Int J Syst Assur Eng Manag (Oct-Dec 2010) 1(4):340–350
DOI 10.1007/s13198-011-0038-9

ORIGINAL ARTICLE

Software fault mitigation and availability assurance techniques

Kishor S. Trivedi · Michael Grottke · Ermeson Andrade

Received: 1 December 2010 / Revised: 24 December 2010 / Published online: 13 April 2011

Abstract Companies are expected to keep their systems up and running and make data continuously available. Several recent studies have established that most system outages are due to software faults. In this paper, we discuss availability aspects of large software-based systems. We begin by classifying software faults into Bohrbugs and Mandelbugs, and identify aging-related bugs as a subtype of the latter. We then examine mitigation methods for Mandelbugs in general and aging-related bugs in particular. Finally, we discuss techniques for the quantitative availability assurance for such systems.

Keywords Aging-related bugs · Mandelbugs · Mitigation techniques · Software rejuvenation · Stochastic availability models

1 Introduction

High availability is being demanded for military as well as commercial applications such as e-commerce systems, financial systems, stock-trading systems, national and international telecommunication infra-structure (e.g., switches and routers) and several types of life-critical and safety-critical systems. Many techniques to achieve high availability from the hardware perspective are known. However, software remains the main bottleneck in achieving high availability. Despite many advances in formal methods, programming methodology, and testing, the software development process has not reached the stage to allow for the routine production of ultra-low defect software systems (Bharadwaj 2008). Yet, complex software-based systems are expected not to fail.

According to (Grottke and Trivedi 2007), during the Gulf War, 28 US Army reservists were killed and 97 were injured on 25 February 1991 when the Patriot missile-defense system at their barracks in Dhahran, Saudi Arabia, failed to intercept an incoming Scud missile. This incident was due to a software fault in the system's weapons-control computer. It turned out that the failure occurrence rate for this fault increased with system runtime, making it a prominent example of software aging. It was also a case in which engineers employed multiple techniques to fight the software bug.

Outages in computer systems can be caused by hardware and software faults. While hardware faults have been studied extensively and varied mechanisms are known to increase system availability in face of such faults, software faults, their mitigation methods and the corresponding reliability/availability analysis have not drawn much attention from researchers. However, software faults are now known to be a dominant source of system outages. During operations software faults are usually dealt with reactively (i.e., after they caused a failure), while failures due to software aging can be avoided by a proactive technique called software rejuvenation (Bernstein and Kintala 2004; Grottke and Trivedi 2007).

K. S. Trivedi (✉)
Department of Electrical and Computer Engineering,
Duke University, Durham, NC 27708, USA
e-mail: kst@ee.duke.edu

M. Grottke
University of Erlangen-Nuremberg, Nuremberg, Germany
e-mail: michael.grottke@wiso.uni-erlangen.de

E. Andrade
Informatics Center, Federal University of Pernambuco (UFPE),
Recife, PE, Brazil
e-mail: ecda@cin.ufpe.br

In general, there are two ways to improve availability: increase time-to-failure (TTF) and reduce time-to-recovery (TTR). To increase TTF, proactive software rejuvenation can be used for aging-related bugs. To reduce TTR, escalated levels of recovery can be used, so that most failures are fixed by the quickest recovery method and only few by the slowest ones. We are also interested in methods to provide quantifiable availability assurance.

In this paper, we first classify software faults and discuss various mitigation techniques. Since approaches to dealing with Bohrbugs are well known, the topic is only covered very briefly. Approaches to deal with Mandelbugs in general and with aging-related bugs in particular are described. This helps us understand the nature of software faults and their impact on system availability and performance and aids in choosing the best possible recovery strategy when a failure occurs. Quantitative assurance of availability by means of stochastic availability models is introduced. It is important to highlight that to deal with the large size of availability models of real systems we typically employ a hierarchical approach. Finally, we discuss quantitative assurance applied to real systems.

The remainder of the paper is organized as follows: Section 2 classifies software faults and discusses the various mitigation techniques to deal with these faults. Section 3 discusses the use of analytic models for availability assurance. Section 4 describes the approaches to deal with Mandelbugs. Section 5 describes the approaches to deal with aging-related bugs. Section 6 discusses quantitative assurance applied to real systems. Section 7 concludes the paper.

2 Classification and treatment of software faults

In this section, we classify software faults and discuss various mitigation techniques to deal with these faults in the testing/debugging phase and in the operational phase of the software system.

