Reflections on Industry Trends and Experimental Research in Dependability

Daniel P. Siewiorek, *Fellow*, *IEEE*, Ram Chillarege, *Fellow*, *IEEE*, and Zbigniew T. Kalbarczyk, *Member*, *IEEE*

Abstract—Experimental research in dependability has evolved over the past 30 years accompanied by dramatic changes in the computing industry. To understand the magnitude and nature of this evolution, this paper analyzes industrial trends, namely: 1) shifting error sources, 2) explosive complexity, and 3) global volume. Under each of these trends, the paper explores research technologies that are applicable either to the finished product or artifact, and the processes that are used to produce products. The study gives a framework to not only reflect on the research of the past, but also project the needs of the future.

Index Terms—Experimental research in dependability and security, computing industry trends.

1 Introduction

For more than four decades, Moore's Law has been a driving force in the computer industry. Doubling on a yearly basis leads to a three orders of magnitude increase in only a decade. Such large increases in capacity (i.e., number of transistors, processing performance, bits of data storage, and communications bandwidth) require fundamental rethinking of all phases of a product's life cycle, from design through usage and maintenance to replacement. Moore's Law also applies to volume as well as capacity. Intel produces more transistors yearly than the number of ants on Earth. Doubling in volume means that every couple of years more computers will be produced than were ever previously produced.

The IT industry has grown in many dimensions. While the Von-Neuman machine is still at the conceptual core, the industry that built at most a few thousand machines in 1970, today ships tens of millions annually. Employment has gone from a few thousand to a few million. And the breadth of the industry spans technology, manufacturing, software, and IT-enabled services, amounting to a world-wide figure in the range of two to three trillion dollars.

This paper identifies three trends in a computer industry fueled by Moore's Law, trends that directly impact computer system dependability and security: *shifting error sources*, *explosive complexity*, and *global volume*. The evolution of three research threads in experimental dependable systems—error monitoring, fault injection, and design methodology—are traced based upon the personal experience of the authors to illustrate how research responds to and, in some cases, anticipates the direction of the computer industry. The first two trends were identified decades ago and, hence, there is a rich history among the research threads with respect to these. The third, just emerging, can be used to predict future directions for research among the three threads.

Section 2 provides background and a framework for motivating the three industrial trends. The remaining sections describe each trend in turn and how experimental research in dependable and secure systems has responded. The more mature trends will have more details related to the three research threads due to their rich history. The emerging trend will be more speculative with respect to the research and, hence, have fewer details. Section 9 provides concluding observations.

2 Trend, Artifact, and Process

From the early days of computers (when vacuum tubes were used to perform logic and arithmetic opera-tions) to today's generation of computing systems, reliability (or, more broadly, dependability) has been considered a fundamental system attribute that determines the system's ability to provide continuous service to the end user. Evidence of early efforts in dependability can be found in publications from the 1960s, e.g., [95] and [8]. An excellent review describing the advances of IBM computer systems in the RAS (reliability, availability, serviceability) area from that time can be found in [52].

An important milestone in the evolution of dependable computing (theory and practice) was the establishment in 1971 of a technical conference on fault-tolerant computing: the First International Symposium on Fault-Tolerant Computing (FTCS). This forum has established itself as a primary arena for presentation, discussion, and dissemination of new ideas in the development of dependable systems.

Over the years, fault/error models have evolved along with the advances in system hardware and software. Table 1 summarizes the changes over the last four decades in terms

- *D. Siewiorek is with the Department of Electrical and Computer Engineering, Carnegie Mellon University, 3519 Newell Simon Hall, Pittsburgh, PA 15217. E-mail: dps@cs.cmu.edu.*
- *R. Chillarege is with Chillarege Inc., 210 Husted Avenue, Peekskill, NY 10566. E-mail: ram@chillarege.com.*
- *Z. Kalbarczyk is with the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1308 W. Main St., Urbana, IL 61801. E-mail: kalbar@crhc.uiuc.edu.*

Manuscript received 16 June 2004; accepted 2 Sept. 2004.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-0088-0604.

TABLE 1
Fault Sources, Levels of Integration, Users, and User Sophistication over the Past Four Decades

Decade	1970s	1980s	1990s	2000s
Typical systems	Mainframes	Workstations	Personal computers	Mobile devices, e.g., cellphones, PDAs
Fault/error sources	Hardware	Hardware, network	Hardware, network, software, human errors	Hardware, software, wireline/wireless networks, environment, e.g., frequent connectivity loss, malicious faults
Integration/ complexity	Close systems; highly custom designs, where both hardware and OS are fully controlled by the vendor	Mostly close systems; network connectivity; standard interfaces exposed to users	Open systems; wide access to network; COTS operating systems; third-party hardware and software	Open systems; proprietary and COTS operating systems; highly integrated PC-like systems
People/users	Tens of thousands	Millions	10 of millions	100 of millions
Level of user sophistication/ training	BS in engineering; 5000 hours	Basic knowledge in computing; 500 hours	Basic computing literacy; 50-100 hours	Training at the time of a purchase of a device; Hours

of the technology, error/fault sources, number of users, and their level of sophistication/training. In Fig. 1, four twisted funnels illustrate the trends in changes—the vertical axis represents time, and the horizontal axis reflects the magnitude of the changes.

The technology has evolved through dramatic changes starting with mainframes in 1970s (where highly skilled personnel were required to operate the systems) through the eras of workstations in 1980s, personal computers in 1990s, and the current generation of mobile/handheld devices (e.g., cell phones, PDAs), where the technology reaches the general public. Today, devices must often operate in highly variable and harsh environments. As a result, the technology must: 1) hide complexity so that a relatively unsophisticated customer can operate the device and 2) operate continuously despite errors/failures.

The initial focus in the 1970s was mainly on hardware errors, as the hardware devices were the major cause of problems. In the 1980s, with introduction of workstations and their network connectivity, made the network an important additional source of errors. In the 1990s, the wide use of personal computers executing commodity software made software a primary source of failures. The current decade is dominated by failures due to the environment and operators.

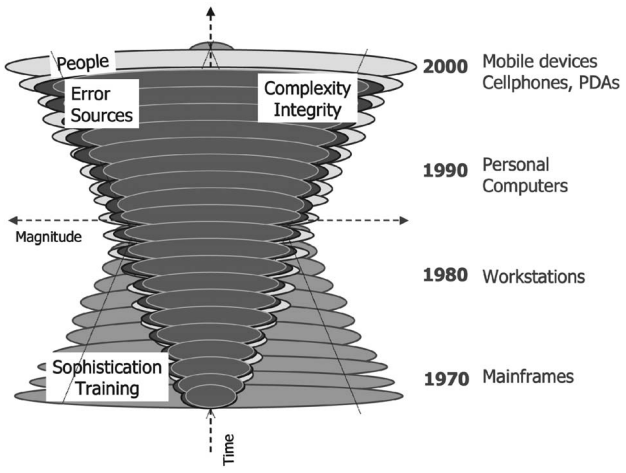


Fig. 1. Changes in technology and user basis.

It should be emphasized that the increasing complexity of systems causes operator errors to become a significant source of failures. For example, the analysis of failure data collected on the Public Switched Telephone Network (PSTN) and on three Internet sites indicate that operator errors contribute respectively to 59 percent and 51 percent of the system downtime [85]. While this study is rather limited in scope, it is a good indicator of the problems one will encounter in the near feature and emphasizes the importance of robust user interfaces. This is one area in which more research is needed to understand and model actual user behavior and to make predictive robustness analysis of systems possible.

Our framework is defined by three elements: *trend*, *artifact*, and *process*. A *trend* refers to an industry trend that has been taking place and has a direct impact on dependability. Each trend is distinct and has been consistently present for a substantial period of time, often decades.

An *artifact* is the product of the industry, be it a piece of hardware, a piece of software, or a service. An example of an artifact is a computer, a piece of shrink-wrapped software, or a cell phone contract. The artifact is the entity of commerce and defines the work-product of engineering effort.

A *process* is the means to produce an artifact. It consists of the engineering methods, tools, or labor the industry employs to create a viable method of manufacturing or development. As we reflect on artifacts and processes, we recognize that much of engineering and research is directed to one of the two, or both.

A body of research and methods can be associated with either the artifact or the process, and in a few instances, with both. If the research is embodied in the product after it is shipped, the association is with the artifact. Alternatively, the body of research helps in production, it is associated

TABLE 2
Industry Trends, Artifacts, and Processes

Industry Trend	Artifact	Process
Trend 1: Shifting Error Sources Failure rates drop in hardware and new sources dominate.	Monitoring Failure data analysis Fault injection	Raise the level of abstraction
Trend 2: Explosive Complexity Growth in system complexity, users, and shrinking user tolerance of failures.	Anomaly detection Trend analysis	Formal methods &model checking ODC Software reliability Testing Standards
Trend 3: Global Volume High level of integration and emerging open systems, a source of new dimensions in failures.	Proactive management Pervasive and cognitive computing Adaptive model of normal behavior	User interaction No test case Tools to assess resilience to both malicious and non-malicious errors

with the process. While quite often, academic research is associated with the artifact (which is understandable given the proximity of the problem set) there are examples where the research is actively involved into the process, e.g., hardware and software verification, testing, software reliability assessment, and development of architectural description languages.

In Table 2, trends are the rows, and artifacts and processes are the two columns. At the intersection of each row and column is the subject of our study. This is where the research methods, tools, techniques, concepts, algorithms, experiments, measurements, simulations, theories, processes, and conjectures lie. The body of work is applied sometimes on the artifact and sometime on the process, and this is what impacts the industry. Some of the work that applies during one part of a decade may not be as applicable at another, or vice versa. When we look at a trend that crosses 30 or more years, there is much change to ac-count for. In this article, we list the research that best fits this mosaic, allowing us to reflect on this metamorphosis of industry and research.

Trend 1—Shifting Error Sources clearly began in the 1980s, but was noticeable toward the end of the 1980s and by mid-1990s had caused a major change in the dependability area. *Trend 2—Explosive Complexity* began in the early 1990s when the cost of computing was dropping substantially and distributed computing was on a growth path. By the mid-1990s, the Internet boom contributed to even larger growth. *Trend 3—Global Volume* is only at its inception and can be argued to have begun with the huge increase in small yet powerful devices flooding the market. The cell phone, the PDA, and the availability of wireless digital networks will have their impact.

3 Trend 1—Shifting Error Sources

One of the dominant trends has been the change in failure rates as well as in the dominant sources of failures. By and large, we can conclude that hardware failure rates are currently down, while the relative contribution of software is up. In addition, as technology matures, the user set changes, and the degree of product sophistication increases, new sources of failures become prominent.

Transient faults have traditionally been associated with the corruption of stored data values. This phenomenon was reported as early as 1954 in adverse operating conditions such as locations near nuclear bomb test sites and later in space applications [121], [83]. Since 1978, dense memory circuits, both DRAM and SRAM, have been known to be susceptible to soft errors caused by alpha particles from IC packaging and cosmic rays. A hardware device can recover its full capability following a transient failure; nevertheless, such failures can be catastrophic for the correct execution of a program. This is because a corrupted intermediate value, if not handled, can corrupt all subsequent computations.

Continuously decreasing feature sizes and supply voltages of devices reduce capacitive node charge and noise margin, even flip-flop circuits inevitably become susceptible to soft-errors [38]. The high clock rate of modern processors further exacerbates the problem by increasing the probability of a new failure mechanism where a momentarily corrupted combinational signal is latched by a flip-flop. Constantly pushing the processor performance envelope will shortly place us in an unfamiliar realm where logically correct implementations alone cannot ensure correct program execution with sufficient confidence. As a result, vendors of high-availability platforms have long incorporated explicit error detection and correction techniques in their architectures. The basic techniques involve information redundancy (e.g., parity and ECC), space redundancy (achieved by carrying out the same computation on multiple, independent hardware at the same time and corroborating the redundant results to expose errors), and time redundancy (where redundant computation is obtained by repeating the same operations multiple times on the same hardware).

