Mathematical Forecasting

Donald I. Good

Technical Report 47 September 1989

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

This report is to appear in a forthcoming American Institute of Aeronautics and Astronautics (AIAA) Progress Series book on Aerospace Software Engineering.

Acknowledgements

I thank William R. Bevier, Robert S. Boyer and William D. Young for their helpful criticism of early drafts of this manuscript.

This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 6082. The views and conclusions contained in this document are those of the author(s) 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

In aerospace and in many other fields of engineering, it is common practice to forecast the behavior of a physical system by analyzing a mathematical model of it. If the model is accurate and the analysis is mathematically sound, forecasting from the model enables an engineer to preview the effect of a design on the physical behavior of the product. Accurate mathematical forecasting reduces the risk of building latent design errors into the physical product. Preventing latent design errors is an important part of successful engineering. If a product contains a latent design error, it can cause operational malfunction. When a latent design error is detected, removing it requires backtracking in the product development cycle. This backtracking can consume large amounts of time, money and human resources.

Although digital computers now are embedded as operational components in many aerospace and other physical systems, capabilities for mathematically forecasting the physical behavior of computer programs are only now beginning to emerge. Without these capabilities, latent design errors in computer programs frequently go undetected until late in program design or until the program is tested or even until it is in actual operation. This increases the dual risks of operational malfunction and high resource consumption caused by developmental backtracking. When computers are embedded in other physical systems, these systems inherit those risks. Mathematically forecasting the physical behavior of computer programs can reduce these risks for software engineering in the same way that it does for aerospace and other fields of engineering. Present forecasting capabilities for computer programs still are limited, but they are expanding; and even the present, limited capabilities can be useful to a practicing software engineer.

1. Mathematics in Engineering

Imagine, if you can, aerospace engineering without mathematics. Suppose that a new anti-mathematics virus suddenly infected all aerospace engineers, making them unable to apply any known mathematical models of the physical world. Suddenly unavailable are all of the mathematical models of the physical laws of motion, gravity, aerodynamics, thermodynamics, ... all of them. What state would aerospace engineering be in? In a word, "Grounded!"

Let's continue this fantasy -- or perhaps nightmare -- a bit further. While aerospace engineers remain mathematically crippled by this virus, what might be done to advance the state of engineering practice? One might look for ways to improve the product development cycle to get better product quality and engineering productivity. One might implement better ways to state complete and consistent product requirements, automate parts of the design process, produce more accurate and detailed documentation, implement more careful configuration management procedures, improve methods for conducting physical experiments, use better manufacturing methods, improve product quality assurance methods, reuse as much previous good work as possible, improve engineering management methods, raise certification standards for engineers, etc. All of these things could help to improve the state of the engineering practice. But without applying accurate mathematical models of physical phenomena to preview product designs, the current, advanced level of aerospace engineering practice would not be possible.

In his introductory textbook on flight [Anderson 89], John D. Anderson, Jr. begins his chapter on basic aerodynamics with the following two quotations:

> Mathematics up to the present day have been quite useless to us in regard to flying. -- From the fourteenth Annual Report of the Aeronautical Society of Great Britain, 1879.
>
> Mathematical theories from the happy hunting grounds of pure mathematicians are found suitable to describe the airflow produced by aircraft with such excellent accuracy that they can be applied directly to airplane design. -- Theodore von Karman, 1954.

The application of mathematics describing the physical laws that pertain to heavier-than-air flight has played a key role in advancing aerospace engineering, and it has played a similar role in advancing many other fields of engineering. The application of mathematical descriptions of physical phenomena enables engineers to predict accurately the physical behavior of products manufactured from their designs.

Without this mathematical forecasting ability, engineers must rely much more on physical experimentation guided by previous experience. To conduct a physical experiment, time, money and human resources must be used to complete a design and manufacture the physical object of the experiment (even if the object is just a prototype). If the experiment detects a design error in the product, the error must be corrected, the product redeveloped and the experiment repeated. With good fortune, the redevelopment may consume less resources than the one before it.

With mathematical forecasting, engineers can reduce the risks of pursuing a poor design. Mathematical forecasting in aerospace engineering does not replace the creativity of human invention, it does not guarantee perfection, and it does not eliminate the need for physical experiments. What it does provide is a way for early detection of design flaws without requiring the manufacturing of a product or even the completion of a design. It is much cheaper to calculate that a rocket engine will not produce enough thrust to achieve escape velocity than it is to design one that won't, manufacture it, and watch it fail!

Now imagine, if you can, software engineering without mathematics. This requires no stretch of the imagination; it happens every day. It is the common, accepted practice.

But software engineering also results in a physical product, just as aerospace engineering does. The product that results from software engineering is a computer program. A computer program is a physical, control mechanism. It consists of physical switches within a computer. Just as a rudder or a wing controls the dynamic, physical behavior of an airplane, these switches control the dynamic, physical behavior of the computer. These switches control the sequence of physical, electronic states that occurs within the computer.

Today almost all software engineering is done without the benefit of mathematically forecasting what effects these switches will have on the computer they control. The current practice of software engineering is dominated by physical experimentation. The result is not much different than if aerospace engineering were done that way, just as when our aerospace engineers were stricken with the anti-mathematics virus. There is a high risk of latent design errors. Encountering them in operation causes program malfunction. Removing them requires developmental backtracking and the consequential consumption of time, money and human resources.

Unlike airplane development, however, almost all resources consumed in program development are consumed in the design stage. The resources required for manufacturing are negligible. Indeed, one of the most remarkable characteristics about computer programs is their manufacturing process. An airplane is built by assembling a collection of physical parts. This requires a large amount of time, labor, energy and materials. A computer program also is built by assembling a collection of physical parts. But these parts are switches that already exist in the computer, and all that remains to be done to "manufacture" the program is to set the switches. In our present day and time, this normally is done by a process called "loading" the program. The manufacturing of even a very large program requires just a tiny amount of time, labor and energy, and it requires no new materials! What an amazing manufacturing bargain for a physical, control mechanism![1]

For all practical purposes, software costs are design costs. It is incredibly easy and inexpensive to manufacture *a* computer program. What is hard is to manufacture the *right* one. Software costs are the design costs of deciding which program to manufacture. Without effective means to forecast the effects of design decisions, programs contain latent design errors which cause operational malfunction and developmental backtracking.

Current software engineering practice is dominated by physical experiment; and, as in other fields of engineering, there are steps that can be taken to improve this practice. But the effectiveness of these steps will be limited until software engineers can forecast accurately the effects that a program design will have on the dynamic, physical behavior of the computer it is controlling.

