Computers & Security, 14 (1995) 719–738

0167-4048(95)00022-4

Refereed paper

Striving for correctness*

Marshall D. Abrams[1] and
Marvin V. Zelkowitz[2]

[1]*The MITRE Corporation, 7525 Colshire Drive, McLean, VA 22102, USA (abrams@mitre.org)*
[2]*Institute for Advanced Computer Studies and Department of Computer Science, University of Maryland, College Park, MD 20742, USA (mvz@cs.umd.edu)*

In developing information technology, you want assurance that systems are secure and reliable, 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, object-oriented design, reuse of existing code are also mentioned. Reliance on these methods involves some element of belief since no validated metrics on the effectiveness of these methods exist. Suggestions for using these methods as the basis for managerial decisions conclude the paper.

Keywords: Assurance, Belief, Correctness, Formal methods, Mathematical models, Metrics, Process models, Risk management, Security testing, Simulation, Silver bullets, Trustworthiness.

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" [1].

1.1 Security and software engineering

Security engineering is part of computer, or Information Technology (IT), engineering, encompassing elements of hardware, firmware, and software. There is a delicate balance in focusing on the security specialization to the exclusion of related fields. More progress in the security specialization probably results from the tight focus, but some relevant events in related fields may not receive the deserved attention. This paper attempts to correct such myopia concerning the software engineering topic of *correctness* or *trustworthiness*. Most of what we have to say in this paper is well known to the software engineer. At the same time, much of it is new, challenging, and perhaps controversial to the security engineer. In order to substantiate our position and provide sufficient pointers for further study, we have perhaps been excessive in the reference citations provided.

1.2 Abstraction layers

One way of thinking about the various technologies upon which security builds is a series of

*A condensed version was previously published in the *Proceedings of the 17th National Computer Security Conference* (USA) under the title "Belief in Correctness."

0167-4048/95/$9.50 © 1995, Elsevier Science Ltd

abstraction layers, conceptually illustrated in Table 1. The security engineer must understand that the device designer, circuit designer, and operating system architect have different viewpoints. Each specialist assumes that the interface provided to him or her by the underlying layer is primitive and trustworthy. This trust is a consequence of specialization. Engineers working at one technological level of abstraction are usually not prepared to investigate and determine the trustworthiness of the resources with which they work. For example, software experts rarely know about hardware design. However, they tend to view hardware as a monolithic entity and to trust it. This trust may or may not be warranted. The hardware may be failure prone due to errors in design or fabrication, the assumptions upon which the hardware is being used may be false, or it may also have been built with malicious intent to sustain the same kinds of attacks as are commonly implemented in software, such as viruses and Trojan horses. See [2] for further discussion.

Similarly, software experts who build trusted computing bases or communications protocol interpreters are users of supporting software, such as compilers and editors. They assume that this supporting software is trustworthy. While this is usually the case, Thompson [3] eloquently advises that one should be careful about extending trust. Recent work has described critical issues related to software trust and has proposed a set of criteria classes for measuring and comparing trust [4].

Addressing the trustworthiness of these layers is a matter of risk management. Absolute risk avoidance would address every possible level of risk. Risks might exist in the design of the chips, the side-effects of instruction set design (especially unimplemented instructions in complex instruction set architectures), or the security flaws in all supporting software. It has been common when confidentiality was the only security policy to assume that mass-produced bedrock was a sufficiently low risk that it could be ignored. Consideration of integrity and availability as security policies may justify reconsideration.

TABLE 1. Abstraction layers

Abstraction layers
Applications
Security subsystem
Operating system
Compilers, loaders, etc.
Circuit design and fabrication
Semiconductor chip design

1.3 The gods have clay feet—the emperor is naked

This paper tends to proclaim that the gods have clay feet or that the emperor is naked. These are never popular sentiments. They are presented as constructively as possible, but we humbly acknowledge that we have no completely satisfactory answer. Our overall challenge to the community is the traditional engineer's problem of finding cost-effective ways of applying the knowledge and skill base to the solution of social problems and requirements. This paper looks at the practical application of research results and finds a lack of evidence to support the very strong beliefs in the efficacy of various methods for increasing IT security.

1.4 Assurance, effectiveness and correctness

Assurance is defined[3] 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 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

[3]Definitions of *assurance*, *correctness*, and *effectiveness* are taken from the Information Technology Security Evaluation Criteria (ITSEC) [5]. Better definitions may be available by the time this paper is published.

comparing the implementation of the countermeasures with their specification. The system is considered correct if the implementation is sufficiently close to its 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*.

