1

Dependability Concepts and Taxonomy

1.1 Introduction

Every single failure in any computing device is a potential cause for concern. Reliable computing and fault tolerance, or, to use a more current term, *dependable computing*, is a longstanding area of research and practical implementation. This broad area of study started in the mid-fifties with John von Neumann's construction of reliable systems from unreliable systems or components. Over the years, significant advancements and deployments have been made in commercial telecommunications, defense, and business applications that address a wide range of potential failures. Today, an explosion in the complexity of systems, applications, and operating systems has resulted in ever-expanding failure sources. That, combined with explosive growth in computing as an enterprise in all areas of human endeavor, has brought forth new challenges and opportunities in designing dependable systems. Further, early detection, rapid concurrent/online diagnosis, and efficient and complete recovery are key to the design of systems that continue to operate in the event of errors. They must be complemented by ongoing analysis and monitoring of failures, supported by strong statistical models. In dependability, an understanding of real failures is critical in the design, implementation, deployment, and validation of reliability techniques. Design and validation must go hand in hand in developing new systems. While dependability techniques protect systems against known faults, their greatest efficacy comes from their ability to safeguard against unanticipated failures due to accidental errors or malicious attacks.

This chapter sets the theme of the book by first placing classic work on dependability techniques in perspective and relating their importance for current computing systems. That assessment is followed by a description of the complexity of systems built using present-day hardware designs, architectures, and software

Dependable Computing: Design and Assessment, First Edition. Ravishankar K. Iyer, Zbigniew T. Kalbarczyk, and Nithin M. Nakka.

Companion website: www.wiley.com/go/iyer/dependablecomputing1

technologies that pose compelling challenges in providing continuous availability against a vast array of potential failures. Examples are provided of the developmental (or changing) trends in these areas that motivate the need for a newer perspective on dependability. The purpose of this chapter is to bring forth the recent challenges and opportunities in the reliability domain. (Possible solutions and techniques for fault tolerance and security will be explained as the book unfolds in the remaining chapters.) The discussion concludes with an introduction of dependability concepts, definitions, a taxonomy of failures, and a sample set of measurements from real systems in preparation for the next chapter's description of basic techniques.

The entire book follows the theme set by this chapter in introducing fundamentals of techniques with examples of prior deployment of the techniques in systems currently in use, with the goal of educating the reader on the applicability of these techniques, and any modifications or adaptations they need for use in modern and upcoming systems.

1.2 Placing Classical Dependability Techniques in Perspective

The earliest diagnostic techniques were developed for testing and failure recovery in the ILLIAC machine at the University of Illinois [1, 2] in the 1950s. When ILLIAC I (1950) and ILLIAC II (1961) were built at Illinois, fault diagnosis consisted of a battery of programs that exercised different sections of the machine. Typically, the test programs compared answers computed in two different ways, or stressed what was suspected to be a vulnerable part. In the ILLIAC II, the arithmetic and control units were designed to operate asynchronously, using a double handshake for each control signal and its acknowledgment. That protocol simplified the fault diagnosis, as it was used as an automatic fault detection mechanism. Most faults caused the control to wait for the next step in the asynchronous handshake protocol; that next step was identified using indicator lights for the flip-flops.

Spaceborne computing systems were one of the earliest avenues for dependability design. Early work on dependability in space-mission systems was performed on the JPL-STAR (Jet Propulsion Laboratory Self-Testing and Repair) computer (1971) [3] and on Voyager [4], leading to work on the Boeing 777 [5]. Although the craft carrying the JPL-STAR computer never went into space, its development resulted in the design and implementation of a range of techniques that are considered standard today. The Voyager computer (launched in 1977) used block redundancy (a form of a standby redundancy whereby redundancy is provided at the subsystem level, e.g., at the altitude control subsystem, rather than internally in each subsystem) for fault tolerance. Heartbeat-based hardware- and

software-implemented techniques were used for error detection. For example, an error would be detected in the hardware if a command for the primary (in the dual-redundant configuration) arrived before the current command had been completely processed, and in software error detection, an error would be detected when the output unit in the primary remained unavailable for more than 14 seconds. Further developments in dependability in aviation were used in the design of the Boeing 777 fly-by-wire system, which used triple modular redundancy for all hardware resources, including the computing system, airplane electrical power, hydraulic power, and communication path.

The basic techniques established for hardware redundancy and software-based fault and failure management, exceptions, and their handling in software, and the use of error codes in memory systems, transmission, and disk systems have been the mainstay of practical and commercial systems such as the AT&T No. 5 ESS [6], IBM S/360, and IBM S/370 [7]. These systems included a combination of hardware and software techniques and diagnostics that significantly advanced the theory and practice of dependable computing. The methods have since been augmented with computational algorithms and protocols to achieve consistency and reliable operation in distributed systems [8].

While parity, ECC (error correcting codes) and redundant array of independent disks have been widely used for commodity systems, the use of massive redundancy in hardware and software has led to high overheads in performance costs, hardware components, and software development costs. For example, the IBM MVS operating system devotes 50% of its software code base to fault management [9], while the IBM G5 processor dedicates 35% of its processor silicon area to fault detection and tolerance hardware [10]. In addition to those overheads, the validation of such systems has become increasingly complex and difficult. Thus, the use of the techniques discussed above to build "one-size-fits-all" architectures has become reserved for high-end, high-cost systems such as those used in military, telecommunication, and financial applications. Until recently, those application domains depended on traditional techniques in which redundancy in the hardware, combined with hooks into the operating system, together supported some level of software redundancy. On the other hand, until recently, failures in commodity environments did not have such a big cost impact and hence were either not addressed or at best marginally addressed.