Applying mathematics to predict the behavior caused by a computer program is *not* a *new* idea. It has been put forward by von Neumann [von Neumann 61], McCarthy [McCarthy 63], Naur [Naur 66], Floyd [Floyd 67], Dijkstra [Dijkstra 68], [Hoare 69] and others. But, as we enter the 1990's, applied mathematics has not been incorporated into the actual practice of software engineering, and both software producers and consumers alike continue to suffer the consequences.

The following sections sketch how the mathematics of recursive functions theory can be applied to forecast program behavior. The approach will be familiar to engineers from other fields. First one develops an accurate mathematical model of the sequence of electronic states that programs will cause to occur in the physical computer. Then one uses this model to analyze the physical effects that a particular program design will cause. This analysis can be done before the program is built or even while it is only

[1]This bargain is a two-edged sword. If it cost as much to manufacture a program as it does an airplane, software engineers probably would be much more highly motivated to preview the effects of their designs before putting them in operation.

partially designed. An analysis can be done to show that the physical behavior of a particular program will satisfy specific behavioral requirements. This mathematical forecasting reduces the risks of operational malfunction and high resource consumption caused by developmental backtracking.

The reader is forewarned that the models for computer programs are expressed in terms of discrete mathematics rather than the continuous mathematics that is common in other engineering fields. However, the engineering benefits of mathematical forecasting are the same in both cases.

2. Mathematics for Programs

To forecast the physical behavior of a computer program, we need an accurate mathematical description of the physical states it will cause to occur. To do that, we need mathematical objects that describe programs and states. With these objects, the sequence of states that a program causes to occur in a computer can be described very accurately by recursive functions.[2]

The line of discussion in the following sections first summarizes how mathematical sequences and mappings can be used to describe programs and states respectively. Then it illustrates how recursive functions on these mathematical objects can define a model of program behavior. Next it shows how that model can be applied to make some forecasts about a simple computer program that controls a physical device on an airplane, and it points out some of the engineering benefits that result. The discussion concludes by addressing the relation between mathematical forecasting and program testing and the important issue of the accuracy of forecasts made from a mathematical model.

2.1 Programs

A computer program can be described accurately by a sequence of symbols from a programming language. The language might be a machine language, in which programs are described by a sequence of zeros and ones, or it might be an assembly language or a higher order language. Whatever the language, it defines various acceptable sequences of symbols, and each such sequence describes a computer program. These sequences of symbols are perfectly acceptable mathematical objects.

To make this discussion concrete, let's focus on the sequence of symbols in Figure 1 from the Gypsy language [Good 86].[3] The sequence of symbols that begins with the symbol **`procedure`** *describes* a program. For future reference, let's denote this sequence by **`m0`**,

```
m0 = procedure validator ... end
```

This sequence of symbols is a mathematical object. Its first element is **`procedure`**, its second is **`validator`**, and its last is **`end`**.

A computer program is a physical mechanism that controls the sequence of physical states that occurs within a computer. It is important to make a careful distinction between the *physical mechanism* that is being described and the *mathematical object* that describes it. There is plenty of room for confusion because it is customary to refer to the sequence of symbols **`m0`** as a "program." But this sequence of symbols is not a *physical mechanism* that controls the sequence of physical states in a digital computer. The physical control mechanism consists of physical switches inside the computer. Just as the number

[2]For this discussion, we restrict attention to synchronous behavior.

[3]This probably unfamiliar language is used in this discussion because there is a relatively concise mathematical model for the behavior of the programs it describes. A major part of this model is stated in Figure 4.

35,000 is a mathematical object that might describe an altitude of an airplane, the sequence **m0** is a mathematical object that describes a physical control mechanism. This is more clear when one thinks about the sequence of bits that **m0** compiles into. That sequence of numbers (zeros and ones) is another mathematical object that describes the settings of the physical switches that comprise the computer program. This is the description from which the program is manufactured -- i.e., from which the physical switches are set. Both the sequence of symbols of **m0** and the sequence of bits it compiles into are different mathematical objects that describe the same physical control mechanism.

In Gypsy, the sequence **m0** describes a control mechanism by describing how it, as a composite mechanism, is made up of component mechanisms. The mechanism described by **m0** is composed of the component mechanisms shown in Figure 2. These components are assembled into a composite by means of the compositions shown in Figure 3. For example, composition 5 composes the two mechanisms **leave** and **update_cmd(c, y, z, e)** into the new mechanism

```
if eox then leave
else update_cmd(c, y, z, e)
end
```

Composition 3 composes **get_cmd(c, eox, x)** with this to give

```
get_cmd(c, eox, x);
if eox then leave
else update_cmd(c, y, z, e)
end
```

Composition 4 gives

```
loop
  get_cmd(c, eox, x);
  if eox then leave
  else update_cmd(c, y, z, e)
  end
end
```

Continuing in this way, sequence **m0** describes how the composite mechanism **validator** is made up of its components. Components 1-4 in Figure 2 are primitive Gypsy mechanisms (ones provided by the implementation of the Gypsy language), and components 5-9 are non-primitive mechanisms that must be composed by the software engineer (ultimately also from Gypsy primitives).

2.2 States

The physical state of a digital computer can be described accurately by a mathematical mapping from a domain of names into a range of values. For example, the mechanism described by **m0** controls a state that can be described by a mapping **s** with four components,

```
s = { (x, x0), (y, y0), (z, z0), (run, run0) }.
```

The first component of **s** is the pair **(x, x0)** with name **x** and value **x0**. The name **x** is the name of the formal parameter **x** of **validator**. The components **y** and **z** are similar to **x**. For **validator**, **x0**, **y0** and **z0** are sequences of elements of type **a_cmd**. At this level of program design, the type of these elements is not relevant. The **run** component of **s** is a special component of the state of every Gypsy program. Its value, **run0**, is either **normal** or **leave**.

The mathematical model in the next section uses two functions on mappings, component selection and alteration.

- **select(s,n)** is the value part of the pair in state **s** with name **n**. Instead of `select(s,n)`, we can use the more concise notation `s[n]`.

- `alter(s,n,v)` is the state which is identical to `s` except that the pair with name `n` has value `v`. That is,

```
alter(s, n, v)[p] = if p=n then v else s[p]
```

For example, given the mapping `s` above,

```
s[x] = x0

alter(s,run,normal) = { (x, x0), (y, y0), (z, z0), (run, normal) }

alter(s,u,0) = { (x, x0), (y, y0), (z, z0), (run, run0), (u, 0) }
```

2.3 Models

With sequences of symbols to describe programs, and mappings to describe states, the sequence of states a program causes to occur within a computer can be described accurately by recursive functions. Just as partial differential equations describe the dynamic, physical behavior of fluids, recursive functions describe the dynamic, physical behavior of digital computers. They describe the physical sequence of states that is caused by a program.