1.5 Major methods and panaceas

This paper exhibits methods used to establish correctness. All current methods contributing to correctness have shortcomings that make it impossible to establish correctness beyond reasonable doubt. That is, establishing correctness becomes a matter of belief, not proof. For each technique we describe attributes for these techniques and show its strengths and weaknesses. We show how to best use that method for increasing our belief in the trustworthiness of our system. 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. Prior panaceas, called silver bullets, include structured programming, the spiral model, computer-aided software engineering (CASE) tools, formal methods, 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. We regret being unable to offer better guidance. We can only suggest that the lack of a definitive answer is characteristic of many management problems where decisions must be made based on insufficient evidence. Perhaps it would be worthwhile if a consensus could be developed in the security engineering community as to what constitutes good practice at the present time.

Security-critical information technology (IT) systems[4] 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, [6]). David Parnas [7], 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 as a way to minimize risk. In the pragmatic end, managerial judgment determines resource allocation to correctness and assurance. Being unable to offer any substantive justification, we observe that recommendations, such as Parnas', are often unsubstantiated and contentious. The mechanism for reaching a consensus is not obvious. Thoughtful discussion, such as this paper, are certainly part of the scientific and technical tradition. In this paper, we focus on practical product correctness and the various problems one has in achieving this correctness.

1.6 Understanding complex systems

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 engi-

[4]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.

neering 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 actual hardware. 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). The mathematics of computer modeling belongs in the realm of applied rather than pure mathematics.

When we use Ohm's law, Kirchhoff'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 "proven" that the circuit works or that the satellite will be exactly where the model predicts it would be. These laws are empirical observations that have stood the "test of time" to represent physical reality. However, when we model a computer program using some method such as Hoare's [8] we have some confidence that the program when executed will behave much as we predict, but perhaps not exactly like we predict (e.g. integer overflow). To accurately model a program's execution 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 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 [9]. 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 speeds close to 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 near overflow conditions. One would need to modify the model to accommodate the overflow behavior. 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 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 to 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 [10] 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" [11]. 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 [12] and the experience factory [13] show the role of metrics in the development process, more needs to be done to totally build an effective measurement model into the development process.

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. Formal methods have long been associated with security-related software [14]. Today, formal models of security policy help perfect understanding and development, especially of new policies. While formal specifications have made some impact 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 techniques for developing source programs [15]. 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 realizes (i.e. builds) that specification as a concrete design or source code implementation. This does not preclude the use of automated tools or an automated deduction system to participate directly in the construction of later design and implementation stages. Using mathematical logic, 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 must develop 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 [8]. 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 proven 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 postcondition when the precondition is true) [16].

Our confidence in the correctness of the program is dependent on our confidence that our target computer is accurately represented by the formal model.

2.1.1 Models of complex systems

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. A distributed application usually has multiple components with a multiplicity of entrance and exit paths. Describing the pre- and postconditions for such systems becomes unmanageable. 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 [9].

One failure of the "formal methods community" in developing such models is that the models often are not grounded in the very problems that software engineers have in developing correct programs. "Doing our work in isolation, and then trying to impose our ideas on the real world is bound to fail" states Harel [17]. The models that the theorists need to use to base their theories on must be grounded in the problems that programmers face in developing correct software.

"Larger examples are necessary to demonstrate how these concepts scale up" [18]. Formal models are often applied to complex systems combined with other belief systems. For example, variants of the Bell-LaPadula security policy model [19] 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. Evaluation of moderate assurance general purpose products by NSA today require a model interpretation if the implementation violates the model, so that the developer can provide justification, through analysis, that the model violation does not violate the security policy. The justification at least requires the developer to think about the problem before violating the formal model, but it is still an informal plausibility argument that still needs to be accepted with skepticism. The value of the rigor of the formal model is seriously diminished by the informal argument.

2.1.2 Assumptions and simplifications

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. Brunnstein [20] puts it this way: "any abstraction makes simplifying assumptions whose consistency with reality is impossible to prove; therefore, formal models reflect only syntactic–semantic levels, they *never* reflect pragmatic levels."

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 implementers.

- 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... [however] 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" [21]. 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 a confidence in the ultimate truth of a theorem. All scientific processes have flaws. Petroski [1] 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. The Tacoma Narrows bridge that went into harmonic oscillation (Fig. 1) is a classic case. We hope that by our continuing (unsuccessful) attempts to model software, we are learning something.

Fig. 1. Tacoma Narrows bridge in oscillation.

This discussion is not to imply that formal methods have no place in software development; it is only to show that putting too much faith in that process can still pose great risks for the success of the project.

- Formal methods can be used to develop high-level design concepts. For example, the Bell-LaPadula security policy model can be used to model the top-level security policy of a system. However, as details are added, the complexity of the formal model becomes too complex to rely on this method as the sole means of verification.

