1)*, the test pattern 0 is 100% effective for proving that *H(n)* is true for all *n>=0*. As another example, for a polynomial *p(x)* of degree *k*, a test pattern of any *k+1* distinct points is 100% effective. Such examples give some hope of the viability of using mathematical analysis of Equation 2 to identify effective test patterns.

Now, let's consider the denominator of the effectiveness measure. The only way we can get a small denominator is with a small *m*n*. There are two ways to do this. Either the acceptance test must involve only a small number of state components and a small number of state transitions, or the system itself must have only a small number of states and transitions.

One very important instance of this is the functional behavior of low level hardware components. These are digital systems for which physical testing can be effective. One of the challenges of hardware engineering is to design systems so that small components and their interconnections can be tested individually, and also so that mathematical analysis can be used subsequently to deduce that the interconnected hardware satisfies a more complex mathematical equation (that cannot be exhaustively tested).

Finally, we should note that analysis of the physical environment in which a system operates also may be effective in reducing the total number of behaviors that need to be considered. The observations made above about numbers of behaviors tell us that, during its useful life time, a digital system will exhibit only a very tiny fraction of its total possible number of behaviors. If, somehow, we can isolate those particular behaviors, we may be able to increase the practical effectiveness of proof by enumeration.

[4] A test that is 100% effective is an "ideal test" in the sense of [Goodenough & Gerhart 75].

3.3.2 Probabilistic Analysis

Another mathematical technique that should not be overlooked is probabilistic analysis. Ideally, we want Equation 2 to be true for all possible system behaviors. However, for some systems, it may be sufficient to show that the probability of Equation 2 being true on any given behavior is sufficiently high. Conversely, it may be sufficient to show that the probability of some particularly harmful behavior is sufficiently low. Or, we might use probabilistic analysis in conjunction with other analysis. We might show that the probability of good behavior is high and prove that there are no harmful behaviors. All of these are plausible, mathematically rigorous approaches to gaining assurance that a system will behave within acceptable limits.

3.3.3 Deduction

Once we have a function, such as $M(s,t,n)$, that describes the physical behavior of a digital system, we can apply the full power of mathematical deduction to analyze the behavior of the physical system. We can use the techniques described above or many others.

There is one deductive approach that may be particularly effective in showing the absence of behaviors that cause harmful effects. It is based on the idea of "hazard analysis" [Leveson 86].

Suppose that a computer system has the potential to cause some particularly hazardous effect. We want to show that not one single behavior causes this effect. To start, we must begin with a complete description of the physical behavior of the system, $M(s,t,n)$. Then, instead of an "acceptance" test, let's think of a "hazard" test and identify a function $Hazard(M(s,t,n))$ that produces T for every possible behavior that can cause the hazardous effect.

Given the complete system description and the hazard test, attempt to prove that the system *does* have some behavior that satisfies the hazard test. Try to prove that there exists some s, t and n such that $Hazard(M(s,t,n))=T$. The attainment of a logical contradiction proves that no such behaviors exist. If a proof is obtained constructively, it will exhibit *some* of the hazardous behaviors.

Finally, let me close this section by returning to the issue of timing. In some applications, timing is the critical requirement of system behavior. I have mentioned the difficulties of physically instrumenting and testing a system for time dependent behavior. For acceptance tests that involve time, there is little hope of proving Equation 2 by enumeration. Thus, it seems that mathematical analysis is the only real hope we have for making accurate predictions about real-time behavior.

This position actually is supported by current engineering practice. When real-time performance is critical, system builders typically resort to assembly language (or lower) so they can get accurate timing information. Then, they add up instruction times or bounds on instruction times. This is simple (or maybe not so simple) mathematical analysis.

3.4 Scaling Up

Many who have read this far may be thinking that it is just not possible to deal with the equations that mathematically describe the physical behavior of a computer system. They are just too complex. I agree that they are complex, but my answer is that we have no alternative but to deal with them and deal with them mathematically. Those are the equations that *do* describe, very accurately, the physical behavior of a digital computer. The equations do not increase the complexity of the computer system. They reveal it. It is not the complexity of mathematics that confronts us; it is the complexity of computing. Mathematics is the best tool we have to master it.

