BELIEF IN CORRECTNESS

Marshall D. Abrams, The MITRE Corporation, 7525 Colshire Drive, McLean, VA 22102, abrams@mitre.org

Marvin V. Zelkowitz, Institute for Advanced Computer Studies and Department of Computer Science, University of Maryland, College Park, MD 20742, mvz@cs.umd.edu

Abstract

In developing information technology, you want assurance that systems are secure and reliable. Correctness is an attribute that one strives for in order to achieve those goals, but you cannot have assurance or security without correctness. We discuss methods used to achieve correctness, focusing on weaknesses and approaches that management might take to increase belief in correctness. Formal methods, simulation, testing, and process modeling are addressed in detail. Structured programming, life-cycle modeling like the spiral model, use of CASE tools, use of formal methods applied informally, object-oriented design, reuse of existing code, and process maturity improvement are also mentioned. Reliance on these methods involves some element of belief since no validated metrics exist. Suggestions for using these methods as the basis for managerial decisions conclude the paper.

1. Introduction

"Engineers today, like Galileo three and a half centuries ago, are not superhuman. They make mistakes in their assumptions, in their calculations, in their conclusions. That they make mistakes is forgivable; that they catch them is imperative. Thus it is the essence of modern engineering not only to be able to check one's own work, but also to have one's work checked and to be able to check the work of others." [Petroski, 1985]

Assurance is defined1 as "the confidence that may be held in the security provided by a Target of Evaluation." Informally, assurance is a "warm fuzzy feeling" that the system can be relied upon to reduce residual risk to the predetermined level. Without delving into psychology, we observe that effectiveness and correctness both contribute to assurance. Effectiveness is determined by analysis of the specifications of the functional requirements; the environment in which the system will be used, the risks, threats, and vulnerabilities; and all the countermeasures, including physical, administrative, procedural, personnel, and technical. The system is considered effective if the result of this analysis is an acceptable residual risk. Correctness is determined by comparing the implementation of the countermeasures with their specification. The system is considered correct if the implementation is sufficiently close to the specification. Note that this definition of correctness is compatible with the concept of risk management and is closer to the concept of trustworthy than to error-free.

This paper shows how correctness can be established. All known methods contributing to correctness have shortcomings that make it impossible to establish correctness beyond reasonable doubt. That is, establishing correctness is a matter of belief, not proof. Under conditions of belief, we caution fiscal prudence in resources invested in assuring correctness. The major methods addressed in this paper are mathematical models, simulation, testing, process models and procedures. Minor methods, called silver bullets, include structured programming, the spiral model, Computer Aided Software Engineering (CASE) tools,

formal methods applied informally, object-oriented (OO) programming, reusing existing code, and process maturity. Cost benefit is offered as a measure for selecting which belief system to embrace. We recommend hedging one's investments by using more than one method.

Security-critical information technology (IT) systems2 are extremely dependent on correctness. In systems involving human life and safety, correctness is paramount. A security-critical IT system must do exactly what is identified in its specification and not do anything that is not so specified. Correctness of software always has to be with respect to a specification.

Various methods may be used to demonstrate correctness, but all are less than perfect and involve some element of belief in relying on the results of using that method. That is, it cannot be proven that a method is "good" or "better." The methods are complementary in contributing to correctness itself as well as in contributing to belief in correctness. There is a growing consensus that, to say the least, no one technique can provide adequate assurance (see, for example, [Butler, 1993]). David Parnas [Parnas, 1990], among others, has suggested that an "assurance tripod" is required: the combination of rigorous testing, evaluation of the process and personnel used to develop the system, and a thorough review and analysis of various products produced during development. In the pragmatic end, managerial judgment determines resource allocation to correctness and assurance. In this paper, we focus on practical product correctness and the various problems one has in achieving this correctness.

We should learn from branches of natural science and engineering that have been trying to understand complex systems far longer than computers have existed. One important objective is to recognize when simplifying assumptions are valid and when they are dangerous. One of the authors learned as a sophomore that "the essence of engineering is to make enough assumptions so that you can solve the problem, without assuming the problem away."

