

American National Standard

AAMI MDS

**Developing safe, effective, and
reliable medical software**

AAMI

Association for the
Advancement of Medical
Instrumentation



Documents
Document Sharing Hub



Association for the Advancement of Medical Instrumentation

1110 N. Glebe Rd., Suite 220
Arlington, VA 22201-4795

© 2000 by the Association for the Advancement of Medical Instrumentation

All Rights Reserved

Copyright and Permissions

Publication, reproduction, photocopying, storage, or transmission, electronically or otherwise, of all or any part of these documents without the prior written permission of the Association for the Advancement of Medical Instrumentation or the copyright holder (if not AAMI) is prohibited by law. It is illegal under federal law (17 U.S.C. § 101, *et seq.*) to make copies of all or any part of these documents (whether internally or externally) without the prior written permission of the copyright holder. Violators risk legal action, including civil and criminal penalties, and damages of \$100,000 per offense. For permission regarding the use of all or any part of these documents, contact AAMI, 1110 N. Glebe Road, Suite 220, Arlington, VA 22201-4795. Phone: (703) 525-4890; Fax: (703) 525-1067.

Violators of this copyright policy should be reported to AAMI's legal counsel:

McKenna & Cuneo, L.L.P.
1900 K Street, N.W.
Washington, DC 20006
Attn: Jacqueline A. Henson, Esq.
Phone: (202) 496-7500

MDS Developing Safe and Reliable Medical Software

Developing Safe, Effective and Reliable Medical Software **Medical Instrumentation Suite 400** **Arlington, Virginia 22201-4598**

In the past 30 years, medical devices have become increasingly dependent on software. Software has enabled devices to perform such functions as medical record-keeping, control of the heart rate, analysis of patient data, and dosage adjustment based on patient feedback. As the functions of medical devices have become more complex, so has the software required to control those functions and so have the development and quality assurance techniques used to assure software safety, effectiveness, and reliability.

This publication is intended to acquaint users of computer-controlled medical devices and managers in the medical device industry with the issues involved in software development and with the basic principles of software quality assurance, hazard analysis, reliability testing, and design validation. It is not meant to be a "cookbook" on how to develop medical software, but rather a general introduction to important concepts.

The development of this publication was inspired by discussion groups convened in 1987 by the Association for the Advancement of Medical Instrumentation (AAMI) for the purpose of identifying new technical areas in which monographs, technical information reports, consensus documents, or other publications would be beneficial to the medical device industry and health care professions. One such discussion group, chaired by Mr. Richard C. Fries and consisting of representatives of industry, the Food and Drug Administration, and health care organizations, concluded that there was a need for a publication that would communicate fundamental aspects of medical software development and quality assurance.

While the chapters of this publication were written by individual experts and mainly reflect the authors' personal views and expertise, the document was subjected to a peer review process to ensure its technical accuracy, utility, and timeliness. Reviewer comments were considered by the authors in preparing the final versions of their chapters. Special thanks are due the following reviewers: Barry J. Altman, CBET, BMET Technology, Mequon, Wisconsin; Alan A. Figler, Figler Consulting, Crystal Lake, Illinois; Rick Beebe and Chas Burr, Hewlett-Packard Company, Waltham, Massachusetts; Diane L. Jachinowski, Nellcor Incorporated, Hayward, California; Roger W. Jelliffe, M.D., University of Southern California School of Medicine, Los Angeles, California; Ira Soller, M.S.E.E., P.E., State University of New York, Brooklyn, New York.

Comments on this publication are welcome and should be sent to: Technical Programs, AAMI, 3330 Washington Boulevard, Suite 400, Arlington, VA 22201.

THE PURPOSE OF SOFTWARE QUALITY ASSURANCE: A MEANS TO AN END

Joseph Jorgens III
Biomedical Engineer
Center for Devices and Radiological Health
Food and Drug Administration
Rockville, Maryland

Introduction

Computer-controlled medical devices are increasingly common. The software is susceptible to design errors that result in unpredictable and often costly mishaps. Designing quality software is essential to the manufacturing process for medical devices containing computers.

