A report on reliable software and communication

Abstract

We discuss the general state of affairs in a variety of related areas ranging from software safety, reliability and testing, protocol specification and verification, network congestion-control and reliability, to communication security and complexity. We intend to identify useful theory and tools, point out connections between different areas, and to a large extent, raise a number of questions whose answers may still lie far beyond the limits of our current knowledge. Some of these areas are still in a very primitive state and call for new ideas, bold approaches, radical thinking and perhaps extraordinary efforts.

Contents

1 Executive summary

1.1 Introduction

Nowadays, computer-controlled systems affect almost all aspects of our lives. These systems are often large, complex and powerful and their failure can have diverse effects ranging from personal inconvenience to catastrophic disaster. Yet, it is a widely known fact that most software systems very likely contain undetected faults with unknown (and perhaps unknowable) consequences. Unlike hardware faults where one circuit can often be backed up by another spare, software faults sometimes have a cascade effect of propagating exponentially far beyond their point of origin resulting in large-scale disruption. Furthermore, because the potent computing power available today is capable of performing massive calculations at dizzying speed, even faults with very low probability or only due to rare combinations of events can easily find ways to reveal themselves. So, one possible analogy is to view software systems as time bombs for which the scheduled time of detonation is unknown.

Facing such an unsatisfactory state of affairs, a natural question is: "Why have we not been able to root out possible faults in large systems?" Indeed, some people including many leading researchers and engineers worry a great deal about such problems (often in their nightmares), but most people only pay attention to these issues after an accident occurs. This is perhaps for the following reasons:

1. It is human nature not to want to face the possibility that some bad scenario might happen.
2. Typically, designing a program is exciting, whereas debugging is boring and frustrating. Consequently, not as much research is done in these areas.
3. Designing a system is regarded as the major endeavor while testing/checking can sometimes be compromised (when facing deadlines for example).
4. It is just too difficult to have really reliable software!

The first three items are easier to deal with than the fourth, which deserves further clarification. *So why is it hard to have reliable software?*

- The enormous size.
  Because of the dazzling progress in computer hardware technology, the raw computing power currently available is almost unimaginable (and increasing), and problems arising in manipulating computation often involve an astronomical number of cases. The sheer volume of a large software system often exceeds millions of lines of code.
- Complexity.
  Because of the discrete nature of software, small changes do not always have small effects. Furthermore, some problems can be inherently undecidable or computationally intractable,

requiring a possible exponential blow-up in running time or memory. In some cases, no amount of testing could prove that a system is correct.

- Interactive and distributed systems.
  Computer programs and large collections of interconnected communication systems usually involve interactive and concurrent processes. Therefore, the potential errors can be very subtle and hard to locate.

- Changeability.
  Software can be changed relatively easily (though not necessarily correctly) and is subject to constant pressure for modifications and additional features. We often see many variations of one software application. For large collections of interconnected systems, constant changeability has to be dealt with.

With such imposing difficulties lying ahead, what can we do now? What we should *not* do is just to give up and turn around to build yet another large system with many potential faults, i.e., "business as usual." Instead, it should be recognized that because of the great difficulties and extreme importance, reliable software offers the *grand challenge* which deserves the attention of everyone involved.

It is the purpose of this report to provide a very rough "road map" to anyone who is willing to take on this grand challenge. The road map consists of two major parts: The first part concerns "*where we now stand*" — the second part involves "*where we are heading*."

Where do we stand? In Section 2, we give a brief overview of a wide range of areas, including software safety, reliability and testing, protocol specification and verification, network congestion-control and reliability, communications security and complexity. We intend to identify useful theory and tools as well as to point out connections between different areas. Detailed surveys of selected areas are included in Part II, Sections 6-10. The selection of topics and the coverage of the surveys merely reflect the personal viewpoints of individual researchers and are not intended to be exhaustive. Some non-technical but vital issues on cost, standards, and regulatory and legal aspects are mentioned in Section 3. In addition, we will consider some reliability issues related to the System Seven (SS7) protocol of the Common Channel Signaling (CCS) system, since its outage in June 1991 affected millions of users in a number of cities and provides a natural motivation for examining related issues in reliable software and communication. Although system failures can be caused by human negligence or oversight, we make no attempt here to discuss such human-factor issues.

Where are we heading? There is indeed an immense gap between where we now are and where we want to be. In an attempt to bridge the gap, we will focus on several crucial areas by posing a number of challenging problems in Section 4. A few of the problems are perhaps easier than others since some approaches already exist although improvements are needed. On the other hand, the answers to many of the other questions might be far beyond our reach at this point. The list of problems is by no means complete or even objective, but we hope it may

provide suggestions for the reader for taking a first step in addressing the grand challenge of reliable software and communication.

1.2 Where do we stand?–An overview of current technology

Errare humanum est.
(To err is human.)

Anything made by man can have errors. So what do we mean by "reliable" software and communication? Is it possible to have any real confidence in complex and perhaps critical systems? On the surface, we see a large grey area instead of black and white ones. However, let us examine the facts and pursue the issues in a more careful manner. Basically, when we deal with a project, there are two major aspects: modeling and implementation. (Here, modeling includes planning, designing, and specification.) Although the implementation might have faults due to physical or practical constraints, the model itself should be subject to rigorous reasoning and is supposed to be consistent and error-free. In spite of the extreme complexity of software systems or the skepticism about correctness of mathematical proofs, substantial progress has been made during the past decade. For example, the heuristics in protocol validation have been greatly improved and the feasible sizes of applications have far exceeded what many people expected only a few years ago. Although large scale applications are not yet in sight, small protocols *can be* verified by using current technology. In addition, research *can be* done to improve the performance of various heuristics involved. So, making large systems reliable is not totally without any glimmer of hope.