Memory arrays can be ECC-protected relatively efficiently because the cost of the coding logic can be amortized over the array. Applying ECC to individual registers in a processor may require a significant amount of overhead and increases the critical path delay. Typically, information redundancy is reserved for memory, caches, and perhaps registers files, whereas space and time-redundant techniques are employed elsewhere in the processor. Time redundancy has the shortcoming that persistent hardware faults may introduce identical errors to all redundant results, making errors indiscernible. Space redundancy has a complementary shortcoming in that a transient failure mechanism may affect the space-redundant hardware identically, again making errors indiscernible.

3.1 ECL to CMOS

The significant change in technology over the past decades is not only an increase in speed and reduction in power consumption, but also an increase in the reliability of the devices. Fig. 2, reproduced from IBM data [103], shows that system outages caused by hardware failure have dropped by two orders of magnitude in two decades.

This dramatic change in reliability has been the driver of a major portion of Trend 1. The shift in circuit technology

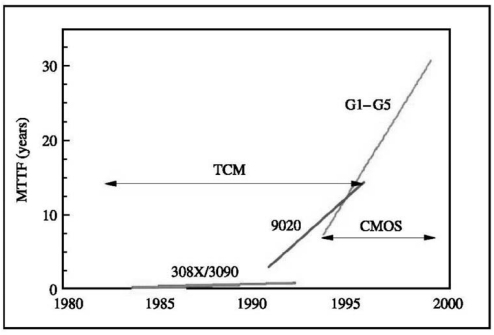


Fig. 2. Failure rate changes in hardware.

from ECL (the technology for the 308X/3090 series of mainframes) to CMOS is dramatic in terms of reliability, as shown in the figure. TCM in the figure stands for Thermal Conduction Module, a ceramic multichip package that is liquid cooled. The shift in circuit technology also changed geometry, power, and cost leading to a major shift in industry.

Considering the beginnings of FTCS in 1971, one can see why the focus in the early years was on hardware fault tolerance. Product dependability was defined by how well one could keep a box running in spite of hardware a malfunction, be it permanent or temporary. During the 1980s, power packaging technology allowed for units with substantially larger number of circuits in one module, this and the complexity of integration made it harder to service failed chips. IBM's thermal conduction modules (TCMs) provided significantly higher density per module, but individual transient failure rates were quite high. To combat transient failures, up to 25 percent of the circuitry was used for error detection and correction. These architectures allowed for very high data integrity, with no data path inside the CPU left unchecked. An instruction-retry mechanism further increased fault tolerance [102].

It was also becoming clear that the next generation of circuit technology, CMOS, would make ECL obsolete. As Fig. 2 illustrates, the MTTF of a high-end machine is more than 30 years, almost two orders of magnitude better than that class of machines two decades ago. As reported in [101], the IBM fourth-generation CMOS microprocessor-based mainframe (G4) achieved fault tolerance superiority over its predecessor ECL mainframe. While CMOS technology offered greater density (e.g., memory) and reduced power, increased density required novel error correcting codes to enable recovery from single chip errors.

The same time period has witnessed the growth of the microprocessor and the PC industry. Today, Intel in the P6 family processors (Pentium Pro, Pentium II, Celeron, Pentium III) brings high-end features to the mass market. All the P6's internal registers are parity-checked, and the 64-bit path between the CPU core and Level-2 cache uses ECC. Built-in diagnostic features allow monitoring and reporting on more than 100 events and variables inside the chip, including cache misses, register contents, and occurrences of self-modifying code. The P6 also improves support for checkpointing (i.e., rolling back the machine to a known state in the event of an error); however, the operating system has to be written to take advantage of machine-check interrupts.

A 1997 *Computer* article [10] concludes with "Eventually, one enterprising chip builder will deliver the first fault-tolerant microprocessor at a competitive price, and soon thereafter fault tolerance will be considered as indispensable to computers as immunity is to humans. The remaining manufacturers will follow suit or go the way of the dinosaurs." Now, about seven years later, while processor manufactures do not always explicitly talk about fault tolerance in their design, the current generation microprocessors employ multiple mechanisms for detecting and recovering from errors, as indicated by the examples given above.

However, there is an issue with the soft-error rate (SER) rising relative to the technology of just a couple of years ago. This occurs as device geometry shrinks and we push physical limits making soft errors (e.g., due to radiation) in memories and logic of microprocessors more likely. Recent studies [97] show that, while SER for single SRAM cells declines slightly with decreasing device sizes and stays relatively constant for latches, the SER for the microprocessor logic circuits increases by orders of magnitude when going from 600nm to 50nm feature size. These changes in error sources and rates should guide future research.

Fortunately, tolerating bit flip errors today has strong parallels with the design philosophies of yesterday's mainframe. Generations of IBM main frames in the late 1980s and early 1990s used thermal conduction modules and were concerned with transient bit flips; this led to designs with no unchecked data path. We will need to recapture the design philosophies and modify/enhance them to the specifics of current technology.

3.2 Software Failures

One of the consequences of the dropping hardware failure rate is that other failure modes have become more prominent. Software, which has also been growing in complexity, has gradually contributed to a larger proportion of system outages. Through the 1980s, while the fault tolerance methods were being developed for hardware and the incidence rate of hardware failures was dropping, software failures became more prominent. At the same time, the focus on software reliability methods was marginal. In the high-end server business, most of the development budgets were focused on new functionality since that was a growth segment all the way into the 1990s. The PC segment was at its inception and the focus was also functionality. As a consequence, software failures—the class of problems that had damaging effects as significant a hardware outage that took the entire system down—became evident.

The high-end server industry responded rapidly. Both fault avoidance and fault tolerance techniques where applied. Just like the hardware platforms for the high-end servers, the software operating systems included more and more recovery code. The result was impressive. Two decades later, a high-end IBM server has almost no cold starts in an entire year.

An important development in providing runtime mechanisms for handling software failures was the inception of design diversity. The seminal academic work

TABLE 3
Examples of Experimental Dependability Research

	1970s	1980s	1990s	2000s
Operational life monitoring	Crash dumps	Error logs	Natural workloads	Human-computer interaction errors
Fault injection	Stuck-at	Memory	API (Application Programming Interface)	Security

materialized as *N-Version Programming* [9], *Recovery Blocks* [2], and *N-Self-Checking* [68] approaches. Extensive experimental studies were conducted to assess effectiveness of the proposed solutions, e.g., [15]. Examples of industrial use span embedded control systems in particular avionics and railway domains, e.g., [63], [111].

3.3 Planned Outages

Faults and failures produce the mental image of uncertainty and catastrophic consequences, which, while they do happen, are far less common in high-end servers. However, high-end computing has a disturbing problem called "planned outage," in which systems need to be shutdown on purpose. Planned outage used to be common with installation and maintenance of hardware; later, it became common with software updates and maintenance. Databases needed to be reorganized or networks reconfigured. While the surprise element was not present, the unavailability and disruption of services caused just as much a problem. With businesses running globally, 24/7 availability was vital, and planned outages accounted for more downtime in the 1990s than unplanned outages.

4 Experimental Research versus Industry Trend 1

Research on experimental evaluation of dependability has advanced following the changes in hardware and software technologies. In general, methods and techniques employed to assess systems correspond to stages in the system's lifetime. In the *design stage*, computer-aided design environments are used to evaluate the design via simulation, including simulated fault injection. In the *prototype stage*, the system runs under controlled workload conditions. In this stage, controlled physical fault injection is used to evaluate the system behavior under faults. In the *installation/operational stage*, a direct measurement-based approach can be used to evaluate systems in the field under real workloads.

Table 3 summarizes the trends in experimental dependability research across four decades, organized by two methods: *monitoring operational systems* and *artificial evaluation by fault injection*.

The individual columns highlight emerging *sources* of data being colleted from systems in the field and fault/error *types* being integrated into fault injection tools and environments. *Analysis of failure data* from operational systems provides insight into the dominant error categories in deployed systems. It also gives valuable feedback for driving fault/error injection experiments. *Fault/error injection* allows acceleration of failure occurrence in the system and, hence, provides very rapid validation of prototype design and further guidance to design decisions.

4.1 Operational Life Monitoring and Failure Data Analysis

Understanding the characteristics of a fault source evolves through several stages. Initial measurements focus on summarizing the underlying statistical distribution with averages such as mean time to an event. Since little is known about the fault source, existing measurement frameworks are used to make estimates. This monitoring may be primary (such as analysis of system event logs) or secondary (such as reports from the field). To discover more about the statistical properties of the source (such as distribution type and distribution parameter values), customized error monitoring systems that are sensitive to the fault source, while, at the same time, filtering out extraneous information on other sources, have to be developed. In the next stage, a deeper semantic understanding of the fault source and how it propagates is used to devise real time anomaly detection so that the onset of a new fault can be discovered and isolated quickly. This pattern of the evolution of stages applies to each fault source, and the depth of understanding of a fault source is directly related to how many of these stages have been explored.

While substantial progress had been made in the area of understanding hard failures, transient faults posed a much harder problem. Once a hard failure had occurred, it was possible to isolate the faulty component; on the other hand, by the time a transient fault manifested itself (perhaps in the form of a system-software crash), all traces of its nature and location were long gone. Methods of detecting transient-induced errors varied widely, from internally detected errors reported in a system event-log file to specially written pro-grams that automatically loaded diagnostics into idle processors, initiated the diagnostics, and periodically queried the diagnostics as to their state to a triply redundant system. Transient faults were seen to be approximately 20 times more prevalent than hard failures. Gross attributes of observed transients were recorded [98]. The data illustrated that the manifestation of transient faults was significantly different from the traditional permanent fault-models of stuck-at-l and stuck-at-0.

4.1.1 Error Distributions

An in-depth understanding of system event-logs enabled analysis of the interarrival times of transient errors. Studies of these times indicated that the probability of crashes decreased with time, i.e., a decreasing failure-rate Weibull function (and not an exponential distribution) was the best fit for the data [77] (DEC computer system), [54] (IBM-VM/SP operating system), and [118] (Windows NT operating system). Because experimental data supported the decreasing failure-rate model, a natural question was "How far could you stray if you assumed an exponential function with a constant, instead of a decreasing, failure-rate?" The difference in reliability—as a function of time between an exponential and a Weibull function with the same parameters—was examined. Reliability differences of up to 0.25 were found. Because the reliability function can range only between 0 and 1, this error is indeed substantial [22].

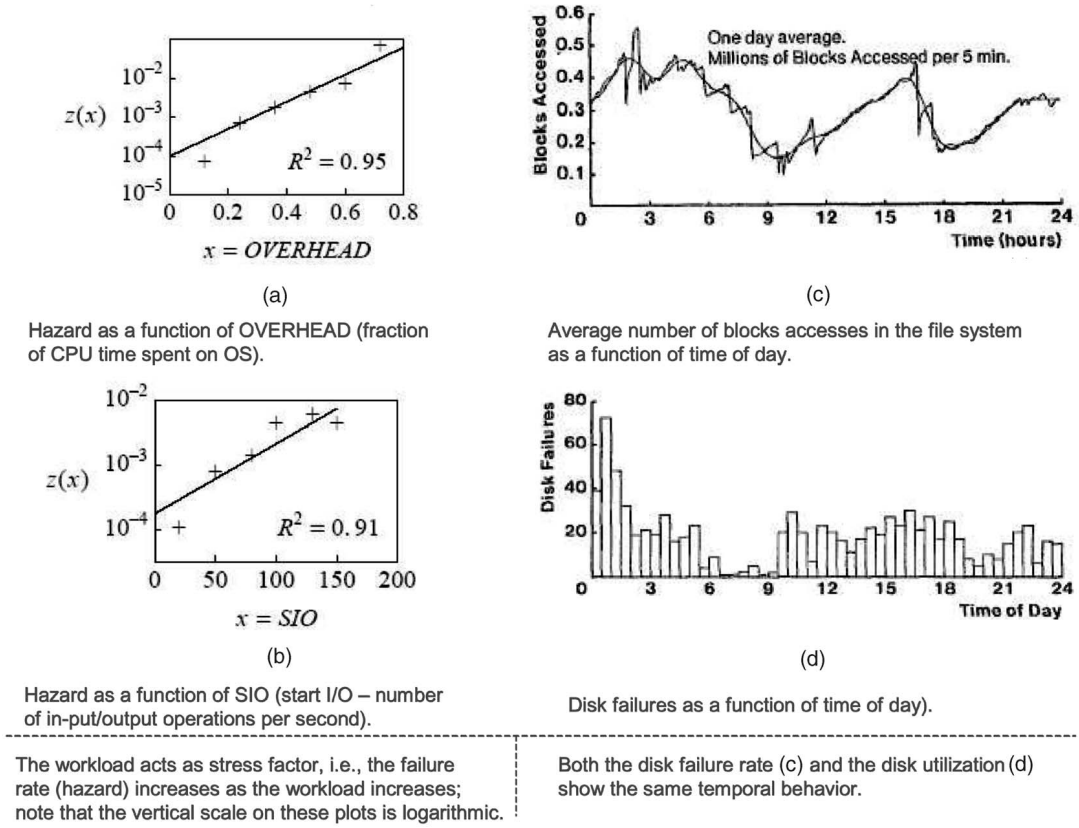


Fig. 3. (a) and (b) Plots of workload hazard for the IBM 3081 system and (c) and (d) profile of disk accesses and disk failures for the DEC system.