2.1 Software fault classification

When faults in a hardware or a software component are activated, they cause errors. An error is defined as that part of the internal state of a running system that is liable to lead to subsequent failure when the error is propagated. According to (Laprie 1992), "a system failure occurs when the delivered service no longer complies with the specifications, the latter being an agreed description of the system's expected function and/or service". If the system comprises of multiple components, an error can lead to a component failure. As various components in the system interact, the failure of one component may introduce faults in other components. For instance, Fig. 1 presents a tree listing all the component failures considered in a model of the IBM SIP Application Server cluster (Trivedi et al. 2008). Hardware component failures are listed in the left branch from the root, whereas software component failures are listed in the right branch.

There are broadly two classes of software faults (or bugs), known as Bohrbugs and Mandelbugs (Grottke and Trivedi 2005b, 2007).

Bohrbugs are, in principle, easily isolated and manifest themselves consistently under well-defined sets of conditions (Raymond 1991).

The behavior of *Mandelbugs* instead appears chaotic or even non-deterministic (Raymond 1991), because their fault activation and/or error propagation are complex. On the one hand, this complexity can take the form of time lags between the fault activation and the final failure occurrence caused by the fact that multiple error conditions need to be traversed (Grottke and Trivedi 2005a); this

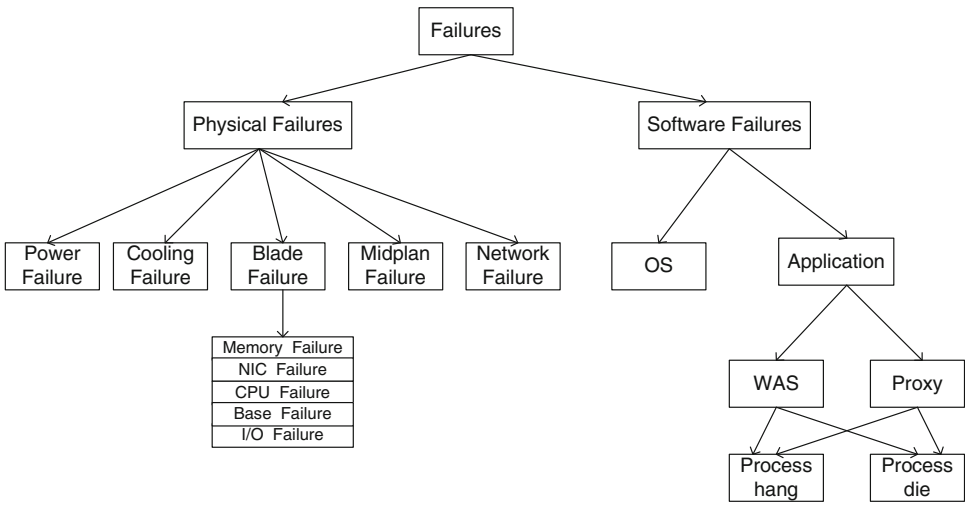

Fig. 1 Component failures in the IBM SIP Application Server cluster

makes it difficult for a user to understand the causes of a failure experienced. A second source of complexity is the influence of indirect factors, like the timing of operations and interactions with other applications, on fault activation and error propagation (Grottke et al. 2010). Software faults causing race conditions are a well-known example of Mandelbugs. Due to the complex fault activation or error propagations of Mandelbugs, retrying the operation carried out prior to a failure occurrence may not result in another manifestation of the fault. Therefore, failures caused by Mandelbugs are typically difficult to reproduce.

Sometimes, the literature also calls such faults Heisenbugs (Gray 1986; Huang et al. 1995). However, Bruce Lindsay, who invented the term deriving it from Heisenberg's Uncertainty Principle, reserves the term for faults that change their behavior when probed or isolated (Winslett 2005). Lindsay's Heisenbugs are actually a subtype of Mandelbugs (Grottke and Trivedi 2005a, 2007).

Another interesting subtype of Mandelbugs (Grottke and Trivedi 2007) has the characteristic that it is capable of causing an increasing failure occurrence rate and/or degraded performance during execution. Such faults have been observed in many software systems and have been called *aging-related bugs* (Avritzer and Weyuker 1997; Garg et al. 1998; Grottke et al. 2006; Grottke and Trivedi 2007; Marshall 1992). There are two possible causes of the aging phenomenon: First, errors caused by the activation of aging-related bugs may accumulate inside the running system. Examples include memory leaks and round-off errors. Second, the activation and/or error propagation of the system may be influenced by the total time the system has been running. The incident with the Patriot missile-defense system mentioned in Sect. 1 is an example for such an aging behavior (Grottke et al. 2008).