A well designed software quality assurance program provides the means to measure and control software quality at each step of the software development process. No system exists which can unilaterally guarantee flawless software. Rather, a software quality assurance program aims to continuously improve the software development process by providing tools and techniques to maximize the efficiency and accuracy of skilled product developers. This approach represents a dramatic improvement over the chaotic, ad hoc practices employed by many medical device manufacturers today. Medical device manufacturers need a software quality assurance program to decrease liability, contain costs, and meet the regulatory expectations of the Food and Drug Administration (FDA).

To illustrate the goals and purposes of a software quality assurance program, this paper will describe (1) some of the mishaps that have occurred because of nonexistent or inadequate software quality assurance programs and (2) the legal, regulatory, and economic implications of software quality assurance programs.

The "end" in the title phrase "a means to an end" may refer to several different goals. The primary goal of a software quality assurance program is to produce a product that performs functionally as intended and in no other fashion. Other desirable goals in a clinical setting include limiting legal liability, reducing costs, and complying with regulatory requirements for medical devices.

The Goal of Correct Functional Performance

It is very enlightening to consider some of the results of failures to achieve this goal.

Non-Medical Mishaps

One of the earliest documented incidents of software failures involved the Strategic Air Command's system for detecting incoming ballistic missiles. Originally, system programming did not ignore naturally occurring phenomena. On one occasion, the system detected a very large, fast-moving object where missiles would be expected. U.S. forces were placed on alert; but, fortunately, an astute operator noticed that the system's antennae were pointed toward the moon and intervened before the fighters actually entered into combat with the attacking orb.

More recently, simulation testing of the software for the F-16 fighter plane found incorrect programming in the navigation system. When the F-16 flew across the equator, the plane flipped upside down!

The F-18 fighter plane also had software problems. The F-18 carries missiles on its wings. When the pilot wishes to launch a missile, he pushes a button, the clamp holding the missile opens, the missile ignites and takes off, and the clamp closes. In an early test of the F-18, this timing sequence was not quite right. When the pilot pushed the button to fire the missile, the clamp opened and the missile ignited, but the timing sequence allowed the clamp to close before the missile left its pod. The thrust generated on that side of the plane was tremendous, and the plane went out of control. Fortunately, the pilot was able to regain control after the missile used up its fuel supply.

In December 1989, AT&T installed new software in all of its switching systems, including a latent programming error. In January 1990, a problem occurred in one switch and the latent error surfaced. This caused a similar problem in 144 switching systems, all with the same latent error and all connected to the system in which the original failure occurred. The resulting chain reaction of failures shut down transcontinental telephone traffic for several hours.

Medical Mishaps

The Food and Drug Administration conducted a study of the medical device recalls that occurred during the years 1983 to 1987. It was found that 84 of the recalls involved software; of those 84 recalls, 79 were caused by software errors that had been introduced during design. Of the software-related recalls, 49 concerned diagnostic devices and 26 concerned devices that were directly involved in patient treatment. The diagnostic devices included ultrasound units, patient monitors, cell counters, blood analyzers, and the like.

The therapeutic devices included lasers, pacemakers, ventilators, electrosurgical units, and infusion pumps. Some detailed examples follow.

A blood gas analyzer was programmed with an incorrect formula for the calculation of certain parameters. The programmer had simply made a mistake in copying the formula out of a book, substituting addition for subtraction in a minor correction term. This error did not matter in most instances because the correction term was a minor fudge factor; for some input data, however, the correction term became a large part of the computation and erroneous values resulted. Another blood analyzer had a maximum display limit of 999; when a value for a parameter exceeded 999, the device presented the user with the value for the previous sample.

A cardiac output measurement device produced erroneous results when used with a particular kind of catheter. The K factor used in the equation for calculating cardiac output was incorrect for certain injectate volumes and catheter sizes.