- For highly critical programs that are small, verification can be used as an aid in correctness. The operative word here is "small" since formal proofs get quite complex quite easily.

- Formal methods deal best with discrete data. Discrete security labels, small integral values and Boolean data can often be handled by formal means. Use of formal models to verify programs employing real data (e.g. floating point values of distances) and programs using character data are harder to manage with formal models.

- It often helps if the underlying program design is based upon a formal model. For example, language compilers usually have finite state automata and context free languages as the basis of their parsing algorithms. This permits properties of those models to be used in the formal proof of correctness. If the underlying program design is not based upon a formal mathematical object, the proof is hard to develop.

- Formal methods can be an aid in program testing (see below). In sections which are easy to verify we have a higher degree of confidence in their trustworthiness than sections which are more difficult to verify. Conversely, sections which are hard to verify need greater testing in order to satisfy our need for trustworthiness.

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.

Related to simulation is the concept of "executable specifications." Using an appropriate higher level language, the interfaces to a system are specified, a simulation of that interface is demonstrated, and then the executable specification provides the framework for the eventual source program that will implement that functionality. Examples of such methods include requirements languages such as Paisley [22], so-called "Fourth Generation Languages" (4GLs) such as FORTH, and user interface generators such as Serpent and Chiron. Use of these systems increases our confidence level since demonstration of the "proof of concept" by executing the simulation provides for an automatic conversion of this concept into the outline of an executable program to implement that concept. Just as we rarely question the correctness of a compiler in generating appropriate machine language for a given programming language statement, use of such executable specification generators obviates the need for some of the next level of design.

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.)

Simulation is most effective when performance of the system is not well understood; simulation provides an early indicator of whether the system will meet performance requirements. Simulation is useful when the overall design allows use of one of the above-mentioned application generators. This permits initial simulation of the top-level design as well as minimizing the chances of additional errors as the top-level design is translated into some other implementation language.

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 other 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 are 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 [23], "testing researchers have defined theoretical ways of comparing testing strategies both in terms of cost and effectiveness (for example, [24]), formal criteria for evaluating testing strategies (for example, [25]), and axioms or properties that any adequacy criterion (rule to determine when testing can stop) should satisfy

(for example, [26])." Analytic results can also indicate when statistically significant measurement results have been obtained [27].

2.3.1 Functional and penetration testing

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 is usually a glass-box strategy using 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 are too familiar with the source program may overlook. As with many of the methods addressed in this paper, the cost-benefit of this added level of assurance must be analyzed [28].

2.3.2 Exhaustive testing

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 $2^{32} \times 2^{32} = 2^{64}$ or over 10^{19} tests. At a rate of even 10^8 tests per second, that would require 10^{11} seconds or over 3000 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 or can we assume the level of hardware abstraction that addition implies? 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 [29].

Testing is the oldest form of support for belief in correctness. But since it represents the most expensive phase of development, there is a strong desire to eliminate testing from the software development life cycle—but it is still with us. Testing is needed in the following circumstances:

- Always! For all but the most trivial programs, testing is needed for validation that the program does behave as expected on at least the well-chosen test data. Even with formally verified programs, testing is needed to demonstrate that the underlying assumptions used to abstract the program's specifications were valid. However, one should not rely only on testing for a thorough belief in correctness.
- End-to-end integration testing should always be performed on complex systems. That is, after each component of a system "checks out", it is

still necessary to integrate the total system and perform tests on its overall behavior. NASA's experience with the Hubble Space Telescope is a classic example of that. Since each component of the telescope checked out on the ground, it was only in orbit when it was discovered that the $2B machine did not work as designed and a $650M fix had to be developed. It could be argued that actually each component did not really "check out" and that the flaw in the lens was actually observed—but misinterpreted and ignored—with ground-based testing; however, that is the reality we have to deal with daily in developing software.

- Unit (e.g. component) testing may or may not be needed, depending upon the context of the development environment. For example, clean-room software development eliminates the need for some unit testing, but not for integration testing.

Finally, you cannot rely only on testing. Other methods must be used as additional aids in believing the correctness of complex systems.

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 frequently 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* [30] (Fig. 2) 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 later 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 [31].