Let us consider whether formal theories of programming are good approximations of real programs executing on real computers. Although the theories are relatively simple, applying them to realistic programs vastly complicates the model. You cannot even assume simple axioms like "For all integers i, i+1>i" on fixed wordsize computers since integer i may "overflow" and have an unspecified, negative, zero, or the same value, depending upon the particular hardware executing the program. Mathematical models of computer programs generally do not accurately represent the subtlety of programs in an environment (i.e., execution on real hardware). In some sense, the mathematics of computer modeling belongs in the realm of applied rather than pure mathematics. When we use Ohm's law, Kirkoff's rules, etc., to design an electronic circuit or use Newton's laws to predict the orbit of a satellite, no one is saying that they have "proved" that the circuit works or that the satellite will be exactly where they said it would be. By the same token, when we model a computer program using some method such as Hoare's [Hoare, 1969] we then have some confidence (maybe little) that the program when executed will behave much as we predict (but perhaps not exactly like we predict-e.g., integer overflow). This requires that even simple programs have complex proofs in order to show that the mathematical properties of the program behave as desired. Simple formalisms for programs are too complex to accurately represent most programs in execution on physical machines.

This insight shows that formalisms in programming are very different from formalisms in the natural sciences. In the natural science, you have a theory (e.g., Laws of Motion) that is a good approximation to the physical interactions among objects. In physics, a sufficiently accurate approximation gives useful

results. In contrast, for programming, you must approximate the program and the hardware (e.g., assume integers are infinite) in order to have any relationship to the formal model. A key difference is lack of continuity. In programming, disastrous examples of integer overflow and other discontinuities show that the supposed approximations are not necessarily close. Use of discrete logic to model these leads to expressions of enormous complexity [Parnas, 1985]. Alternatively, models could incorporate known characteristics and limitations of the computer to increase their veracity. We do not wish to compare good models of physics with bad models of computers. Newton's laws do not work well for objects at near the speed of light or for objects that are not in inertial frames of reference. Likewise, a Hoare model of computer system behavior is a poor representation if the integer values are at or near the overflow. One would need to modify the model to accommodate the overflow behavior. Once having done so, the model would be better.

Several methods have been developed and been accepted over time to demonstrate the correctness of computer programs. None of these heuristics are true in the sense that they portray absolute infallibility of the method. Each has proponents and detractors. In the next section, we describe these methods, explore ways in which each accomplishes its task, and draw some conclusions from this analysis.

2. Correctness methods

Several techniques are regularly employed to show that a computer program does exactly what it is supposed to do and nothing else. The first two described below, formal methods and simulation, analyze the program and derive properties about it. The third, testing, experiments with program behavior, perhaps using some information derived by application of the first two techniques. The fourth technique, process models and procedures, looks at the development process itself under the assumption that good development practices result in good software.

Each method is described briefly, emphasizing its advantages, disadvantages, and contributions to our belief system. A common distraction with all methods is the complexity of execution. The steps, processes, or manipulations that constitute the practice of the method can be so overwhelming that perspective is lost. We agree with Hamming [Hamming, 1962] that "the purpose of computing is insight" and that it is difficult to retain perspective and insight in the face of complexity. It is very easy to get caught up with all the mechanics of employing a method so that in practice the mechanics get emphasized at the expense of understanding.

"When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind" [Kelvin, 1881]. Metrics of correctness need to be developed and applied to individual methods and combinations of methods. We need to replace belief with analysis if at all possible. While early work on the Capability Maturity Model [Paulk, 1993] and the Experience Factory [Basili, 1992] show that we may develop such metrics, more needs to be done.

2.1 Formal Methods

The use of formalisms stems from two related observations: natural language tends to be imprecise, and in achieving precision, there is the potential for automation. Mathematical notation has the advantage of precision and is

associated with rigorous, logical thinking that assists in reducing ambiguity. In principle, formal models of IT systems can support all phases of the system development process -articulation of policy for use, high-level architecture, design, and implementation. Today, formal models of security policy help perfect understanding and development, especially of new policies. While formal specifications are used in Europe, they have not made much of an impact in the United States. No language is likely to be a cure- all in achieving higher levels of abstraction, and more natural models of problem spaces, for all problem spaces.