If diagnostic devices produce incorrect results, the errors may be readily noticed or may be inconsistent with other clinical signs. Thus, the risk to the patient may be less than in the case of software-driven devices that directly affect patients. An infant warmer in which a software feedback loop controlled the temperature locked up in certain situations. Such a lock-up could markedly increase the baby's temperature, possibly causing serious injury or death.

An infusion pump continuously infused insulin when the value "0" was entered as the maximum volume to be infused. In another pump, the alarm indicating an empty solution container would not sound if the batteries were weak; therefore, air could be infused without warning, potentially creating an air embolus. When the volume on another pump was set to less than 1 milliliter (ml) per hour, the pump ignored the command and delivered the previous setting, up to 700 ml/hour. A potentially deadly drug, such as sodium nitroprusside, could produce disastrous results.

There have also been software-related problems with pacemakers. A certain command sequence in a particular brand of pacer programmer caused the program to enter a random state. In another incident, a company had produced a "universal" programmer that, in theory, could be used to program any of a particular line of pacemakers. Unfortunately, in certain situations, one of the pacemakers in the series did not respond properly, even though the universal programmer indicated that the pacemaker was programmed correctly. Another pacer programmer signaled that the proper values had been transferred to the pacemaker, when, in fact, the pacemaker's values had not changed at all.

In one type of ventilator, the patient disconnect alarm would not sound in certain circumstances because of a software error. In the same ventilator, the gas concentrations could decrease without any activation of the alarms and without any change in the operator's digital display.

Incidents involving a linear accelerator had tragic consequences. This particular type of linear accelerator had been in therapeutic use for over two years. There were two treatment modes. In the electron therapy mode, an electron beam was generated, providing a prescribed amount of radiation. In the x-ray mode, this electron beam's current was increased, perhaps a thousand-fold, which was the level necessary to produce the prescribed amount of x-rays when the electron beam impacted a heavy metal target. In several instances, operators typed in commands that were intended to initiate the second treatment mode, causing the target to move into place. Because the operators had become so proficient, the commands were typed in faster than the commands could be accepted by the software program, and thus the heavy metal target did not move into place. The end result was that a number of patients were exposed to a drastic increase in radiation dose over the prescribed amount. The beam burned several patients so severely that they died.

The Goal of Limiting Legal Liability

Limiting legal liability is another goal of software quality. With the increasing use of software in medical

devices comes an increase in the application of product liability laws to software-driven devices. Among the most common theories of liability advanced in lawsuits are express warranty, implied warranty, negligence, and strict liability.

Express Warranty

An express warranty is one which is stated explicitly, either orally, in a contract of sale, or in the labeling. For instance, let us assume that a clinical engineer requests and receives a written or oral statement that the medical device manufacturer followed a certain reliability protocol or that the medical device and its software will perform in a specific fashion. If the medical device causes an injury because the device was not developed according to the stated reliability protocol or because it did not function as warranted, the manufacturer faces liability under the express warranty theory.

Implied Warranty

Sometimes a warranty is not stated explicitly. By introducing a product into commercial distribution, the manufacturer implicitly warrants that the product is reasonably fit for the purposes that similar products are intended to serve. For instance, if all pacemaker manufacturers have reliability programs to ensure proper functioning of their products, and a new pacemaker manufacturer begins commercial distribution, there may be an implied warranty that the new manufacturer has a similar program in place.

Negligence

Under this theory, a manufacturer that does not exercise reasonable care or fails to meet a reasonable standard of care in the manufacture, handling, or distribution of a product may be liable for any damages caused. For instance, if it can be established that not having a reliability program constitutes a failure to meet an industry-wide practice that is found to be an applicable standard of care, then the manufacturer may be subject to liability for negligence.

Strict Liability

Strict liability may become an issue if an injury is caused by a product that has a defect that is unreasonably dangerous.

The Goal of Fiscal Responsibility

The total cost of producing a medical device includes the cost of design and development, materials, testing, production, and maintenance. Unfortunately, maintenance costs for software-driven products are very high. Maintenance and software support functions include analyzing and removing errors and providing enhancements. Some have estimated that maintenance currently constitutes approximately 50 percent of the cost of a software product.