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* (Fig. 3) has been gaining favor [32]. 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

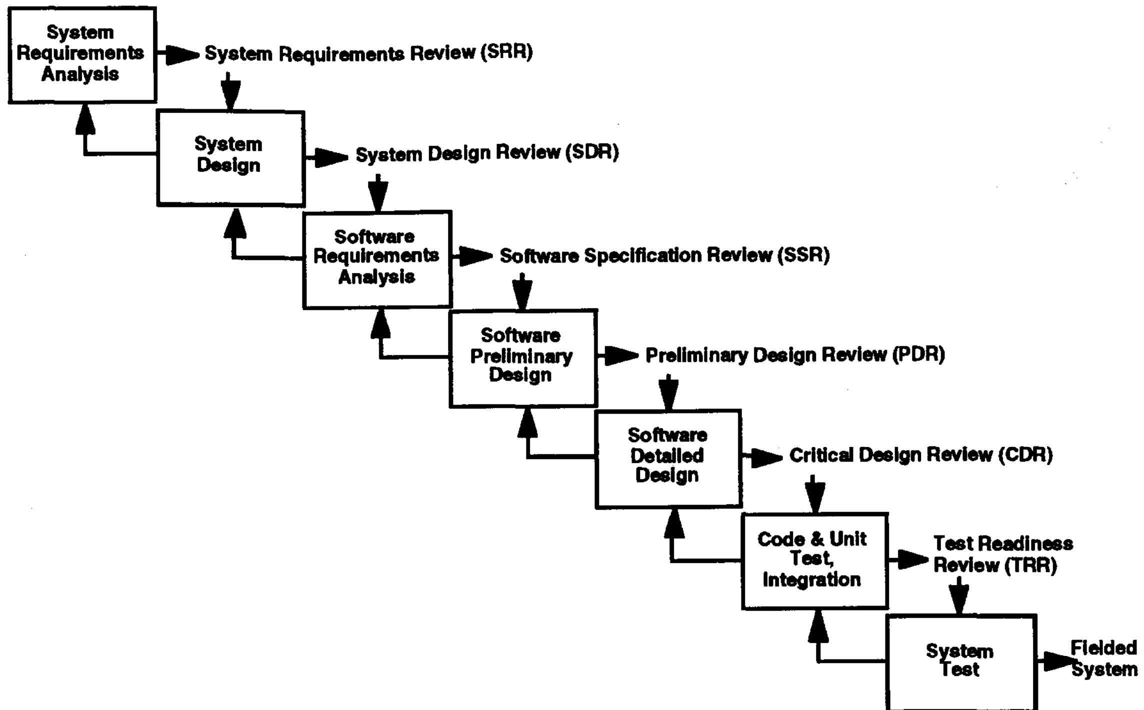

Fig. 2. The waterfall model of software development.

matures, but such activities as design, implementation, testing, evaluation, and planning are related. Changing requirements are more easily accommodated. 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.

Despite proper procedures, things will go wrong. Returning to civil engineering, we call your attention to the Citicorp Center built in New York City in 1994 [33]. The discovery and patching of a design/implementation flaw that had the potential to cause the building to collapse disastrously under high winds was surprising only in that it was so recent; there were some unique properties to this building. The question is not so much how to prevent problems but how to respond and recover. Prevention is only an optimization of this process and should not overwhelm the process.

Cleanroom software development represents an interesting variation on the traditional waterfall model [34]. The concept is to embed formal methods into a practical development strategy. All software is verified (at least informally) during the design and development stage, and *no* unit testing is performed. The formal proof represents all the validation that is done before the program undergoes integration testing. Surprisingly, by eliminating unit testing, quality goes up, not down.

The reasons for this have more to do with human nature than with the science of programming. Programmers as people often look for easy solutions. As such, understanding the logic of a diffi-