Figure 4 illustrates a recursive function `g(m,s)` that describes the physical states caused by programs described by sequences of symbols from the Gypsy language. The function `g(m,s)` describes the final state that results from applying the mechanism (described by the sequence of symbols) `m` to the initial state (described by the mapping) `s`.[4] The model in Figure 4 has one equation for each kind of primitive mechanism and one for each kind of composite mechanism. The terms `<T>` in these equations are syntactically well-formed sequences of symbols of the appropriate kind.[5]

To get some feel for the nature of these equations, consider applying the sequential composition of the mechanisms `var e:a_context` and `reset(y)` to an initial state

```
s = { (x, x0), (y, y0), (z, z0), (run, normal) }.
```

The final state `v` produced by the composite mechanism is

```
v = g(var e:a_context ; reset(y), s)

  = g(reset(y), g(var e:a_context, s))              Eq. 8

  = g(reset(y), s1)                                 Eq. 2

    where s1 = alter(s, e, default(a_context,D))

  = copyout(reset, y, s2, s1, D)                    Eq. 7

    where s2 = g(D[reset], copyin(reset, y, s1, D))
```

[4]A more detailed description could be given by having `g(m,s)` produce the entire sequence of states that results from applying `m` to `s`; but having `g(m,s)` produce just the final state is sufficient for many analytical purposes.

[5]The existence of a function `g(m,s)` that satisfies the equations in Figure 4 is an important issue that is not discussed here.

First the mechanism **`var e:a_context`** is applied to **`s`** to produce the state[6]

```
s1 = { (x, x0), (y, y0), (z, z0), (run, normal),
       (e, default(a_context,D)) }.
```

Then the mechanism **`reset(y)`** is applied to **`s1`** to produce **`v`**. State **`v`** is the result of calling procedure **`reset`** with actual parameter **`y`** on state **`s1`**. If we assume that the **`reset`** mechanism has the behavior stated for it in Figure 8, then by virtue of the **`copyin`** and **`copyout`** functions (whose definitions are not shown in Figure 4),

```
v = { (x, x0), (y, <>), (z, z0), (run, normal),
      (e, default(a_context,D)) }.
```

The state **`v`** is the same as **`s1`** except that the value of the **`y`** component is the empty sequence **`<>`**.

The function **`g(m,s)`** provides some important insight into the difficulty of software engineering. The function has no continuity! Typically engineers in other fields rely on continuity to help predict system behavior. If the system is changed just a little, its behavior will change just a little. This continuity often is used to design safety factors into a system. Not so in software engineering! Small changes in either **`m`** or **`s`** can produce dramatic changes in **`g(m,s)`**. Because of this, the behavior of a computer program can be quite counter-intuitive and therefore even more difficult to predict. Consequently, the need for accurate mathematical forecasting is even greater in software engineering than in some of the more traditional engineering fields.

2.4 Forecasting

Some of the important benefits that mathematical forecasting can bring to software engineering can be illustrated by a simple example. Suppose that a master and a slave computer are to be embedded on an airplane with the master sending commands to the slave. The slave uses the commands to control some physical device. But, for some reason, the slave cannot fully trust the master to issue a sensible sequence of commands. Therefore, perhaps for the safety of the airplane, the slave needs to issue only valid commands to the device it is controlling, and ones that are invalid should be returned to the master. What is needed is a validator program such as the one described in Figure 1.

One of the first benefits that comes from having a mathematical model of the physical behavior of a program is the ability to state precise behavioral requirements for it. The model has mathematical objects that describe programs and states. Behavioral requirements of the program can be stated as mathematical

---

[6]The **`D`** that appears in the equations of the model is a dictionary which is a mapping. It maps from names, which are symbols, into values. The values are the sequences of symbols that comprise the type and procedure declarations contained in the body of Gypsy text being interpreted. For example, from the text given in Figure 1, **`D`** would have the pairs

```
(a_cmd,     pending)

(a_cmd_seq, sequence of a_cmd)

(a_context, pending)

(validator, procedure validator(var x, y, z:a_cmd_seq) =
            begin
              ...
            end)
```

and **`D[validator]`** would be the same sequence of symbols as **`m0`** in Section 2.1. For the examples in this discussion, **`D`** also contains components for the Gypsy procedure declarations given in Figures 6-9.

relations on these objects -- for example, as a relation **`R(s,g(m0,s))`** between the initial and final state of **`m0`**. This provides a very precise way of stating requirements; and once stated in this form, the requirements **`R(u,v)`** themselves also can be the object of rigorous mathematical analysis.

For example, the required behavior of **`validator`** is stated in the "**`BEHAVIOR:`**" part of Figure 1 as a relation **`R(u,v)`** which is

```
v[y] = all_valid(x0) and v[z] = all_invalid(x0)
```

The state **`u`** is the initial state that the **`validator`** is applied to, and **`v`** is the final state that it produces. The state **`u`** has components **`(x,x0)`**, **`(y,y0)`** and **`(z,z0)`**. The names of these components, **`x, y`** and **`z`**, are the names of the formal parameters of **`validator`**. The values of these components, **`x0, y0`** and **`z0`**, are the initial values of the corresponding actual parameters. The state **`u`** also has a **`run`** component with value **`normal`**. Whenever the **`validator`** mechanism is applied, the **`copyin`** function in the Procedure Call Primitive equation in Figure 4 creates a state of this form, and the **`validator`** mechanism is applied to it to produce the final state **`v = g(D[validator], u)`**. **`D[validator]`** is the sequence of symbols that describes the validator mechanism, **`procedure validator(...) = begin...end`**. **`D[validator]`** is the same as the sequence **`m0`** cited in Section 2.1.

The required behavior of the **`validator`** mechanism is that the relation **`R(u,v)`** stated above be satisfied *for every* value of **`u`** where **`u`** and **`v`** are defined as in Figure 1. The value **`x0`** of the **`x`** component of **`u`** is assumed to be the sequence of commands that the slave computer receives from the master. The **`y`** component of the final state produced by the **`validator`** mechanism must be the sequence of all valid commands in **`x0`**. These are the commands that the slave computer can send to the device it is controlling. The **`z`** component of the final state must be the sequence of all invalid commands in **`x0`**. These are the commands that the slave will return to the master computer.

Similarly, the required behavior of the component mechanisms of **`validator`** is stated in Figures 6-9. Stating these behavioral requirements requires the use of several new mathematical functions, such as **`all_valid(x)`** and **`all_invalid(x)`**, which are **`not`** part of the model of the program behavior. These functions are shown in Figure 5. They do not appear anywhere in the mathematical model of the behavior of Gypsy programs. Instead, they are functions of the problem domain. They help describe the problem that the **`validator`** mechanism is expected to solve.