With the explosion of computing devices, and in particular a variety of mobile/handheld devices in a wide variety of applications, computing has become a social enterprise. Massive computing data centers are distributed geographically, logically, and physically, servicing networked entities from telecom to Internet service providers to banks (i.e., high-dependability domains). On the one hand, the likes of Amazon and Google have increasingly adopted and invested in high-performance computing systems. On the other hand, ubiquitous computing,

present in everyday appliances such as washing machines and microwaves, vehicles such as automobiles and airplanes, and applications such as e-commerce and health monitoring, has dramatically changed the impact of computing system failures on the world's social and economic machinery. With computing now a common enterprise, such outages can no longer be ignored or brushed aside with a marginal or cursory solution. Dependability requirements for these systems are nearly as high as those for the legacy systems that extensively used redundancy throughout the system. However, the cost margin for high availability is typically small, precluding the use of traditional techniques for commodity systems. New, low-cost techniques that are tailored to the specific needs of the application are required for the emerging domains. On the other end of the spectrum from embedded, ubiquitous computing are new large-scale, high-performance computing systems (i.e., supercomputers) for which dependability (or the ability to compute through failures) is paramount for providing sustained performance at scale. Such systems pose another important challenge with respect to dependability. The domain-specific requirements of the varied systems discussed thus far, failures during recovery in any system significantly change the dependability dynamics of the system [6, 11]. However, this aspect has not been adequately considered in either the design or the assessment of computing systems.

1.3 Taxonomy of Dependable Computing

In this section, we introduce dependability concepts and definitions as preparation for the descriptions of dependability techniques that follow in the rest of the book.

Avižienis and Laprie [12] described a taxonomy for dependable systems and certain dependability measures. A subsequent publication by Avižienis et al. [13] described a taxonomy that also included secure computing, as well as methodologies for evaluating the availability of a system with greater rigor. It introduced dependability as a global concept that subsumes attributes such as reliability, availability, safety, integrity, and maintainability. The consideration of security brings in concerns for confidentiality, in addition to availability and integrity. Although there have been attempts at quantification, e.g., [14], a consensus has not yet emerged on how to measure dependability and security.

Figure 1.1 shows a refined dependability and security tree as described by Avižienis et al. [13].

Since this book is primarily focused on dependability design and assessment, this section will present a taxonomy related to dependability that draws on the work mentioned above. We direct the reader to the referenced papers for more in-depth understanding.

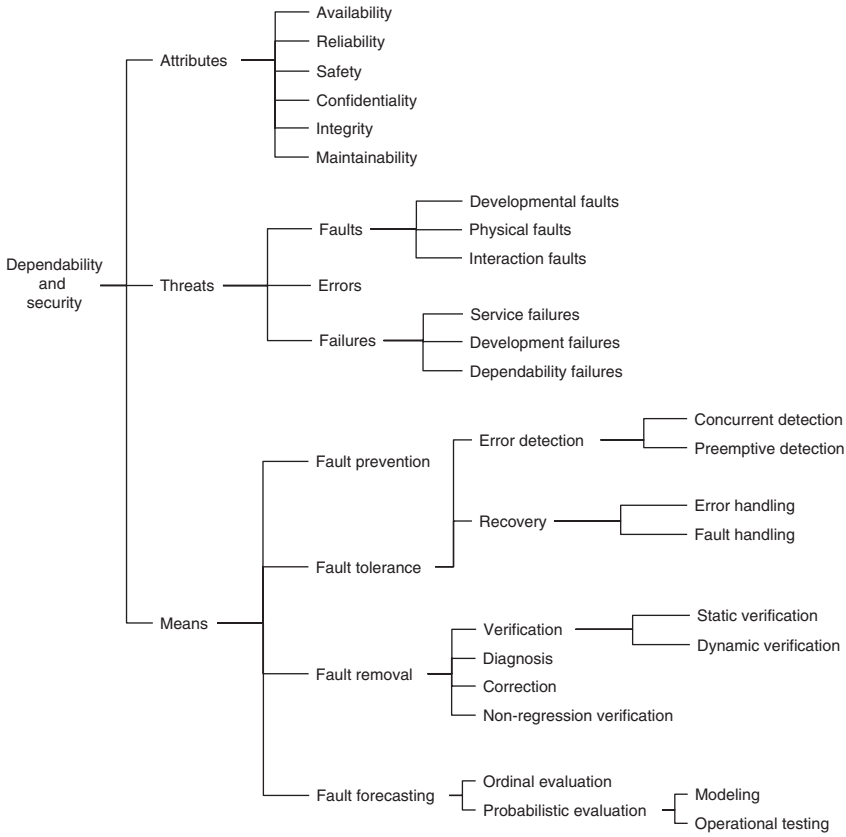

Figure 1.1 A refined dependability and security tree. *Source*: Avižienis et al. [13]. © IEEE.

1.3.1 Faults, Errors, and Failures

Within a computing system, it is useful to distinguish among faults, errors, and failures as follows:

- A *failure* is a deviation of the delivered service from compliance with the specification, i.e., a failure is a departure from the service required of a system and is a property of the external behavior of a system.
- An *error* is part of the system state that has been damaged by the fault and, if uncorrected, can lead to a failure. Here, fault tolerance is based on detection of latent errors before they become active, followed by replacement of the erroneous part of the state with an error-free state.
- A *fault* is the adjudged or hypothesized cause of an error.

Example: A programmer's mistake is a fault in the written software. When the system executes the erroneous instructions with certain data values, the fault becomes active and produces an error. When the erroneous data affect the delivered service, a failure occurs.

The ultimate result of a failure is loss of system reliability and availability. The sections that follow discuss faults, errors, and failures and their relationships in more detail and begin to consider approaches to meeting reliability requirements in spite of them.

1.4 Fault Classes

One way to classify faults, based on Siewiorek and Swarz [6], is based on their temporal persistence:

- *Permanent faults*, sometimes called *hard faults*, are continuous and stable. They include random failures, natural radiation, hardware and software design errors, system upgrades, and changes to requirements.
- *Intermittent faults* are present only occasionally and are due to instability in the hardware or software. They include natural failures and faults triggered by transient power failures. They are the most difficult faults to diagnose; they may appear transient and may be hard to recreate.
- *Transient faults*, sometimes called *soft faults*, result from temporary environmental conditions. Transient faults include power failures, switching failures, natural radiation, single upsets, multiple upsets, and software faults. They originate in hardware or are due to software bugs such as synchronization and timing errors. Ninety percent of field errors are due to transient faults [15].

Faults may also be classified by their origin, again following [6]:

- *Physical faults* stem from physical phenomena internal to the system, such as shorts, or from external changes, such as vibration.
- *Human-made faults* include design and procedural faults as well as human-machine interaction faults that occur during maintenance or operation.