The old adage that "quality is free" could be changed (for software) to "quality is not a cost item, but the lack of quality is." Thus, any action that decreases maintenance costs is actually a cost saving. In fact, considering that enhancements are really requirements that were not initially considered, the entire maintenance phase and its associated cost can be dramatically reduced if the software is of high initial quality.

Furthermore, a business may lose repeated sales if the software has many bugs or does not meet the user's requirements. The cost of this loss of future business can be very high. Increasingly, users are asking software developers and distributors to warrant their products, forcing developers to determine the cost-effectiveness of defect prevention and removal throughout the development process and after release.

The Goal of Regulatory Compliance

The Food and Drug Administration has the responsibility and authority to regulate medical devices. A company that does not comply with the appropriate regulations cannot lawfully place a device into

commercial distribution.

The depth of FDA review of the software contained within a particular medical device is based on the risk that the entire device poses to patients and on the extent to which the software contributes to that risk. The FDA distinguishes the risks associated with software-driven medical devices into categories of minor, moderate, or major concern.

For all device software, FDA desires a software quality assurance program to be in place. For the devices of "minor concern" from a risk standpoint, FDA wants little documentation of the software development process. For devices of "moderate concern," FDA requests additional information on the software development process. For devices of "major concern," FDA requires assurance that significant thought has gone into the design of the device and its associated software. For such devices, FDA must be provided with documentation that a hazard analysis and appropriate testing were performed for both the device and its software. The functional requirements and system specifications should be described, as well as the software design and development procedures and the revision control procedures.*

Summary

From public health and ethical standpoints, software produced for a medical device, whether as a component or as a stand-alone package, should be quality software. Software mishaps can be critical in a medical device. Software developers have every reason to produce quality software and no reasons to produce poor software except lack of knowledge and understanding. Incentives for producing quality software include financial benefits, reduced legal liability, and an easier path through FDA. A clear understanding of the benefits should convince anyone that a software quality assurance program is necessary for any medical device that contains a computer.

Bibliography

*Food and Drug Administration. Computerized drug processing; source code for process control application programs (Compliance Policy Guide 7132a.15). April 16, 1987.

*Food and Drug Administration. Draft policy guidance for regulation of computer products; availability. *Federal Register* 52 (186): p. 36104; September 25, 1987.

*Food and Drug Administration. Software development activities; reference materials and training aids for investigators. July 1987.

*Food and Drug Administration. Draft reviewer guidance for computer-controlled medical devices. August 1, 1988.

*Food and Drug Administration. Draft FDA policy for the regulation of computer products. November 13, 1989.

SAFETY OF COMPUTER-CONTROLLED DEVICES

Jonathan Jacky, Ph.D.
Research Assistant Professor
Department of Radiation Oncology
University of Washington
Seattle, Washington

What is Safety?

This chapter is an overview of system safety considerations for devices that include computers. It is not exclusively devoted to software, but also considers the interaction of hardware and software in systems that have safety implications.

In recent years, it has become economical to use computers to replace many of the electromechanical components that have traditionally equipped instruments and control systems. Software can substitute for "hard-wired" relay logic, and keyboards and colorful video displays can substitute for switches and dials. Vendors are especially intrigued by the possibility of periodically improving products without making changes to the hardware; they simply distribute a new copy of the control software.

These advantages do not come without a price: producing high-quality software turns out to be surprisingly difficult. A never-ending "software crisis" is characterized by projects that take too long, cost too much, or deliver unsatisfactory products (Shore 1985). In medicine, computers introduce possibilities for new kinds of hazards. For example, the linear accelerators used for cancer radiation therapy treatments may pose risks to patient and personnel from electrical shock, radiation exposure, and mechanical collisions caused by runaway programs. In fact, several deaths and injuries have been caused by faults in therapy machine control software (Joyce 1986, 1987). Those accidents prompted concern and action on the part of the U.S. Food and Drug Administration (FDA), AAMI, and other organizations.