For example, functions **`all_valid(x)`** and **`all_invalid(x)`** define precisely what is meant by sequences of valid and invalid commands. Both are defined in terms of the function **`valid(c,x)`**. This function defines what it means for a command **`c`** to be valid with respect to a sequence of commands **`x`**. The functions **`all_valid(x)`** and **`all_invalid(x)`** use **`valid(c,x)`** to determine the validity of each command **`c`** with respect to the sequence of commands that preceded it. This allows the **`validator`** to review each command it receives with respect to all preceding commands.

The requirements of the **`update_cmd`** component refer to two additional functions, **`context(x)`** and **`check(c,e)`**, with the property stated in Figure 5 that

```
valid(c, x) = check(c, context(x))
```

These functions are introduced to allow the **`update_cmd`** mechanism to operate efficiently. The function **`context(x)`** gleans from the sequence of commands **`x`** only that information needed to make the validity decision on the command **`c`**. Thus, **`update_cmd`** can operate just by retaining this context information, and it does not need to retain the full history of all commands it receives.

Figure 5 does not provide definitions for the functions **`valid(c,x), context(x)`** or `check(c,x)`. These definitions are not needed to show that `validator` satisfies its requirements. All that is needed is to assume that they exist, and that they satisfy the relation stated for them. These definitions, or at least additional properties of `context(x)` and `check(c,x)`, would be needed to analyze `update_cmd`, but they are not needed to analyze `validator`.

By analyzing `g(m0,s)` where `m0=D[validator]`, it is possible to show that

- *if* the problem domain functions `valid(c,x), context(x)` and `check(c,x)` satisfy their assumed relation and
- *if* the component mechanisms of `validator` satisfy their stated requirements,
- *then* the `validator` mechanism will satisfy its requirements.

Here is a sketch of the analysis. The `validator` can be decomposed as follows:

```
m0 = procedure fp = m1

fp = validator(var x, y, z:a_cmd_seq)

m1 = begin m2 ; m3 end

m2 = var c  :a_cmd;
     var eox:boolean;
     var e  :a_context;
     reset(y);    reset(z);
     init_context(e)

m3 = loop m4 end

m4 = get_cmd(c, eox, x);
     if eox then leave
     else update_cmd(c, y, z, e)
     end
```

As stated in the behavioral requirements of `validator`, let

```
u  = { (x, x0), (y, y0), (z, z0), (run, normal) },

v  = g(m0, u).
```

First let's show that `v[y]=all_valid(x0)`. A sketch of the analysis follows, and the rationale is discussed below.

```
v[y] = g(m0, u)[y]

     = g(m3, g(m2, u))[y]

     = g(m3, s1)[y]

       where s1 = { (x, x0), (y, <>), (z, <>), (run, normal),
                    (e, context(<>)) }

     = s1[y] @ all_valid(s1[x])

     = <>    @ all_valid(x0)

     = all_valid(x0)
```

Showing that `s1` is the state given above requires knowing the definitions of `copyin` and `copyout` in the Procedure Call Primitive equation and assuming that `reset` satisfies its requirements. The next step above uses the following general property of the **m3** loop:

```
g(m3, s)[y] = s[y] @ all_valid(s[x]).
```

The effect of the **m3** loop is to append to **s[y]** the sequence of all the valid commands in **s[x]**. Under the assumptions that **get_cmd** and **update_cmd** satisfy their requirements and that **valid(c,x), context(x)** and **check(c,x)** satisfy their assumed relation, this property can be shown by a straightforward induction on the length of **s[x]**. The analysis sketched above shows that the **v[y]=all_valid(x0)** requirement is satisfied, and a similar one shows that the **v[z]=all_invalid(x0)** requirement is satisfied. Thus, the **validator** satisfies its behavioral requirements.

This preceding analysis shows that the **validator** mechanism satisfies its requirements *for every* possible initial state **u** that it might be applied to, *provided* that its component mechanisms satisfy their requirements and that the problem domain functions **valid(c,x), context(x)** and **check(c,x)** satisfy their assumed relation. This illustrates the power of mathematical forecasting to preview the effects that will be caused by a particular program design. The analysis has shown that *if* the component mechanisms of **validator** are built to the requirements assumed for them in the analysis, then **validator** will have its required behavior. If the components are designed to meet their assumed requirements, there will be no need to backtrack and redo the design of **validator**.

The previewing power of mathematical forecasting reduces the risk of developmental backtracking, but it does not eliminate it. It is possible that the behavioral requirements used in the analysis of **validator** might be incorrect or incomplete in some way. Or, for some component, we might not be able to find an efficient design that meets behavioral requirements assumed for it in the analysis of **validator**. Any of these situations could require backtracking to redesign **validator**.

An important benefit of the kind of analysis done on **validator** is that it does show that the behavioral requirements of the components are sufficient *with respect to* the requirements of **validator**. The analysis shows that the requirements stated for the component mechanisms are sufficient for the composite mechanism to satisfy its requirements. In this sense, the analysis identifies "build to" requirements that are sufficient for the component mechanisms.

Another benefit of mathematical forecasting is the ability to predict accurately the effects of program modifications. Many programs change almost continually throughout their lifetimes. Certainly a program design will change as it evolves from initial conception to completion. Program maintenance is another common cause of program modification. Even in the best of worlds, after a good design is completed and the program is built and put into operation, better designs are discovered and requirements change. Because of the generally discontinuous behavior of programs, the effects of program modifications on their behavior often are very difficult to predict.

The way in which mathematical forecasting can predict the effect of program modifications can be illustrated by a trivial example. For the **validator**, it is easy to show that

```
g(reset(y) ; reset(z), s) = g(reset(z) ; reset(y), s)
```

The **reset(y)** and **reset(z)** mechanisms can be applied in either order to produce the same result. If, for some reason, we wish to reverse the order of **reset(y)** and **reset(z)** in the **validator**, this simple reversibility property tells us that the new version of the **validator** produces the same final state as the old one. The change has no effect on the final state. From this, we also know that the new version still satisfies the same behavioral requirements that were satisfied by the old one. Rather than redoing the entire analysis for the new version, we use the reversibility and *reuse* the analysis of the old version.

It also should be noted that the requirements stated for components of the **`validator`** can be used defensively to limit the effects of design changes. When we showed that the **`validator`** satisfies its requirements, we did so under very minimal assumptions about its component mechanisms. We assumed only that they satisfied their requirements. Nothing else was assumed about the components. The requirements stated for the component are a precise statement about the assumed interface between the component and its composite mechanism. Thus the design of a component can be changed without effecting the analysis of the composite *so long as* the modified component still satisfies it requirements. In this way, the requirements of a component can be used to limit the effects of changes to the component on the analysis of the composite mechanism.