Here we will briefly overview the general state of affairs in a variety of topics related to reliable software and communication; detailed surveys in a number of areas are presented in Part II Sections 6-10. We will first discuss reliability issues in large software systems where neither rigorous models nor systematic testing methods are yet available and only statistical process control or software management are currently in use. We then proceed to recent developments in the area of protocol specification and verification, where both validation and testing have had some limited success. We will also mention fundamental combinatorial methods and problems that can be used to either improve the heuristics or to analyze the complexity of algorithms. Some of these methods have been applied to areas of communication security and program checking when the function of a program can be clearly defined. We will mention reliability issues related to CCS/SS7 throughout the discussion of various areas with emphasis on congestion control and network reliability.

A. Software

We often hear, "Programming is more art than science." In spite of the tremendous power and excruciating precision of computer technology, its developments still mostly depend upon experience and intuition. The creativity of the relatively few "super-hackers" is heavily valued, but large systems must rely on the coordination of large teams. It is not easy to discuss reliability

without involving the heart of software design, implementation and maintenance. Still, we will try to focus upon the following topics.

- Software safety
  In spite of its importance, there are relatively few researchers in the area of computer safety, which is concerned with reducing the risk of using software. Leveson[1] publicized the importance of computer safety and initiated the techniques of fault-tree analysis. Basically, a hazard or risk would first be identified, assessed as to criticality/likelihood and then back-traced by exhaustively searching all possible causes of errors. In order to avoid too many possibilities from branching, some "pruning" of cases has to be performed, mostly by human judgment. The final phase of the analysis consists of suggestions indicating which hazards or risks are to be eliminated or controlled, if feasible, by designing lockout, warning, isolation or containment devices. Although these approaches are still in very primitive state, they have been used by industry and the Federal Aviation Administration (FAA) for aerospace software projects among others (see Horgan's survey in Section 6).

  Some researchers have proposed introducing redundancy in software by having several independently designed modules for the same task, using techniques similar to those in hardware. However, the cost-tradeoffs and the effectiveness of this approach for software systems was not resolved.

  Quis custodiet ipsos custodes?[2]
  (Who is guarding the guards?)

  Based on the experience of several major incidents, it seems that the safeguards implemented in a complex system for preventing faults are, in fact, quite likely to be the causes for large scale failures. For example, the recent telephone disruption in several cities resulted from incorrect implementation of the safeguard features of the CCS/SS7 protocol. Perhaps special attention should be paid to performing risk analysis of safeguard installation.

- Software testing
  There are at least two schools of thought in testing. One is to treat a software program as a "black box." Combinations of inputs are selected and checked to see that they give the proper outputs with the goal of meeting the requirements in the design or specification documents. The other school is the so-called the "glass-box" method, aiming at testing all paths through the code. The glass-box approach is usually incorporated in the design of the program. After each module is coded, a test program is also prepared and updated, serving as tools throughout the design process. Although there are still limitations such as difficulties in automating tests of interactive parts of the software, the glass-box method can be very effective and useful (see Section 6).

[1] Nancy G. Leveson, Software safety in embedded computer systems, *Comm. ACM*, 34, No. 2, (1991) 35-46.
[2] Juvenal (Decimus Junius Juvenalis) born AD 55-60?

- Software debugging
  There has been slow but steady progress in debugging. Some tools now available can offer users a structured view of dynamic events and let programmers interactively identify subsets of interest. For example, some "slicing" tools allow users to backtrack from checkpoints without reexecuting the program to reach recent prior states. Current debugging tools are mostly visibility tools rather than intelligent ones in the sense that the presentation and organization of data have substantially improved but the selection and elimination of massive cases still depend on experience and judgement.

- Software measures
  Many different software measures, commonly referred to as metrics, have been proposed. Some of the measures are statistical and a good number of them are graph-theoretical since a program can be viewed as a directed graph (see Section 6). The relevance of such measures to the characteristics of software development have, for the most part, not yet been firmly established. Some research papers argue that the most significant measure for predicting field faults is still the number of non-commented source lines[3]. Although software reliability measurement technology is currently unable to provide adequate confidence that high reliability and safety have been achieved, they can demonstrate that it has not been achieved and thus provide useful information during software development and maintenance[1].

  The Software Engineering Institute (SEI) further considers measures for scrutinizing and controlling the process of developing software instead of software itself. In their "software maturity" model, there are five levels of maturity ranging from chaotic and no management procedure (Level 1) to well-organized and self-improving (Level 5) defined as, for example, forming a software process group, developing measurements and analyzing defect-cause, etc. In spite of the controversy about the wisdom and effectiveness of their models, the SEI programs have been used by several companies including Hughes Aircraft[4] (see Section 7).

- Statistical process control
  The traditional software reliability model attempts to estimate the probability of failure in software by monitoring the process and applying data analysis techniques. There are a large number of research papers on software reliability models and some recent work tries to provide indicators for the amount of testing needed (see Section 7). At present statistical techniques are the most common methods for estimating software quality. However software reliability depends heavily on the structure and complexity of the software as well as the nature and quality of testing the software has endured. These aspects are largely ignored by the statistical reliability models which therefore limits their potential usefulness. In addition, there are always questions as to the validity of the underlying

[3] A.R. Feuer and E.B. Fowlkes, Relating computer programs maintainability software measures, *AFIFS National Conference Expo, Conference Proceedings*, 48,(1979) 1003-1012.

[4] Watts S. Humphrey, Terry R. Snyder and Ronald R. Willis, Software process improvement at Hughes Aircraft, *IEEE Software* July 1991 11-23

probabilistic assumptions. Various models, techniques and practices are summarized in Section 7.

- Visualization
  Visualization techniques can be very helpful in monitoring and analyzing data or in statistical process control and network performance analysis. There seem to be ample opportunities for interaction among a number of areas including statistics, cognitive science, learning, computational geometry and, in particular, interactive and animation techniques (see Section 7).