Software is not, in itself, unsafe. Only the physical systems that it may control can do damage. Safety considerations hardly arise for programs that perform conventional data processing or scientific computation. In these applications, the computer only displays results on paper or on video screens. It is presumed that the users will review these results, bringing their informed judgment to bear before acting upon them. It is only when computers are used to directly control systems which are themselves potentially unsafe (such as radiation therapy machines) that safety issues arise. The results of control computations are usually not subject to human review and cannot be discarded or ignored. The controlled system can never be returned to its prior state. Systems that include computers that directly control motors, valves, electrical transducers, or other physical devices are called *process control systems* or *embedded systems*. Some medical devices are process control systems. The remainder of this paper will be limited to the discussion of these systems.

For safety reasons, process control computer hardware requires special design and construction techniques that are not needed for consumer electronics and personal computers. Process control software also requires special techniques, techniques that are unnecessary in conventional data processing and scientific programming. Because of the interaction between the software and the controlled devices, an overall systems perspective is needed. Safety efforts that concentrate only on software or on hardware will be inadequate. Organizations should not naively assume that experience with less demanding projects prepares them to build process control systems. They must make a well-informed decision on whether they can bear the additional cost and call upon the requisite expertise.

Designing for Safety

Safety engineering is a recognized engineering discipline that emerged when it became clear that the complexity and risk of modern technologies demanded a systematic approach to controlling hazards (Rodgers 1971; Roland and Moriarty 1983). The most important lesson of safety engineering is that safety is an important system requirement in its own right and must be designed into a product, not added on as an afterthought.

Safety requirements may actually conflict with other system requirements, suggesting a quite different design than would be obtained if cost and performance were the only considerations. Resolving such conflicts in a consistent and intelligent manner, rather than by default or at the whim of some individual, demands that safety requirements be explicitly separated out and that the responsibility for meeting them be assigned to someone with authority.

Today, government regulations, liability laws, and purchase contracts often require that vendors apply safety engineering practices. Safety standards for computer systems, including software, are being produced (for example, Department of Defense 1984); and the FDA includes computer software used to control some

types of medical devices within its regulatory authority (FDA 1987; 1989).

Safety Versus Reliability

It is important to distinguish safety from reliability. Not all software faults cause safety problems, and not all software that performs according to specification is safe. Reliability is a measure of how well the system does exactly what it is intended to do. Safety, on the other hand, deals with protection from hazards and limiting damage in the event of a failure. A safe system protects from hazards, whether its intended function is performed correctly or not. In fact, safety is often most concerned with what happens when the system does not work as expected. Specifications usually focus on what the system *shall* do. Safety requirements must also include what the product *must not* do. Safety analysis cannot assume that a system is free from faults. It must also consider how the system behaves in the presence of faults, or when the normal or specified operating conditions are violated in various ways--in other words, analysis of failures.

There is a huge literature on software quality assurance and software reliability, but most of it is oriented towards data processing and is not concerned with safety. There is a small but rapidly growing literature that specifically deals with the safety of computer-controlled systems, including some case studies (Baylis 1983; Bologna and Leveson 1986; Houston 1987; Lauber 1980; Leveson 1986a, 1986b; Quirk 1986). Designers responsible for product safety would also benefit from reading accounts of accidents involving technologically complex equipment (Perrow 1984). Incidents specifically involving computers are indexed in Neumann (1989). These accounts illustrate how seemingly unlikely combinations of events, poor design decisions, and human and institutional behavior interact to cause accidents. They act as antidotes to a normal human tendency to hope for the best and discount the importance of apparently unlikely problems, and they encourage a constructive attitude toward actively preventing hazards.

The Unique Problems of Computer Control

Computer-based process control systems present some new problems, problems quite unlike those encountered in electromechanical control systems. There is a large literature on the use of electromechanical controls in safety systems, and specialists in particular fields, such as radiation therapy, have available very detailed manuals that explain how to design and test safety systems for their applications (Post 1971). Corresponding manuals do not exist for computer control systems. The following sections review some of the difficulties.