4.1.2 Impact of Workload on System Failures

One of the seminal contributions from data analysis was an understanding of the relationship between the system load and the system error/failure rate. The workload/failure dependency issue was studied in the early 1980s. Two primary approaches were employed: 1) *statistical quantification* of dependency between workload [17], [54] and failure rate and 2) *stochastic modeling* as function of workload [21]. The key conclusion from these studies indicated that there is a strong correlation between workload and failure rate and, as a consequence, dependability models must account for the impact of system workload.

A performance-reliability model for computing systems introduced in [20] gives quantitative results about how much a user can expect from a system as a function of the workload and reliability. The study indicated a four to one range in mean time to system failure as a function of system load. Subsequent in-depth analysis of system event-log entries and system load led to the derivation of a new system model—the cyclostationary model—that predicted failures involving hardware and software errors (see Figs. 3c and 3d). This model was an excellent match to the measured data and also exhibited the property of a decreasing failure-rate. A physical test demonstrated that, at the cost of some modeling accuracy, the Weibull function was a reasonable approximation to the cyclostationary model, with the advantage of less mathematical complexity [21].

Concurrently, statistical analysis of failure data from SLAC (Stanford Linear Accelerator Center) also indicated a strong relationship between failure rate and system utilization (a factor of 5 change in basic component reliability due to variation in load) [17]. Subsequent study [53] of internal CPU errors at SLAC introduced a load hazard model to measure the risk of a failure as system activity increases. In addition, the analysis showed that most of the errors are transient or intermittent. The load hazard model was applied to the software failure and workload data collected from an IBM 3081 system at SLAC running VM operating system [54]. Analysis in [55] showed that the probability of a CPU-related error increases nonlinearly with increasing workload. The resulting increase in the error probability can be 50 to 100 times more than that at a low workload (see Figs. 3a and 3b).

4.1.3 Trend and Symptom Analysis

A natural extension of work with system event-logs was to analyze log entries to discover trends [113]. From a theoretical perspective, the trend analysis of event-logs was based on the common observation that a hardware module exhibits a period of (potentially) increasing unreliability before final failure. Trend analysis developed a model of normal system behavior and watched for a shift that signified abnormal behavior. By discovering these trends, it was possible to predict certain hard failures (and even to discern hardware/software design-errors) prior to the occurrence of catastrophic failure.

One trend-analysis method employed a data-grouping or clustering technique called *tupling* [113]. *Tuples* were clusters, or groups, of event-log entries exhibiting temporal or spatial patterns of features. The *tuple* approach was based on the observation that, because computers have mechanisms for both hardware and software detection of faults,

single-error events could propagate through a system, causing multiple entries in an event-log. *Tupling* formed clusters of machine events whose logical grouping was based primarily on proximity and time in hardware space. A single *tuple* could contain from one to several hundred event-log entries.

A methodology of automatic recognition of the symptoms of persistent errors in large system by statistically relating the different manifestations of the same problem was proposed in [56]. The approach enabled differentiation between transients and intermittent errors. Lee and Iyer [70] analyzed the reports on software failures in the Tandem GURDIAN90 operating system to derive failure symptoms, which can be used for identification of recurrent software failures. Results show that 72 percent of reported field software failures are recurrences of known software faults and 70 percent of the recurrence groups have identical characteristics.

4.1.4 Online Monitoring and Diagnosis

The research was slowly progressing toward the online diagnosis of trends in systems. There were three basic parts to the monitoring and diagnostic process, and correspondingly, three basic requirements for building a system to implement the process:

- *Gathering data/Sensors*. Sensors must be provided to detect, store, and forward performance and error information (e.g., event-log data) to a diagnostic server whose task it is to interpret the information.
- *Interpreting Data/Analyzers*. Once the system performance and error data have been accumulated, they must be interpreted or analyzed. This interpretation is done under the auspices of expert problem-solving modules embedded in the diagnostic server. The diagnostic server provides profiles of nor-mal system behavior as well as hypotheses about behavior exceptions.
- *Confirming Interpretation/Effectors*. After the diagnostic server interprets the system performance and error information, a hypothesis must be confirmed (or denied) before issuing warning messages to users or operators. For this purpose, there must be effectors for stimulating the hypothesized condition in the system. Effectors can take the form of diagnostics or exercisers that are downline loaded to the suspected portion of the system and then run under special conditions to confirm the fault hypothesis or to narrow its range.

One of the first projects exploring online monitoring and diagnosis was CMU's Andrew System, a distributed personal computing environment based on a message-oriented operating system. When a fault was exercised, error events propagated from the lowest hardware error-detectors, through the microcode level, to the highest level of the operating system. The error dispersion index—the occurrence count of related error events—was developed to identify the presence of clustered error-events that might have caused permanent failure in a short period of time [71].

TABLE 4
Impact of Correlated Errors/Failures on System Availability

System	VAX1			VAX2
Model	5-out-of-7	4-out-of-7	3-out-of-7	3-out-of-4
Measured	$4.2x10^{-4}$	$1.9x10^{-4}$	$1.3x10^{-4}$	$2.5x10^{-5}$
Independ.	$3.5x10^{-6}$	$1.4x10^{-8}$	$3.3x10^{-11}$	$2.0x10^{-6}$
Meas./Ind.	$1.2x10^{2}$	$1.3x10^{4}$	$4.0x10^{6}$	1.2x10

Subsequently the Dispersion Frame Technique (DFT) [72] was developed to effectively extract error-log entries (which were caused by individual intermittent faults) and a set of rules that could be used for fault-prediction. In ANDREW networks, these rules were able to predict 93 percent of the physical failures with recorded error-log symptoms including both electromechanical and electronic devices. The predictions ranged from 1 to more than 700 hours prior to actual repair actions, with a false-alarm rate of 17 percent. A portable version of the DFT was implemented in *Dmod* [96], an online system dependability measurement and prediction module.

4.1.5 Impact of Correlated Failures on System Dependability

In constructing reliability/availability models of systems, one often makes simplifying assumptions to handle complexity and to enable meaningful analysis. One of common assumptions is independence of errors/failures in different components. Experience shows that this can result in overestimating system reliability/availability by orders of magnitude. For example, analysis of failure data from two VAX clusters showed that correlated failures involved 27 percent (VAX1) and 9 percent (VAX2) of all errors and occurred due to error propagation between machines. Table 4 shows the impact on system availability of the assumption of failure independence. Assuming independent failures may result in overestimating availability by one to six orders of magnitude [108].

4.1.6 User-Perceived System Dependability

More recent studies on failure data analysis attempt to measure dependability as it is perceived by the customers/users. From the user perspective, the dependability characterizes the ability of a machine/system to provide service, not just to stay alive. Experience shows that there can be a significant difference in the availability from the system and from the user/application perspectives.

For example, in [60], a study of the LAN Windows NT-based mail servers indicates that while the measured availability of the system was 99 percent, the user-perceived availability was only 92 percent, i.e., the system often can be alive but not able to provide a required service. A more recent study [100] reports similar figures (99 percent) on system availability of Windows NT and 2K workstation and servers. The need to account for the user's perception of system dependability is also stressed in the analysis of Windows 2000 dependability [81].

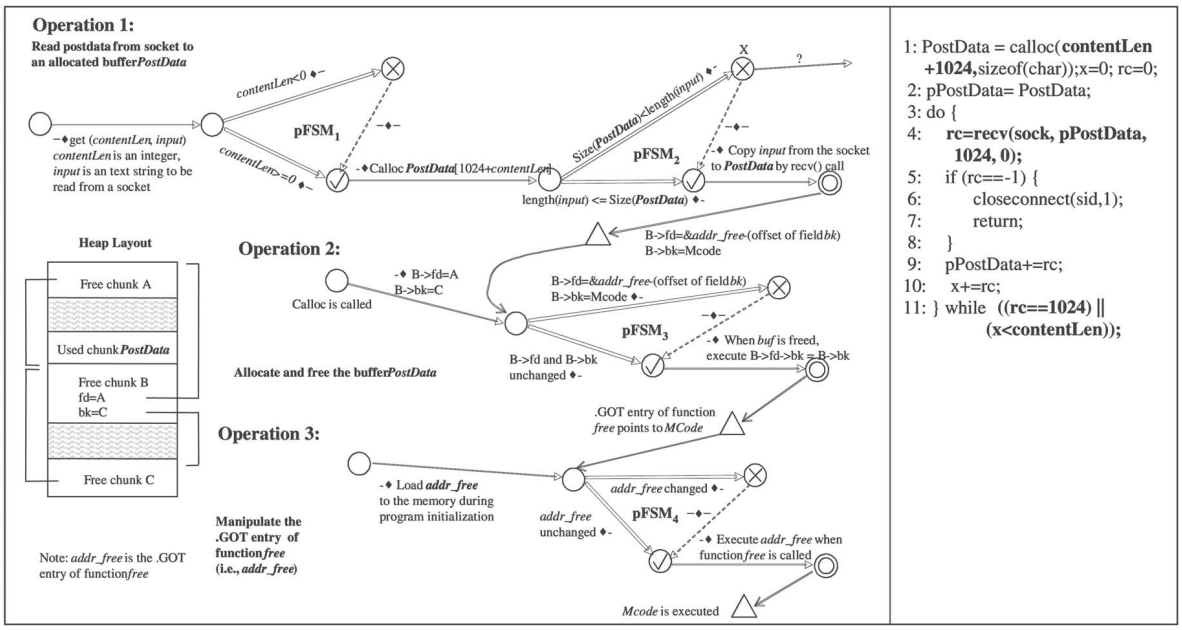


Fig. 4. FSM for NULL HTTPD heap overflow vulnerability.

4.1.7 Use of Measurements

Employing a single measure (e.g., an average) to characterize system dependability attributes may be somewhat misleading (many studies have shown that the distributions are highly skewed, e.g., [72], [54]). For example, study of Internet host reliability [59] showed that on average, a host remains unavailable to the user for 6.5 hours (during the 40-day experiment, i.e., approximately 2.5 days per year), an availability of 99.9 percent Examining the distribution indicated: 1) 45 percent of hosts had a total downtime ranging from 1,000 seconds to 7,000 seconds and a median downtime of nearly an hour (approximately 9.5 hours per year), 2) 49 percent of hosts had a total downtime ranging from 7,000 seconds to 70,000 seconds and a median downtime of about 4.5 hours (approximately 40 hours per year), and 3) 6 percent of hosts had a total downtime ranging from 90,000 seconds to 120,000 seconds and a median downtime of about 2.2 days (approximately 20 days per year).

4.1.8 Measurement-Driven Security Vulnerability Analysis

Challenged by the increasing number and severity of malicious attacks, security has become an issue of primary importance in designing dependable systems. Several studies have proposed classifications to abstract observed vulnerabilities into easy-to-understand classes. Representative examples include Protection Analysis [13], Landwehr's taxonomy [67], Aslam's taxonomy [7], and the *Bugtraq* classification [123]. Similarly, taxonomies for intrusions have been proposed. Examples include Lindqvist's intrusion classification [73] and the Microsoft STRIDE model [50]. In addition to providing taxonomies, [67] and [73] perform statistical analysis of actual vulnerability data, based on the proposed taxonomies.