3.4.1 Equations for Scaling

One of the main aspects of the complexity that confronts us is the sheer number of possible system behaviors. The structure of Table 3-1 and Equation 1 provide some valuable insight into how to manage this aspect of complexity. We need to reduce the number of rows and columns of the matrix.

The number of rows in the matrix is determined by the number of observable state transitions. The number of columns is the number of observable state components. To reduce the number of possible behaviors we have two choices; to reduce the number of rows or reduce the number of columns. And surely, we have to do both. But to know that a computer does not cause certain physical effects, we must be able to make a statement about every one of those state components and transitions, and those state components and transitions are real, physical things. Every row of that matrix describes a real, physical object, composed of atomic matter. Every column describes an event in real time. How can we possibly reduce the size of this matrix?

The answer is to control what can be *observed* about this real, physical object. We limit the observability of various rows and columns in the matrix.[5] This, however, must be done very carefully. We must be absolutely certain that the ability to observe the digital system is limited in exactly the way we think it is.

We cannot *change* the behavior of a physical digital system. But by controlling the way in which its physical behavior can be observed, we can create an illusion about its behavior. This illusion can reduce the observable number of possible system behaviors.

Figure 3-1: Virtual System

```
                 M2
           o----------->o
           |            ^
           |            |
 decompose |            | compose
           |            |
           v            |
           o----------->o
                M1
```

One way to do this is with a scaling equation such as

$$M2[s,t,n] = compose(s, M1[decompose(s,t),t,steps(s,t,n)], t, n) \qquad \textbf{(4)}$$

provided *ready(s,decompose(s,t),t)*

This equation describes a *virtual* system *M2* that is created by limiting the observation of a physical system *M1*. The function *steps* controls observation of the state transitions, and the functions *compose* and *decompose* control observation of the state components. The relation *ready* describes the conditions which are sufficient for *M1* to provide the *M2* illusion. The mathematical relationship between *M1* and *M2* is illustrated in Figure 3-1.

The virtual system *M2* is truly an illusion. *M1* describes the physical system that is composed of atomic matter. The behavior of that physical material is not changed in any way by creating the illusion. All we

[5] Limiting the observability of these rows and columns is similar in purpose to information hiding as described in [Parnas 79].

have done is create a new way of observing it. But we have done it in a way so that *M2* appears like another digital system -- i.e., *M2(s,t,n)* is a function in the same form as *M1(s,t,n)*.

By virtue of the scaling equation, *M2* now also provides an accurate description of the physical system. Therefore, it too, can be used as a basis for predicting accurately the physical behavior of the system. This kind of scaling up can be continued for as many levels as desired, *M1, M2, M3, ..., Mn*.

Examples of equations that scale a microprogrammable microprocessor, which is comparable in complexity to a PDP 11, from gates to machine language, can be found in [Hunt 87]. Equations that scale a machine language program of 750 16-bit words into a simple multi-tasking operating system kernel are described in [Bevier 87a]. How this kind of scaling can be continued upward to define the equations for a small subset of a high level programming language, Gypsy [Good 86], is described in [Bevier 87b].

But, let's be careful. A virtual system such as *M2* provides an accurate description of the physical system, but does it provide a complete description? This depends on the actual statement of the scaling equation -- i.e., on the definitions of the functions *compose*, *decompose* and *steps*. If these functions really do provide an illusion that hides some of the observable behavior described by *M1*, then *M2* definitely is *not* a complete description.

Does this incompleteness mean that a virtual system cannot be used to prove the absence of system behaviors that cause physical effects? Again, this depends on the particular scaling equation. In a physical digital system, some state components will be observable to the physical world and others will not. This is why we connect cables to computers. For the description of a virtual system to be effective in predicting the absence of a behavior that causes a physical effect, the scaling equation must provide a complete description *only* of those state components that are observable to the physical world. But, it *must* describe accurately *every* state component that *is* physically observable.