- Disciplined and controlled environments
  Some approaches to producing quality software emphasize controlled environments for developing software. Typical techniques include requirements for design, code walk-through, specifications of design techniques or coding style, and enforced sign-off criteria for life cycle phases.

  One of the examples is the "Cleanroom" model, which dogmatically separates design, implementation, and analysis from execution and testing. This partitioning of activities forces careful analysis in order to prevent defects rather than debugging. Another example is the "spiral" model, which takes an entirely different approach. In this model, the code-test-debug phases form a loop with constant feedback (for details see Section 6).

- Formal methods
  There are basically two approaches in rigorous verification. One approach, called formal methods, relies on logic and views programs as symbolic manipulators. The other is based on model-checking which will be described later in the overview of protocol validation. Formal methods have a long history, based on the work of Floyd, Hoare, Boyer, Moore, Dijkstra, Milner and others. Currently, many research groups in Europe actively advocate using formal methods for specification while their counterparts in the U.S. tend to work with tools and to focus on proofs for properties of specifications. There are a number of case-studies of real applications of formal methods using mainly specification. In general, there are difficulties in scaling up and also in asserting correctness of the proofs (see Section 6).

B. Protocols

A protocol provides the rules of interaction between different parts of a large system or across systems. Each rule may be very simple (such as: "if the light is green and no pedestrian is present, you can make a right turn"), but a combination of many rules can become very complicated since, after all, the process is concurrent and non-terminating. Nowadays, telecommunication systems are governed by many layers of communication protocols. Here we will describe some current developments in the following areas.

- Specification languages
  Three protocol specification languages have been developed by the International Con-

sultative Committee for Telephone and Telegraph (CCITT) and International Standards Organization (ISO):

1. The Specification and Description Language (SDL) was developed by CCITT in 1976, revised in 1982, 1985, and finalized in 1987.
2. The Language of Temporal Ordering Specification (LOTOS) was developed by ISO in 1989.
3. Estelle was developed by ISO in 1989.

A rough comparison of the three Formal Description Techniques can be described as follows: Estelle's strengths and weaknesses stem from its PASCAL origins and its closeness to implementation. LOTOS's strengths and weaknesses stem from its algebraic formality. SDL's strengths and weaknesses stem from its informal nature. Currently, the most commonly used of the three is SDL. A substantial amount of the Bellcore SS7 specification is in SDL in graphical form (SDL includes two equivalent syntactic forms: the graphical form and the language form). There are many tools available for converting the graphical form to the language form (and vice versa) and for automatic translation into C, CHILL, ADA, CORAL, PASCAL and PLM[5] [6] [7] [8]. There are many other specification languages which have been developed by different groups, each having its possible advantages and disadvantages (see Section 8).

- Protocol validation
  The word "verification" usually means getting an answer—either "yes, the protocol is correct (under certain circumstances)" or "no, it is not". To some people, this provides relatively little information and gives no indication of its trustworthiness. A more *practical* way for verification is called model checking: For a given protocol, build a "model-checker" so that queries can be posted and answers can be generated automatically. Therefore, a large set of answers can either provide confidence in the protocol or be used to construct counterexamples. The quality of the validation depends heavily on the choice of queries and the limitations of the language (see Section 8). Although the model checker is based on sound theoretical foundations, the algorithms involved are often heuristics which are efficient but sometimes not exhaustive. Model-checking is perhaps the most automated approach for protocol verification. We will describe several model-checking tools.

[5] G.V. Bochmann, Usage of protocol development tools, the results of a survey, *7th IFIP Symposium on Protocol Specification, Testing and Verification*, Zurich, May (1987) 139-161.

[6] R. Sarocco and P.A.J. Tilanus, CCITT SDL: Overview of the Language and its Applications, *Computer Networks and ISDN Systems* 13 (1987) 65-74.

[7] R. Tinker, A. Hopcropt, R. Lewis, H. Ichikawa, M. Itoh and M. Shibasaki, A system design environment, *Proceeding of the IEEE, Sixth International Conference on Software Engineering for Telecommunication Switching System*, The Netherlands, April 1986, 216-220.

[8] Yoshikiro Nitsu and Osamau Mizuno, Interactive specification environment for communication service software, *J. on Selected Areas in Communications*, Vol. 8, No. 2 (1990) 181-188.

– Holzmann's PROMELA
The model-checker, developed by Holzmann at AT&T, is perhaps the most simple and neatly implemented of the three models we discuss here. The available memory is utilized in a way that each state is represented by one bit. For small protocols with fewer states than the memory (say about 10^{10} states), an exhaustive search can be conducted. Otherwise, partial searches[9] [10] can be used for somewhat larger state spaces. The major drawback of PROMELA seems to be its inflexibility to deal with state space explosion (see Section 8).

– Kurshan's COSPAN
There are two major factors behind the power of COSPAN – the refinement technique and the use of the Binary Decision Diagram (BDD). The refinement technique is based on algebraic methods and can decompose a problem into small modules or simpler abstract levels. However, the implementation of this technique requires human adjustments. Binary Decision Diagram[11] uses succinct (on the average) representations of Boolean functions as labelled directed graphs. BDD was the key factor in the rapid improvement of model checkers during the past two years and has caused a dramatic increase in the size of state spaces which can be dealt with in various applications (see Section 8).

– Clarke's SMV
This model checker, called Symbolic Model Checker and developed by a group at Carnegie Mellon, has its theoretical foundations in temporal logic. The reduction techniques are similar but somewhat different from COSPAN. BDD has been incorporated into this model and there are reports of successful analysis with state spaces in the range of 10^{1300} states and beyond.

The above three model checkers all have their own languages, although it is possible to have a front-end interface with languages such as SDL. There are many other model checkers and validation approaches such as N. Lynch's I/O automata, D. Harel's statecharts, L. Lamport's temporal logic of action, the ESPRIT project at University of London on executable temporal logic, C. A. R. Hoare's recent work on timed transition system methods, some of which are described in Section 8.