A study presented in [24] combines an in-depth analysis of real data on security vulnerabilities with a focused source-code examination to develop a finite state machine (FSM) model for depicting and reasoning about security vulnerabilities. The developed formal reasoning uncovers the process of exploiting the vulnerabilities, and it can be further exploited to extract the logic predicates that need to be met to ensure vulnerability-free system implementation.

In the FSM approach, each predicate is represented as a primitive FSM, or pFSM. The primitive FSM consists of four transitions and three states. In the initial state, input specifications are checked. The other two states reflect rejection or acceptance of the input (marked in Fig. 4 with an X or checkmark, respectively). A hidden potential transition (shown in the figure as a dotted line) reflects a vulnerability that causes an input that should be rejected according to the specifications, to be accepted. While our objective here is to reason that a vulnerability (violation of a derived predicate) is not present in the implementation, the process of this reasoning can also allow us to uncover a previously unknown vulnerability.

For example, in the process of constructing the FSM model for the known vulnerability of the null HTTPD application (a multithreaded web server for Linux and Windows platforms), a new and as yet unknown vulnerability (*Bugtraq*, ID 6255) was discovered. To see why, observe that the null HTTPD heap overflow can be modeled as a series of four pFSMs as shown in Fig. 4. (The right side of the figure gives the source code for the vulnerable function, *ReadPOSTData*.) pFSM1 designates the predicate that checks *contentLen* against the specification. Similarly, pFSM2 designates a predicate that checks the actual length of the supplied input.

The input should be rejected if its length is larger than allocated buffer size, i.e., it takes the transition marked "?". Source code Line 11 controls the termination condition of *recv*(source code Line 4). However, due to a logic error (operator || should be && in source code Line 11), *recv* does not terminate before the entire input string is read from the socket. Thus, the outgoing transition (marked with

a "?") from state X does not exist, and instead, the hidden transition to the accept state is taken. A malicious user can supply the right *contentLen* but an arbitrary-length string input to overflow the buffer *PostData*. Constructing the FSM allowed us to uncover this new vulnerability [24].

If the checks corresponding to the predicates depicted by pFSM1 and pFSM2 are not in place, the impact of the vulnerability is further analyzed using pFSM3, which describes the operation of manipulating the heap layout (as shown on the left side of Fig. 4). Specifically, in this example, the attacker exploits this vulnerability and overwrites the GOT (Global Offset Table) entry of the function *free()* so that it points to the location of malicious code *MCode*. pFSM4 depicts the consequence of the corruption of the GOT entry of *free()* (i.e., *addr_free*). Finally, when *free()* is called again, *Mcode* is executed. In summary, this model consists of three operations. The first operation encompasses two activities, each described by an independent pFSM (pFSM1 and pFSM2). Operation 2 and operation 3 each consist of a single pFSM. Cascading these four pFSMs allows us to reason through all of this vulnerable code.

The proposed FSM methodology is demonstrated by analysis of several types of vulnerabilities reported in the *Bugtraq* database: stack buffer overflow, integer overflow, heap overflow, file race condition, and format string vulnerabilities, which constitute 22 percent of all vulnerabilities in the database. For the studied vulnerabilities, three types of pFSM were identified that can be used to analyze operations involved in exploiting vulnerabilities and to identify the security checks needed at the elementary activity level.

4.1.9 Security Monitoring—"Know Your Enemy"

An important activity in characterizing, understanding, and predicting security attacks is creation of efficient tools and methods for collecting and sharing (with security community) data on every attack and exploit against different target systems. Currently, multiple sources of data on security vulnerabilities, secu-rity attacks and different level statistics are widely available, e.g., *Bugtraq* [123], *CERT* [124], and *Security Tracker* [125].

Also, several projects evolved to provide directed and systematic means of collecting and sharing data on security attacks. Most prominent example is the *Honeynet* project [49], which has been collecting and archiving information on *blackhat* (adversary) activity. The primary goals of *Honeynet* include: 1) making the community at large aware of security threats and 2) providing early warning and prediction of attacks (by identifying trends and methods employed by the attackers, it may be possible to predict an attack and react before it happens). An important component of *Honeynet* is a concept of *honeypots* [104], i.e., systems designed to be compromised by an attacker. Once compromised, such systems can be used to detect and alert about intruders or to serve as a deception mechanisms. More information on the project and relevant publications can be found at [126], [127].

4.2 Fault/Error Injection

Fault injection has been used for more than three decades and it is generally recognized as an important method for dependability analysis. It is usually employed either in the early design stage (simulation-based tools) or at the prototype stage (hardware and software based physical fault injection). Fault/error injection can be employed to conduct detailed studies of the complex interactions between fault/error and fault/error handling mechanisms.

The earliest fault injectors use a pin-level injection, e.g., [4], [66]. One important step in the evolution of fault injection was formalization of the key components of a fault injection experiment: the faults set, the set of activations, the set of readouts, and the derived measurements [5], [11]. These concepts were further extended by formalizing *failure acceleration* (i.e., injecting faults at a high rate to increase the number of failures observed) and applying it to an IBM 370 mainframe [25]. The seminal work on abstracting the notion of failure was presented in [88].

The biggest advance in fault injection in the late 1980's was the idea of using software instead of hardware to inject faults because the hardware faults injectors often damaged the target system and were expensive to build. This idea is called software implemented fault injection (SWIFI). As result of an intense research multiple tools were developed e.g., FIAT [94], FERRARI [61], FINE [64], ASPHALT [120], Xception [19], NFTAPE [106], MAFALDA [90].

Another important method of injecting faults was the use of radiation sources to induce single event upsets (SEUs) on exposed ICs [43]. Power supply fault injections attempt to mimic the effects of transients on the power bus of a circuit or system [115]. Normal operation of the system is likely to generate varying levels of current demands, and power supply injection effectively simulates any worst-case current demands that the system might endure. A detailed analysis of three physical fault injection techniques—pin-level, heavy-ion, and EMI (electromagnetic interferences)—together with the comparison with respect to SWIFI is presented in [6].

Laser fault injection (LFI) has emerged as a preferred contact-less method of inducing SEUs in semicon-ductor circuits. In this approach, the laser beam mimics the effects of heavy-ion radiation [93]. Several simulation-based tools have been used to test fault-tolerant designs by emulating the effects of faults, e.g., DEPEND [40], MEFISTO [58].

Analysis of monitored data drives the development (or extension) of fault injection tools. For example, in [112], stress-based fault injection is employed to evaluate one of the first UNIX-based fault-tolerant systems developed by Tandem (now a division of HP). The stress-based approach ensures fault/error injection to system components when they are heavily used (i.e., highly stressed). This allowed meaningful comparison of systems and was an important step towards system benchmarking.

While fault/error injection methods and techniques have been extensively studied in academia, industry also employs fault injection. Work in [79] reports on fault injection-based testing of recovery and serviceability in the IBM ES/9000 systems. Fault injection and software testing were used by Ansaldo-Cris, Italy to assess dependability of new generation of Railway Control Systems [1]. In [34], physical fault injection at the pin-level was employed to

validate error-handling mechanisms of teraflops supercomputer developed by Intel. Software implemented fault injection assists in evaluation of embedded flight control system [114]. The discussion in the following sections provides several examples of fault injection studies, employed to assess system dependability.

4.2.1 Fault Injection at the Memory Level

Understanding an operating system's sensitivity to errors and identifying error propagation patterns is important in selecting a computing platform and in assessing trade offs involving cost, reliability, and performance. In order to provide insight into these issues, a series of fault/error injection experiments was conducted to obtain an insight into how the Linux kernel responds to errors that impact kernel code, kernel data, kernel stack, and processor system registers, and how processor hardware architecture (instruction set architecture and register set) impacts kernel behavior in the presence of errors [42]. Two target Linux-2.4.22 systems were used: the Intel Pentium 4 (P4) running RedHat Linux 9.0 and the Motorola PowerPC (G4) running YellowDog Linux 3.0. The study finds for example that: 1) the activation of errors is generally similar for both processors, 2) less-compact fixed 32-bit data and stack access makes one of the platforms less sensitive to errors, and 3) the most severe crashes (those that require a complete reformatting of the file system on the disk) are caused by reversing the condition of a branch instruction. Since the recovery from such failures may take tens of minutes, those failures have a profound impact on availability.

4.2.2 Fault Injection at the Operating System API Level

The success of many products depends on the robustness of not only the product software, but also operating systems and third party component libraries. But, until now, there has been no way to quantitatively measure robustness. Ballista changes this by providing a simple, repeatable way to directly measure software robustness without requiring source code or behavioral specifications. Ballista is a "black box" software testing tool, and it was demonstrated on testing the APIs of Commercial Off-The-Shelf (COTS) software. [65] provides a comprehensive assessment of 15 POSIX-compliant operating systems and libraries as well as the Microsoft Win32 API.

Each of the 15 different operating system's robustness has been measured by automatically testing up to 233 POSIX functions and system calls with exceptional parameter values. Overall, only 55 to 76 percent of tests performed were handled robustly, depending on the operating system being tested. Hardening can be accomplished by first probing a software module for responses to exceptional inputs that cause "crashes" or "hangs." When these robustness bugs have been identified, a software wrapper can be automatically created to filter out dangerous inputs, thus hardening the software module. More details and ideas on use of protective wrappers may be found in [3], [35], [91].

4.2.3 Security Threat of Firewall Data Corruption Due to Transient Errors

Recently, an exiting research avenue where fault/error injection was applied is the exploration of the possibility of security violations due to errors. In [119], it was shown that naturally occurring hardware errors can cause security vulnerabilities in network applications such as an FTP (file transfer protocol) and SSH (secure shell). As a result, relatively passive but malicious users can exploit the vulnerabilities. While the likelihood of such events is small, considering the large number of systems operating in the field, the probability of such vulnerabilities cannot be neglected. In another study, fault/error injection was employed to experimentally evaluate and model the error-caused security vulnerabilities and the resulting security violations on two Linux kernel-based firewall facilities (IPChains and Netfilter) [23]. Using data on field failures, data from the error injection experiments, and system performance parame-ters such as processor cache miss and replacement rates, a SAN (Stochastic Activity Network) model was developed and simulated to predict the mean time to security vulnerability and the duration of the window of vulnerability under realistic conditions. The results indicate that the error-caused vulnerabilities can be a nonnegligible source of security violations.

4.2.4 Dependability Benchmarking

Continuously increasing sophistication (support of new fault/error models and targets), representativeness (ability to create real life failure conditions), and automation of fault/error injection experiments creates an opportunity for fault/error injection to become an enabling technology for dependability benchmarking. The dependability benchmarking aims at providing cost-effective experimental methods and procedures to evaluate the behavior of components and computer systems in the presence of faults, allowing the quantification of dependability attributes and characterization of systems in terms of well-defined dependability classes.

An important initiative has been undertaken by the IFIP Working Group 10.4, which has created a Special Interest Group (SIG) on Dependability Benchmarking to promote the research, practice, adoption, and dissemination of benchmarks for computer-related system dependability [122]. While there is still a lot of work to be done before an actual dependability benchmark(s) is/are specified and, more importantly, accepted by the industry and the user community, a fault/error injection should be considered as an important (if not crucial) technology in addressing this challenge.

5 Trend 2—Explosive Complexity

Complexity arises from several factors, and with complexity, issues in dependability become more diverse. The lines between preship and postship are increasingly blurred by our ability to ship fixes instantly on the Internet. Customizability and interoperability of services further makes products, vendors, and services less distinguishable.

Wireless technology makes device and user locations transparent, and the demands 24/7 availability spread across the industry.

5.1 Growth and Complexity

This huge growth is not just due to selling more computers and increasing the installed base of software. Applications, infrastructure, and services have exhibited compounded growth. Thus, while individual segments of the industry show a steep linear growth, the complexity of the systems built for them grows nonlinearly.

While it is always hard to measure individual engineer productivity, which we believe has grown due to better software engineering tools, the labor market growth figures illustrate substantial growth. Since development continues on larger components, which are themselves growing in features and functionality, the ability to synthesis larger and more complex systems has grown substantially. For instance, a modern Web application is built on a number of standard components—operating system, middleware, Web server, transaction manager, database system, and the rendering layers supporting streaming media—all built upon a transport layer or network that is configured separately. The application and business logic is an industry-specific product layered upon these other components. Integrating the various components and dealing with the complexity of such systems is a new layer of conceptualization with its own dependability issues.