In discussing formal methods, we have to be sure to differentiate them from formalized methods, such as Computer Assisted Software Engineering (CASE) tools, structured analysis, and other mechanized methods for developing source programs [Rushby, 1993]. In using formal methods, one traditionally begins with a formal description of the specification of a software system according to some underlying mathematical model and the realization of that specification as a concrete design or source code implementation. (Other possibilities are to start with a description of how the system is to be used, or to let an automated deduction system participate directly in the construction of later design and implementation stages.) Using mathematical principles, one shows that the program agrees with the model. For example, axiomatic verification, perhaps the oldest of the formal techniques, assumes we have a program S, a precondition (specification) P that is true before the execution of S, and a postcondition (output specification) Q. We need a proof that demonstrates: (1) the relationship among S, P, and Q that determines the effect program S has on P to assure that Q will be true after execution terminates, and (2) program S does indeed terminate if P is true initially [Hoare, 1969]. If we derive a set of axioms for each statement type in our language (e.g., rules for describing the behavior of the if statement, the while statement, the assignment statement), then we have tied program correctness to the problems of generating correct mathematical proofs. But we still have not proved that the program when executed on a specific computer is correct because of the very problems raised earlier. At best we have shown that the formal description of the program satisfies its specification (i.e., produces the given post condition when the precondition is true) [Fetzer, 1988]. Our confidence in the correctness of the program is dependent on our confidence that our formal model is an accurate representation of the target computer.

As described previously, when we use formal models we need to suppress details to make the models tractable. Unfortunately, many of the details suppressed in the formal models are implementation dependent and security relevant. Formal models are losing ground to the complexity of networked and distributed systems. It is difficult to scale up the traditional use of formal methods to large complex systems. While they may appear to work satisfactorily on small "toy" problems, there has been little evidence that they scale up very well [Parnas, 1985].

"Larger examples are necessary to demonstrate how these concepts scale up" [Youngblut, 1989, p. 58]. Formal models are often applied to complex systems combined with other belief systems. For example, variants of the Bell-LaPadula security policy model [Bell, 1974] are often cited as the basis of operating system security, but the actual implementations also include security- relevant processes, called trusted or privileged, that are not formally modeled. Belief that security is preserved after introduction of these processes is often established by non- formal means. Practitioners of formal modeling sometimes appear to forget about the assumptions and simplifications that were made to make their models tractable and fail to caveat the applicability of their

results to the real world. This is an error on the part of the practitioners. A great deal of the simplifying assumptions are made because the modelers simply do not know how to model some of these features (although many are certainly susceptible to being modeled), or the resources available do not permit modeling the necessary details. Perhaps the practitioners are not experienced enough in this kind of mathematics.

Within the limits imposed by the simplifications and assumptions made for the sake of tractability, formalism can be used both to determine correctness of the implementation and adherence of the system to certain properties. We can prove that a given procedure must return a certain value and also show that certain policies are never violated. Many observers believe that formal policy models have their maximum benefit in removing inconsistencies, ambiguities, and contradictions in the natural language policy statement. The process of formalizing the policy aids in clarifying the policy. This process then has the secondary benefit of making a clearer statement of policy to the implementors.

Although formal methods are based on mathematical proofs, we must realize that even mathematical proofs may have flaws. "Outsiders see mathematics as a cold, formal, logical, mechanical, monolithic process of sheer intellection... [but] Stanislaw Ulam estimates that mathematicians publish 200,000 theorems every year. A number of these are subsequently contradicted or otherwise disallowed, others are thrown into doubt, and most are ignored. Only a tiny fraction come to be understood and believed by any sizable group of mathematicians" [DeMillo, 1979, p. 272]. Although mathematicians do not like to admit it, correctness can be likened to a social process-it is only the test of time where no flaw has been discovered that builds our confidence in the ultimate truth of a theorem. All scientific processes have flaws. Petroski [Petroski, 1985] argues that failure is an important part of engineering design. It is only when things fail that we understand how to make them better. How well would we be designing bridges if none ever collapsed? Either we have overbuilt them to a point of economic stupidity, or we have never stressed them sufficiently. We hope that by our continuing (unsuccessful) attempts to model computers, we are learning something.