All of the analyses of **`validator`** illustrated in this section can be done before the components of composite mechanism are designed and even before their requirements are fully defined. This illustrates how mathematical forecasting can be used to preview the physical effects of the design of a composite program, and by virtue of that, to reduce the risk of latent design errors and the consequential risks of operational malfunction and developmental backtracking.

2.5 Testing

Effective engineering commonly involves both accurate mathematical forecasting and physical experimentation. Aerospace engineering without mathematics would be a nightmare. It also would be a nightmare without physical experimentation. Wind tunnels are still used. Prototypes are still built. Test flights are still made. Even with the best mathematical forecasting, physical experimentation still is needed to demonstrate physical operability *and* to validate the accuracy of the mathematical models. Every airplane flight provides additional, experimental evidence of the accuracy (or inaccuracy) of the relevant mathematical models.

Program testing is physical experimentation. To test any physical thing, it must exist. A computer program is no exception. A computer program can be tested only by physically running it. In contrast, mathematical analysis is logical deduction that is performed on a description of a program. This predictive analysis can be done long before the program can be run physically.[7]

The analysis of the **`validator`** in Section 2.4 showed that it satisfies its requirements for *every possible* initial state **`u`** to which it might be applied. Let's also select a finite set of test cases **`{u1, ..., uk}`** and conduct **`k`** physical experiments by observing what final state **`vi`** results when the actual **`validator`** mechanism is applied to each initial state **`ui`**. Finally, let's suppose that every result **`vi`** satisfies the physical requirements of the **`validator`**.

By mathematical analysis, it is *forecast* from the model **`g(m,s)`** that the mechanism described by **`m0`** *will* satisfy (future tense) the requirement **`R(u,g(m0,u))`** for every possible value of the initial state described by **`u`**. The accuracy of the forecast depends on the accuracy of the model, the accuracy of the requirement and the soundness of the analysis. By testing, it was *observed* that the physical mechanism *did* satisfy (past tense) its physical requirement for the *k* test cases. The accuracy of the testing depends on our ability to perform physical experiments and observations.

If **`u`** ranges only over a set of values that is small enough so that the physical mechanism can be tested successfully on every one of them, then this tells us that the mechanism once actually *was* observed to work correctly for every possible initial state. This is a stronger statement than the corresponding

[7]The analysis of the behavior of a program on a single initial state sometimes also is referred to as "program testing." However, I prefer to call this "single-point analysis" and to let "program testing" refer strictly to physical experimentation.

mathematical forecast that it *will* work correctly because there is no question about the accuracy or the completeness of the mathematical forecast. There is no replacement for a demonstrably successful operational history.

Typically, however, **`u`** ranges over such an enormous set of values that the physical mechanism could be tested on only a very tiny fraction of them in a reasonable amount of time. For example, if **`u`** ranges over a state of just 64 bits, exhaustive testing would require 2^64 (more than 1.84*10^19) test cases. Even if each case required only 100 nano-seconds, testing them all would require over 58,000 years! And who or what is going to evaluate all those results? When one considers that it is common for a modern program to control states of 2^32 or more bits, these numbers become truly unimaginable! In contrast, however, if **`u`** ranges over a state of just 16 bits and each test requires 1 second, then exhaustive testing requires only a little more than 18 hours.

A serious difficulty with non-exhaustive, physical testing is that a successful test on one case **`ui`** tells us nothing about the outcome of testing a different case **`uj`**. But this problem is not unique to computer programs. It is a characteristic of experimental, physical testing. In order to extrapolate from a successful test on **`ui`** to what the outcome of testing **`uj`** might be, a predictive model usually is required, such as **`g(m,s)`**! Sometimes a useful degree of predictability can be achieved just by assuming that there exists a predictive model and that it is continuous or monotonic. For example, if a chair is tested successfully to support 200 kilograms, it probably also will support any number of kilograms less than 200, and it probably will even support 201. But for computer programs, discontinuities abound and intuition serves us poorly. If a program passes a test for 200, who knows what it might do for 201 or even 199? To extrapolate accurately from the physical behavior of a program for 200 to its behavior for other cases, an accurate predictive model is needed.

2.6 Accuracy

Mathematical forecasting is of benefit to an engineer to the extent that it provides *accurate* predictions about the future behavior of a physical system. Inaccurate predictions about physical behavior are of little interest. Those are made commonly every day with the well-known, unsatisfactory results.

How accurate a forecast is depends on how accurately the mathematical model describes the physical system, and on the soundness of the mathematical analysis. One would not expect to get an accurate forecast about the behavior of a spring from a mathematical model of a pendulum. And one would not expect to get an accurate forecast, even from an accurate model, if the mathematical analysis contained logical errors.

How does one obtain an accurate mathematical model of the behavior of a physical object? One does not construct a mathematical proof that a model accurately describes a physical system. Instead, one must conduct physical experiments which either affirm or deny that a particular model describes the appropriate physical observations. For example, Newton did not construct a mathematical proof of the law of gravity. The physical effects of the force of gravity were observed, and in time, Newton proposed a mathematical model to describe those effects. By careful measurement and observation, many subsequent physical experiments confirmed the accuracy of Newton's model. Newton's model certainly is not a *complete* description of the force of gravity. To this day, we do not know *how* gravity works. But Newton's mathematical model does describe the *effect* of gravity accurately enough so that engineers can, and do, make very accurate forecasts about the attractive forces between physical objects.

To establish the accuracy of a mathematical model of a computer program, ultimately, a similar experimental process is required. What must come out of these experiments is confirmation that a particular mathematical model is an accurate description of the effects caused by programs running on some physical computer. How this might be done is an important subject that is beyond the scope of this discussion. The scope of this discussion is the role of mathematics in forecasting the behavior of computer programs. Therefore, it is important to set forth clearly what can and cannot be done with mathematics. Ultimately, mathematics cannot replace the process of experimentally confirming the accuracy of a model of program behavior.

What can be done mathematically is to prove that one model of a physical system logically follows from another one. If *A* is a model of some physical system, and model *B* logically follows from model *A*, then *B* describes the physical system just as accurately as *A* does. Accuracy-preserving transformations of one model into another by sound mathematical deduction are common practice in other fields of engineering, and it also is possible in software engineering.

For example, suppose that we are given a mathematical function **`f(x)`** that describes the behavior of the some digital processor at the level of machine language. This function gives the final state produced by the processor when its initial state is described by the sequence of binary digits **`x`**. Both the program and the state would be contained in the bit sequence **`x`**, and the model would allow even self-modifying programs.