5.2 Redefining Failure

For decades, we have assumed that *failure* is a well-understood concept. The IEEE/IFIP definition of failure is "a system not performing up to its specification" [69]. This is an apt definition for a piece of hardware, or an IC chip, a microprocessor, or a UNIX command that has been unchanged for 20 years. In each of these cases, one can find a specification. But, increasingly, piece parts (components) do not have specifications. And, if the specifications do exist, they are vague, incomplete, and never really meant to provide a clear binary answer to whether a situation is failing or is working as designed. When there is a specification for a new product or service, it may only be in the form of commercial material and not a detailed technical specification.

How then does one recognize a failure? A good place to start is the customer service desk. When customers call, the problem diagnosis process usually tries to answer the question: Is it working as designed, or is something wrong? If it is not working as designed, is it because the user did something incorrectly, or is the product or service broken? In many cases, the answers are reasonably clear. However, there is an increasing trend regardless of where blame is assigned or when blame cannot be assigned at all: The desire to retain customers and develop a better product motivates fixing the situation so that it does not occur.

From this perspective, the definition of failure needs to be rethought. It is the expectation in the eyes of the customer, the satisfaction of the customer, and to a much lesser degree, the technical specification of the product or service [30].

5.3 Constant Development

There was a time when a product was built, assigned a product number, and shipped. Other than for service, the product and its functionality was encapsulated in that part number and never changed. This was the day before microcode and downloadable software patches. It is conceivable (and one could argue we are already there) that we can buy a widget and then determine what it does.

Assuming that the process of delivering the product to an end user takes zero time, the challenge becomes the process of developing the product and verifying that it works. The unintended consequence is that the development process never really has a clear start and end, which historically was forced by the product delivery mechanism. Now, one can argue that the product delivery mechanism is not what should define produce start and stop, but product and investment management.

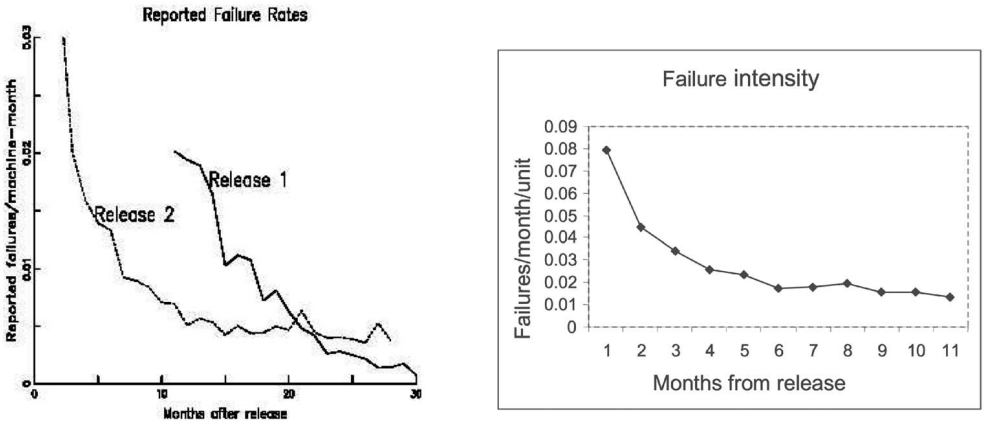
The consequences are fascinating, especially to dependability. Constant development and delivery places much lower premium on dependability. Since the change cycle is assumed to be fast, it tends to make design far more reactive. It allows a larger field test of situations that otherwise would need to be tested in-house. The processes of development, service, and dependability design get rolled into one big cycle that has few distinctions between these elements. From product manageability perspective, it can give rise to a far greater number of generations of a product coexisting in the field. This latter point makes it much harder to deliver service and maintain the installed base.

5.4 Standards

Historically, the computer industry has used abstractions as a means to handle complexity and allow independent development on opposite sides of the boundary. One of the first highly successful abstractions was the separation of instruction set design from implementation. In the early 1960s, IBM defined the IBM System 360 instruction set architecture. Over the next four decades, hardware designers created tens of different hardware implementations increasing performance a thousand fold but guaranteeing that even the software written in the 1960s could run unmodified. Meanwhile, software designers could write new applications without worrying about the moving target of hardware technology.

The contemporary embodiment of abstractions is industrial standards. Standards define functionality that service providers can implement and service consumers can use without concern about the implementation of the service. A research opportunity is to apply previously successful concepts to this new level abstraction. For example, the Ballista approach to probing the exception handling capabilities at the API (Application Programming Interface) could be extended and adapted to standards definitions.

As another example, the mnemonic reminder approach to minimizing the generation of design errors might also be extended to both the providers and consumers of standards. For example, the CHILDREN mnemonic [76] has been demonstrated to decrease the number or errors due to common software programmer omissions and commissions.



(a) (b)

Fig. 5. Field failure rates for software, measured from service data for widely distributed software: (a) IBM product circa 1994 [27]. (b) Microsoft product circa 2004 [57].

Mnemonics aid recall of issues to consider while writing code.

6 Experimental Research versus Industry Trend 2

The issues in Trend 2 that drive greater complexity bring home a central theme in the research area of dependability: breadth. What was once an area that was better defined by failures and faults from well-known sources is now dispersed across a broader set of sources and relationships. Thus, the concepts based on clear fault models with known specifications and design are muddied by less clear notions of failure and the blurred lines among the design, development, and field life of the product.

Our reflection on research that addresses these issues consequently covers a broader range of topics. It also makes it much harder to be complete or exhaustive. Hence, we discuss topics and areas that we know to have a direct relationship to experimental and empirical issues in dependability.

6.1 Software Reliability

Measurement of software reliability is complicated by a data-collection problem and partially by a definitional problem. Thus, after almost 30 years of software reliability engineering research, there are only a few studies that measure software *mean-time-between-failures (MTBF)* from field data [27], [41], [57], [117]. In sharp contrast, there are several hundred, if not a thousand studies that propose models (e.g., [39], [62]). Many models try to exploit test data to predict a reliability measure and use that as a guide to the test or development process. Since the test process can be instrumented to collect data much better than can be done in the field, it is possible to be sophisticated in terms of workload, usage profiles, and time. A goal of software reliability engineering [82] has also been to provide assessment and feedback to the development process. The difficulty with this approach is that it occurs very late in the development cycle. By the time the software is stable enough to be put through systematic testing and measurement, it is often so late in the development cycle that it is hard to impact the particular release. However, the information may be useful for follow-up releases. Furthermore, the measured or estimated reliability is just one metric, and the causal chain that leads up to it has many factors. We need research to establish the factors and their significance to the resultant reliability if this discipline is to gain greater impact on the development process. This would be a significant value to the software engineering community since it helps greater predictability and the development of best practices that are measurement-based.

As an engineering community, we need measures of MTBF for software, just like we have for hardware. Therefore, field measurements are of particular importance to provide a baseline for computing MFBF. Two studies, one from IBM [27] and another from Microsoft [57] conducted using field data from service calls, give insight on the nature of reliability and its order of magnitude. The studies are of particular interest since they use similar methodologies, report similar metrics, and both concern widely distributed products. The installed base of the Microsoft product is at least two orders of magnitude greater than for the IBM product, and the studies are about 10 years apart. Fig. 5 (graphs reproduced from [27] and [57]) shows that the order of magnitude of MTBF changes from around 50 days to 200 days over the course of 12 months after release.

Research in mechanisms to make the collection, estimation, and projection of software reliability automatic and scalable is necessary. There is work in tools for the systems management aspect of software and its maintenance. When these are coupled with measures and analytical tools for reliability, they would provide considerable value to the customers and the vendors.

6.2 Use of Formal Methods

Formal methods use mathematical techniques to represent a design and enable analysis of computer hardware and software. By creating an appropriate (at a certain abstraction level) formal representation/model of a system (hardware and or software), one can attempt to predict the

system properties. Over the years, many notations have been proposed to formally represent system components, and many tools have been developed to automate theorem proving. Despite of all these efforts (and unquestionable successes), acceptance of formal approaches in validation and verification of systems in industry is rather limited (a wonderful discussion on formal methods can be found in [92], [44], [14]). It should be, however, recognized, that in certain areas of industry, e.g., safety critical applications employed in railway control or in avionics, the formal methods are widely accepted [47], [110]. In addition, some of the major standards, e.g., IEC 61508 (international, generic standard for the development of safety critical systems) recommend use of formal methods in developing critical applications.

An important area where formal methods were very successful is model checking, applied in the design and verification of finite state systems, e.g., distributed algorithms and protocols. This is achieved by verifying whether the model, derived from a hardware or software design, satisfies a logical specification (often expressed as temporal logic formulas). Early work [32], [89] provides foundation for current generation of model checking tools such as SPIN [48].

6.3 ODC Technology

Orthogonal Defect Classification (ODC) is a technology that has evolved over the past decade, bringing measurement and sophistication into development process analysis. It is tied closely to defects, as the name suggests, but it is primarily about methods to understand and control the development process.

6.3.1 Concept

The beginnings of ODC, interestingly, go back to beginnings of the software implemented fault-injection (SWIFI) era. As we realized that fault-injection held possibilities not merely for evaluation but also for learning about the system, we looked in greater detail into fault occurrence and detection mechanisms. The ODC *Trigger* was born out the need to understand the error mechanisms for fault injection by studying real field failures [107]. As we exploited defect information for the development process, pressing problems of the development process gained visibility, which led to today's ODC.

A *defect*, defined as *a necessary change to software* [28], is the object of the final repair actions that follow the discovery of a fault. Faults, by their nature are often not observable except through inspection. Alternatively, failures and errors are the events and states that allow the fault to be revealed/detected. Dependability's primary concern are faults that escape development and go into the field, but the processes that lead up to product delivery are inundated with faults and have a direct bearing on the overall dependability. The number of defects found in the development phase usually peaks during the coding stage, although most literature points to the fact the requirements and design are the likely sources of a large number of them. Regardless, defects exist through the entire development process and field life of a product. As far as measurement is concerned, defects are an ideal source of information, not merely for their wealth of information, but also for being available abundantly across all stages of the process.

ODC requires that each defect be categorized/characterized by a prescribed attribute value set. A rigorous empirical process is employed to determine the attribute values [26]. When so classified, these attribute-value sets do not merely describe the defect but result in a specific measurement on the development process. For example, *Defect Type* and *Trigger* are two key ODS attributes. The *Defect Type* captures the meaning of the fix, expressed in a value set describing design or programming terms and maps on the *development process* via a set of associations and probabilities. The defect type attributes were initially established empirically [27]. The *Trigger* maps into the test process and expresses conditions leading to a defect to surface.

To illustrate the power of ODS approach, we discuss two applications that have gained considerable interest in the industry: *test effectiveness*, a specific analysis made possible through ODC, and *root cause analysis*, a broader and more open-ended application that is made more powerful through ODC.

6.3.2 Application—Test Effectiveness

A simple yet powerful concept that has grown due to ODC is the notion of test effectiveness. This helps tailor a test strategy depending on the nature of defects present in the process. In contrast, the classical approaches have a sequence of test stages that begin with unit test and culminate in requirements or scenario tests. While the classical approaches to test use metrics such as coverage and reliability, they suffer broadly due to the lack of insight into the nature of the specific defects present in the specific target artifact being tested.

The cross product of ODC defect type and trigger provides an effectiveness measure that helps us understand the nature of defects and the appropriate triggers that are most effective in detecting them. IBM, the prime IT vendor, used this method to enhance testing of systems for the 2000 Sydney Olympics [16]. To gain an understanding of the usage profile of applications and the nature of defects, an earlier release of the game and scoring software was studied. Fig. 6a shows the triggers that activated defects yielding in customer-found defects in the field. The triggers are subgrouped under Function, System, and Performance test, which are the most likely test phases to generate the triggers. The data illustrate that a rich spectrum of triggers are present: several (system test subgroup) triggers that were inadequately represented in the prerelease testing and several (function test subgroup) that could be eliminated with better functional test design. Fig. 6b gives us further insight through the cross product of type and trigger, illustrating the types of triggers that are most effective for specific defect types. These insights resulted in a targeted design of the test strategy, yielding a new version with far fewer escapes.