2.2 Simulation

Simulation is the development of a simplified version of a system's specification by eliminating non-critical attributes to develop a system that exhibits relevant properties. By ignoring certain properties, it is often possible to quickly and inexpensively build simpler versions of a system. Using this simulation, security-related principles can be more readily developed and examined. This increases our belief in the ultimate specification since we have demonstrated the existence of an implementation that already has the desired properties.

While we can simulate a system to test the security policies, the interaction of these policies with the assumed-away specifications of the complete system severely lowers our belief in the correctness of the overall system with respect to security. By definition, one is "abstracting away" non-essential aspects of the system when doing simulation and modeling-yet it is very hard to develop "non-interference proofs" for those missing aspects, so that you have confidence that they really won't change the behavior of interest in the "real" system. It is only by testing (and/or formalism) applied to the complete system that adds to our belief in this product-although the existence of a simulation that implements our security policy does provide a sort of existence proof on policy

and increases our confidence (i.e., belief) in a complete implementation. (See spiral model discussion, below.)

2.3 Testing

Testing demonstrates behavior by executing a system using a selected set of data points to show that the system executes correctly on those points. The assumption is made that if the set of data points is chosen appropriately, then the behavior of the system for most data points will be analogous to the selected data points. If we believe that the selected data points are representative of the domain of data in which we interested, we have confidence in the correctness of our implementation. Choosing the selected data points and the best method of testing our program are our major decision steps toward determining our belief in the correctness of this system. Knowledge gained from formal methods, code analysis, and simulation can help focus the selection. As pointed out by Leveson [Leveson, 1992], "testing researchers have defined theoretical ways of comparing testing strategies both in terms of cost and effectiveness (for example, [Weyuker, 1991]), formal criteria for evaluating testing strategies (for example, [Goodenough, 1975]), and axioms or properties that any adequacy criterion (rule to determine when testing can stop) should satisfy (for example, [Weyuker, 1986])." Analytic results can also indicate when statistically significant measurement results have been obtained [Mamrak, 1979].

Testing methods can be divided into functional, performance, failure-mode, and, for security, penetration. Functional testing includes testing against a catalog of flaws previously discovered in this or other systems. The major thrust of security testing is in penetrating (i.e., violating the security policy), thereby measuring the resistance to anticipated threats. The presence of anticipated threat actions, possibly by a malicious adversary, distinguishes the security concerns in a system.

Testing functional specifications is usually achieved by black- box testing, in which the tester only has access to the specifications of the program, while testing specific program behavior by understanding the design is achieved by glass-box (a.k.a. white-box) testing, in which the tester has access to the internal source code of the program. Security testing of high-assurance systems proceeds with extensive documentation of design and implementation. Varying degrees of assurance are obtained according to the information available to the testers, including security kernel code, design documentation, and formal models. The value of penetration testing depends on the experience of the testers and the methodology employed. IV&V (Independent verification and validation), where a group independent from the developers is charged with testing a system, is sometimes effective in finding errors that developers who "know" the source program sometimes overlook. However, IV&V is expensive and many applications, especially ones without high reliability requirements, do not benefit from this added level of assurance [Page, 1985].

The classical example by Dijkstra shows that exhaustive testing cannot prove correctness of any implementation. To prove the correctness of "a+b=c" on 32-bit computers would require 232x232 = 264 or over 1019 tests. At a rate of even 108 tests per second, that would require 1011 seconds or over 3,000 years. Perhaps we should ask ourselves whether we really have so little understanding of the operation of a computer that we have to test addition, for example, for all possible addends to be convinced that the addition function is working correctly? Under what conditions can we state a general argument that works in the face of overflow? Although it is recognized that testing cannot be

exhaustive, testing has a very strong intuitive appeal and constitutes a very strong basis for belief in correctness.

Testing always involves comparing the actual results of execution with anticipated results. One way to capture anticipated results is to test an executable specification of a prototype. Once this is done, it is possible to automatically execute the system being tested and its specification in parallel, and to automatically compare the results, thereby greatly increasing the number of feasible test cases [Taylor, 1985].