3.4.2 Objectives for Scaling

Scaling equations can be used to scale an accurate mathematical description of a physical digital system from gates to micro code to machine language to an operating system to a high level programming language. It is useful to consider what that particular scaling needs to accomplish to reduce the apparent complexity of the physical system. Let us return to the issue of limiting the observability of the rows and columns in Table 3-1.

The length of a row is the number of observable state transitions. Reducing this number is what we attempt to do with higher level control structures. We attempt to deny the observability of certain intermediate state transitions. Sometimes we succeed. Often we do not.[6]

The length of a column is the number of observable state components. Reducing this number is more of a challenge.

We can start by making those bits that are not relevant to a particular computation unobservable. This is the strategy of *separation*. We attempt to partition the system state so that, over a particular period of time, the behavior of one set of state components is completely independent of another set. For example, if we can partition the state into two completely different sets of size *p* and *q* so that the behavior of those two partitions have absolutely no effect on each other, then we only have 2^(p*n)+2^(q*n) possible

---

[6]Control structures that make visible only those states that satisfy known mathematical relationships provide a particularly effective basis for mathematical analysis [Dijkstra 76, Hoare 85].

behaviors instead of $2^{(p*n)}*2^{(q*n)}$. Divide and conquer works! This separation strategy is familiar in operating systems. It also is achieved in higher level programming languages with varying degrees of success.

What else can we do to reduce the number of observable state components? We can group sets of *n* bits together in ways so that they have less than 2^n values. This means creating equivalence classes. If we take the five state components in Table 3-1 and group them into an equivalence class with just 20 values, instead of 32, the number of possible behaviors reduces from $32^7 (=3.43*10^{10})$ to $20^7 (=1.28*10^9)$. Again, this effect is achieved in higher level programming languages with varying degrees of success.

Truly effective, mathematically rigorous solutions to these problems have yet to be found, but they indicate important objectives for the design of operating systems and higher level programming languages. We can scale up, but scaling needs to have a purpose. Reducing apparent complexity is a good one.

Chapter 4

ENGINEERING DIGITAL SYSTEMS

The scientific and mathematical foundations that are needed to advance the practice of digital system engineering are simple to state. Digital system engineers need equations that accurately and completely describe the behavior of the physical system they are going to install in the real world of atomic matter. They need equations that describe the higher level virtual systems that they will be using on that computer. They need mathematics to apply the equations.

4.1 Future Practice

Let's take a look at what digital system engineering based on these foundations might be like.

The primary value of these foundations is that it enables a digital system engineer to analyze the behavior of a system *before* it actually is constructed. This provides a rigorous, objective way of making informed design and implementation decisions. An engineer can make much better informed decisions because s/he can determine accurately what the effects of various decisions will be. Such information is essential to building a system to given specifications within given resource limitations.

Digital system engineering based on these foundations would involve many applications of the following three steps:

1. Identify a mathematical function, such as *M(s,t,n)*, that describes the behavior of the physical or virtual system in question.
2. Identify an acceptance test *A(M(s,t,n))* that conclusively distinguishes acceptable from unacceptable behavior.
3. Mathematically demonstrate that system behaviors pass the acceptance test.

The application of these steps would begin in the construction of computer hardware. The main goals of the hardware engineering would be to provide, at reasonable cost, digital systems in which electronic behavior conformed to a particular mathematical function, such as *M(s,t,n)*, *and* to tell the rest of the world exactly what that function is. Hardware engineering also would be responsible for developing methods for testing and maintaining the physical computing system so that its behavior will conform to the mathematical function.

Operating systems and programming languages also will need to identify the mathematical functions they provide. Only in this way can a software engineer build programs in which computer behavior can be predicted accurately. This will require a rigorous mathematical definition of operating systems and programming languages, and it will require rigorous mathematical demonstrations that the scaling equations are satisfied.

Based on these foundations, a software engineer will be able to make rigorous, objective decisions about alternative software design and implementation questions. Making these decisions will be based on the mathematical analysis of alternative program structures. This analysis will be based on mathematical evaluation rather than on physical testing. This analysis will be performed on the program text, possibly with the aid of mechanical tools that amplify the engineer's ability to perform the mathematical analysis and reduce the probability of human analytical error.