Computer Hardware is Delicate

Computer hardware is less robust than electromechanical hardware. Modern solid-state electronics, including discrete logic modules, microprocessors, and computer systems, are far more vulnerable to environmental stresses than are relays. Extremes of heat and cold, modest electronic overvoltages, even static electric charge carried on an operator's clothing can temporarily disrupt or permanently damage solid-state circuitry. Extremely brief electronic transients or "noise spikes," which can only be detected by special test equipment, may cause mystifying and unreproducible system behavior. Control programs stored in what is supposed to be permanent, read-only memory may fade away as components age. Electrical interference from unexpected sources can induce serious hazards. There are several reports of programmable heart pacemakers being radically reprogrammed by interference from sources such as microwave ovens and department store anti-shoplifting devices (Neumann 1989).

Because solid-state electronics are more delicate than electromechanical devices, it is usually not possible to simply replace electromechanical controls with functionally equivalent solid-state equipment. The new equipment often fails because it proves to be vulnerable to electrical interference and other environmental disturbances that the older equipment could easily tolerate. It is usually necessary to provide the computer with a more protected electrical environment, constructed according to good packaging, grounding, and shielding practice. In addition, special signal conditioning and isolation circuitry is often required (Goodwin 1975; Nissley 1976; Cooling 1986). Techniques sufficient for personal computers and other consumer

electronics are not always adequate for more demanding process control environments.

Software Can Fail Catastrophically

Electromechanical components usually fail one at a time; consequently, many functions may continue to work even after one or more components have failed. It is not unusual for all software functions to fail simultaneously. In computer-controlled systems, many different functions are usually performed by a single processor. Distributed or multiprocessor control systems having more than one computer usually have more functions than they have processors. Different functions are controlled by different parts of the control program, which are run in rapid sequence on a single processor. This control program, or software, replaces, in effect, a large number of relays or other discrete components. Concentrating so much complexity into software provides much of the economy and flexibility of computer-controlled systems, but also makes them more vulnerable to errors.

Most programming languages provide some way to divide a program text into sections variously called subroutines, procedures, functions, modules, tasks, or processes (among other things). These sections are sometimes called software components, but this analogy is misleading. Even when the texts of two program sections appear to be completely independent (that is, they do not share any variables), they are in fact much more tightly coupled than is usual for electronic components; they share a vital resource--the processor itself.

Certain kinds of software errors can cause the processor to interrupt its normal sequence of operations and enter an abnormal state that prevents it from doing useful work. Such an event is colorfully termed a "crash." Crashes can be caused by many programming errors that are easy to commit: attempting to divide by zero, attempting to compute a number larger or smaller than can be accommodated by the processor hardware, attempting to read or write into a memory location that is not populated by a memory chip, attempting to use an array element whose subscript is larger than the size of the array, and so forth. Crashes can also be induced by hardware failures such as intermittent faults or electrical interference.

The behavior of a crashed program is completely unpredictable. It may halt in some apparently random state, or it may continue on, generating random output. Another kind of global program failure occurs when a particular program section seizes control of the processor and will not release it. These occurrences, in which the computer appears to be "stuck" or "hung," result from common programming errors, such as infinite loops (in which the program cycles endlessly through the same series of instructions) and deadlocks (in which tasks wait endlessly for events that never occur). All functions in a crashed or hung program stop working, not just the one containing the error. Systems in which several processors share common memory can be vulnerable because errors in one processor's program may cause it to corrupt instructions or data needed by other processors.

In data processing and scientific computing, program crashes do not contribute to accidents because they do not release energy directly. In these environments, programs run under control of a supervisory program called an operating system, which can usually recover control from a crashed or stuck program; if the operating system itself crashes, the operator can shut down and restart the computer, which often clears the problem. Process control systems, on the other hand, often have no operating system or include customized program sections that perform some of the functions of an operating system. There may be no opportunity for the operator to intervene in any useful way. Consequences of failure can be very serious: a runaway program could drive a radiation therapy machine gantry into a patient; a hung program could fail to terminate a radiation exposure and deliver an overdose.