2.4 Process Models and Procedures

All of the previous techniques depend upon subjecting a program to one of the discussed methods to increase confidence that the program exhibits correct behavior. However, as we have often stated, this is extremely difficult to do. As an alternative, perhaps it is easier to understand the mechanisms used in developing the program under the belief that correct methods yield correct programs. The idea underlying process models is that understanding what you are doing is a necessary step to improvement. By using a simple, well understood process to develop software, we have belief that the ultimate product best meets our needs. Two process models currently enjoy favor: waterfall and spiral. The United States Department of Defense (DoD) standards imply (but do not require) use of the former in management of software development.

The waterfall model [Royce, 1970] conceives of software development as a linear process based upon a set of deliverable artifacts. There are easily recognized milestones between steps in the process. Although the mechanisms of the process are generally obscure-only the results of the process are visible. Therefore, the waterfall model uses these products-a specifications document, a design document, a source file, and the results of testing, for example. These milestones can support a management strategy of schedules and reviews. Recognition that the process is not perfect led to the introduction of feedback paths in the model. If drawn as a waterfall of steps, the feedback paths suggest salmon swimming upstream. The feedback paths represent knowledge gained in latter steps that affect activities and decisions made earlier. It may be necessary to adjust, or even abandon, earlier work as a consequence of feedback. In practice, schedules tend to not allow for such corrective action. Non-technical project managers are often determined to meet their schedules, no matter what the consequences [Stillman, 1993].

Because of all of these deficiencies, belief in the waterfall model as a useful methodology for developing software that satisfies its specification has been slowly decreasing, and an alternative spiral model has been gaining favor [Boehm, 1988]. The spiral model emphasizes the process of developing software rather than the resulting products. It is also called a risk- reducing model, since the basic premise is to develop and prototype a solution, evaluate the risks of adding specifications, and repeat the process. Each cycle of the model creates a more complex version of the system, with the ultimate prototype being the final system itself. At each stage, we use Occam's razor to simplify our solution, we make the process of development as visible as possible, and we try to quantify the risks involved in continuing development. Thus, our belief in the solution should be higher than with the hidden processes inherent in the waterfall model. The spiral model emphasizes the repetition of basic activities at progressive stages of a project. The exact activities change as the project matures, but such activities as design, implementation, testing, evaluation, and planning are related. Changing requirements are more easilly accomodated. The cost is represented by the radial distance in a polar coordinate system and the

activities occur at a specified polar angle. Progress is assumed proportional, or at least related to, cost. While the theory of the spiral model accommodates redesign and backtracking, the imposition of schedules can have exactly the same effect as on the waterfall model.

3. Choosing among alternative beliefs

Software engineers promote one technique after another as the "silver bullet" [Brooks, 1987] solution to all our problems. This section examines the most popular silver bullets.

3.1 Tarnished Silver Bullets

To address correctness in system development, many techniques have been proposed as potential solutions (e.g., see [Chang, 1993; Davis, 1993]). All techniques involved a measure of belief as groups of professionals argued among themselves regarding the appropriateness of their favorite method. None has completely provided the warm fuzzy feelings we want :

a. Structured programming (e.g., "goto-less programming" of the 1970s) makes programming easy and correct. Twenty years of experience have shown that quality has improved, but not to the level initially proposed. There is a relationship between the restrictions imposed by using only the appropriate control structures and formal verification of the source code produced; however, errors still occur in such programs [Zelkowitz, 1990].

b. The spiral model is superior to the waterfall model. The spiral model was an improvement in that it emphasized the process of software development with attendant interest in the management, risk evaluation and reduction, and prototyping aspects of the process. Note that this is an example of Petroski's theses. Because the waterfall methodology has proved inadequate to produce good software, a new methodology (spiral) has been introduced. When it is determined that the spiral also is inadequate, creative people will develop a new system. Since we do not have good measures of correctness, it is difficult to know how to make the process better. Note also that the spiral model and the waterfall model that it replaced both represent a similar set of practices as actually implemented by many organizations.

c. CASE tools will supplement the intelligence lacking in today's programmers. Unfortunately, the tools have not added much intelligence and today's programmers could still use additional help. Case tools suffer from the same problem as the other software we are discussing: they have errors (all software has errors), and they are only as smart as their developers.