Figure 2 summarizes the classification of software faults. Based on the relationships between the fault types, we can partition the space of all software faults into Bohrbugs, aging-related bugs, and those Mandelbugs that are nor aging-related bugs. We refer to the latter set as non-aging-related Mandelbugs.

As we will see in Sect. 2.2, possible methods to deal with a fault depend on its type. The relative importance of the mitigation methods is thus influenced by the fault type proportions. To get an idea of these proportions in real-world systems, Grottke et al. (2010) classified the 520 software faults detected in 18 NASA/JPL space missions after launch. While 2.1% of the faults could not be classified based on the information available, 61.4% were Bohrbugs and 32.1% turned out to be non-aging-related Mandelbugs; the remaining 4.4% were aging-related bugs.

Fig. 2 Venn diagram of software fault types

2.2 Software fault mitigation techniques

Figure 3 assigns potential methods for treating software faults in the test phase or in the operational phase to the fault types described in Sect. 2.1.

The classical approach to deal with software faults introduced during development is to *debug/test* the software. It is specifically applicable to Bohrbugs, which by definition manifest consistently. The Bohrbugs that caused failures in the testing phase can thus rather easily be isolated and removed from the code. Likewise, if a failure due to a Bohrbug occurs in production, it can be reproduced in the original testing environment, and a patch correcting the bug or a workaround can be issued. Due to their seemingly non-deterministic nature Mandelbugs are much more difficult to understand and detect in the software code. Therefore, run-time techniques to recover from failures caused by Mandelbugs (or to even prevent such failures) are usually more cost- and time-effective than debug/test approaches. We will discuss these techniques below.

The *design diversity* approaches have specifically been developed to tolerate design faults in software arising out of wrong specifications and incorrect coding. Two or more variants of a piece of software developed by different teams, but to a common specification, are used. These variants are then used in a time- or space-redundant manner to achieve fault tolerance. Popular techniques which are based on the design diversity concept for fault tolerance in software are N-version programming (Avižienis and Chen 1977), recovery blocks (Horning et al. 1974) and N-self-checking programming (Laprie et al. 1987). The design diversity approach was developed mainly to deal with Bohrbugs. Of course, it can also mitigate Mandelbugs, provided that Mandelbugs leading to a failure in a specific situation are not contained in several (or even all) of the different designs; however, there are less expensive approaches to dealing with Mandelbugs.

Data diversity relies on the observation that software sometime fails for certain values in the input space and this failure could be averted if there is a minor perturbation of input data which is acceptable to the software. This approach can work well with Bohrbugs and is cheaper to implement than design diversity techniques. Data diversity can also deal with Mandelbugs since retrying the operation (with or without changing the input data) may not lead to another failure occurrence.

Although *environment diversity* has been used for long in an *ad hoc* manner, only recently has it gained recognition and

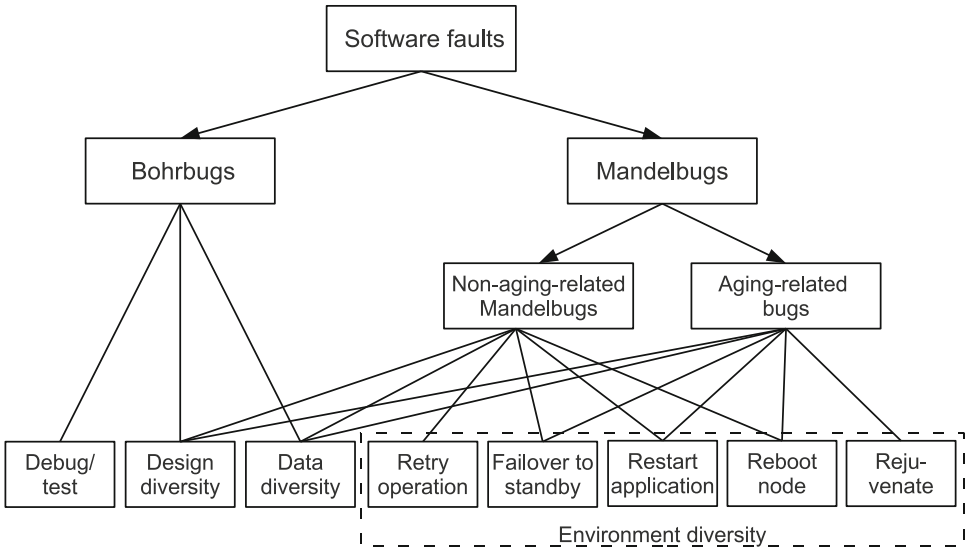

Fig. 3 Software fault types and mitigation methods

importance. Having its basis on the observation that many software failures are caused by Mandelbugs and thus transient in nature, environment diversity involves reexecuting the software in a different environment. Adams (1984) proposed restarting the system as the best approach to masking software faults. Environment diversity, a generalization of restart, comprises a range of techniques, including operation retry, failover to an identical standby copy, application restart, and node reboot. The retry and restart operations can be done on the same node or on another spare (cold/warm/hot) node. These techniques deal effectively with Mandelbugs by exploiting the fact that their activation and/or error propagation depend on complex causes, which may be removed by changing the environment.