1.5 The Fault Cycle and Dependability Measures

Four important metrics measure system dependability: reliability, maintainability, availability, and safety. They are defined in terms of the concepts of *mean time to failure* (MTTF) or *mean time to error* (MTTE), *mean time between failures* (MTBF), *mean time to catastrophic failure* (MTTCF), *fault latency*, *error latency*,

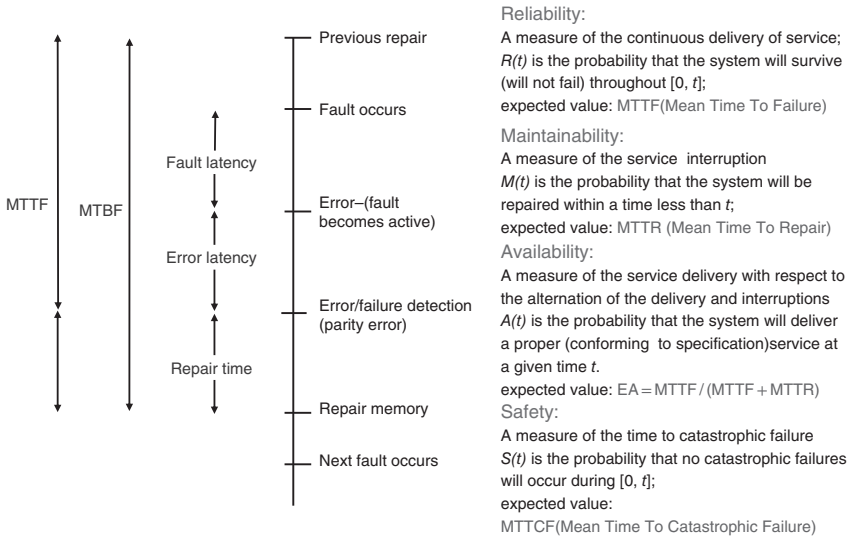

Figure 1.2 Dependability measures in relation to the fault cycle.

and repair time; all are understood with reference to the fault cycle, as shown in Figure 1.2.

Note that increased error latency increases the likelihood that multiple errors will occur, placing a greater burden on recovery mechanisms. It also increases the likelihood of error propagation, which is why error containment boundaries are important.

Figure 1.2 also defines basic dependability metrics: (i) *reliability* (a measure of the continuous delivery of services), (ii) *maintainability* (a measure of the service interruption), (iii) *availability* (a measure of service delivery with respect to the alternation of the delivery and interruptions), and (iv) *safety* (a measure of the time to catastrophic failure).

Note that in theory, high availability can be achieved with a low MTTF, as long as the MTTR is very low. In practice, however, those two quantities are closely related: they tend to increase together. In other words, it is very unlikely that a system that crashes frequently will offer a very fast and robust recovery service.

1.6 Fault and Error Classification

This section provides a brief, somewhat generic discussion of broad classes of faults/errors. The purpose is not to be comprehensive, but to articulate major differences between hardware and software faults/errors.

1.6.1 Hardware Faults

Hardware fault modeling is better understood than software fault modeling. Hardware fault models include four types: the gate level or stuck-at model, the module-level model, the functional-level model, and the system-level model.

The *stuck-at* model assumes the following:

1) The lines in the gate-level circuit are stuck at 0 or 1.
2) Faults are located at inputs and outputs of gates.
3) The basic functionality of gates remains unchanged.

Examples: Physical failures in MOS circuits; a faulty contact; a transistor stuck open or closed; open metal lines; or shorts between adjacent metal lines.

The *module-level* model applies to hardware components that are more complex than simple logic gates, such as decoders and programmable logic arrays. *Examples:* In a decoder, an incorrect line is activated instead of the desired line, or an incorrect line is activated in addition to the desired line.

The *functional-level* model applies to memories, ALUs (arithmetic logic units), and network switches. *Examples:* Memory in which one or more cells are stuck at 0 or 1, one or more cells fail to undergo 0–1 or 1–0 transition, two or more cells are coupled, a 1–0 transition in one cell changes contents in another cell, more than one cell is accessed during a READ or WRITE, or a wrong cell is accessed during a READ or WRITE.

The *system-level* model applies to higher-level system components, e.g., several CPUs, or a memory, network, or disk subsystem. *Example:* In a parallel processor topology, a processor or a communication link between processors is faulty.

1.6.2 Software Faults and Errors

Software faults are often referred to as software *bugs*. Computer pioneer Grace Hopper described how that term was coined after a malfunction in an early electromechanical computer was caused by a "moth trapped between points at Relay #70, Panel F, of the Mark II Aiken Relay Calculator while it was being tested at Harvard University, 9 September 1945. . .. The operators affixed the moth to the computer log, with the entry: First actual case of bug being found [. . .]". They thus coined the term *bug* and joked that they had "debugged" the machine – thus also introducing the term *debugging* now used to describe removal of software faults from a computer program [16].

Most software bugs are mistakes made by people during the design or coding of a program, and a few are caused by compilers that produce incorrect code. Software bugs have been classified as permanent and transient; Gray [17], for example, classified software faults into "Bohrbugs" and "Heisenbugs." *Bohrbugs*,

whose name was inspired by the Bohr model of the atom, are permanent design faults. By nature they are very nearly deterministic. During testing and debugging (i.e., the early deployment phase), Bohrbugs can be easily identified and weeded out. On the other hand, *Heisenbugs* (whose name was inspired by the Heisenberg uncertainty principle in physics) are intermittent and, as such, belong to the class of temporary internal faults. Heisenbugs are, in essence, permanent faults with activation conditions that occur rarely and are not easy to reproduce. Such faults give rise to transient failures, and if the software is restarted, the failures will not necessarily recur. Heisenbugs are likely to circumvent testing, as the bug-catcher may perturb the situation enough to make the Heisenbug disappear [17].

The majority of studies on failure data have shown that a significant proportion of software failures are transient [15, 17] and are caused by phenomena such as timing, overloads, and exception errors. Studies of failure data from Tandem's fault-tolerant computer system reported that 70% of its failures were transient, triggered by faults such as race conditions and timing problems [11].

1.6.2.1 The GUARDIAN90 Operating System

Lee and Iyer [11] presented an analysis of software failure data on the Tandem GUARDIAN90 operating system. The analysis examined a collection of memory dumps of field reports on software failures, and resulted in the classification scheme presented in Table 1.1. A total of six classes of software errors were distinguished.