d. Formal methods applied informally (e.g., languages like VDM and Z) can improve the process. While this seems to be true, it has yet to be demonstrated that this approach results in the correctness that we need for security-related systems. It is not clear that our belief in these specification techniques will be high enough to eliminate the need for alternate mechanisms. Nor is it clear that our beliefs are the only ones that count. See [MacKenzie, 1992] for a discussion of mathematical arguments that qualify as proof in a court of law.

e. Object-oriented (OO) programming and design will replace conventional design techniques, and languages that implement such processes (e.g., C++) will replace other languages (e.g., Pascal, Ada, FORTRAN). This concept represents one of the newer trends in program design. We do not have enough evidence to judge the effects of OO design on security. This technique does encapsulate some

of the formal data-structuring mechanisms into the programming language; however, it must still be observed what effects it will have on overall system correctness. (Note that this is just the current version of the traditional silver bullet, "Language X will make programming easier." In the 1960s, we had COBOL and then PL/I, in the 1970s we had Pascal, in the 1980s we had Ada, and now we have C++.). Each language is perceived to have failed in achieving some objective. Hence, someone develops a new language to correct the flaws. This cycle will probably never end, as it is not likely that any one language will be perfect for all applications.

f. Reusing existing code is the solution. Since code proven correct once need not be so proven again, one only needs to create a library of reusable components. While reusing existing code is an admirable goal, we still do not have the technology to implement this process. While we can create write-only libraries of reusable components, we have no process available that enables us to determine the specifications of an existing library component and whether it fulfills the specifications for another application. Current interests in domain-specific architectures and faceted classification schemes are both attempts at understanding the functionality of reusable components. We reuse hardware components all the time, in the sense that we manufacture identical copies of circuit packages and other components. Each component conforms to some specification of performance and behavior that is described in components manuals. Why can't we do something similar with software?

g. Process maturity improvement is today's salvation [Paulk, 1993]. Current thinking is that improving only the process without looking at the ultimate product being produced is all that is necessary to produce quality software. While it should greatly improve the production of software from many organizations that currently have no such process, as shown often in the past, this is a naive approach to producing correct software.

We do not mean to say that the above techniques are failures. All, to some extent, improve upon the quality and correctness of the resulting program that is produced. Programming as taught in the universities and practiced in industry today is radically different from that of the 1960s. However, the important point is that none of them achieves the level of correctness that would support our belief in that technique over all others.

3.2 Which Belief System to Embrace

Resources must be allocated among the correctness methodologies. While management has been described as the art of making decisions based on inadequate information, the quality of decisions is often improved by providing more information. Installation and use of security-critical IT systems cannot wait for proofs of efficacy or development of metrics for determining cost-benefit. Managers will need to continue to make decisions whether or not to employ IT. The managerial authorization and approval granted to an IT system to process sensitive data in an operational environment is, in theory, made on the basis of analysis and certification of the extent to which design and implementation of the system meet pre-specified requirements for achieving adequate security. Security objectives can be met by a combination of technical means within the system and physical and procedural means outside the system. In this theory, when management accredits the system, management is accepting the residual risk.

How can we address this residual risk? While we have no clearly defined metric for this, we do have examples of systems that seem to adequately address our security concerns. One avenue of research is increased study of these

"artifacts"-the systems, designs, and specifications that have helped produce acceptable solutions. This knowledge should enable us to produce better models in the future. However, today there is no way to measure the residual risk, nor is there a metric for cost-benefit. So, how is a decision made? Since computer and management science cannot help verify a decision, the experienced manager's intuition cannot be dismissed. Experience probably includes comparison with previous efforts, the correctness of which has become better known over time. One must be careful to distinguish between management saying "I did this before and it worked" versus "I feel safe using this since I used it before, while this new technique is unknown to me." The first statement encapsulates the experiences of good management, while the second statement reinforces unscientific prejudices. The real problem is how to differentiate among good science, common sense, and stubborn stupidity.

In deciding which belief system to embrace, the prudent manager probably hedges by using more than one system. Various combinations of formalism, testing, simulation, and process may be employed. Since cost is one of the attributes we need to address in evaluating the overall quality of the product, it is prudent that management should adequately choose from among the techniques those that meet required cost constraints yet still meet functional requirements for the product.