- Conformance testing
The goal of conformance testing is to check whether a given protocol implementation meets all requirements in its specification. By using an external tester, the conformance test is done by applying a sequence of inputs and verifying that the corresponding output sequence is as expected. Many reasons contribute to the difficulties of conformance testing

[9] Gerald J. Holzmann, *Design and Validation of Computer protocols*, Prentice Hall, Engelwood Cliffs, New Jersey, 1991.

[10] C. H. West, Protocol validation by random state exploration, *Protocol specification, testing and verification VI*, B. Sarikaya and G.V. Bochmann eds., North-Holland, Amsterdam Pub (1986) 233-242.

[11] Randale E. Bryant, Graph based algorithms for Boolean function manipulation, *IEEE Trans. on Computer*, Vol. C-35, No. 8 (August 1986) 677-691.

such as limited controllability (the tester cannot place the protocol implementation into a predetermined state), limited observability (the tester cannot directly observe the state of the protocol implementation), time-dependent errors, and the unbounded number of choices for input sequences.

Although in practice conformance test generation is mostly ad hoc, there has been steady progress in the systematic generation of test sequences. Some techniques have their origins in the 60's in switching circuit testing and automata theory, and some recent work uses methods in algorithmic design and computational complexity. For example, a protocol can be modeled by a finite state machine, represented by a directed labelled graph in which the nodes are the states, and the edges are labelled by the input/output messages determined by the protocol. The problem of generating a conformance test sequence for checking every state transition is then equivalent to the so-called "Chinese postman problem" (or some of its many variations) of traversing every edge at least once. Test sequences are used for various problems such as confirming a state, determining a preset state and fault detection. For each of these problems, questions can be asked such as whether a test sequence exists, if it exists, how short the test sequence can be, and how hard it is to construct such test sequences. Recent results and surveys on conformance testing can be found in the references[12] [13] [14] [15].

Conformance testing standards have been developed by ISO for Open System Interconnection, known as ISO 9646, which is in the process of being adopted by CCITT without significant changes. The standards include the test language, called Tree and Tabular Combined Notation (TTCN), test realization and requirements for the conformance assessment process. The conformance testing work for SS7 provides test specifications for the message transfer part and the telephone user part; extension to other parts of SS7 and compatibility testing for checking the interworking of two implementations are still under study.

C. Network congestion control and reliability

Congestion control is perhaps the most important but the least understood issue in real-time communication networks. When congestion occurs, the tasks of the network management system are to minimize the congested area, to shorten the duration of congestion and to avoid escalation of disruptions (see Section 9). The principles for handling these tasks are not so different from what an old-fashioned telephone operator did. Namely, traffic is restricted from entering the congested area, calls to and from unaffected areas are rerouted around the congested area and

[12] Alfred V. Aho, Anton T. Dahbura, David Lee and M. Ümit Uyar, An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours, *Protocol Specification, Testing, and Verification VIII*, S Aggarwal and K. Sabnani eds., Elsevier (North-Holland) IFIP, (1988) 75-86

[13] Anton T. Dahbura, Krishan K. Sabnani, and M. Ümit Uyar, Algorithmic generation of protocol conformance tests, *AT&T Technical Journal* (1990) 101-118

[14] Richard J. Linn, Jr, Conformance evaluation methodology and protocol testing, *IEEE J. on Selected Areas in Communications*, Vol. 7, No. 7, (1989) 1143-1158.

[15] Mihalis Yannakakis and David Lee, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing* (1991), 476-485.

information is provided to the callers to reduce reattempts. Nowadays, these functions are carried out by various provisions in the design of a system at a much greater speed (in the range of milliseconds for SS7 systems). Since congestion can be caused by unpredictable events, there are numerous potential traps in the process of congestion control, some of which we mention here.

- Can network management messages flood the system?
  In order to control congestion, management messages must be exchanged among congested nodes and uncongested ones. Not only do such messages contribute to additional traffic, but some of them might not get through to do their jobs. In addition, they might be delayed or discarded and thus generate further messages. Therefore, this could cause possible message looping and an explosion of management messages.

- Can the increase of the traffic load and the reduction of the rate of call completion be locked into a vicious cycle?
  When the traffic load is heavy, the ratio of calls which get connected on first attempts decreases. This causes an increase in call reattempts which places further demands on the real-time processing power. Coupled with the increase in management messages, it is possible to enter a nasty cycle of load increase and rate of call completion decrease.

- Can the failure of a single node bring down the whole system?
  For example, can the congestion of one STP(Signaling Transfer Point) spread to the whole SS7 system in a LATA network? Can the congestion within a LATA cause a national blockage?

- Can congestion be properly monitored and visible management information be provided?
  The high speed of the automated systems could make monitoring difficult during traffic congestion. It would be very useful to have visible management information for determining the sources of a possible signaling fault and for speedy recovery of the system.

- Could congestion control procedure be unstable with an oscillating behavior?
  There is the possibility that under certain circumstances the congestion level might oscillate and generate its own burstiness[16].

- How can large queue sizes or buffer capacity help the performance of the network?
  Can additional redundancy or connectivity increase the reliability of the system? For certain malfunctions in a network, additional redundancy or connectivity can provide alternate routes and increase the survivability of the network [17]. However, for some faults

[16] A. Erramilli and L. Forys, Oscillations and chaos in a flow model of a switching system, *IEEE J. on Selected Areas in Communications*, Vol. 9, No. 2, (1991) 171-178.

[17] Ali Zolfaghari, Network simulation study as a base for survivability standards, *Teletraffic and Datatraffic in a period of Change*, ITC-13 A. Jensen and V. B. Iversen (Editors) Elsevier Science Publishers B. V. (North-Holland), (1991), 377-382.

which have a cascade effect, increasing buffer sizes at queues to some very large value does not seem to help since network resources can be exhausted very quickly[18] [19].