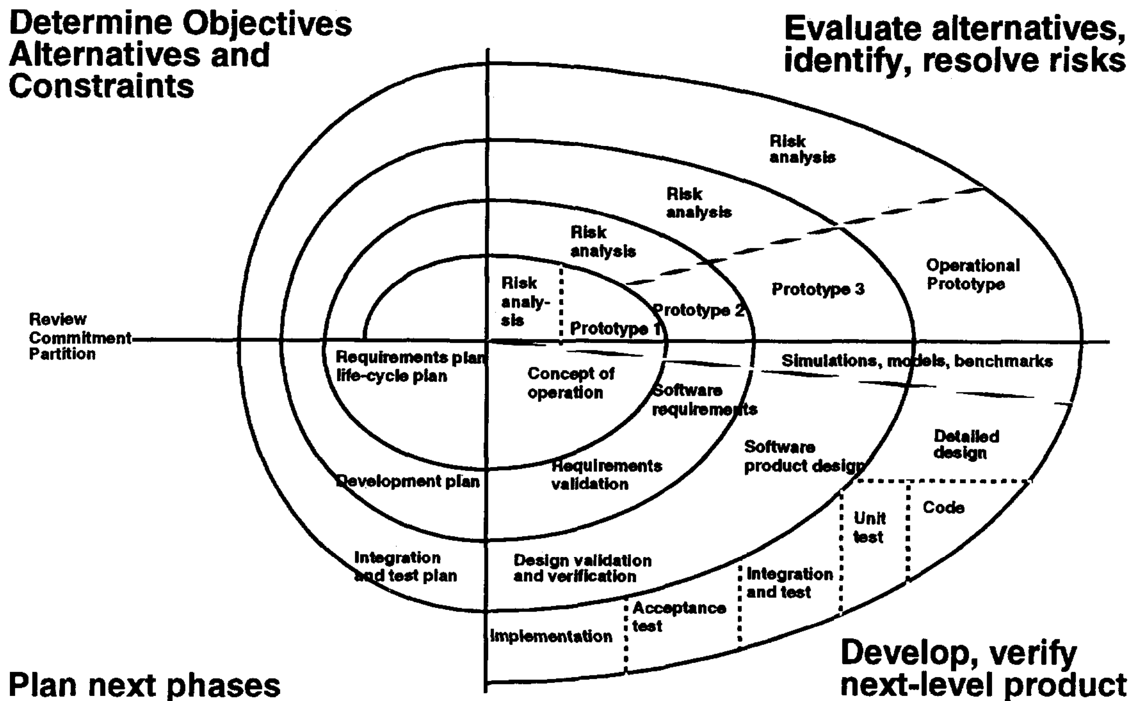

Fig. 3. The spiral model of software development.

cult algorithm is often delayed with the comment "I'll find the error in testing." With cleanroom, since there is no testing phase, the logic must be understood as it is written. While this slows down coding somewhat (which upsets unenlightened management), it has the effect of forcing the programmer to truly understand the program. This has the beneficial effect of greatly eliminating later costly integration testing errors.

Within the software engineering community today, the concept of *process improvement* is of great importance. The Software Engineering Institute has developed the capability maturity model (CMM) [12] as a means to assess an organization's process and as a means to suggest improvements to that process.

The CMM is based somewhat on hardware analogies; however, we have to be careful in applying hardware rules to software. For example, within the manufacturing domain, international standard IS 9000 governs the way products are manufactured in order to assess consistent manufacturing quality. For example, the typical home light bulb will burn within a few per cent of 1100 hours. The *quality* of this light bulb is outside of the domain of the standard. Whether 1100 hours is good quality or bad quality is not so specified. However, within software development there is the implied assumption that a good development process means a good resulting product. All that the CMM (and related process models) attempt to assure is that the process is repeatable. We need to look outside of the development process to get greater confidence in the developed products.

Process models do have a place. The problem today is that most organizations have no formal procedures for developing software, for develop-

ing requirements, for testing, for performing maintenance and enhancements on their products. As former President Eisenhower said after World War II "plans are worthless, but planning is everything" [35]. We need to encourage planning, and understanding process models at least forces organizations to undergo some initial planning. Going from CMM level 1 to CMM level 2 causes organizations to undergo the indispensable planning mentioned almost 50 years ago by Eisenhower and should prove to be effective. However, whether "walking up the CMM ladder" from level 1 through level 5 increases product quality at each level still needs to be determined.

3. Choosing among alternative beliefs

Software engineers promote one technique after another as the "silver bullet" [36] 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 [37, 38]). 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, although all have their value as part of a coordinated approach towards trustworthiness:

- 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 [39]. For the most part, this concept is now a "non-issue." Students are all taught structured programming and the goto statement is fast becoming an artifact of history. However, we need to do more to remove all programming errors.

- The spiral model described earlier 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 was perceived 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. We so far have little experimental data comparing both techniques, and in reality, both models have many elements in common.

- 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. One danger of CASE is that it has been oversold in industry. Since it has not proven to eliminate all errors, many managers avoid all CASE products—both good and bad. Current interest in environment design and the ability to integrate tools upon an infrastructure platform of environmental services holds promise for developing effective CASE tools in the near future [40].

- 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 [41] for a discussion of mathematical arguments that qualify as proof in a court of law.

- 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.

  Jones [42] identifies four risks in using an object-oriented design model: (1) while much of the literature asserts substantial productivity gains using OO techniques, most publications do not provide quantitative data to back up those claims; (2) productivity improvement results from reduced defect insertion and improved defect removal. It is claimed that OO inheritance and reuse lowers the defect insertion process, but there is no empirical evidence to back up that claim (although there is no counterevidence either); (3) OO proponents claim vastly increased levels of reuse, but there is little published data in this area; and (4) any technique has limitations, yet there are few published instances of projects where an OO design was abandoned and deemed inappropriate. OO technology may indeed prove to be beneficial; however, until we have a body of empirical evidence generated under recognized scientific principles, there is still a level of risk in madly adhering to an OO process without a good analysis of the problem first.