1.6.2.2 IBM-MVS (zOS) and IBM Database Management Systems

Table 1.2 presents a classification of error types obtained from failure reports from the IBM-MVS operating systems [18] and two large IBM database management systems, DB2 and IMS [19]. A total of 12 classes of faults were distinguished in these studies. It was observed that the overall error-type distributions for the three

Table 1.1 Error classes for Tandem GUARDIAN90.

Incorrect computation	Arithmetic overflow or use of incorrect arithmetic function
Data fault	Use of an incorrect constant or variable
Data definition fault	Fault in declaring data or in defining data structure
Missing operation	Omission of a few lines of source code
Side effect of code update	Not all dependencies between software modules were considered when software was being updated
Unexpected situation	Routines to handle rare but legitimate operational scenarios not provided

Table 1.2 Error classes for IBM-MVS and IBM database management systems.

Allocation management	Module uses a memory region after deallocating it.
Copying overrun	Program copies data past end of buffer.
Pointer management	Variable containing address of data was corrupted.
Wrong algorithm	Program works but uses wrong algorithm.
Uninitialized variable	Variable used before it is initialized.
Undefined state	System goes into state designers did not anticipate.
Data error	Program produces or reads wrong data.
Statement logic	Statements executed in the wrong order or are omitted.
Interface error	Module's interface is defined or used incorrectly.
Memory leak	Program does not deallocate memory it has allocated.
Synchronization	Error occurs in locking or synchronizing code.

products (IBM-MVS, DB2, and IMS) differ significantly. However, some aspects of the error type distributions are invariant, for instance, the dominating error type, *undefined state*.

An important contribution to the classification and understanding of software faults/errors was the development of Orthogonal Defect Classification (ODC) [20]. ODC was proposed by R. Chillarege at IBM T.J. Watson to bring measurement into the software development process. By measuring and analyzing software defects, ODC provides methods for understanding and controlling the design and test process.

ODC's roots go back to the early stages of software-implemented fault injection (SWIFI). The concept of *ODC Trigger* reflects the need to understand the error mechanisms for fault injection by studying real field failures [18]. A *defect*, defined as a necessary change to software that must be made for the software to be usable [21], is the target of the final repair actions that follow the discovery of a fault. Since defects can exist through the entire development process and field life of a product, they are an excellent source of information that can be used to understand and characterize all stages of the software development 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 that describe the defect and produce a specific measurement of the development process [22]. For example, *Defect Type* and *Trigger* are two key ODC attributes. The *Defect Type* captures the meaning of the fix, expressed in a value set that describes design or programming terms and maps to the development process via a set of associations and probabilities. The defect type attributes were initially established empirically [20]. The *Trigger* maps into the test process and expresses conditions that lead to the surfacing of a defect.

The classes of software faults identified using ODC were used as a basis for emulation of software faults using fault injection. Based on the observation that a large percentage of faults can be characterized in a precise way, a set of emulation operators was established to mimic software faults. A software fault injection technique (G-SWFIT) based on emulation operators derived from field study was proposed [23].

1.7 Mean Time Between Failures

We also know that mean time between failures (MTBF) can vary greatly between systems designed for fault tolerance and other systems. For example, consider the data in Table 1.3.

The Los Alamos National Laboratory collected and published data regarding the failure and usage of 22 of their supercomputing clusters [24]. The root causes of the failures were classified into the following categories: (i) Facilities, (ii) Hardware, (iii) Human Error, (iv) Network, (v) Undetermined, and (vi) Software. The data were previously analyzed by Schroeder and Gibson [25] from CMU to study the statistics of the data in terms of the root causes of the failures, mean time between failures, and the mean time to repair.

Table 1.3 provides the details of the systems. In the "Production Time" column, the range of times specifies the node installation-to-decommissioning times. A value of "N/A" indicates that the nodes were installed before the observation period began. For those machines, the beginning of the observation period (January 1996) was used as the starting date for calculating the production time. For some machines, multiple ranges of production times were provided because the systems were upgraded during the observation period; each production time range corresponds to a set of nodes within the system. For the sake of simplicity, for all such machines, we consider the production time to be the longest range among the specified ranges.

The failure data for a single system include the following fields:

System: The number of the system under study (as specified by the Los Alamos National Laboratory).
Machine type: The type of the system (smp, cluster, numa).
Number of nodes: The number of nodes in the system.
Number of processors per node: Notably, some systems are heterogeneous in that they contain nodes with different numbers of processors per node.
Number of processors total: Total number of processors in the system $= \sum_{i}^{k} n_i \times p_i$,

where, for a system, k is the number of node types in the system, n_i is the number of nodes of type i, and p_i is the number of processors in a node of type i.

Table 1.3 MTTF, MTTR, and availability data for various real-world systems.

System	MTTF (h)	MTTR (h)	Availability
Tandem GUARDIAN98	98	N/A	N/A
Tandem NonStop UNIX	480 to 2040	N/A	N/A
Network of 69 SunOS Workstations	96.44	0.72	99.25%
LAN of NT machines (using system logs from 68 mail servers)	284	1.97	99.31%
Blue Waters Petascale system	154.04	5.16	96.76%
Amazon EC2	99.95	0.05	99.95%

Type	Cat	System	# of procs	# of nodes	# of procs/node	Network topology	Production time			
A	Smp	7	8	1	8	N/A	N/A-12/99	34 315.6	4.43	99.987%
B	Smp	24	32	1	32	N/A	N/A-12/03	69 377.7	6.26	99.991%
C	Smp	22	4	1	4	N/A	N/A-04/03	63 522.3	5.73	99.991%
D	Cluster	8	328	164	2	GB Ethernet Tree	04/01-11/05	34 300.8	34.30	99.90%
E	Cluster	21	512	128	4	Dual Rail Fat Tree	09/01-01/02	2 925.6	2.40	99.918%
F	Cluster	14	256	128	2	Single Rail Fat Tree	09/03-11/05	19 629.5	98.52	99.500%
G	Cluster	2	6152	49	128/80[a]	Multi Rail Fat Tree	01/97-11/05	77 438.0	2.37	99.997%
H	Numa	15	256	1	256	N/A	11/04-11/05	9 477.7	2.31	99.976%

[a]Depending on the function of the processor, if it is used for front-end graphics it has 80 procs per node. If it is used for back-end computing/graphics, it has 128 procs per node.