- How should the congestion control in different modules be coordinated?
  A large network usually consists of a number of modules, each of which may operate independently. It is possible that these modules do not communicate their congestion status until they have been removed from service. This can lead to lower completion rates than expected.

- How fast can recovery be made?
  This is perhaps the most important question because there is no guarantee that the preventative measures can always succeed and the problems mentioned above can be satisfactorily resolved. A wide range of issues are involved here including anticipating various scenarios during breakdowns, designing systems and tools for tracing the sources of errors, developing strategies to contain and isolate the effect of failures, and, in particular, designing robust system that can recover effectively.

D. Theoretical foundations and their applications

Because of the enormous size and complexity of software, any significant progress perhaps will not result from incremental improvements but depend on "quantum leaps" instead. There are two possible ways which might lead to such major advancements — ingenious new ideas or connections with known powerful methods and tools. Possible sources of such methods could lie in many different arenas and, in particular, in some areas of mathematics where a rich body of knowledge has been established over hundreds of years (mostly in the traditional continuous domain) but connections with discrete structures are relatively less developed and remain to be explored. Here we will briefly describe some theoretical topics with known or potential applications.

- Protocol analysis and synthesis
  Since the late 70's, a rather significant and elegant theory has been developed to address the problem of protocol design and validation in a uniform way. When a protocol specification is written in some suitably chosen formal logic, there is a procedure that decides if the specification is satisfied. Furthermore, if the specification is satisfiable, the decision procedure is constructive and can produce programs that are provably correct. The main drawback of this approach is that these algorithms require exponential running times in the worst case. Hence these methods, in their full generality, are not applicable. However, this theory touches the heart of protocol design and validation in a fundamental way and there is great potential to reveal some of the key principles involved. Such principles may suggest heuristical approaches and, in fact, special features of these schemes have been used in the model-checkers of Holzmann and Clark.

[18] H. J. Fowler and W. E. Leland, Local area network traffic characteristics with implications for broadband network congestion management, *IEEE Journal on Selected Areas in Communications* **9,** 1139-1149, Sept. 1991.
[19] P. T. Brady, private communication 1991.

- Cryptography
  There are many connections between cryptography and secure/reliable communication. Possible faults in a communication system can be viewed as adversaries in a cryptographic system. In particular, cryptography can be useful for many essential aspects involving privacy, authorization, and access control in systems with shared memory. Security can often be closely coupled with reliability in the following sense: The software is partitioned into pieces which are implemented in separated environments so that a failure in supplementary pieces does not impact the main software line. There has been a great deal of progress in the areas of cryptography and interactive communication. The study of "zero-knowledge" protocols has not only led to many interesting developments in complexity theory, but also to applications in authentication schemes (see Section 10).

- Self-stabilizing systems
  One of the major problems in distributed computing is to design self-stabilizing systems which can be initialized in an arbitrary state and effectively reach a legitimate state. Such systems can tolerate any transient errors and recover from faults quickly. There has been active research in finding self-stabilizing systems. Some recent work[20] [21] concerns a self-stabilizing protocol which only requires local checking and correction without the use of unbounded counters. Such a protocol can be superimposed on any other distributed protocol and provides a reset service that may be invoked at any node.

- Program checking
  Program checking as introduced in the reference[22] is a way to certify the correctness of program output and applies mainly to computational problems where correctness could be described as a relation between input and output. The correctness of output is ensured with high probability on each run rather than on proving the program correct overall. Some of the recent work[23] deals with problems of checking storage and memory retrieval and on quickly checking the correctness of any very long computation (see Section 10).

- Fault tolerant routing and design
  Motivated by various applications in switching networks, there has been a long history of research in permutation networks and non-blocking networks. One of the basic building blocks in such networks is the so-called "expander graph" which guarantees sufficient access for arranging disjoint routes for requested pairs of nodes. During the past five years, fundmental breakthroughs have been made on the explicit constructions of expander graphs by establishing connections between eigenvalues of graphs and other areas of mathematics[24].

[20] Y. Afek, E. Gafni and A. Rosen, Slide- a technique for communication in unreliable networks, *Proceedings of the ACM Symposium on principles of Distributed Computing,* 1992.

[21] Baruch Awerbuch, Boaz Patt-Shamir and George Varghese, Self-stabilization by local checking and correction, *Proc. 32nd Foundations of Computer Science* (1991), 268-277.

[22] M. Blum and S. Kannan, Designing programs that check their work,*Proc. 21st ACM Symposium on Theory of Computing* (1989) 86-97.

[23] M. Blum, W. Evans, P. Gemmell, S. Kannan and M. Naor, Checking the correctness of memories, *Proc. 32nd Foundations of Computer Science* (1991) 90-99.

[24] Fan R.K. Chung, Constructing random-like graphs, *American Math. Soc. Short Course Lecture Notes*, 1991.

There are numerous applications of expander graphs in a variety of topics including parallel sorting, fault-tolerant routing, pseudo-random number generation and approximation algorithms.

Another direction in studying network reliability is to build redundancy within a network, to define and measure redundancy, to relate redundancy to reliability and to use optimization techniques to design networks with an adequate number of alternative routes. Applications along this line include SONET rings[25] for optical fiber networks [26].

- Graph theory and algorithms
  Graph theory is the study of discrete structures and their relations. Many fundamental properties have led to either efficient heuristics or the understanding of the limitation of existing methods. For example, Binary Decision Diagrams upon which recent major improvements in model-checking are based are directed graphs. Expander graphs have become a very useful tool in many areas of computation. Also, graph separators have found many applications in recurrence algorithms and parallel architectures, among many other areas.