  Furthermore, encapsulation, the OO theory that is most attractive from a security viewpoint, has often not been implemented in practice; Gemstone is one exception. For example, [43] points out: "First, and most seriously, C++ encapsulates classes rather than objects; that is, a method on a C++ object *o* can access the private state of *any* object in *class* (*o*). Second, some models assume that the methods can write only in ways mediated by the OODBMS, but method code is usually allowed to invoke any system capability. Finally, for high-assurance systems, verifying that a language's encapsulation is enforced may require assuring a substantial part of the compiler."

- 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. Of course, the effectiveness determination needs to be made anew, but that is outside the scope of this paper. While reusing existing code is an admirable goal and is quite successful in limited application domains, we still do not have the technology to implement this process on a larger scale. 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 fulfils 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?

- Process maturity improvement is today's salvation [12]. 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. A CMM evaluation collects no data on the quality of the products that are being produced by the development organization. While a capability evaluation and changes suggested by 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 radially 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 Just build hardware

An alternative approach to the correctness of software problem has often been expressed by the sentiment "since hardware is easy to build and is correct, we should eliminate software and build only hardware." Although often said in jest, many deeply believe that this may indeed pose a solution to the trustworthiness issue. In reality, however, the distinction between software and hardware is moving in the opposite direction.

Today microprocessors are becoming increasingly complex with today's processors often containing over 4 million transistor-equivalents per chip. It is rare, today, to see a new microprocessor that is not first delayed in introduction or quickly modified due to initial errors in its fabrication. The hardware design process often includes many of the following steps [44]:

- Breadboarding (prototyping) a design and testing it, although that is becoming increasingly difficult as the number of circuits per chip increases.
- Designing a chip using abstraction and a divide-and-conquer strategy to define each functional unit on the chip.
- Using design automation tools (e.g. design languages) to describe circuit functionality.
- Simulating chip-level functionality.
- Verifying formal timing constraints and functional correctness of circuits [45].

This list looks surprisingly like the list of software correctness methods we have been describing in this paper. Hardware design is rapidly taking on the structure of software design, and with the increasing size and complexity of such circuits, hardware correctness is becoming as much a problem as its software counterpart.

3.3 Trustworthiness is just risk management

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 certi-

fication 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.

Missing from most of this discussion are the quantifiable results of using the various methods which are necessary in order to apply Lord Kelvin's definition of science given earlier. Measurement research in software development is relatively rare and there are only a few long-range studies of the development process. The NASA/Goddard Space Flight Center Software Engineering Laboratory has been studying software development activities since 1976 [46], but there are few other groups involved in such long-range evaluations of the software process. Many more such activities need to be funded and undertaken.

We do not imply that only software has such a poor track record in building reliable products. On 17 July 1981 the skywalk of the Hyatt Regency Hotel in Kansas City collapsed, killing over 100. The problem was ultimately traced to a problem that is typical in the software domain—poor translation from specifications to design. In this case, the design called for a single rod to thread through two layers of skywalk (Fig. 4(a)) whereas it was implemented as segmented rods (Fig. 4(b)), thus causing undue stress where the two segments met [1].

Problems in translation between specification and implementation is something software people are all too familiar with. The problem with software is just that failures occur so often when compared to mature technologies such as bridge building. The difficulties of achieving correct results are common to all engineering fields. But software engineering hasn't the long history and empirical database of civil engineering. Whenever new or novel solutions are attempted the probability of failure increases. We should not be surprised that since security engineering appears to be trying to solve new problems very frequently; the probability of achieving correctness is small.

3.4 The prudent manager

Without such quantifiable results, 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.

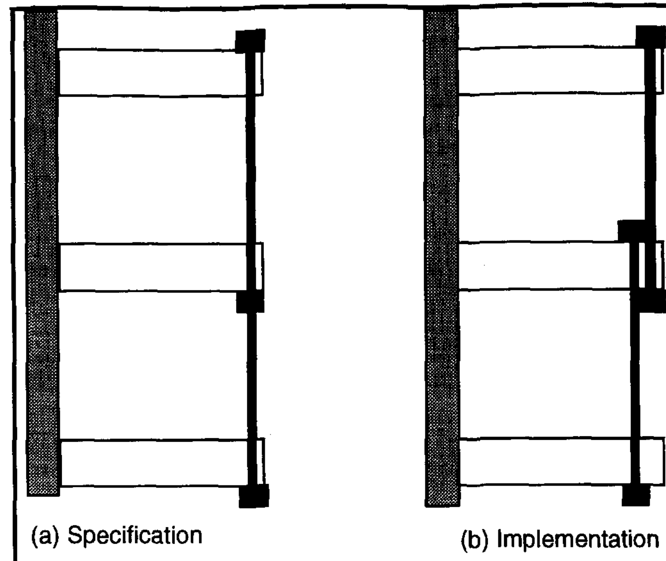

Fig. 4. Details of Hyatt skywalk: (a) as specified; (b) as implemented.