4. Recommendations

Given the absence of metrics for any of the belief systems, the inherent difficulty in using any of them, and the lack of a repository of correctness artifacts to study and evaluate, the authors do not propose to solve this problem with a pronouncement of correct technique. Our focus is to increase the awareness of the technical and managerial segments of the IT security community to the limitations of each of these techniques. We attempt to increase understanding of the need to address more than one solution to the multifaceted correctness problem.

We view the glass as being half full. We do not advocate that anyone abjure his belief(s) in correctness. Rather, we suggest that attempts to prove beliefs are bottomless pits. Unless some breakthrough occurs, we advocate treating this aspect of software engineering pragmatically. Just as engineers built steam engines (see [Leveson, 1992] for further analogy) before the science of thermodynamics was developed, the software engineering community can build software systems based on intuitive and pragmatic notions of how to attain correctness and other aspects of quality. At least now, we should acknowledge practicing an empirical discipline.

At the risk of appearing cautiously optimistic, we hesitantly endorse four interrelated strategies. The exact allocation of resources among the strategies remains a technical management decision. Looking at the mature methods available today, we tend to agree with the perceived consensus that a combination of the following should be employed:

* Evaluation of process, personnel, and abilities to identify and reinforce positive attributes

* Thorough review and analysis of intermediate products during development with sufficient time and resources allocated to correct deficiencies

* Rigorous testing based on the preceding analysis

* Recognition of critical points in system development

  - Point of diminishing return for application of any method
  - When a development should be terminated for cause or to stop hemorrhaging

Looking forward, we see promise in combining aspects of program reuse and object orientation. The possibility of employing object self-protection in security architecture should be considered.

Each of the techniques described in this paper has an aspect that help increase our belief in the correctness of an implementation, yet each is fraught with some dangers. Each technique comes with some, generally high, cost for its use. It is imperative that management addresses each as aids in developing security-critical IT systems and not arbitrarily dismiss any of them. We should:

* Be cognizant of the limitations of each

  - Belief in correctness should be relative

* Be prudent in establishing realistic assurance requirements for a given system that are measurable, achievable, and cost- effective

* Resist the temptation of unachievable elegance and perfection

* Differentiate between research and operations

  - Define achievable specifications
  - Accept residual risk

5. Acknowledgments

We appreciate the contributions from the following individuals on previous drafts of this paper: Rochelle Abrams, Sharon Fletcher, Lester Fraim, John Gannon, David Gomberg, Ronald Gove, Chuck Howell, Jay Kahn, Carl Landwehr, John McLean, Jonathan Millen, Jonathan Moffett, Jim Purtilo, Jim Williams, John P. L. Woodward, and the anonymous reviewers. Research support on this activity for Marshall Abrams was provided by the National Security Agency under contract DAAB07-94-C-H601, and for Marvin Zelkowitz was partially provided by NASA grant NSG-5123 from NASA/Goddard Space Flight Center to the University of Maryland.

6. References

Basili, V. R., G. Caldiera, and G. Cantone, 1992 "A Reference Architecture for the Component Factory," ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 1, pp. 53-80.

Bell, D. Elliott, and Leonard J. LaPadula, April 1974, Secure Computer Systems: Unified Exposition and MULTICS Interpretation, MTR 2997,. The MITRE Corporation, Bedford, MA. Available from National Technical Information Service, AD/A 020 445.

Boehm, B., May 1988, "A Spiral Model of Software Development and Enhancement," IEEE Computer, Vol. 21,~ No. 5, pp. 61-72.

Brooks, F., 1987, "No Silver Bullet: Essense and Accidents of Software Engineering," IEEE Computer, Vol. 20, No. 4, pp. 10-19.

Butler, R. W., and G. B. Finelli, 12 January 1993, "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software,"IEEE Transactions on Software Engineering , Vol. 19, No. 1, pp 3-12.

Chang, C., 5 September 1993, "Is Existing Software Engineering Obsolete?," IEEE Software , Vol. 10, No. 5, pp. 4-5.

Commission of the European Communities, 28 June 1991, Information Technology Security Evaluation Criteria (ITSEC): Provisional Harmonized Criteria, Luxembourg: Office for Official Publications of the European Communities, Version 1.2.

Davis, A., 5 September, 1993, "Software Lemmingineering," IEEE Software, Vol. 10, pp. 79-84.