Suppose further that we construct a mathematical proof that **`g(m,s) = display(f(compile(m,s)))`** where **`g(m,s)`** is the Gypsy model of program behavior described in Figure 4, **`compile(m,s)`** is a function that maps Gypsy descriptions of programs **`m`** and states **`s`** into bit sequences, and **`display(x)`** maps bit sequences back into Gypsy states. The mathematical deductions that comprise this proof show that the Gypsy model **`g(m,s)`** logically follows from the machine language model **`f(x)`**. Thus, the accuracy of the machine language model is preserved in the Gypsy model -- i.e., the Gypsy model is just as accurate as the machine model is. If a compiler program can be produced that performs the transformation **`compile(m,s)`**, then one can use **`g(m,s)`** to make forecasts about the physical behavior of compiled Gypsy programs with the same degree of accuracy that one gets by using **`f(x)`** to make forecasts about machine language programs.

Some specific examples of different models of program behavior and mathematical proofs that one logically follows from another can be found in [Bevier 89a], [Hunt 89], [Moore 89] and [Young 89]. These examples involve three languages: Gypsy, the assembly language Piton, and FM8502 machine language. The FM8502 is a 32-bit micro-processor of complexity comparable to a PDP-11. Gypsy is compiled into Piton, and Piton is compiled into FM8502 machine language. Each of these three languages has a predictive model that forecasts the physical behavior of programs described in that language. By mathematical deduction, it has been proved that the Gypsy model logically follows from applying the Piton model to Gypsy compilations [Young 89], and the Piton model logically follows from the FM8502 machine language model applied to Piton compilations [Moore 89]. Finally, it also has been proved that the FM8502 machine language model logically follows from the gate-level design of the FM8502 [Hunt 89]. In this way, accuracy of the gate-level model is preserved in the FM8502 machine language model, in the Piton model and in the Gypsy model. The Gypsy model is just as accurate as the FM8502 gate-level model!

Another important example of deducing that one model logically follows from another is described in [Bevier 89b]. Here it is proved that a model of a simple separation kernel, Kit, logically follows by applying the machine language model of an FM8502-like processor to a particular machine language program description. The program description which the model is applied to is the actual machine language code for the separation kernel. This kernel provides separation for a fixed number of communicating processes running on a single FM8502-like processor.

Making an accurate forecast about the physical behavior of a computer program not only requires having an accurate model of program behavior, but also, to preserve the accuracy inherent in the model, it requires making sound mathematical deductions. Because accurate mathematical models of programs are highly discontinuous, our mathematical intuition about these models is not very well developed; therefore, we must be especially careful to perform the mathematical analysis of these models correctly.

The mathematical deductions used to perform the analysis of the **`validator`** example are sufficiently simple so that they can be done reliably with pencil and paper (even the important induction step that was left out of the presentation). In some cases, this kind of manual analysis can be quite tractable and useful. In addition, structuring a program description so that its mathematical analysis does become tractable manually can have considerable engineering benefit because it tends to keep the program structured in a way so that we understand it better.

However, because the current state of technology for modeling program behavior is still quite primitive, analyzing a model also can pose a high volume of mathematical deduction. Mechanical theorem provers are one way of dealing with this volume. For example, to obtain maximal assurance that the mathematical deductions cited above in [Hunt 89], [Bevier 89c], [Moore 89] and [Young 89] were done without logical errors, all of them have been confirmed with a mechanical theorem prover. With appropriate human guidance, the Boyer-Moore theorem prover [Boyer & Moore 88] has confirmed that the conclusions drawn from the models of Gypsy, Piton, Kit and the FM8502 logically follow from the axioms of the Boyer-Moore logic [Boyer & Moore 79] by applying only precisely stated, well-understood rules of logical deduction. It is important to emphasize that these mechanical deductions are *confirmations of*, not *replacements for*, human mathematical deductions.

3. Into the Future

Let’s take von Karman’s 1954 statement about airplane design and rephrase it for the design of computer programs, those physical control mechanisms that are composed of very large numbers of very small switches.

> Mathematical theories from the happy hunting grounds of pure mathematicians are found suitable to describe the *state sequences* produced by *computer programs* with such excellent accuracy that they can be applied directly to *program* design.

If the practice of software engineering could advance to this state, both providers and consumers of computer software systems could receive the benefits that applied mathematics brings to engineering.

Historically, the incorporation of applied mathematics is an important step in the evolution of a field of engineering. Strictly speaking, applied mathematics is neither necessary nor sufficient for successful engineering. But accurate mathematical models and sound analyses provide engineers with a very powerful way to forecast the physical behavior of systems constructed from their designs. This forecasting capability does not replace the creativity of human invention, it does not guarantee perfection, and it does not eliminate the need for physical experimentation. But it can reduce the risk of operational malfunction and the risk of high resource consumption caused by developmental backtracking.

The current practice of software engineering falls well short of the state of aerospace engineering described by von Karman in 1954. Current software engineering practice is more like aerospace engineering in 1879.

> Mathematics up to the present day have been quite useless to us in regard to *programming*.

The application of mathematics to forecast the physical behavior of computer programs is virtually non-existent in current practice. Certainly it is common for mathematical equations describing airflow or other physical phenomena to be deduced from various models and then handed over to a "programmer" for "coding." But how many times has that "programmer" applied a mathematical model to the Fortran or C or Ada "code" to predict how the program (those physical switch settings) produced from that "code" will control the electronic states caused by the physical computer? Zero, I would guess. How many times does that program cause unpredicted behavior? Many, and there is no need to guess.

These simple observations raise an important question. Computer programs are control mechanisms for physical machines. These machines *do* cause effects in our physical world. Many of them have been engineered without adequate means to forecast accurately what effects they *will* cause. Therefore, many of them probably are capable of causing unforeseen effects. What risks do these potential unforeseen effects pose to our physical world? This important question has yet to be answered.

What is to be done to advance software engineering beyond this current state of affairs? Part of the answer is to incorporate effective, applied mathematics for computer programs into mainstream software engineering practice. To do this, it will necessary to

- develop models for the notations engineers use to describe programs,
- develop models of requirements for programs,
- develop mathematics and tools to apply these models effectively,
- integrate this mathematics into the software engineering process,
- educate software engineers in how to use it, and
- transfer it into engineering practice.

All of this is much easier said than done, and there is much to be done before mathematics can be applied as extensively in software engineering as it currently is in aerospace engineering. But even today, software engineers can begin to use some of the mathematics that is becoming available.

At some future time, there will need to be standard, mathematical models for whatever notations[8] are being used in software engineering practice. If we were in this state today, there would be, for example, ISO standard models of Fortran, C and Ada. Software engineering students would be learning these models and how to apply them while aerospace engineering students were learning the mathematical models of gravity and aerodynamics. Practicing engineers would be applying these standard models to predict the behavior of programs described in these languages. These predictions might be made by direct application of the models, or they might be made by applying libraries of general theorems derived from the standard models. For every language implementation that conformed with the standard, predictions made from the model would be accurate. This would have the important practical consequence of making the predicted behavior truly portable among all conforming implementations. This would be a significant advance beyond the "pseudo portability" that often exists today in which, although the same program description may compile and run under different implementations, its behavior may be different.