Production time: The time between the installation and decommission of the system. If the system was in production at the end of the observation period, we consider that to be the end time.
Mean time between failures (MTBF) (in hours, h): The average time between failures of the system, calculated as the ratio of the production time and the number of failures.
Mean time to repair (MTTR) (in hours, h): The average time for which the system was unavailable due to repair.
Availability: Calculated as $\left(\frac{(\text{MTBF} - \text{MTTR})}{\text{MTBF}}\right)$.

1.8 User-perceived System Dependability

Some studies on failure data analysis have attempted to measure dependability as it is perceived by customers/users. From the user's perspective, dependability characterizes the ability of a machine or system to provide service, not just to survive.

In 2000, Chandra and Chen [26] evaluated the hypothesis that generic recovery techniques such as process pairs can survive most application faults without using application-specific information. They examined three large, open-source applications examined. Information from the bug reports and source code were used to classify faults related to the faults' dependence on the operating environment. A quite large percentage of the faults (72–87%) were found to be independent of the operating environment, and recovery from the failures caused by these faults required the use of application-specific knowledge. Of the remaining faults, half depended on a condition in the operating environment that was likely to persist on retry, and the failures caused by these faults were also likely to require application-specific recovery. Only 5–14% of the faults were triggered by transient conditions (e.g., timing and synchronization) that inherently self-correct during recovery. Results from this study showed that classical application-generic recovery techniques (e.g., process pairs) would be insufficient to enable applications to recover from the majority of failures caused by application faults.

For example, a study [27] of a LAN of Windows NT-based mail servers indicates that while the measured availability of the system was 99%, the user-perceived availability was only 92%, i.e., the system was often alive but unable to provide a required service. A later analysis [28] reports similar figures on system availability of a Windows NT cluster consisting of two thousand workstations and servers. The need to account for the user's perception of system dependability is also stressed in an analysis of Windows 2000 dependability [29].

1.9 Technology Trends and Failure Behavior

As technology matures, the user set changes, the degree of product sophistication increases, and new sources of failures become prominent. It is important to understand these trends to determine the techniques that would be most effective in preventing, detecting, and recovering from the new sources of errors. One of the main trends driving the design of modern dependable systems has been the change in failure rates as well as the change 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 in failures is up.

Through the years, transient faults have typically been associated with the corruption of stored data values. The phenomenon was reported in adverse operating conditions as early as 1954, first in areas close to nuclear bomb test sites and later in space applications [30, 31]. Since 1978, dense memory circuits (both DRAM and SRAM) have been known to be at risk from soft errors caused by alpha particles from both IC packaging and cosmic rays. Following a transient failure, a hardware device can recover its full capability. Nonetheless, such a failure can be catastrophic for the correct execution of a program; a corrupted intermediate value, if not handled, can corrupt all subsequent computations.

Continuously decreasing feature sizes and supply voltages of devices will lessen capacitive node charge and noise margin. Inevitably, even flip-flop circuits will become susceptible to soft errors [32]. The high clock rate of modern processors makes the problem worse by increasing the probability of a new failure mechanism whereby a momentarily corrupted combinational signal is latched by a flip-flop. Continual pushing of the processor performance envelope will soon place users and developers in an unfamiliar realm where logically correct implementations alone cannot guarantee correct program execution with acceptable confidence. Explicit error detection and correction techniques have long been incorporated into the architectures provided by vendors of high-availability platforms. The basic techniques involve space redundancy (achieved by carrying out the same computation on multiple independent pieces of hardware at the same time and corroborating the redundant results to expose errors), information redundancy (e.g., parity and ECC), and time redundancy (whereby 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. Application of 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 register files, whereas space- and time-redundant techniques are employed elsewhere in the processor. A shortcoming of time redundancy is that persistent hardware faults may introduce identical errors to all redundant results, making errors indiscernible. A complementary shortcoming of space redundancy is that a transient failure may affect the space-redundant hardware identically, again making errors indiscernible.

Table 1.4 summarizes the changes over the last four decades in terms of the technology, error/fault sources, number of users, and their level of sophistication/training [33]. One consequence of the dropping hardware failure rate is that other failure modes have become more prominent. Increasing software complexity has gradually contributed to a larger proportion of system outages. Throughout the 1980s, as 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, as that was a growth segment into the 1990s. The PC segment was at its inception, and functionality was the focus there as well. As a consequence, software failures – a class of problems whose damaging effects could be as significant as hardware outages that took down entire systems – became more evident.

An important development in providing runtime mechanisms for handling software failures was the inception of design diversity. The seminal academic work materialized as the N-Version Programming [34], Recovery Blocks [35], and N-Self-Checking [36] approaches. Extensive experimental studies (e.g., [37]) were conducted to assess the effectiveness of the proposed solutions. Examples of industrial use span embedded control systems, in particular in the avionics and railway domains; e.g., [38, 39].

The following sections briefly discuss the issues that have emerged at the hardware and platform levels. We then turn to a discussion of cloud computing as an example of a computing paradigm that encompasses new technological trends while still being impacted by the aforementioned challenges.

1.10 Issues at the Hardware Level

- If standard redundancy techniques were to be applied as is, they would scale costs by a factor of the degree of integration of the processing elements.
  - For instance, consider this quick back-of-the-envelope calculation. For the $200 million Blue Waters supercomputer deployed at the University of

Table 1.4 Fault sources, levels of integration, users, and user sophistication over the past four decades.

Decade				
Topics	**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, wired/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 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	Tens of millions	Hundreds 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; couple of hours

Source: Siewiorek et al. [33]. © IEEE.

Illinois [40], requiring a hardware overhead of 35% would incur approximately $70 million in costs ($200M × 0.35), a sizeable amount.