The environment diversity approaches discussed so far are reactive in nature, recovering from failures caused by Mandelbugs. However, software rejuvenation (Bernstein and Kintala 2004; Grottke and Trivedi 2007; Huang et al. 1995) is a proactive environment diversity technique trying to prevent future failure occurrences, for example by regularly stopping and rebooting the running software system. The removal of accrued errors and the resetting of the runtime of the system caused by the rejuvenation lead to a decrease in the failure rate and the alleviation of performance problems due to aging-related bugs.

In this paper, we focus on Mandelbugs in general and aging-related bugs in particular.

3 Quantified availability assurance

In practice, availability assurance is provided qualitatively by means of verbal arguments or using checklists. Quantitative assurance of availability by means of stochastic availability models constructed based on the structure of the system hardware and software is very much lacking in today's practice (Smith et al. 2008; Trivedi 2000; Trivedi et al. 2006). While such analyses are nowadays supported by software packages (Bolch et al. 2006; Sahner et al. 1996), they are not routinely carried out on what are touted as high availability products; there are only islands of such competency even in large companies.

Engineers commonly use reliability block diagrams or fault trees to formulate and solve availability models because of their simplicity and efficiency (Sahner et al. 1996; Trivedi 2001). But such non-state-space models cannot easily incorporate realistic system behavior such as imperfect coverage, multiple failure modes, detection and recovery delays, or hot swap (Smith et al. 2008). In contrast, such dependencies and multiple failure modes can easily be captured by state-space models such as Markov chains, semi-Markov processes (Trivedi 2001), and Markov regenerative processes (Bolch et al. 2006).

However, the construction, storage, and solution of these state space models can become prohibitive for real systems. The problem of large model construction can be alleviated by using some variation of stochastic Petri nets (Bolch et al. 2006), but a more practical alternative is to use a hierarchical approach based on a judicious combination of state space models and non-state-space models (Sahner et al. 1996). Such hierarchical models have successfully been used on practical problems including hardware availability prediction (Lanus et al. 2003), OS failures (Smith et al. 2008; Trivedi 2000; Trivedi et al. 2006) and application software failures (Garg et al. 1995). Furthermore, user and service-oriented measures can be computed in addition to system availability. Computational methods for such user-perceived measures are just beginning to emerge (Sato et al. 2007).

As an example, the IBM BladeCenter is a system where the complexity of the system precludes modeling as a

single-level state space model. The number of BladeCenter components subject to failure is close to 140. If each component were to be in one of two states only (actually some components have more than two states), the size of the state space of the overall Markov chain would be 2^{140}. However, as dependencies exist in the system, an overall non-state-space model will not suffice. Dependencies within subsystems are modeled in (Smith et al. 2008) using homogeneous continuous-time Markov chains. Independence across subsystems is assumed, thus a non-state-space model is used to combine the subsystem availabilities into the overall system availability. The top-level model is a fault tree because some of the component failures affect several different portions of the system at the same time. Such effects are captured by fault trees with repeated events but cannot be captured by reliability block diagrams (Sahner et al. 1996; Trivedi 2001). Other methods to reduce the state space size include state truncation, applicable to high-level model descriptions such as stochastic Petri nets (Liu et al. 2005; Vaidyanathan et al. 2001; Wang and Trivedi 2009) and fixed-point iterations (Mainkar and Trivedi 1996; Tomek and Trivedi 1991). Besides availability assurance, such models can also be used to find availability bottlenecks (Sato et al. 2007).

Subsequently, parameter values are needed to solve the models and predict system availability and related measures. Model input parameters can be divided into failure rates of hardware or software components; detection, failover, restart, reboot and repair delays and coverages; and parameters defining the user behavior. Hardware failure rates (or equivalently, MTTFs) are generally available from vendors, but software component failure rates are much harder to obtain. Alcatel-Lucent uses residual failure intensity based on a software reliability growth model as the failure rate in operation (Mendiratta et al. 2007).