The current practice of software engineering is far from achieving this futuristic state of affairs. The important role of mathematical models of programs in software engineering is only beginning to be recognized. There are no "off-the-shelf" models, standard or otherwise, for Fortran, C or Ada. What presently is needed is to begin building and validating some of these models. This will not be easy because mathematical models of program behavior were not even considered in the design of most current languages. In the future, these mathematical models need to be developed in parallel with the design of new languages or the evolution of current ones. In contrast to the way that most languages currently are developed, this will require that language development efforts put a high priority on defining the mathematical semantics of the language. For example, the Gypsy model **`g(m,s)`** described in Section 4

[8]These future notations may, or they may not, resemble the programming languages used in current practice.

defines part of an operational semantics for Gypsy. Strictly as a practical engineering consideration, the mathematical semantics of programming languages can no longer be ignored. Without these semantic foundations, accurate mathematical forecasting will not be possible.

But for now, we can begin with what we have. It is not necessary to wait for the definition of international standard models for current languages. A wise project manager can build or adopt a model for a particular project. The project model can be restricted to a particular language subset as it is implemented by a particular combination of compiler, operating system and machine. The manager can constrain project programming to these restrictions. Where this is possible, it can be a useful start. Just by defining a project-wide model, one makes explicit a precise set of assumptions about how programs behave on a particular project. This, at least, enables all project software engineers to work from the same set of *assumptions* about program behavior, and it allows one to begin validating the assumptions of the project model from practice and experience. With a bit of luck, a project model could be generalized and used on other projects, thus amortizing the cost of developing the model. A specific, project-wide standard is far removed from a generally accepted, international standard, but at least it can bring the benefits of applied mathematics to a particular project.

One of the important uses of mathematical forecasting is to predict that the behavior of a program will meet certain behavioral requirements. Making the forecast requires a mathematical model of the requirements (such as the problem domain functions shown in Figure 5). Developing models of requirements is another major problem that confronts software engineering. This development will require requirements discovery and requirements modeling. Have all of the right requirements been discovered? Have they been modeled accurately? These are important questions that require answers because mathematical forecasting is no better than its models. Once a model is stated, mathematical analysis can provide some help in answering these questions. For example, one can analyze the consistency of the requirements. One also can make deductions from the requirements, and see if these logical conclusions conform to expected program behavior. Ultimately, however, physical experimentation and observation, through previous experience and prototyping, will need to play a major role in requirements discovery and modeling.

A well-developed mathematical theory already exists which can provide the foundations for describing very accurately the state sequences produced by computer programs. It is the theory of recursive functions. What is not yet well-developed are the means to apply this theory effectively in engineering practice. This body of applied mathematics is beginning to emerge, and enough is presently available so that it can be useful to a practicing engineer. Various aspects of mathematical reasoning about programs are discussed in [Boyer & Moore 88], [Chandy 88], [Dijkstra 76], [Gehani 86], [Gries 81], [Hayes 87], [Hoare 85], [Jones 80] and [Jones 86]. In response to this mathematics, one often hears the objection that it is just too complex to be useful. It is complex. But the complexity is not caused by the mathematics. This mathematics just accurately describes the complexity of computing. Is computing too complex to be useful? Mathematics is one of the most effective ways we have to manage complexity.

If software engineering is to reach the level of maturity of aerospace and other successful engineering fields, it needs to incorporate effective, applied mathematics. Applied mathematics is one of the important crystal balls that engineers use to forecast the future behavior of their systems. This reduces the dual risks of operational malfunction and high resource consumption caused by developmental backtracking. There is not a single point in the engineering process where aerospace engineers "do the mathematics" to certify that their system will meet its requirements. Mathematics is applied in many ways and in many places throughout the product development process. This same kind of integration of applied mathematics into the software development process is needed in software engineering. Certainly there is much more to engineering than applied mathematics. But engineering without it is risky business.

4. Figures

Figure 1: Validator Mechanism Description

```
DESCRIPTION:

  type a_cmd     = pending;
  type a_cmd_seq = sequence of a_cmd;
  type a_context = pending;

  procedure validator(var x, y, z:a_cmd_seq) =
  begin
    var c  :a_cmd;
    var eox:boolean;
    var e  :a_context;
    reset(y);    reset(z);
    init_context(e);
    loop
      get_cmd(c, eox, x);
      if eox then leave
      else update_cmd(c, y, z, e)
      end
    end
  end


BEHAVIOR:

  v[y] = all_valid(x0),

  v[z] = all_invalid(x0),

  where u = { (x, x0), (y, y0), (z, z0), (run, normal) },

        v = g(D[validator], u).
```

Figure 2: Validator Components

```
1. var   c:a_cmd
2. var eox:boolean
3. var   e:a_context
4. leave
5. reset(y)
6. reset(z)
7. init_context(e)
8. get_cmd(c, eox, x)
9. update_cmd(c, y, z, e)
```

Figure 3: Validator Compositions

```
1. procedure <A> = <B>
2. begin <A> end
3. <A> ; <B>
4. loop <A> end
5. if <A> then <B> else <C> end
```

Figure 4: Mathematical Model

```
1. Procedure Composition.

   g(procedure <F> = <B>, s) = if s[run]=normal
                               then g(<B>, s)
                               else s

2. Local Variable Primitive.

   g(var <I> : <J>, s) = if s[run]=normal
                         then alter(s, <I>, default(<J>, D))
                         else s

3. Begin Composition.

   g(begin <A> end, s) = if s[run]=normal
                         then g(<A>, s)
                         else s

4. If Composition.

   g(if <I> then <B> else <C> end, s) = if s[run]=normal
                                        then if s[<I>]
                                             then g(<B>, s)
                                             else g(<C>, s)
                                        else s

5. Loop Composition.

   g(loop <A> end, s) = if s[run]=normal
                        then if g(<A>,s)[run]=leave
                             then alter(g(<A>, s), run, normal)
                             else g(loop <A> end, g(<A>, s))
                        else s

6. Leave Primitive.

   g(leave, s) = if s[run]=normal
                 then alter(s, run, leave)
                 else s

7. Procedure Call Primitive.

   g(<I> ( <A> ), s) =  if s[run]=normal
                        then copyout(<I>, <A>, v, s, D)
                        else s

   where v = g(D[<I>], copyin(<I>, <A>, s, D))

8. Sequential Composition.

   g(<A> ; <B>, s) = if s[run]=normal
                     then g(<B>, g(<A>, s))
                     else s
```