- Process variations are an additional factor that has grown more important over time because of the decreasing feature size and increasing complexity that are resulting from greater and greater levels of integration.
- With decreasing feature sizes, issues such as negative bias temperature instability (NBTI) [41] have arisen as key reliability concerns at the device level.
  - The bug in Intel's Cougar Point SATA (Serial Advanced Technology Attachment) port on the 6-Series Chipset is an example of the impact of failures on device-level integration [42]. Cougar Point (Intel's 6-Series Chipsets, H67/P67) has two sets of SATA ports: four that support 3-Gbps operation, and two that support 6-Gbps operation. Each set of ports requires its own PLL (phase-locked loop) source. On 31 January 2011, Intel announced that it had identified a bug in the 6-Series Chipset, specifically in its SATA controller [43]. Intel stated that "in some cases, the Serial-ATA (SATA) ports within the chipsets may degrade over time, potentially impacting the performance or functionality of SATA-linked devices such as hard disk drives and DVD drives." Intel began shipping the corrected version of the chipset in late February of 2011. The recall reduced Intel's revenue by around $300 million, and it cost around $700 million to completely repair and replace affected systems.

1.11 Issues at the Platform Level

The use of virtual machine-based systems has drastically transformed the system view by introducing the hypervisor in between the operating system and the hardware. The relationship between the hypervisor/virtual machine monitor (VMM) and a guest operating system is analogous to the former relationship between the operating system and the application processes running atop it. The hypervisor introduces a new set of interactions (with both the operating system and applications) and consequent system failure modes.

The non-uniform, dynamic geographic distribution of nodes (in the current cloud computing environment) violates the assumptions of traditional distributed systems regarding communication overheads, resulting in a need to adapt these systems for efficiency in spite of the additional performance-deterring factors. Legacy techniques such as synchronous and asynchronous checkpointing already incur significant overhead and cannot be applied naïvely in the new scenario without investigation. Added to that are the high and nondeterministic costs that result from the dynamic nature of distributed systems.

1.12 What is Unique About this Book?

While this book reflects the experience and expertise of the authors in design and validation of dependable systems, it also introduces a holistic view of dependability and discusses techniques and approaches for achieving dependability from the system perspective.

System or application failures can originate in faults/errors at different levels of the system hierarchy (hardware, operating system, communication layer, middleware, or application). Therefore, a system (hardware and software) perspective is needed to properly handle design issues in dependable computing. Figure 1.3 depicts a basic system hierarchy and poses relevant questions that a system designer or developer must answer before building a dependable computing system.

At each system level – applications, middleware, communications, operating system, hardware, and network – we can ask ourselves what reliable dependability-supporting mechanisms are provided and what mechanisms can be added. In answering those questions, we must consider how hardware and software techniques can be combined; specifically, how fast error detection in hardware and high-efficiency detection and recovery in software can be coordinated, bearing in mind that fault-tolerance mechanisms in both hardware and software must themselves be self-checking and fault tolerant. We must also consider how to assess whether the dependability achieved is sufficient to meet system requirements. State-of-the-art techniques available to us at each of the four levels (and discussed in this book) are shown in Figure 1.4.

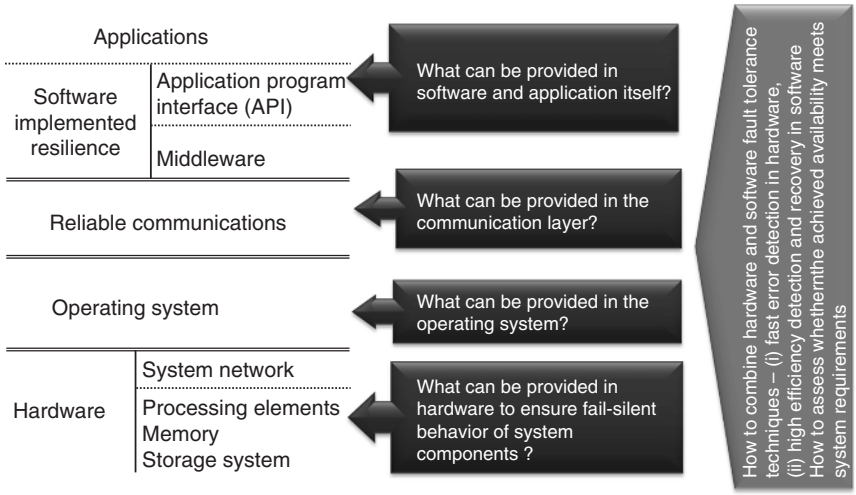

Figure 1.3 Dependability requirements at various system levels.

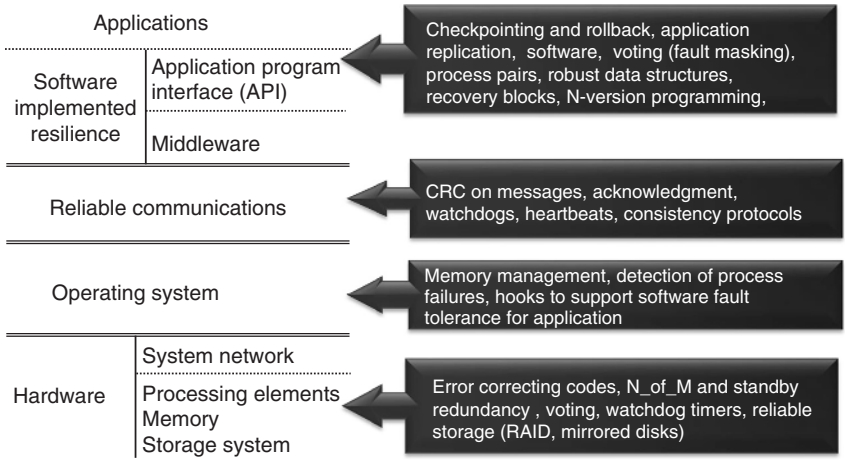

Figure 1.4 Techniques available at each system level.

1.13 Overview of the Book

The intended audience of this book ranges from students who wish to learn the fundamentals of dependable computing, to dependability researchers and engineers and software and hardware developers. The goal of this book is to present the fundamentals of designing computer systems for dependability. We will begin by introducing key terms and concepts and go on to illustrate and explain a wide range of design and implementation techniques. Because the design of dependable systems is as much an art as a science, our approach will be to provide, through real-life examples, a repertoire of ideas and techniques that you, the designer, can draw upon and adapt according to your constraints and requirements. We will discuss issues and techniques from both hardware and software viewpoints in the context of selected dependable systems, including distributed systems and networks.