  In the area of algorithmic design and analysis, there has been substantial and continuing research. Work is being done both on classifying difficulty levels of generic problems and on extensions to parallel algorithms, on-line algorithms and probabilistic algorithms. Each of these topics deals with some aspects of dynamic structures and fault tolerance. The trend seems to be towards creating and using more sophisticated combinatorial tools in order to deal with harder problems. Although most of the work is theoretical in its nature, it is hard to tell which part will be of significant use in the future. In fact, numerous applications are already in use, as mentioned in the above areas.

1.3 Benefits and needs

There are a number of crucial but non-technical issues related to research on reliable software and communication. Here we mention the current status in the areas of cost, standards, regulatory and legal aspects concerning reliability of software and communication.

- Cost
  In the past three decades, the performance-price gain of computer hardware has been about 30 percent per year[27]. Consequently, software is in growing demand (at about 25 percent a

[25] T.-H. Wu, D. J. Kolar and R. H. Cardwell, High-speed self-healing ring architectures for future interoffice networks, *IEEE GLOBECOM*, Nov. 1989, 23.1.1-23.1.7

[26] Frank Hwang, Clyde Monma and Fred S. Roberts, DIMACS workshop on reliability of computer and communication networks, Technical Report 89-23 of DIMACS (Center for Discrete Mathematics and Theoretical Computer Sciences)

[27] The National Challenge in Computer Science and Technology, *National Academy Press, Washington*, DC 1988.

year) [28]. However, software development has undergone only incremental improvement in productivity so far[27]. Furthermore, the cost of software testing and software maintenance has been soaring[29].

It is worth mentioning that the cost of testing/quality-control usually falls on the vendors while the consequences of unreliable software are felt mainly by the users or the engineers who use the purchased software for large systems. Research in reliable software and communication can provide the approaches for evaluating the quality of vendor-software and thus increase mutual understanding/trust, as well as improving the cost of testing and, in many cases, reducing the cost of design.

- Standards
  In spite of the great difficulties in assessing reliability of software, efforts have been made in the areas of standards and regulations. Here we outline the current state in these areas and we make no attempt to discuss to what extent these rules have been complied with or verified.

  The National Security Agency issued the DoD standard in 1983 (revised in 1985) known as the Trusted Computer System Evaluation Criteria (TCSEC, usually called The Orange Book after the color of its cover)[30] for access control or, in general, for systems with shared memory. TCSEC provides a hierarchy of seven sets of evaluation classes, in increasing order of "trustworthiness" (D, C 1/2, B 1/2/3, and A1). Similar standards either exist or are in the process of being developed in Canada, France, Germany and the Netherlands[31].

  Currently, the National Institute for Standards and Technology is developing a FIPSPUB (a Federal Information Processing Standards Publication – a preliminary to a standard) on the Assurance of Software for High Integrity Systems. Another FIPSPUB concerns the Security Requirements for Cryptographic Modules (FIPSPUB 140-1). The IEEE is also in the process of developing a draft standard (P1228) for software safety plans which emphasizes the use of rigorous or formal methods.

  The International Telegraph and Telephone Consultative Committee (CCITT) is responsible for setting the standards for communication protocols. CCITT provides the standardized specifications, Description Languages and test languages. However, the CCITT standards do not cover implementation aspects of the protocol or the software reliability issues.

  In May 1989 the UK Ministry of Defense (MoD) promulgated Interim Draft Standards MoD-STD-0055 (Requirements for the Procurement of Safety) [32] which required the use

[28] Keeping the U.S. Computer Industry Competitive, Computer Science and Technology Board, *National Academy Press, Washington*, DC 1990.

[29] B.W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ 1981.

[30] National Computer Security Center, Trusted Computer Security Evaluation Criteria, United States Department of Defense, May 1983.

[31] Information Technology Security Evaluation Criteria: Harmonized Criteria of France, Germany, The Netherlands, and United Kingdom, May 2, 1990.

[32] Requirements for the Procurement of Safety Critical Software in Defense Equipment Requirements for the Analysis of Hazard in Safety Critical Software, Interim Defense Standards 0055 and 0056, UK Ministry of Defense,

of formal mathematical methods for specifying and verifying all safety-critical elements in a MoD contract project. The companion standard to 0055 is MoD-STD-0056 (Requirements for the Analysis of Safety Critical Hazards) which requires a thoroughly documented hazard analysis to be conducted. These standards were circulated for review and comment to industry, government agencies and academia over a two-year period and have recently entered into "discretionary use" status. The original versions of the standards contain some extreme conditions such as prohibiting unsafe programming practices (e.g., interrupt programming, dynamic memory allocation, recursion), prescribing only certain methods as acceptable and requiring independent safety review. The recently published Interim MoD Standard relaxes most of the above features by placing them in an advisory guidance section. However, the chief requirement for using "formal mathematical methods" remains and is likely to be included in the final version of the standard [33].

- Regulatory
  There have been increasing efforts by various regulatory agencies to address issues of risk and dependability through regulation, certification and education. For example, the Federal Aviation Administration (FAA) is in the process of upgrading the US Air Traffic Control Systems. Techniques such as fault-tree analysis are used in some parts of the development of the Traffic Collision Avoidance System[34]. FAA is also upgrading the certification requirement for avionics systems in commercial aircraft.

  Because of the rapid increase of software-control systems for medical instruments and treatment equipment, coupled with several serious related accidents, the Food and Drug Administration has begun to explore the use of rigorous techniques to the development of computer-based medical systems. Also, the Securities and Exchange Commission has shown an interest in the use of formal techniques of system design and development, as a result of recent computer-related difficulties with the program trading systems at the New York Stock Exchange and other stock exchanges in the world. To date, this interest has been manifested only by participation in the NIST FIPSPUB on Assuring Software for High Integrity Systems[33].