It is important to note that including error checking and recovery coding in a program may not protect against the consequences of a crash, since a hung or crashed program may not even allow the error-handling code to run. Of course, errors less dramatic than crashes and infinite loops can also have serious consequences. In many process control systems, simply computing an incorrect value, or even taking too long to compute a correct value, can cause harm.

Software Reliability Cannot be Predicted from Component Performance

Hardware components and subsystems usually have known failure histories, and there are published values for such reliability measures as mean time to failure (MTTF) or mean time between failures (MTBF). When a system is entirely composed of components whose reliability statistics are known, the reliability of the entire system can be estimated by combining the reliability of the components according to the mathematical laws of probability. Such calculations are the source of assertions that certain safety-critical systems, such as aircraft controls, have a very low probability of failure (for example, one failure in a billion flight hours).

Estimates like these are difficult to establish with confidence, even for systems satisfying the assumptions of the theory, namely, those that break or wear out in a more or less random fashion. Control system hardware does satisfy the assumptions, since it is usually built from standard digital logic circuits, microprocessors, and minicomputers for which failure statistics may be available. (In practice, the reliability estimates for complex new components are uncertain. For example, units from early production runs of the popular Intel 80386 microprocessor, used in many personal computers, performed multiplication incorrectly, in some cases, due to a design flaw that was only discovered in the field [Spectrum 1987].) With regard to new software, however, there is no sound foundation for quantitative statistical failure estimates, such as MTBF. Software faults are design errors, not random equipment failures. Control software is customized for each product. A new control program is therefore a unique artifact with no performance history of any kind. The MTBF will be zero if the system is tested with an input that excites a fault.

Failure data should be collected during development and early field experience. It is usual to discover many faults early in a product's lifetime. The fault discovery rate gradually declines as more subtle problems are unearthed. There exist statistical software reliability models that attempt to predict the number of undiscovered faults remaining, based on the past history of failures (Goel 1985; Musa et al. 1987), but these models are necessarily less trustworthy than statistics gathered from mass-produced items. In practice, it is not possible to predict when a program will next fail, and it is not realistic to assign failure probabilities or measures, like MTBF, to programs just entering service. The main practical lesson of the software reliability models is that a program that has recently exhibited a lot of failures is likely to continue to fail. It is also important to note that a new version of a program may need to be considered a completely new program from the point of view of failure history. It seems difficult to accept that a new version containing only 5 percent new code is a new product, but flaws in that 5 percent could interact with the rest of the system in unforeseen ways and cause the whole system to fail. In fact, attempts to correct previously undiscovered errors are a major source of new errors.

Software Cannot Be Exhaustively Tested

Software, like any product, must be thoroughly tested. The huge literature on software testing describes testing criteria that are generally accepted as reasonable (Bezier 1984; Myers 1979). For example, tests should be designed to cause every statement in the program to be executed at least once, to probe every alternative of every control statement, and to exercise every function of the product. Unfortunately, a test set meeting these criteria will probe only a small sample of possible program behaviors. It is almost never possible to subject a program to all possible combinations and sequences of inputs, because the number of combinations is so large. Moreover, computer systems include memory, so programs can be sensitive to the order in which tests are presented.

It is difficult to generalize from any test result because software behaves discontinuously. There is no guarantee that the program will perform similarly with similar inputs. For example, if the program computes the expression $a/(x-y)$, an apparently innocent combination of inputs, where x very nearly equals y , will cause a program to crash with a divide-by-zero error, while an almost identical combination, where the value of x or y is slightly different, may be handled without mishap. This phenomenon contrasts with the behavior of hardware, where it is often very informative to test a device at a few points in its normal operating range and at the extremes of its range, because device behavior is constrained by physics to be more or less