6.3.3 Application: 10X on Root Cause Analysis

Root cause analysis is at the core of process improvement. However, root cause analysis, defect prevention, and learning are among the most difficult practices to execute

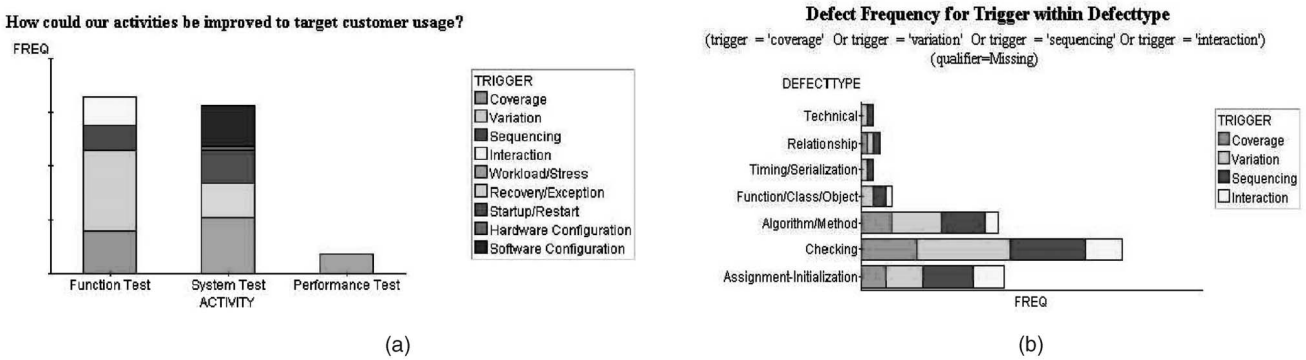


Fig. 6. (a) Trigger distribution of customer-found defects from a prior release of the software and (b) cross product of defect type and trigger of the customer-found defects.

and sustain. Thus, it is not surprising that they reappear every so often under a new banner: *quality circle*, *defect prevention*, *Level 5*, and *Six Sigma*. The significance of learning cannot be understated and any means to help institutionalize and sustain it is vital.

One of the limiting elements of the classical root cause analysis has been its cost. Typical analysis cost runs around one person-hour per defect. ODC changes this equation for root cause analysis making it far more practical and scalable. The cost of analysis is reduced from one hour per defect to around 4 minutes per defect. After 8 hours of training, students classify defects at 6 minutes per defect in their very first week. Experts, regularly clock ODC at 2 minutes per defect, in retrospective mode. At this low cost, all the defects in a process can be classified and are subject to analysis, compared to the classical root cause analysis, which is usually limited to a sample.

Classifying defects by ODC is only the first step. Actual root cause analysis is done by an ODC process analyst with skills in multidimensional analysis and statistical tools. This quantitative approach to root cause analysis has multiple side benefits:

1. Not everyone on the development team needs to be involved in the root cause analysis, or at least not all of them in great depth as is common with the classical defect prevention.
2. There is greater coverage of the defect data given lower running costs.
3. The quantitative methods allow easier comparison of one release with another.
4. When multiple actions are involved, the data make it far more tractable to prioritize and roll up actions.
5. Finally, communicating the results is far more systematic.

These benefits allow the methods to be scaled to larger projects and rolled out to organizations more readily, yielding a larger impact. Two case studies are noteworthy.

A large IBM product needed to go through a multiyear quality improvement process to reduce cost of operations. The goal was to significantly enhance the code quality and reduce maintenance costs. A couple of years of classical methods of quality improvement had yielded a 4x improvement in quality and then reaching a plateau. Over the following next few years, ODC-driven analysis and feedback yielded a ~15x improvement in quality. The original starting point was in the range of ~1,500 defects per million lines of code; after ODC it reached ~20 defects per million lines of code, resulting in code quality that rivals the best in the industry. The overall savings were ~$100 million. Since warranty costs accrue annually, the total savings over the life of product is even higher.

A Nortel implementation of ODC for root cause analysis and directing development to strategically tackle a difficult development situation has a similar story. In about five years, the overall savings exceeded $250 million when including reduced cost in warranty, critical situation handling, and critical accounts.

6.3.4 Research Avenues

ODC illustrates one of our themes, namely, the increasing abstraction level of fault models and their mapping into process space to deal with growing complexity. The concepts are not specific to software engineering, and therefore have the promise of being adaptable to other domains. Thus, this research is rich with opportunities in at least three visible directions: the data model for new domains, models for prediction, and evaluations of design processes and tools. The 5.11 release of ODC [84] software has been stable for almost 10 years. Its application ranges from systems code, microcode, and applications and has been implemented in approximately 100 projects so far. It has been extended at Carnegie Mellon to robotics, and useful inferences have been possible with a limited body of prior data [99].

6.4 Software Testing

Software testing is a vast field (which includes theory and practice) and requires its own treatment, which is far beyond the scope of this paper. There is a large volume of literature covering the theory and practice of software testing, e.g., [45] (a special section on software testing edited by R. Hamlet), [51], [37], [109], [116]. We must, however, mention that there are some fundamental issues that need focus due to the explosive complexity of Trend 2. While much of the software testing research addresses issues at a program level, be it *white box* or *black box*, the issues of complexity quickly dwarf this level of testing. The challenge of integration and multiple layers of functionality have

complicated the problem enormously. The corresponding weak specification or lack thereof makes a difficult problem worse. Thus, heuristics for test generation at higher levels of abstraction are welcome in the industry.

There have been a few notable pieces of research for issues of complexity that tend to be domain-specific. Parsimonious test generation for screens and system configuration gained tremendously through the ideas of orthogonal arrays [87] furthered by heuristics such as Telcordia's AETG [33]. State charts [46] that blend graphical representation with functional properties have had success in areas where protocols and algorithms are better specified. Model-based testing holds promise to deal with complexity, but the terrain has been difficult due to problems at the test formulation stage and in simplifying for broader use.

The issue of dealing with multiple layers of software, incomplete specification of standards, and assumed interfaces and services needs conceptual models that can break us out of the morass. Along with this, we need more appropriate notions of coverage (other than those at the code level) to match the higher levels of abstraction that we need to work with.

7 Trend 3—Global Volume

The numbers of end users has grown by orders of magnitude in the IT industry. With this comes a pleth-ora of changes in every aspect of the business. The user's tolerance for failures is lower, and the higher levels of integration are a source of new dimensions in failures. These trends often require altering long-held assumptions. In other respects, ideas that were considered impossible only a decade ago, are already in production.

7.1 Form Factor and Mobility

The smaller geometry of circuits and lower power has reached the point where devices such as hand-held PDAs rival the compute power of a desktop computer of only a few years ago. While that trend has been predicted in Moore's law for more than two decades, it is only now that the hand-carried or wearable device has the power for significant computing. Coupled with the corresponding decrease in cost, the volume of devices grows by a couple of orders of magnitude over desktop and lap top computers. No longer are computers numbered in hundreds of thousands to a million per year, but in tens to hundreds of millions, with each model having a useful operating life of a couple years. The cell phone is an excellent example of a form factor, technology, and accepted functionality that will drive more CPU sales with the power of a desktop computer than previously conceived.

This large volume of devices is not mere power in the hands of the end users. There is an infrastructure that needs to be laid down to enable the end-user device, manage the end user device, and create a channel for new features and services. All this may need to be accomplished with location transparency and, in the ideal situation, without forcing direct contact with the end user.

7.2 Lower Training Threshold

The increase in volume of users leads to lower levels of training of the average end user. While this is quite evident in the consumer segment, such as cell phone users, it is also true in other segments. Clearly, the need for specialized skill and the expectation that there will be the necessary training is always true in the high-end servers network markets. However, the desktop era demonstrated the need to build computing elements that required lower and lower levels of training for the end user thereby enabling mass markets. Along with this, there has also been the trend to place systems management and repair in the hands of technicians who do not have a degree in computer science and who can be trained rapidly. The growth of education programs in information technology that focus on solutions and capabilities and less on engineering design reflects this trend.

Dependability demands accelerate with lower training. What may have once been considered an acceptable level of difficulty in use and maintenance of systems is no longer acceptable. Thus, what are minor irritations in historic systems become faults today. What would historically be routine maintenance becomes a serious failure. Thus, the design assumptions on products themselves have dramatically changed to accommodate this trend.

7.3 Dependability in Systems Management

With more diversity of devices on the market, the task of installing, delivering, maintaining, servicing, and upgrading are all now far more challenging. Some tasks that could once be done manually cannot be done manually. Failures in these systems have their effect multiplied by the number of users they serve. For instance, installing the wrong version of an application that leads to updating databases around the world with incorrect pricing information can cause a huge disruption in accounting. Such an error can be triggered by a human error, lack of training, or a programming error in version management, or it could be a secondary problem due to the integrity of a database.

8 Experimental Research versus Industry Trend 3

Commencing with the workstation boom in the early 1980s, many of the traditional system management procedures fell to the user. Since workstation behavior was not visible to a central site, the research community developed techniques in trend analysis to monitor activity and report deviation from normal behavior. Thus, rather than attempt to reconstruct sequences of events after the fact, operational personnel would merely be notified of unusual events. Operational personnel could focus their attention on events that were likely to be meaningful. In addition, the trend analyzers provided data surrounding the event so no reconstruction would be required.

8.1 Adaptive Model of Normal Behavior

Trend analysis creates a model of normal behavior and looks for deviation between current behavior and normal. Since systems and their applications continually evolve, the model must also learn the new behavior and factor it into a

new model of normal. In its early stages of development, Harbinger [74] saw traffic grow by an order of magnitude in a matter of months on an Ethernet backbone. Harbinger would flag events that were plus or minus one or two standard deviations away from prior behavior. Operational personnel could examine these behaviors and, if they considered normal, do nothing. The Harbinger model would adapt such that any change that persisted for more than two weeks would become part of the new normal behavior. Harbinger reduced the number of events of interest by several orders of magnitude [75]. The case study on using models of normal behavior for voicemail systems in Telco shows that alarm analysis could predict failures up to a few weeks in advance [36]. Setting up the alarm measurement and analysis system decreased the mean time to repair at least by a factor of two, with a corresponding drop in unavailability.

8.2 User Interaction

In the face of the computer pervasiveness, the human-computer interaction becomes one of the factors determining the end-user's perception of system dependability. Response latency is one of the important quality of service (QoS) metrics. An early study by Miller [80] indicated that: 1) a response within 0.1 seconds is perceived by the user as an instantaneous reaction of the system, 2) a response within 1.0 second keeps the user's attention to an interactive dialog, and 3) a 10-second delay is about the limit for holding user attention to the task at hand. For longer delays, the user is distracted and may move to another task. Similar conclusions are drawn from the study on a cognitive coprocessor (a user interaction manager) [18]. More recent work conducted in HP Laboratories on the user-perceived latency of Web-based services reinforces these findings and indicates that delays of around 11 seconds represent the threshold beyond which it is difficult to keep a user's attention on the task [12]. For designers of dependable systems, these findings show the importance of fast error detection and rapid recovery in minimizing system down-time due to errors and, hence, reducing the perceived response latency.

Another, slightly different yet important aspect of the human-machine interaction is the unplanned outage due to operator error. As new devices, services, and consumer appliances flow into the market, user/operator errors can be argued to be *usability design flaws*. The problem is that often, in practice, the usability issues are not considered in the configuration of a system. As a result, the actual testing of user interfaces and, in particular, their usability occurs in the finally built system, which may be too late.

8.3 Pervasive Computing

For the past decade, computer science researchers have been defining new services and architectures for pervasive and ubiquitous computing environments [86]. Pervasive/ubiquitous computing environments encompass hundreds of embedded computers throughout the physical environment that can provide services such as displays and printing. As mobile users and devices move through the environment, services must be discovered and protocols established for communicating and combining services. Basic research in pervasive/ubiquitous computing may provide some of the mechanisms for on-the-fly monitoring and reconfiguration.