DeMillo, R., R. Lipton and A. Perlis, May, 1979, "Social Processes and Proofs of Theorems and Programs," Communications of the ACM, Vol. 22, No. 5, pp. 271-280.

Fetzer, J. H., September 1988, "Program Verification: The Very Idea," Communications of the ACM, Vol. 31, No. 9, pp. 1048-1063

Goodenough, J. B., and S.Gerhart, June 1975, "Toward a Theory of Test Data Selection," IEEE Transactions on Software Engineering Vol. SE-1, No. 2.

Hamming, R., 1962, Numerical Methods for Scientists and Engineers, McGraw Hill.

Hoare, C. A. R., October, 1969, "An Axiomatic Basis for Computer Programming," Communications of the ACM, Vol. 12, No. 10, pp 576-583.

_____ August 1986, "Mathematics of Programming," Byte, pp 115-121.

Kelvin W. T., 1881-1884, Popular Lectures and Addresses.

Knight , J. C. and D. M. Kienzle, 1992, "Preliminary Experience Using Z to Specify a Safety-Critical System," Proceedings of 1992 Z Users Workshop, Springer-Verlag.

Leveson, N. G., May 1992, "High-Pressure Steam Engines and Computer Software," Proceedings International Conference on Sostware Engineering, Melbourne, Australia.

Mamrak, S. A. and M. D. Abrams, December 1979, "A Taxonomy for Valid Test Workload Generation," Computer, pp. 60-65.

MacKenzie, November 1992, "Computers, Formal Proofs, and the Law Courts," Notices of the American Mathematical Society, Vol. 39, p. 9.

M. C. Paulk, B. Curtis, M. B. Chrissis and C. V. Weber, "Capability Maturity Model for Software, Version 1.1," IEEE Software, Vol. 10, No.4, (July, 1993) pp. 18-27.

Page G., F. E. McGarry and D. N. Card, June, 1985, Evaluation of an independent verification and validation methodology for flight dynamics, NASA/GSFC Technical Report SEL 81-110.

Parnas, D. L., December 1985, "Software Aspects of Strategic Defense Systems," Communications of the ACM, Vol. 28, No. 12, December 1985, pp. 1326-1335.

Parnas, D. L., A. John van Schouwen, and Shu Po Kwan, June 1990, "Evaluation of Safety-Critical Software," Communications of the ACM.

Petroski, H., 1985, To Engineer is Human: The Role of Failure in Successful Design.publisher

W. W. Royce, 1970, "Managing the Development of Large Software Systems: Concepts and Techniques," Proceedings IEEE Wescon..

Rushby, J., December 1993, Formal Methods and the Certification of Critical Systems, Technical Report C3L-93-7, Stanford Research Institute.

Stillman, R., March 22, 1993, "Software Development: Neither Economics nor Engineering," keynote address, Third Annual Software Engineerring Economics Conference, The MITRE Corporation.

Taylor, T., October 1989, "FTLS-Based Security Testing for LOCK," Proceedings of the 12th National Computer Security Conference, pp 136-145.

Weyuker, E. J., December 1986, "Axiomatizing Software Test Data Adequacy," IEEE Transactions on Software Engineering Vol. SE-12, No. 12, pp. 1128-1138.

Weyuker, E. J., S. Weiss, and D. Hamlet, October 1991, "Comparison of Program Testing Strategies," Proceedings of the Fourth Symposium on Software Testing, Analysis and Veriofication (TAV4), Victoria, B.C., Canada, pp 1-10.

C. Youngblut, et al, February 1989, SDS Software Testing and Evaluation: Are View of the State-of-the-Art in Software Testing and Evaluation with Recommended R&D Tasks, Institute for Defense Analysis Report p. 2132.

Zelkowitz, M. V., November 1990, "A Functional Model of Program Verification," IEEE Computer Vol 23, No. 11, pp. 30-39.

1 Definitions of assurance, correctness, and effectiveness are taken from the Information Technology Security Evaluation Criteria (ITSEC) [Commission of European Communities, 1991]. Better definitions may be available by the time this paper is published.

2 The term IT system includes all sizes of computer systems, from super mainframes to desktop units to embedded components and controllers, as well as networks and distributed systems.