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 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 are disappointed that we cannot end this paper on a stronger note, but think that we have presented a realistic assessment of the situation as it stands today. There are many points of dispute; we would be delighted to learn that the situation is better than we thought. Table 2 summarizes the salient points we wish to leave with the reader. Reflecting the subjective nature of the value judgments presented in this paper, the symbols used in the table are vague and imprecise.

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 [23] 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:

TABLE 2. Characteristics of correctness methods

Method	When used	Skill required	Cost	Cost effectiveness	Applicable to complexity
Formal methods	All	▲	▲	▼	▼
Simulation	All	▲	▲	▼	▼
Testing	After	▼	■	■	▲
Process modeling	Before, during	▼	▼	■	▲
Structured programming	During	▼	▼	■	▲
CASE tools	During	▲	▼	▼	▲
Object-oriented methods	Before, during	■	■	▼	▲
Code reuse	During	▲	■	■	▲

Key: *design* tool, during *coding*, *after* completion, during *all* stages ▲ above average, rare ■ average ▼ below average, common.

- 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. This includes understanding of risk of failure and cost/benefit analysis of reducing this risk further:
  — 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 helps 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
  — understand risks involved, costs to decrease those risks, and accept residual risk.

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, Bill Herndon, 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.

References

[1] H. Petroski, *To Engineer is Human: The Role of Failure in Successful Design*, St. Martin's Press, 1985.

[2] V.E. Hampel and C.F. Bender, Covert corruption of integrated circuits and possible strategies for correction, in *Proceedings of the First Conference on Hostile Intelligence Threat to Software, Firmware and Algorithms Embedded in U.S. Army Weapon Systems*, Defense Technical Information Center, Alexandria, VA, 1988.

[3] K. Thompson, Reflections on trusting trust, *Communications of the ACM*, 27 (8) (Aug. 1984) 761–763.

[4] E. Amoroso, T. Nguyen, J. Weiss, J. Watson, P. Lapiska and T. Starr, Toward an approach to measuring software trust, in *Proceedings of the* 1991 *IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 1991, IEEE Computer Society Press, pp. 198–218.

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

[6] R.W. Butler and G.B. Finelli, The infeasibility of quantifying the reliability of life-critical real-time software, *IEEE Transactions on Software Engineering*, 19 (1) (Jan. 1993) 3–12.

[7] D.L. Parnas, A. John van Schouwen and Shu Po Kwan, Evaluation of safety-critical software, *Communications of the ACM*, 33 (6) (June 1990) 636–648.

[8] C.A.R. Hoare, An axiomatic basis for computer programming, *Communications of the ACM*, 12 (10) (Oct. 1969) 576–583.

[9] D.L. Parnas, Software aspects of strategic defense systems, *Communications of the ACM*, 28 (12) (Dec. 1985) 1326–1335.

[10] R. Hamming, *Numerical Methods for Scientists and Engineers*, McGraw Hill, 1962.

[11] W.T. Kelvin, *Popular Lectures and Addresses*, 1881–1884.

[12] M.C. Paulk, B. Curtis, M.B. Chrissis and C.V. Weber, Capability maturity model for software, Version 1.1, *IEEE Software*, 10 (4) (July 1993) 18–27.

[13] V.R. Basili, G. Caldiera and G. Cantone, A reference architecture for the component factory, *ACM Transactions on Software Engineering and Methodology*, 1 (1), 53–80.

[14] C. Landwehr, Formal models for computer security, *ACM Computing Surveys*, 13 (3) (Sept. 1981) 247–278.

[15] J. Rushby, *Formal Methods and the Certification of Critical Systems*, Technical Report C3L-93-7, Stanford Research Institute, Dec. 1993.

[16] J.H. Fetzer, Program verification: the very idea, *Communications of the ACM*, 31 (9) (Sept. 1988) 1048–1063.

[17] D. Harel, Will I be Pretty, will I be rich? *ACM Symposium on Principles of Database Systems* (May 1994) 1–3.

[18] C. Youngblut, B.R. Brykczynski, J. Salasin, K.D. Gordon and R.N. Meeson, *SDS Software Testing and Evaluation: View of the State-of-the-Art in Software Testing and Evaluation with Recommended R&D Tasks*, Institute for Defense Analysis Report IDA-P 2132 (Feb. 1989).