8.4 Cognitive Assistants That Learn

The DARPA PAL (Personal Assistant that Learns) program's goal is to create cognitive systems that use a variety of learning techniques to discover user preferences and, thus, to proactively anticipate user needs. By communicating through cognitive agents, the user that has discovered a technique to work around a problem could share have information with other users. Thus, rather than having a dedicated, skilled staff to help users work through problems, the entire user community could share what has been discovered. (A recent study of remote user security problems indicated an average problem resolution time of 60 hours and up to 100 hours for problems originated in the networks [105].)

8.5 Proactive Management

The number of operational parameters in a computing system has grown explosively. Companies have been founded whose sole product is to discover network configurations and set the parameter values of the various switches to optimize performance. One can envision such services reaching down to individual users and their mobile devices. Early research in artificial intelligence led to the R1 system [78] for configuring VAX high-end computers for Digital Equipment Corporation (DEC). Since there were a large number of possible configurations, R1 examined the purchase order and created a bill of materials adding in missing components that were implied by the rest of the configuration. Research in such technology could carry beyond the manufacturing phase into the operational life phase.

9 Conclusions

Our framework, defined by three elements—trends, artifacts, and processes—has allowed us to reflect on overarching industry trends and technologies that impact the industry's artifacts and/or processes. Looking back at Table 2, following our discussions, lets us reflect on where we have been and where opportunities for research lurk.

Trend 1—Shifting Error Sources and *Trend 2—Explosive Complexity* are well underway with a substantial body of research. Nevertheless, there remains a need for more research, especially on issues of complexity and security (e.g., measurement-based analysis of system security). *Trend 3—Global Volume* is upon us but is young in terms of research. We have articulated several topics that are currently visible and certainly others will reveal themselves. The pace of industry and its needs for research has grown by the sheer volume of business and domains of application. Thus, the need for this research is imminent.

Acknowledgments

The authors would like to thank the reviewers for their insightful comments and constructive suggestions.

REFERENCES

[1] A. Amendola et al., "Experimental Evaluation of Computer-Based Railway Control Systems," *Proc. Int'l. Conf. Fault-Tolerant Computing Systems (FTCS-27),* 1997.

[2] T. Anderson et al., "Software Fault Tolerance: An Evaluation," *IEEE Trans. Software Eng.,* vol. 11, no. 12, Dec. 1985.

[3] T. Anderson et al., "Protective Wrapper Development: A Case Study," *Proc. Second Int'l Conf. OTS-Based Software Systems (ICCBSS),* 2003.

[4] J. Arlat, Y. Crouzet, and J.-C. Laprie, "Fault Injection for Dependability Validation of Fault-Tolerant Computer Systems," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-19),* 1989.

[5] J. Arlat et al., "Fault Injection for Dependability Validation: A Methodology and some Applications," *IEEE Trans. Software Eng.,* vol. 16, no. 2, 1990.

[6] J. Arlat et al., "Comparison of Physical and Software-Implemented Fault Injection Techniques," *IEEE Trans. Computers,* vol. 52, no. 8, Aug. 2003.

[7] T. Aslam, I. Krsul, and E. Spafford, "Use of A Taxonomy of Security Faults," *Proc. 19th NIST-NCSC National Information Systems Security Conf.,* 1996.

[8] A. Avizienis, "Design of Fault-Tolerant Computers," *Proc. AFIPS Fall Joint Computer Conf.,* vol. 31, 1967.

[9] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Trans. Software Eng.,* vol. 11, no. 12, Dec. 1985.

[10] A. Avizienis, "Toward Systematic Design of Fault-Tolerant Systems," *Computer,* vol. 30, no. 4, 1997.

[11] A. Avizienis, J.-C. Laprie, and B. Randell, "Fundamental Concepts of Dependability," *Proc. Third Information Survivability Workshop,* 2000.

[12] N. Bhatti, A. Bouch, and A. Kuchinsky, "Integrating User-Perceived Quality into Web Server Design," *Proc. Ninth Int'l WWW Conf.,* 2000.

[13] B. Bisbey II and D. Hollingsworth, "Protection Analysis Project Final Report," Technical Report ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Inst., 1978.

[14] J. Bowen and M. Hinchey, "Seven More Myths of Formal Methods," *IEEE Software,* vol. 12, no. 4, 1995.

[15] S. Brilliant, J. Knight, and N. Leveson, "Analysis of Faults in an N-Version Software Experiment," *IEEE Trans. Software Eng.,* vol. 16, no. 2, 1990.

[16] M. Butcher, H. Munro, and T. Kratschmer, "Improving Software Testing via ODC: Three Case Studies," *IBM Systems J.,* vol. 41, no. 1, 2002.

[17] S. Butner and R. Iyer, "A Statistical Study of Reliability and System Load at SLAC," *Proc. Int'l. Symp. Fault-Tolerant Computing (FTCS-10),* 1980.

[18] S. Card, G. Robertson, and J. Mackinlay, "The Information Visualizer: An Information Workspace," *Proc. ACM CHI '91 Conf.,* 1991.

[19] J. Carreira, H. Madeira, and J.G. Silva, "Xception: A Technique for the Evaluation of Dependability in Modern Computers," *IEEE Trans. Software Eng.,* vol. 24, no. 2 1998.

[20] X. Castillo and D. Siewiorek, "A Performance-Reliability Model for Computing Systems," *Proc. Int'l. Symp. Fault-Tolerant Computing (FTCS-10),* 1980.

[21] X. Castillo and D. Siewiorek, "Workload, Performance, and Reliability of Digital Computing Systems," *Proc. Int'l. Symp. Fault-Tolerant Computing (FTCS-11),* 1981.

[22] X. Castillo, S. McConnel, and D. Siewiorek, "Derivation and Calibration of A Transient Error Reliability Model," *IEEE Trans. Computers,* vol. 31, no. 7 1982.

[23] S. Chen et al., "Modeling and Evaluating the Security Threats of Transient Errors in Firewall Software," *Int'l J. Performance Evaluation,* vol. 56, nos. 1-4, 2004.

[24] S. Chen et al., "A Data-Driven Finite State Machine Model for Analyzing Security Vulnerabilities," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '03),* 2003.

[25] R. Chillarege and N. Bowen, "Understanding Large System Failures—A Fault Injection Experiment," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-19),* 1989.

[26] R. Chillarege, W. Kao, and R. Condit, "Defect Type and its Impact on the Growth Curve," *Proc. 13th Int'l Conf. Software Eng.,* 1991.

[27] R. Chillarege et al., "Orthogonal Defect Classification—A Concept for In-Process Measurements," *IEEE Trans. Software Eng.,* vol. 18, no. 11, 1992.

[28] R. Chillarege, "ODC for Process Management, Analysis, and Control," *Proc. Fourth Int'l Conf. Software Quality,* 1994.

[29] R. Chillarege et al., "Measurement of Failure Rate in Widely Distributed Software," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-25),* 1995.

[30] R. Chillarege, "What is Software Failure?" *IEEE Trans. Reliability,* vol. 45, no. 3, 1996.

[31] R. Chillarege, "The Marriage of Business Dynamics and Software," *IEEE Software,* vol. 19, no. 6, 2002.

[32] E. Clarke and E. Emerson, "Synthesis of Synchronization Skeletons for Branching Time Temporal Logic," *Logic of Programs: Workshop,* 1981.

[33] D. Cohen et al., "The AETG System: An Approach to Testing Based on Combinatorial Design," *IEEE Trans. Software Eng.,* vol. 23, no. 7, 1997.

[34] C. Constantinescu, "Validation of the Fault/Error Handling Mechanisms of the Teraflops Supercomputer," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-28),* 1998.

[35] J. DeVale and P. Koopman, "Robust Software—No More Excuses," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '02),* 2002.

[36] L. Dorron and R. Chillarege, "Early Warning of Failures through Alarm Analysis—A Case Study in Telcom Voice Mail Systems," *Proc. Int'l Symp. Software Reliability Eng.,* 2003.

[37] J. Duran and S. Ntafos, "An Evaluation of Random Testing," *IEEE Trans. Software Eng.,* vol. 10, no. 4, 1984.

[38] F. Faccio et al., "Single Event Effects in Static and Dynamic Registers in a 0:25um CMOS Technology," *IEEE Trans. Nuclear Science,* vol. 46, no. 6, 1999.

[39] A. Goel, "Software Reliability Models: Assumptions, Limitations and Applicability," *IEEE Trans. Software Eng.,* vol. 11, no. 12, 1985.

[40] K. Goswami, R. Iyer, and L. Young, "DEPEND: A Simulation-Based Environment for System Level Dependability Analysis," *IEEE Trans. Computers,* vol. 46, no. 1, 1997.

[41] J. Gray, "A Census of Tandem System Availability between 1985 and 1990," *IEEE Trans. Reliability,* vol. 39, no. 4, 1990.

[42] W. Gu, Z. Kalbarczyk, and R. Iyer, "Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '04),* 2004.

[43] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-Ion Radiation," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-19),* 1989.

[44] A. Hall, "Seven Myths of Formal Methods," *IEEE Software,* vol. 7, no. 5, 1990.

[45] R. Hamlet, "Special Section on Software Testing," *Comm. ACM,* vol. 31, no. 6, 1988.

[46] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The STATE-MATE Approach.* McGraw-Hill, 1998.

[47] C. Hennebert and G. Guiho, "SACEM: A Fault Tolerant System for Train Speed Control," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-23),* 1993.

[48] G. Holzmann, "The Model Checker SPIN," *IEEE Trans. Software Eng.,* vol. 23, no. 5, 1997.

[49] The Honeynet Project, *Know Your Enemy: Revealing the Security Tools, Tactics, and Motives of the Blackhat Community.* Addison-Wesley, 2002.

[50] M. Howard and D. LeBlanc, *Writing Secure Code.* Microsoft Press, 2001.

[51] W. Howden, *Functional Program Testing and Analysis.* McGraw-Hill, 1987.

[52] M. Hsiao et al., "Reliability, Availability, and Serviceability of IBM Computer Systems: A Quarter Century of Progress," *IBM J. Research and Development,* vol. 25, no. 5, 1981.

[53] R. Iyer and D. Rossetti, "A Statistical Load Dependency Model for CPU Errors at SLAC," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-12),* 1982.

[54] R. Iyer and D. Rossetti, "Effect of System Workload on Operating System Reliability: A Study on the IBM 3081," *IEEE Trans. Software Eng.,* vol. 11, no. 12, Dec. 1985.

[55] R. Iyer and D. Rossetti, "A Measurement-Based Model for Workload Dependency of CPU Errors," *IEEE Trans. Computers,* vol. 35, no. 6, June 1986.

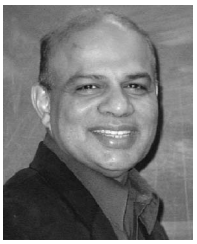
[56] R. Iyer, L. Young, and K. Iyer, "Automatic Recognition of Intermittent Failures: An Experimental Study of Field Data," *IEEE Trans. Computers,* vol. 39, no. 4, Apr. 1990.

[57] P. Jalote and B. Murphy, "Reliability Growth in Software Products," *Proc. Int'l Symp. Software Reliability Eng.,* 2004.