- Legal
  There have been an increasing number of cases in the US involving product liability claims concerning computer systems which indicate a growing awareness on the part of the courts of faulty computer systems as a source of injury. While there are many cases in which damage awards are sought and awarded in connection with faulty computer systems, there are also cases in which the courts have ruled against awarding damages for such computer failures. However, because of the trends in standardization and regulations, the burden of proof might be shared by the vendors/developers to prove that they used the "best possible practices"[33].

Directors of Standardization, May 9 1989.

[33] Ted Ralston, Trends in formal methods, MCC Technical Report STP-FT-200-91.

[34] Radio Technical Commission for Aeronautics DO-178B, Software Considerations in Airborne Systems, June 1990.

Also, in the rise is the number of law firms that offer a computer law practice with emphasis on personal injury, and recently large US insurance companies are offering personal liability insurance coverage designed specially for software developers[33].

We have now discussed *why* we need research in reliable software. We also mention several critical concerns on *how* research can be performed in an effective way.

- User-aided value is vital for software research.
  Software differs substantially from hardware in its evolution. The fruits of software research usually just set up the initial stage of the software life cycle which can only evolve through the growing and maturing phases by user interaction. One of the major factors for the success of the Unix operating system® was its early adoption by academic and research labs outside, while it was mostly ignored initially within the Bell System. Indeed, the transfer of technology from research to product is a complex route. There are many examples of research done that was first commercially developed outside and then ultimately found its way back to its birthplace in terms of a real product [35]. For example, linear predictive coding, a technique for speech synthesis, was invented at Murray Hill, but its first commercial use was the Speak-and-Spell product developed by Texas Instruments.

- Long-term efforts can lead to long-term payoff.
  "There is no Silver Bullet" is a well-known saying in software engineering[36]. The road to reliable software and communication is likely to be long and hard. Researchers must have the freedom to fail and to take risks by investigating directions that might lead nowhere. Although some of the approaches that are mentioned in the report could be useful even now or in the near future, many of the problems concerning the essence of computing have no easy answer and will require continuing long-term efforts.

1.4 Where are we heading?–Several challenging problems

Based on the brief review in the previous sections, it is quite evident that the current technology is far from being able to deal with the serious obstacles on the road to reliable software and communication. We usually hear about the lack of automated tools for testing and verification, but the truth is the lack of knowledge. We can imagine the analogous situation of trying to solve problems in quantum chemistry by applying the "art" of alchemy.

What we really need at present is a sound scientific foundation upon which the advanced technology in broadband communication and multimedia networks can be built. Giants, such as Alan Turing and John von Neumann, laid the logical foundations for the conception and

[34] Unix is a registered trademark of Unix System Laboratories, Inc.

[35] A. Michael Noll, The future of AT&T Bell Labs and Telecommunications Research, Telecommunications Policy, April 1991, 101-104.

[36] Frederick P. Brooks, Jr. "No Silver Bullet" Essence and Accidents of Software Engineering, *Computer*, April 1987, 10-19.

construction of computers, which led to the technological revolution of computer hardware. The next phase of the technological revolution will be centered on software in order to control and utilize the beasts generated by the hardware revolution. New ideas and concepts of the same magnitude and depth as Turing's or von Neumann's will be required to spawn the coming software revolution. So far, we do not even have a reasonable measure for the complexity of a software program, for example. In order to wake up potential sleeping giants, we here include several challenging problems to function as Socratic "gadflies" for the scientific foundations of reliable software and communication:

Problem 1: What are the underlying principles of reliable software?

Such principles can be crucial for building models, collecting data, or managing complexity. In particular, testing and verification depend heavily on quantitative parameters or measurements which capture the essense of computing. So, this problem actually lies at the heart of software quality.

To deal with a complex problem, one general approach is by levels of abstraction. Namely, the problem is reduced to a simpler form using techniques such as mathematical reasoning, logic, or data abstraction languages. In particular, what are the appropriate levels of abstractions for thinking about reliability?

Problem 2: How can we cope with complexity?

A major bottleneck in testing, debugging and verification is the astronomical number of states that might possibly be encountered. For example, suppose a message must be routed through every node in a network of 100 nodes as quickly as possible. A crude calculation shows that there are 100! or about 10^{150} different ways to route the message. Brute-force case-by-case analysis is completely infeasible for this problem because today's fastest computer can barely perform about 10^{12} operations per second. On the other hand, since the beginning of the universe (according to the current accepted estimates), there have been altogether about 10^{37} femto-seconds. (A femto-second is perhaps the shortest period in which anything physically significant can happen). Therefore, at most 10^{37} cases can actually occur, which is a very small portion of the previous estimate. This opens up a whole range of problems concerning how human understanding of the systems can reduce their complexity. For example, which problems are inherently intractable or undecidable and which problems have efficient algorithms? If a problem is provably intractable, can feasible heuristics or fast approximation algorithms be found? Sometimes, a problem that is intractable in general turns out to have efficient algorithms for special structures or under additional conditions. What is the impact of reliability on complexity? Perhaps, theoretical studies can provide some insight in these directions.

Problem 3: What software architectures promote reliability?

Is it possible to build fault-tolerant software systems (or even self-healing systems)? What are the tradeoffs in system performance for reliability ? How can we measure and predict the behavior of a software system when it is under stress? Can "firewalls" be designed to prevent errors which have a cascade effect? Will "orthogonal" systems increase software

diversity? To address these questions might well require combining our knowledge in logical design and performance analysis.

Problem 4: How do we model and engineer concurrent and real time systems?

The current telecommunication systems is basically a large distributed system. One of the crucial issues is to model concurrency and real time. This is essential for developing machine-processable specifications and for the ultimate goal of standardizing tools for automating the process of the design and implementation of distributed systems.

Problem 5: How do we specify reliable systems and how do we test whether a system meets its requirements?