To interest readers in the potential applications of dependability, in Chapter 5, we start with a set of illustrative case studies of systems that have paved the way for modern dependability and still remain relevant. At the same time, we present a spectrum of computing paradigms in which dependability has applications. We are going to present a system-level perspective that encompasses both hardware and software. To do so, we will first introduce hardware redundancy techniques. We will then present a variety of error-detecting and -correcting codes as well as noncoding detection techniques. Fault-tolerance techniques employed at the processor level in academic research and commercial systems will be presented. We will describe software fault tolerance techniques, including process pairs, robust

data structures, recovery blocks, and N-version programming, giving examples of an online transactional processing system and a controller database.

An entire chapter is devoted to dependability measurement of real and laboratory systems, in particular how to measure the robustness of operating systems and the effectiveness of techniques introduced at the operating system level. The fault models and primitives used in evaluating computing systems through software-implemented fault injection are presented in the chapter that covers fault injection techniques/approaches.

Network-specific issues include mechanisms and algorithms for providing consistent data and reliable communications in a network of distributed applications. We present broadcast, agreement, and commit protocols and illustrate them in the context of a reliable platform. We also compare the effectiveness and costs of a reliable design that relies on replication and one that relies on self-checking.

We then introduce the fundamentals of checkpointing and recovery techniques, followed by a discussion of the additional considerations involved in checkpointing large-scale systems. Checkpointing and rollback recovery are illustrated using three examples: a commercial, distributed database system (a "macro" example); checkpointing of multi-threaded processes (a "micro" example); and an operating system that supports checkpointing and recovery.

Validation is a key step, often overlooked or inadequately executed, in the design of any dependable system. Early validation of a system using modeling has been dealt with in detail by many others, e.g., Trivedi [44] and Sahner and Trivedi [45] (SHARPE), Clark et al. [46] (Möbius), and Ciardo and Miner [47] (SMART). In this book, we will address validation through field measurements and fault injections. Fault injection is an effective and common method for validating a dependability technique. Fault injection techniques can be implemented at various levels in a system, either in hardware or in software.

The book concludes with a chapter on understanding the dependability challenges of modern-day systems and paradigms: virtualization and cloud systems in software and system architecture, multicore processors in hardware, and stream processing in applications. We look into the dependability techniques applied in those areas in light of the techniques discussed in all the previous chapters.

References

1 Iyer, R.K., Sanders, W.H., Patel, J.H. et al. (2004). The evolution of dependable computing at the University of Illinois. *Building the Information Society: IFIP 18th World Computer Congress Topical Sessions* (ed. R. Jacquart), Toulouse, France (22–27 August 2004), 135–164. Boston, MA, USA: Kluwer Academic Publishers.

2 Abraham, J.A., Metze, G., Iyer, R.K. et al. (1987). The evolution of fault tolerant computing at the University of Illinois. In: *The Evolution of Fault Tolerant Computing. Dependable Computing and Fault-Tolerant Systems*, vol. 1 (eds. A. Avižienis, H. Kopetz and J.C. Laprie), 271–288. Vienna, Austria: Springer-Verlag.

3 Avižienis, A., Gilley, G.C., Mathur, F.P. et al. (1971). The STAR (self-testing and repairing) computer: an investigation of the theory and practice of fault-tolerant computer design. *IEEE Transactions on Computers* C-20 (11): 1312–1321.

4 Jones, C.P. (1979). Automatic fault protection in the Voyager spacecraft. AIAA Paper No. 79-1919, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA.

5 Yeh, Y.C. (1996). Triple–triple redundant 777 primary flight computer. *Proceedings of the IEEE Aerospace Applications Conference*, vol. 1, Aspen, CO, USA (10 February 1996), 293–307. IEEE.

6 Siewiorek, D.P. and Swarz, R.S. (1998). *Reliable Computer Systems: Design and Evaluation*, 3e. Natick, MA, USA: A K Peters/CRC Press.

7 Carter, W.C., Montgomery, H.C., Preiss, R.J. et al. (1964). Design of serviceability features for the IBM system/360. *IBM Journal of Research and Development* 8 (2): 115–126.

8 Garg, V.K. (2002). *Elements of Distributed Computing*. Wiley.

9 Iyer, R.K. and Velardi, P. (1985). Hardware-related software errors: measurement and analysis. *IEEE Transactions on Software Engineering* SE-11 (2): 223–231.

10 Spainhower, L. and Gregg, T.A. (1999). IBM S/390 Parallel Enterprise Server G5 fault tolerance: a historical perspective. *IBM Journal of Research and Development* 43 (5): 863–874.

11 Lee, I. and Iyer, R. (1993). Faults, symptoms, and software fault tolerance in the Tandem GUARDIAN90 operating system. *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, Toulouse, France (22–24 June 1993), 20–29. IEEE.

12 Avizienis, A. and Laprie, J.C. (1986). Dependable computing: from concepts to design diversity. *Proceedings of the IEEE* 74 (5): 629–638.

13 Avizienis, A., Laprie, J.C., Randell, B. et al. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1 (1): 11–33.

14 Sharma, V.S. and Trivedi, K.S. (2007). Quantifying software performance, reliability and security: an architecture-based approach. *Journal of Systems and Software* 80 (4): 493–509.

15 Gray, J. (1990). A census of Tandem system availability between 1985 and 1990. *IEEE Transactions on Reliability* 39 (4): 409–418. http://bnrg.eecs.berkeley.edu/~randy/Courses/CS294.F07/10.1.pdf (accessed 22 October 2020).

16 Naval History and Heritage Command (2020). NH 96566-KN: the first "computer bug." https://www.history.navy.mil/content/history/nhhc/our-collections/

photography/numerical-list-of-images/nhhc-series/nh-series/NH-96000/NH-96566-KN.html (accessed 25 May 2020).

17 Gray, J. (1985). Why Do Computers Stop and What Can Be Done About It? Technical Report 85.7. Tandem Computers.

18 Sullivan, M. and Chillarege, R. (1991). Software defects and their impact on system availability: a study of field failures in operating systems. *Digest of Papers, the 21st International Symposium on Fault-Tolerant Computing*, Montreal, Quebec, Canada (25–27 June 1991), 2–9. IEEE.