[58] E. Jenn et al., "Fault Injection into VHDL Models: The MEFISTO Tool," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-24),* 1994.
[59] M. Kalyanakrishnam, R.K. Iyer, and J. Patel, "Reliability of Internet Hosts: A Case Study from End User's Perspective," *Proc. Int'l Conf. Computer Comm. and Networks,* 1996.
[60] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer, "Failure Data Analysis of LAN of Windows NT Based Computers," *Proc. 18th Symp. Reliable and Distributed Systems (SRDS '99),* 1999.
[61] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-22),* 1992.
[62] K. Kanoun et al., "SoRel: A Tool for Reliability Growth Analysis and Prediction From Statistical Failure Data," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-23),* 1993.
[63] H. Kantz and C. Koza, "The ELEKTRA Railway Signaling System: Field Experience with an Actively Replicated System with Diversity," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-25),* 1995.
[64] W. Kao, R.K. Iyer, and D. Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the Unix System Behavior under Faults," *IEEE Trans. Software Eng.,* vol. 19, no. 11, Nov. 1993.
[65] P. Koopman and J. DeVale, "The Exception Handling Effectiveness of POSIX Operating Systems," *IEEE Trans. Software Eng.,* vol. 26, no. 9, 2000.
[66] J. Lala, "Fault Detection, Isolation, and Reconfiguration in FTMP: Methods and Experimen-tal Results," *Proc. Fifth AIAA/IEEE Digital Avionics Systems Conf. (DASC),* 1983.
[67] C. Landwehr et al., "A Taxonomy of Computer Program Security Flaws, with Examples," *ACM Computing Surveys,* vol. 26, no. 3, 1994.
[68] J.-C. Laprie et al., "Definition and Analysis of Hardware-and-Software Fault-Tolerant Ar-chitectures," *Computer,* vol. 23, no. 7, July 1990.
[69] J.-C. Laprie et al., "Dependability: Basic Concepts and Terminology," *Dependable Computing and Fault-Tolerant Systems,* 1992.
[70] I. Lee and R. Iyer, "Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating Systems," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-23),* 1993.
[71] T. Lin and D. Siewiorek, "Architectural Issues for On-Line Diagnostics in A Distributed Environment," *Proc. Int'l. Conf. Computer Design,* 1986.
[72] T. Lin and D. Siewiorek, "Error Log Analysis Statistical Modeling and Heuristic Trend Analysis," *IEEE Trans. Reliability,* vol. 39, no. 4, 1990.
[73] U. Lindqvist and E. Jonsson, "How to Systematically Classify Computer Security Intrusions," *Proc. Symp. Security and Privacy,* 1997.
[74] R. Maxion, "Distributed Diagnostic Performance Reporting and Analysis," *Proc. Int'l Conf. Computer Design,* 1986.
[75] R. Maxion and K. Tan, "Anomaly Detection in Embedded Systems," *IEEE Trans. Computers,* vol. 51, no. 2, Feb. 2002.
[76] R. Maxion and R. Olszewski, "Eliminating Exception Handling Errors with Dependability Cases: A Comparative, Empirical Study," *IEEE Trans. Software Eng.,* vol. 26, no. 9, Sept. 2000.
[77] S. McConnel, D. Siewiorek, and M. Tsao, "The Measurement and Analysis of Transient Errors in Digital Compute Systems," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-9),* 1979.
[78] J. McDermott, "R1: A Rule-Based Configurer of Computer Systems," *Artificial Intelligence,* vol. 19, no. 2, 1982.
[79] A. Merenda and E. Merenda, "Recovery/Serviceability/System Test Improvements for the IBM ES/9000 520 Based Models," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-22),* 1992.
[80] R. Miller, "Response Time in Man-Computer Conversational Transactions," *AFIPS Fall Joint Computer Conf.,* vol. 33, 1968.
[81] B. Murphy and B. Levidow, "Windows 2000 Dependability," Microsoft Research Technical Report MSR-TR-2000-56, 2000.
[82] J. Musa, *Software Reliability Engineering.* McGraw Hill, 1998.
[83] E. Normand, "Single Event Upset at Ground Level," *IEEE Trans. Nuclear Science,* vol. 43, 1996.
[84] ODC ODC-511, Web Resources, 2004, www.chillarege.com; www.research.ibm.com/softeng.
[85] D. Patterson et al., "Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies," CS Technical Report UCB/CSD-02-1175, Univ. of California at Berkeley, 2002.
[86] *IEEE Pervasive Computing,* special issue on integrated environments, vol. 1, no. 2, 2002.
[87] M. Phadke, *Quality Engineering Using Robust Design.* Prentice Hall, 1989.
[88] D. Powell, "Failure Mode Assumptions and Assumption Coverage," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-22),* 1992.
[89] J. Queille and J. Sifakis, "Specification and Verification of Concurrent Systems in Cesar," *Proc. Fifth Symp. Programming,* 1981.
[90] M. Rodriguez et al., "MAFALDA: Microkernel Assessment by Fault Injection and Design Aid," *Proc. Third European Dependable Computing Conf. (EDCC-3),* 1999.
[91] M. Rodriguez, J.-C. Fabre, and J. Arlat, "Wrapping Real-Time Systems from Temporal Logic Specifications," *Proc. Fourth European Dependable Computing Conf. (EDCC-4),* 2002.
[92] J. Rushby, "Formal Methods and the Certification of Critical Systems," Technical Report CSL-93-7, CS Laboratory SRI, 1993.
[93] J. Samson, W. Moreno, and F. Falquez, "A Technique for Automated Validation of Fault Tolerant Designs Using Laser Fault Injection," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-28),* 1998.
[94] Z. Segall et al., "FIAT—Fault Injection Based Automated Testing Environment," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-18),* 1988.
[95] F. Sellers, M. Hsiao, and L. Bearnson, *Error Detecting Logic for Digital Computers.* McGraw-Hill, 1968.
[96] Y. Shi, "A Portable, Self Hosting System Dependability Measurement and Prediction Module," technical report, Electrical and Computer Eng. Dept., Carnegie Mellon Univ., 1999.
[97] P. Shivakumar et al., "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '02),* 2002.
[98] D. Siewiorek et al., "A Case Study of C. mmp, Cm*, and C. vmp," *Proc. IEEE,* vol. 66, no. 10, 1978.
[99] J. Silberman, "Robot Orthogonal Defect Classification Towards an In-Process Measurement System for Mobile Robot Development," technical report, Robotics Inst. Carnegie Mellon Univ., 1998.
[100] C. Simache, M. Kaâniche, and A. Saidane, "Event Log Based Dependability Analysis of Windows NT and 2K Systems," *Pacific Rim Int'l Symp. Dependable Computing (PRDC '02),* 2002.
[101] L. Spainhower and T.A. Gregg, "G4: A Fault-Tolerant CMOS Mainframe," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-28),* 1998.
[102] L. Spainhower et al., "IBM's ES/9000 Model 982's Fault-Tolerant Design for Consolidation," *IEEE Micro,* vol. 14, no. 1, 1994.
[103] L. Spainhower and T. Gregg, "IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective," *IBM J. Research and Development,* vol. 43, nos. 5/6, 1999.
[104] L. Spitzner, *Honeypots: Tracking Hackers.* Addison-Wesley, 2003.
[105] A. Steinfeld et al., "An Examination of Remote Access Help Desk Cases," Technical Report CMU-CS-03-190, CMU-HCII-03-100, School of Computer Science, Carnegie Mellon Univ., 2003.
[106] D. Stott et al., "Dependability Assessment in Distributed Systems with Lightweight Fault Injectors in NFTAPE," *Proc. Fourth Int'l Computer Performance and Dependability Symp.,* 2000.
[107] M. Sullivan and R. Chillarege, "Software Defects and Their Impact on System Availability—A Study of Field Failures in Operating Systems," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-21),* 1991.
[108] D. Tang and R. Iyer, "Analysis and Modeling of Correlated Failures in Multicomputer Systems," *IEEE Trans. Computers,* vol. 41, no. 5, May 1992.
[109] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet, "Software Statistical Testing," *Predictably Dependable Computing Systems,* 1995.
[110] A. Tiwari, J. Rushby, and N. Shankar, "Invisible Formal Methods for Embedded Control Systems," *Proc. IEEE,* vol. 91, no. 1, 2003.
[111] P. Traverse, "Dependability of Digital Computers on Board Airplanes," *Proc. First Int'l Working Conf. Dependable Computing for Critical Applications,* 1989.
[112] T. Tsai et al., "Stress-Based and Path-Based Fault Injection," *IEEE Trans. Computers,* vol. 48, no. 11, 1999.
[113] M. Tsao and D. Siewiorek, "Trend Analysis on System Error Files," *Proc. Int'l Symp. Fault Tolerant Computing (FTCS-13),* 1983.
[114] T. Vardanega et al., "On the Development of Fault-Tolerant On-Board Control Software and its Evaluation by Fault Injection," *Proc. Int'l. Symp. Fault-Tolerant Computing (FTCS-25),* 1995.

[115] K. Wagner and E.J. McCluskey, "Effect of Supply Voltage on Circuit Propagation Delay and Test Application," *Proc. Int'l Conf. Computer-Aided Design,* 1985.
[116] E. Weyuker and T. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," *IEEE Trans. Software Eng.,* vol. 6, no. 3, 1980.
[117] A. Wood, "Softare Reliability from the Customer View," *Computer,* vol. 36, no. 8, 2003.
[118] J. Xu, Z. Kalbarczyk, and R. Iyer, "Networked Windows NT System Filed Failure Data Analysis," *Proc. Pacific Rim Int'l Symp. Dependable Computing (PRDC '99),* 1999.
[119] J. Xu et al., "An Experimental Study of Security Vulnerabilities Caused by Errors," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '01),* 2001.
[120] C. Yount and D. Siewiorek, "The Automatic Generation of Instruction-Level Error Mani-festations of Hardware Faults," *IEEE Trans. Computers,* vol. 45, no. 8, Aug. 1996.
[121] J. Ziegler et al., "IBM's Experiments in Soft Fails in Computers," *IBM J. Research and Development,* vol. 40, no. 1, 1996.
[122] http://www.dependability.org/wg10.4/SIGDeB, 2004.
[123] http://www.securityfocus.com, 2004.
[124] http://www.cert.org, 2004.
[125] http://securitytracker.com/learn/statistics.html, 2004.
[126] http://www.honeypots.net/honeypots/links/, 2004.
[127] http://www.honeynet.org, 2004.

Daniel P. Siewiorek received the BS degree in electrical engineering from the University of Michigan and the MS and PhD degrees, both in electrical engineering, from Stanford University. He is a Buhl University professor of computer science and electrical and computer engineering at Carnegie Mellon University, is currently the director of the Human Computer Interaction Institute. He helped to produce the Cm* multiprocessor system and contributed to the dependability design of 24 commercial computer systems. He has published more than 400 technical papers and eight text books. He was elected an IEEE fellow in 1981 for contributions to the design of modular computer systems; in 1988 received the Eckert-Mauchly Award for his contributions to computer architecture, was elected as a member of the 1994 Inaugural Class of ACM fellows and elected to the 2000 class of the National Academy of Engineering. He is a member of the ACM, Tau Beta Pi, Eta Kappa Nu, Sigma Xi, and the IEEE Computer Society.



Ram Chillarege received the PhD degree from the University of Illinois, Urbana-Champaign in computer engineering, the ME and BE degrees from the Indian Institute of Science, Bangalore, and the BSc degree from the University of Mysore. He is a management consultant in software engineering optimization. He had been the Executive Vice President of Software and Technology at Opus360, and prior to that was with IBM Research for 14 years, where he founded and headed the Center for Software Engineering. He received the IEEE Technical Achievement for inventing Orthogonal Defect Classification (ODC) and an expansive body of work, which ushers new methods to the business of managing software. In the mid 1990s, he led the effort, which culminated in IBM establishing a corporate-wide software testing practice initiative. He has authored around 50 peer-reviewed articles and is an active speaker at conferences. He serves on the IEEE steering committees of Software Reliability and Dependability symposiums, several conference committees, and the US National Science Foundation panels. He is a fellow of the IEEE.

Zbigniew T. Kalbarczyk recieved the PhD degree in computer science from the Technical University of Sofia, Bulgaria. He is currently a principal research scientist at the Center for Reliable and High-Performance Computing in the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign. After receiving his doctorate, he worked as an assistant professor in the Laboratory for Dependable Computing at Chalmers University of Technology in Gothenburg, Sweden. His research interests are in the area of reliable and secure networked systems. Currently, he is a lead researcher on the project to explore and develop high availability and security infrastructure capable of managing redundant resources across interconnected nodes, to foil security threats, detect errors in applications and the infrastructure components, and recover from failures. His research involves also development of automated techniques for validation and benchmarking of dependable and secure computing systems. He has published more than 70 technical papers. He served as a program cochair of Performance and Dependability Symposium (PDS), a track of Conference on Dependable Systems and Networks (DSN '02) and is regularly invited to work on the program committees of major conferences on design of fault-tolerant systems. He is a member of the IEEE and the IEEE Computer Society.