Both of the above problems are central and, in fact, touch almost all the issues we have discussed so far. Such hard problems have no easy answers and require efforts in many related areas in order to develop systematic and automated methods.

One common problem in building a large network is the coordination of heterogeneous entities. For example, the telephone network involves many versions of SS7 implementations by many vendors, each with their own interpretation of the requirements. What kind of interface specifications and protocols are sufficient? How do we deal with ambiguity? Any understanding of this problem would be very helpful to the process of system integration.

Another question about reliability is "who is guarding the guards?" Perhaps a better statement of the question is "who is guarding the guards who are guarding the guards who are...?" The complication of the statement reflects the difficulties involved. Should the "guarding" system be entirely independent of the "guarded" system? How many iterations of audit programs are needed? Should requirements writers specify what the program should not do, rather than just say what it should do? These vital problems are intimately related to system management.

Problem 6: How do we measure and verify the reliability of a design?

One simple principle [37] [38] [39] is the divide-and-conquer strategy. Namely, to solve a large problem, first we partition the problem, if possible, into two or more parts in such a way that the interaction between the parts is "small" (i.e. feasible for current technology), and then proceed to solve each part and combine the solutions of the parts to yield a solution to the original problem. For example, can we design a protocol which can be recursively partitioned into smaller parts such that if the smaller parts are verified, then their combination can be easily verified?

The model-checking tools described in the previous section use reduction techniques that need to be improved to deal with this problem, and current methods are still far from being adequate. We also note that better software measure could be useful for partitioning. "Modularity" is one of the most popular buzzwords, but there is rarely used with precision.

[37] E.W. Dijkstra, Notes on structured programming, in *Structured Programming*, O.J. Dahl, E.W. Dijkstra and C.A.R. Hoare, Academic Press, NY (1972) 1-82.

[38] D. Parnas, On the criteria for decomposing systems into modules, *Comm. ACM* 15, 12, Dec. 1972, 1053-1058.

[39] N. Wirth, *Systematic Programming, An Introduction*, Prentice Hall, Englewood Cliffs, New Jersey, 1973.

The principle of divide-and-conquer can be very powerful for handling large problems. For example, suppose hypothetically a program of twenty thousand lines of code can be verified and suppose furthermore there are efficient and feasible schemes to verify the combination of two parts that have already been verified. We can then continue combining verified parts into larger ones. After eight iterations, the total size can reach well beyond 2 million lines. Of course, the number of lines is perhaps not a good measure, but it is used just to provide a rough estimate of the size.

We remark that the divide-and-conquer strategy could also be very helpful for software maintenance. When changes are incorporated into the existing problems, instead of verifying the whole software system, we only have to verify the updated modules that contain the changes while most of the previous verification involving unchanged portions will still be useful.

Problem 7: Can a systematic approach be developed to ensure software reliablity?

Some current computer systems (such as the connection machine CM-5) devote as much as 30 percent of their hardware design to ensuring the reliability of the machine itself. Since software is much more complex than hardware, it does not seem unreasonable to predict that even a larger percentage of software resources will be used to enhance software reliability. Of course, current software practice falls far short of this figure because of the lack of "know-how". Although our understanding and knowledge are limited, some of the methods such as the glass-box approach, monitoring/visualization, formal methods and model checking are already available. Hopefully, a coherent systematic approach (from design, architecture, implementation to maintenance) can eventually be developed from these seeds.

There is no question that all the above problems are fundamental and difficult, requiring new ideas, insight and persistent efforts. Some of the problems are "top-down" while others are "bottom-up" (such as the last one). Perhaps the most important problem now is to formulate crisp questions which are useful, say, in comparing existing methods or reliability models. In Sections 6-10, more problems are raised and discussed. There may indeed be no silver bullet for reliability problems. Long-term and large-scale efforts are needed in order to accelerate the continuous dialogue between research and practice, theory and applications, knowledge and tools.

1.5 Acknowledgement

I am indebted to numerous people for their help, without whom this report could not possibly have been written within the limited time available. In particular, I am grateful to Al Aho for getting me started on this unusual project and helping me throughout the preparation of this report. Also, I wish to express my deep gratitude to Bob Martin for his invaluable and extensive comments.

I have greatly benefited by the discussion on software research with Manuel Blum, Richard Demillo, Stuart Feldman, Richard Lipton, and Peter Weinberger. My understanding on protocols relies heavily on the generosity of Gerard Holzmann, Bob Kurshan, Mike Merrit, Debasis Mitra, Krishan Sabnani, M. Ümit Uyar. I have learned about network management and Signaling System 7 from Amir Atai, Jack Brady, Pete Brady, Don Branflick, Ashok Erramilli, Len Forys, Richard Goldberg, Bichlien Hoang, Donnie Thompson, and Stan Wainberg.

For many invaluable discussions, special thanks are due to Baruch Awerbuch, Richard Becker, David Bergland, Sandeep Bhatt, Ernie Brickell, Jane Cameron, Bob Cardwell, Fred Chang, Geoff Clemm, Stephen Eick, Tomas Feder, Sandy Fraser, John Guttag, Shlomo Halfin, Gary Hayward, Hank Kluepfel, Ming-Lee Lai, Tom Leighton, Charles Leiserson, Arjen Lenstra, Mike Lesk, Tom Mevaskey, Chao-Ming Lin, Nancy Lynch, Clyde Monma, Brent Morris, Jakob Nielsen Teun Ott, John Peoples, V. Ramaswami, Jeff Sanders, Celia Schahczenski, Lynn Streeter, Mike Todd, Guda Venkatesh, and Ali Zolfaghari. In addition, Ron Graham, Jim McKenna, Milena Mihail and Peter Winkler have contributed in numerous ways. Finally, I wish to recognize the crucial help of Nancy Davidson for searching a large number of references, coordinating schedules and designing the format. I also want to thank Nancy Gerstl for her timely help on many occasions.