19 Sullivan, M. and Chillarege, R. (1992). A comparison of software defects in database management systems and operating systems. *Digest of Papers, the 22nd International Symposium on Fault-Tolerant Computing*, Boston, MA, USA (8–10 July 1992), 475–484. IEEE.

20 Chillarege, R., Bhandari, I.S., Chaar, J.K. et al. (1992). Orthogonal defect classification: a concept for in-process measurements. *IEEE Transactions on Software Engineering* 18 (11): 943–956.

21 Chillarege, R. (1994). ODC for process management, analysis, and control. *Proceedings of 4th International Conference on Software Quality*, McLean, VA, USA (3–5 October 1994).

22 Chillarege, R., Kao, W.-L., and Condit, R.G. (1991). Defect type and its impact on the growth curve. *Proceedings of the 13th International Conference on Software Engineering*, Austin, TX, USA (13–16 May 1991), 246–255. IEEE.

23 Durães, J. and Madeira, H. (2006). Emulation of software faults: a field data study and a practical approach. *IEEE Transactions on Software Engineering* 32 (11): 849–867.

24 Nakka, N. and Choudhary, A. (2010). Failure data-driven selective node-level duplication to improve MTTF in high performance computing systems. In: *High Performance Computing Systems and Applications. HPCS 2009. Lecture Notes in Computer Science*, vol. 5976 (eds. D.J.K. Mewhort, N.M. Cann, G.W. Slater, et al.), 304–322. Berlin, Heidelberg: Springer.

25 Schroeder, B. and Gibson, G.A. (2006). A large-scale study of failures in high-performance computing systems. *Proceedings of the International Conference on Dependable Systems and Networks*, Philadelphia, PA, USA (25–28 June 2006). IEEE.

26 Chandra, S. and Chen, P.M. (2000). Whither generic recovery from application faults? A fault study using open-source software. *Proceedings of the International Conference on Dependable Systems and Networks*, New York, NY, USA (25–28 June 2000). IEEE.

27 Kalyanakrishnam, M., Kalbarczyk, Z., and Iyer, R. (1999). Failure data analysis of a LAN of Windows NT based computers. *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, Lausanne, Switzerland (22 October 1999). IEEE

28 Kalyanakrishnan, M., Iyer, R.K., and Patel, J.U. (1999). Reliability of Internet hosts: a case study from the end user's perspective. *Computer Networks* 31 (1–2): 47–57.

29 Murphy, B. and Levidow, B. (2000). Windows 2000 Dependability. Microsoft Research Technical Report MSR-TR-2000-56.

30 Ziegler, J.F., Curtis, H.W., Muhlfeld, H.P. et al. (1996). IBM's experiments in soft fails in computers. *IBM Journal of Research and Development* 40 (1): 3–18.

31 Normand, E. (1996). Single event upset at ground level. *IEEE Transactions on Nuclear Science* 43 (6): 2742–2750.

32 Faccio, F., Kloukinas, K., Marchioro, A. et al. (1999). Single event effects in static and dynamic registers in a 0.25 μm CMOS technology. *IEEE Transactions on Nuclear Science* 46 (6): 1434–1439.

33 Siewiorek, D.P., Chillarege, R., and Kalbarczyk, Z.T. (2004). Reflections on industry trends and experimental research in dependability. *IEEE Transactions on Dependable and Secure Computing* 1 (2): 109–127.

34 Avižienis, A. (1985). The *N*-version approach to fault tolerant software. *IEEE Transactions on Software Engineering* SE-11 (12): 1491–1501.

35 Anderson, T., Barrett, P.A., Halliwell, D.N. et al. (1985). Software fault tolerance: an evaluation. *IEEE Transactions on Software Engineering* SE-11 (12): 1502–1510.

36 Laprie, J.-C., Arlat, J., Beounes, C. et al. (1990). Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer* 23 (7): 39–51.

37 Brilliant, S.S., Knight, J.C., and Leveson, N.G. (1990). Analysis of faults in an *N*-version software experiment. *IEEE Transactions on Software Engineering* 16 (2): 238–247.

38 Kantz, H. and Koza, C. (1995). The ELEKTRA railway signalling system: Field experience with an actively replicated system with diversity. *Digest of Papers, 25th International Symposium on Fault-Tolerant Computing*, Pasadena, CA, USA (27–30 June 1995), 453–458. IEEE.

39 Traverse, P. (1991). Dependability of digital computers on board airplanes. *Proceedings of the 1st International Working Conference on Dependable Computing for Critical Applications*, Santa Barbara, CA, USA (August 1989), 133–152 (ed. A. Avižienis and J. Laprie), *Dependable Computing and Fault-Tolerant Systems*, vol. 4. Vienna: Springer.

40 About Blue Waters (2020). National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign. http://www.ncsa.illinois.edu/BlueWaters/ (accessed 31 July 2020).

41 Schroder, D.K. and Babcock, J.A. (2003). Negative bias temperature instability: road to cross in deep submicron silicon semiconductor manufacturing. *Journal of Applied Physics* 94 (1): 1–18.

42 Shimpi, A.L. (2011). The source of Intel's Cougar Point SATA bug. *AnandTech* (31 January). http://www.anandtech.com/show/4143/the-source-of-intels-cougar-point-sata-bug (accessed 31 July 2020).

43 Intel Newsroom. (2011). Intel identifies chipset design error, implementing solution. https://newsroom.intel.com/news-releases/intel-identifies-chipset-design-error-implementing-solution/#gs.09015p (accessed 31 January).

44 Trivedi, K.S. (2002). *Probability and Statistics with Reliability, Queuing and Computer Science Applications*, 2e. Chichester, UK: Wiley.

45 Sahner, R.A. and Trivedi, K.S. (1987). Reliability modeling using SHARPE. *IEEE Transactions on Reliability* R-36 (2): 186–193.

46 Clark, G., Courtney, T., Daly, D. et al. (2001). The Möbius modeling tool. *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, Aachen, Germany (11–14 September 2001), pp. 241–250. IEEE.

47 Ciardo, G. and Miner, A.S. (1996). SMART: simulation and Markovian analyzer for reliability and timing. *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, Urbana-Champaign, IL, USA (4–6 September 1996), 60. IEEE.