There is much research and development that needs to be done to reach the point where digital system engineers routinely can apply this kind of mathematics to constructing systems. To put it bluntly, the mathematical foundations and methods still need to be developed. Effective means for applying them in engineering practice also need to be developed, implemented and adopted. We also need to start thinking about designing hardware, operating systems and programming languages to facilitate this kind of mathematical analysis. We need to discover the most effective kinds of mathematical functions upon which to base that design. We also need to discover acceptance tests that distinguish acceptable from unacceptable behavior for various kinds of systems.

4.2 Current Practice

The current practice of digital system engineering is far removed from practice that is based on science and mathematics. It is instructive to think about how far removed it really is. Let me pose some questions.

What kind of information is there that describes the physical behavior of the computer systems that are operating today and causing effects in our physical environment? Does this information describe exactly and completely what next state will be produced from the current state and how long it will take? Does the accuracy and precision of this information approach that of a mathematical function such as $M(s,t,n)$?

Consider the information about computer system behavior that a hardware engineer typically provides to a software engineer. What is it? It's a reference manual. How close does that information come to a mathematical function that is a complete description of the physical behavior of the system? If we're going to make accurate statements that hardware will not cause harmful effects, we need to know exactly what happens to every observable bit in the system on every observable state transition, and we need to know how long it takes. Does the hardware engineer even know what mathematical function the hardware *does* provide to the software engineer? Doesn't this leave the software engineer in the same position as having first to *discover* Ohm's Law before s/he can even begin to think about *applying* it?

What kind of information does a software engineer have? More manuals. Programming language manuals! These languages range from the micro code level on up to the "Higher Order Languages." How many of these manuals describe exactly the next observable state the physical machine will produce from the current one? How many tell how long it will take? For how many programming languages is there a mathematical function that describes accurately the behavior of its programs on a physical machine?

How can a software engineer produce a program that makes a computer behave in a predictable way if s/he doesn't know how the program will make the computer behave? We're going to use Ada to program systems with predictable real-time behavior. Right!

How many engineers identify any kind of objectively measurable acceptance test criteria for a system? Do the criteria distinguish conclusively between acceptable and unacceptable for every possible system behavior? How can anyone predict that the future behavior of a computer system will be acceptable without identifying the limits of acceptability?

How many engineers use any kind of mathematical analysis to demonstrate that their systems pass an acceptance test? How can they, without mathematical functions that describe system behavior and an acceptance test?

Current engineering practice is based primarily on physical testing. Exhaustive testing is practical only for the tiniest of systems. On larger systems, how often is any kind of systematic statistical testing done?

Software engineering practice is in a very immature state. Our expectations far exceed our capabilities. In many cases, we can't even be precise about what our expectations are. Often, even though we may have 25 pounds of requirements documentation, we simply do not know what we want a computer system to do! So, because it's so much fun to build these systems and watch them do something, we just build them and see if they do something we like.[7] Often they do. However, sometimes they also do something we did not foresee and cause harmful effects.

This sandbox engineering practice can carry us only so far. When applied to systems that can cause major effects in our physical environment, it's only a matter of time before it carries us into real, physical harm. For some, it already has.

[7]One of the things that encourages this kind of practice is that the raw materials that programs are composed of are free! No one charges for assignment statements. Just get a new one out of an endless warehouse. The only thing that's going to cost anyone any money is "just" the cost of someone's labor to hook it up with other assignment statements and stuff. Most other engineer fields don't have this "blessing." I suspect one of the reasons the Empire State building was built only once was that someone had to pay for concrete and steel.

Chapter 5

CONCLUSION

Science and mathematics are not required to engineer physical systems, but they sure do help! Pre-historic engineers built bridges by laying stones and logs across a stream. But it took science and mathematics to build the Golden Gate. Benjamin Franklin was able to go fly a kite and discover electricity. But, how many electrical systems were engineered successfully without Ohm's Law and the mathematics to apply it?

What is the Ohm's Law of digital system engineering? It is the mathematical function that describes the physical behavior of a digital computer. From science, we need to identify the mathematical functions that describe accurately and completely the physical behavior of computers. From mathematics, we need the ability to apply these functions effectively. Without these foundations, the capabilities of digital system engineering will remain very limited.