Figure 5: Problem Domain Functions

```
Let c be a command,

    e be a context,

    x be a sequence of commands,

   <> be the empty sequence.


For a non-empty sequence of commands,


    last(x) is the last element of x,

    nonlast(x) is the rest,

    nonlast(x) :> last(x) = x.


    first(x) is the first element of x,

    nonfirst(x) is the rest,

    first(x) <: nonfirst(x) = x.


Assumed Functions:

  valid(c,x)

  context(x)

  check(c,e)


Assumed Relations:

  valid(c,x) = check(c,context(x))


Defined Functions:


  all_valid(x) = if x = <> then <>
                 else if valid(last(x),nonlast(x))
                      then all_valid(nonlast(x)) :> last(x)
                      else all_valid(nonlast(x))

  all_invalid(x) = if x = <> then <>
                   else if valid(last(x),nonlast(x))
                        then all_invalid(nonlast(x))
                        else all_invalid(nonlast(x)) :> last(x)

  eos(x) iff x=<>
```

Figure 6: Get_Cmd Mechanism Description

DESCRIPTION:

```
  procedure get_cmd(var   c:a_cmd;
                    var eox:boolean;
                    var   x:a_cmd_seq) = pending
```

BEHAVIOR:

```
  v[c]   = if v[eox] then c0 else    first(x0),

  v[eox] = eos(x0),

  v[x]   = if v[eox] then x0 else nonfirst(x0),

  where u = { (c, c0), (eox, eox0) (x, x0), (run, normal) },

        v = g(D[get_cmd], u).
```

Figure 7: Init_Context Mechanism Description

```
DESCRIPTION:

  procedure init_context(var e:a_context) = pending


BEHAVIOR:

  v[e] = context(<>),

  where u = { (e, e0), (run, normal) },

        v = g(D[init_context], u).
```

Figure 8: Reset Mechanism Description

```
DESCRIPTION:

  procedure reset(var y:a_cmd_seq) = pending

BEHAVIOR:

  v[y] = <>,

  where u = { (y, y0), (run, normal) },

        v = g(D[reset], u).
```

Figure 9: Update_Cmd Mechanism Description

```
DESCRIPTION:

  procedure update_cmd(c:a_cmd;
                var y, z:a_cmd_seq;
                var    e:a_context) = pending


BEHAVIOR:

  v[c] = c0,

  v[y] = if check(c0, e0) then y0 :> c0 else y0,

  v[z] = if check(c0, e0) then z0       else z0 :> c0,

  v[e] = context(h0 :> c0),

  where u = { (c, c0), (y, y0), (z, z0), (e, e0), (run, normal) },

       e0 = context(h0),

        v = g(D[update_cmd], u).
```

References

[Anderson 89] John D. Anderson, Jr.
Introduction to Flight, Third Edition.
McGraw-Hill, 1989.

[Bevier 89a] William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, William D. Young.
An Approach to Systems Verification.
The Journal of Automated Reasoning 5(4), November, 1989.

[Bevier 89b] William R. Bevier.
Kit: A Study in Operating System Verification.
IEEE Transactions on Software Engineering 15(11), November, 1989.

[Bevier 89c] William R. Bevier.
Kit and the Short Stack.
The Journal of Automated Reasoning 5(4), November, 1989.

[Boyer & Moore 79]
R. S. Boyer, J S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

[Boyer & Moore 88]
R. S. Boyer, J S. Moore.
A Computational Logic Handbook.
Academic Press, Boston, 1988.

[Chandy 88] K. Mani Chandy, Jayadev Misra.
Parallel Program Design, A Foundation.
Addison Wesley, 1988.

[Dijkstra 68] E.W. Dijkstra.
A Constructive Approach to the Problem of Program Correctness.
BIT 8-3, 1968.

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

[Floyd 67] R.W. Floyd.
Assigning Meanings to Programs.
In J.T. Schwartz (editor), *Proceedings of a Symposium in Applied Mathematics*. R.W. Floyd, American Mathematical Society, 1967.
Vol. 19.

[Gehani 86] N. Gehani, A. D. McGettrick.
Software Specification Techniques.
Addison-Wesley, 1986.

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

[Gries 81] Gries, D.
The Science of Computer Programming.
Springer-Verlag, 1981.

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

[Hoare 69] C.A.R. Hoare.
An Axiomatic Basis for Computer Programming.
Communications of the ACM 12-10, 1969.

[Hoare 85] C. A. R. Hoare, J. C. Shepherdson, Eds.
Mathematical Logic and Programming Languages.
Prentice Hall International Series in Computer Science., 1985.

[Hunt 89] Warren A. Hunt, Jr.
Microprocessor Design Verification.
The Journal of Automated Reasoning 5(4), November, 1989.

[Jones 80] Cliff B. Jones.
Software Development: A Rigorous Approach.
Prentice Hall, 1980.

[Jones 86] Cliff B. Jones.
Systematic Software Development Using VDM.
Prentice Hall, 1986.

[McCarthy 63] J. McCarthy.
A Basis for a Mathematical Theory of Computation.
In P. Braffort and D. Hershberg (editors), *Computer Programming and Formal Systems*. North-Holland Publishing Company, Amsterdam, The Netherlands, 1963.

[Moore 89] J Strother Moore.
A Mechanically Verified Language Implementation.
The Journal of Automated Reasoning 5(4), November, 1989.

[Naur 66] P. Naur.
Proof of Algorithms by General Snapshots.
BIT 6, 4, 1966.

[von Neumann 61]
Herman H. Goldstine, John von Neumann.
Planning and Coding of Problems for an Electronic Computing Instrument, Part II, Vols 1-3.
In A. H. Taub (editor), *John von Neumann, Collected Works, Volume V*, pages 80-235. Pergamon Press, Oxford, England, 1961.

[Young 89] William D. Young.
A Mechanically Verified Code Generator.
The Journal of Automated Reasoning 5(4), November, 1989.

Table of Contents

1. Mathematics in Engineering 1
2. Mathematics for Programs 3
   2.1. Programs 3
   2.2. States 4
   2.3. Models 5
   2.4. Forecasting 6
   2.5. Testing 10
   2.6. Accuracy 11
3. Into the Future 13
4. Figures 16

List of Figures

Figure 1: Validator Mechanism Description 16
Figure 2: Validator Components 17
Figure 3: Validator Compositions 18
Figure 4: Mathematical Model 19
Figure 5: Problem Domain Functions 20
Figure 6: Get_Cmd Mechanism Description 21
Figure 7: Init_Context Mechanism Description 22
Figure 8: Reset Mechanism Description 23
Figure 9: Update_Cmd Mechanism Description 24