[19] D.E. Bell and L.J. LaPadula, *Secure Computer Systems: Unified Exposition and MULTICS Interpretation*, MTR 2997, The MITRE Corporation, Bedford, MA, 1974. Available from National Technical Information Service, AD/A 020 445.

[20] K. Brunnstein, University of Hamburg, private communication, 1994.

[21] R. DeMillo, R. Lipton and A. Perlis, Social processes and proofs of theorems and programs, *Communications of the ACM*, 22 (5) (May 1990) 271–280.

[22] P. Zave, An operational approach to requirements specification for embedded systems, *IEEE Transactions on Software Engineering*, 8 (3) (1982) 250–269.

[23] N.G. Leveson, High-pressure steam engines and computer software, *Computer*, (Oct. 1994) 65–73.

[24] E.J. Weyuker, S. Weiss and D. Hamlet, Comparison of program testing strategies, *Proceedings of the Fourth Symposium on Software Testing, Analysis and Verification* (TAV4), Victoria, B.C., Canada, Oct. 1991, pp. 1–10.

[25] J.B. Goodenough and S. Gerhart, Toward a theory of test data selection, *IEEE Transactions on Software Engineering*, SE-1 (2) (June 1975).

[26] E.J. Weyuker, Axiomatizing software test data adequacy, *IEEE Transactions on Software Engineering*, SE-12 (12) (Dec. 1986) 1128–1138.

[27] S.A. Mamrak and M.D. Abrams, A taxonomy for valid test workload generation, *Computer* (Dec. 1979) 60–65.

[28] G. Page, F.E. McGarry and D.N. Card, *Evaluation of an Independent Verification and Validation Methodology for Flight Dynamics*, NASA/GSFC Technical Report SEL 81-110, 1985.

[29] T. Taylor, FTLS-based security testing for LOCK, in *Proceedings of the 12th National Computer Security Conference*, Oct. 1989, pp. 136–145.

[30] W.W. Royce, Managing the development of large software systems: concepts and techniques, in *Proceedings IEEE Wescon*, Los Angeles, CA, 25–28 August 1970, pp. 1–9.

[31] R. Stillman, Software Development: Neither Economics Nor Engineering, Keynote Address, Third Annual Software Engineering Economics Conference, The MITRE Corporation, 22 March 1993.

[32] B. Boehm, A spiral model of software development and enhancement, *IEEE Computer*, 21 (5) (May 1988) 61–72.

[33] J. Morgenstern, The fifty-nine-story crisis, *The New Yorker*, LXXI (14) (29 May 1995) 45–53.

[34] H.D. Mills, M. Dyer and R.C. Linger, Cleanroom software engineering, *IEEE Software*, 4 (5) (1987) 19–25.

[35] D.D. Eisenhower, National Defense Executive Reserve Conference, 14 November 1967.

[36] F. Brooks, No silver bullet: essence and accidents of software engineering, *IEEE Computer*, 20 (4) (1987) 10–19.

[37] C. Chang, Is existing software engineering obsolete? *IEEE Software*, 10 (5) (Sept. 1993) 4–5.

[38] A. Davis, Software lemmingineering, *IEEE Software*, 10 (Sept. 1993) 79–84.

[39] M.V. Zelkowitz, A functional model of program verification, *IEEE Computer*, 23 (11) (Nov. 1990) 30–39.

[40] M.V. Zelkowitz, Use of an environment classification model, in *Proceedings of the ACM/IEEE 15th International Conf. on Soft. Eng.*, Baltimore, MD, May 1993, pp. 348–357.

[41] D. MacKenzie, Computers, formal proofs, and the law courts, *Notices of the American Mathematical Society*, 39 (9) (Nov. 1992) 1066–1069.

[42] C. Jones, Gaps in the object-oriented paradigm, *IEEE Computer*, 27 (6) (1994) 90–91.

[43] A. Rosenthal, W. Herndon, J. Williams and B. Thuraisingham, A fine-grained access control model for object-oriented DBMSs, in *Proceedings of the 8th IFIP Working Conference on Database Security*, Hildesheim, Germany, Aug. 1994.

[44] W.M. van Cleemput and H. Ofek, Design automation for digital systems, *IEEE Computer*, 17 (10) (Oct. 1984) 114–122.

[45] R.E. Bryant, Symbolic Boolean manipulation with ordered binary-decision diagrams, *ACM Computing Surveys*, 24 (3) (Sept. 1992) 293–318.

[46] V. Basili, M. Zelkowitz, F. McGarry, J. Page, S. Waligora and R. Pajerski, SEL's software process-improvement program, *IEEE Software*, 12 (6) (1995) 83–87.