A good engineer must do the best s/he can with what s/he has. An engineer builds the best possible product from the resources that are available. A wise engineer will not try to engineer systems beyond the limits of available technology.

Stones and logs have carried many a person safely from one side of a stream to another. For some purposes, that was, and still is, good, sound technology. The physical laws of nature have not changed. But, when pushed beyond its limits, to try to get across some water that was too swift or too wide or too deep, this technology presents a real, physical danger.

Digital system engineering is already pushing beyond the limits of available technology. It is limited by the absence of scientific and mathematical foundations. Without them, digital systems will continue to have unpredictable behavior that will cause physical harm. What harm will be caused will depend on what physical effects a digital system is allowed to control.

Acknowledgements

Many of the observations I have made here have been made by others. What I have done is interpret them from the view that computing is a physical science, just like physics or chemistry. It matters not that the object of study, a computer, is the product of a manufacturing process. A computer is still a part of the physical world of atomic particles. The laws of nature still apply. The manufacturing process does not suspend them. The most effective way we have to predict the behavior of the physical world is the traditional scientific method. Discover the mathematical equations that describe the object of study, and apply them to predict its physical behavior.

I am particularly indebted to J Moore, Warren Hunt, Bill Bevier and Bill Young for surrounding me with a mountain of proved mathematical statements about computer hardware, operating systems and programming languages, and to Bob Boyer and J Moore's wonderful deduction machine [Boyer & Moore 88] that was used to build that mountain. The scaling equations that I have presented primarily are a result of that work.

I especially thank Bob Morris for insisting that I think about silicon. This was the catalyst from which my view of computing as a physical science emerged.

This paper also appears in the Proceedings of COMPASS '88 (3rd Annual Conference on Computer Assurance).

References

[Bevier 87a] William R. Bevier.
A Verified Operating System Kernel.
Technical Report CLI-11, Computational Logic, Inc., October, 1987.
Also Ph.D. Thesis, The University of Texas at Austin, 1987.

[Bevier 87b] William R. Bevier, Warren A. Hunt, Jr., and William D. Young.
Toward Verified Execution Environments.
Technical Report CLI-5, Computational Logic, Inc., 1987.
Also appears in "Proceedings of the 1987 IEEE Symposium on Security and Privacy".

[Boyer & Moore 88]
Robert S. Boyer, J Strother Moore.
The User's Manual for A Computational Logic.
Technical Report CLI-18, Computational Logic, Inc., February, 1988.

[Dahl 72] E.W. Dijkstra.
Notes on Structured Programming.
Structured Programming.
Academic Press, 1972.

[Dijkstra 76] E.W. Dijkstra.
A Discipline of Programming.
Prentice-Hall, 1976.

[Good 86] Donald I. Good, Robert L. Akers, Lawrence M. Smith.
Report on Gypsy 2.05 - January 1986
Computational Logic, Inc., 1986.

[Goodenough & Gerhart 75]
J. B. Goodenough and S. L. Gerhart.
Toward a theory of test data selection.
IEEE Trans. on Software Engineering 2, June, 1975.

[Hayes 87] Ian Hayes (Editor).
Specification Case Studies.
Prentice-Hall, 1987.

[Hoare 85] C. A. R. Hoare.
Programs are Predicates.
Mathematical Logic and Programming Languages.
Prentice Hall International Series in Computer Science., 1985.

[Hunt 87] Warren A. Hunt, Jr.
The Mechanical Verification of a Microprocessor Design.
Technical Report CLI-6, Computational Logic, Inc., 1987.
See also "FM8501: A Verified Microprocessor", Ph. D. Thesis, The University of Texas at Austin, 1985.

[Leveson 86] Nancy G. Leveson.
Software Safety: Why, What, and How.
ACM Computing Surveys 18,2, June, 1986.

[Parnas 79] D.L. Parnas.
Designing Software for Ease of Extension and Contraction.
IEEE Transactions on Software Engineering 5-2, March, 1979.

Table of Contents

List of Figures

List of Tables