Fault injection experiments can be used to estimate detection, restart, reboot, and repair delays (Hsueh et al. 1997), as in the IBM SIP/SLEE modeling exercise (Smith et al. 2008; Trivedi et al. 2006). Statistical inference methods for the estimations are well known (Arlat et al. 1993; Lee and Iyer 1995; Meeker and Escobar 1998; Nelson 1982; Tobias and Trindade 1995). However, passing the confidence intervals of input parameters through an analytic availability model is relatively unexplored (Devraj et al. 2010).

Due to many simplifying assumptions made about the system, its components, and their interactions and due to unavailability of accurate parameter values, the results of the abstract models cannot be taken as a true availability assurance. Monitoring and statistically inferring the observed availability is surely much more satisfactory assurance of availability. Off-line and on-line monitoring (Pietrantuono et al. 2010) of deployed system availability and related metrics can be carried out. The major difficulty is the amount of time needed to get enough data to obtain statistically significant estimates of availability.

4 Recovery from failures caused by Mandelbugs

Reactive recovery from failures caused by Mandelbugs has been used for some time in the context of operating system failures, where reboot is the mitigation method (Hunter and Smith 1999; Trivedi et al. 2006). Restart, failover to a replica, and further escalated levels of recovery such as node reboot and repair are being successfully employed for application failures. Avaya's NT-SwiFT and DOORS systems (Garg et al. 1999), JPL REE system (Chen et al. 2002), Alcatel Lucent (Mendiratta 1999; Mendiratta et al. 2007; Vilkomir et al. 2005), IBM x-series models (Vaidyanathan et al. 2001), CORBA (Narasimhan et al. 2005; Pertet and Narasimhan 2004; Pertet and Narasimhan 2005), and IBM SIP/SLEE cluster (Smith et al. 2008; Trivedi et al. 2006) are examples where applications or middleware processes are recovered using one or more of these techniques. Replication of software system has been used as viable fault tolerance technique to improve reliability/availability. A standby copy to failover to can be either an active or a passive replica. For example, the IBM SIP/SLEE system uses active replication (Trivedi et al. 2008), while Avaya's SwiFT system uses passive replication (Garg et al. 1999). In active replication, both copies serve different requests at the same time and constant synchronization of data might be required if the data is not partitioned across replica. In passive replication, only one replica, the primary, executes at any one time while one or more backups are waiting to take over when the primary fails (Dumitras et al. 2005). Passive replication can be further divided into two categories: warm and cold (Garg et al. 1999). Warm replicas are periodically updated with state information while cold replicas are not. The chosen way to organize the replicas has both performance and availability impact. Performance penalty will be larger as we move from cold to warm to active replication while the availability will likely improve. Detailed availability and performance models can be developed for the three schemes as in (Garg et al. 1999).

To support recovery from Mandelbug-caused failures, multiple run-time failure detectors are employed to ensure that detection takes place within a short duration of the failure occurrence (Trivedi et al. 2008). In all but the rarest cases, manual detection is required. As, by definition, failures caused by non-aging-related Mandelbugs cannot be anticipated and must be reacted to, current research is aimed at providing design guidelines as to how fast recovery can be accomplished and obtaining quantitative assurance on the availability of an application.

Recovery should be tailored to different kinds of failures and only touch the affected system components. However, a recovery technique may not always successfully recover an application from the current failure. Since it is not known in advance which recovery technique should be used after a failure occurrence, a sequence of recovery procedures consisting of specific escalated levels or stages of recovery should be employed. Typically, the techniques are ordered according to the expected length of time needed for their execution: the fastest technique is tried first, while the last recovery stage may be slow but guarantees recovery. An example of such a sequence is: micro-rebooting of an individual software component in an application, restart of the application, fail-over, reboot of the entire node, and full system repair. Since the number of possible recovery actions is small, preliminary research suggests that an exhaustive search is adequate to determine the optimal sequence (Grottke and Trivedi 2008).

Stochastic models discussed in the previous section are beginning to be used to provide quantitative availability assurance (Chen et al. 2002; Garg et al. 1999; Vaidyanathan et al. 2001). Besides system availability, models to compute user-perceived measures such as dropped calls in a switch due to failures are beginning to be used (Trivedi et al. 2010). Such models can capture the details of user behavior or the details of the call flow (Vaidyanathan et al. 2001) and its interactions with failure and recovery behavior of hardware and software resources. Difficulties we encounter in availability modeling are model size and obtaining credible input parameters (Bolch et al. 2006; Nicol et al. 2004; Smith et al. 2008). To deal with the large size of availability models for real systems, we typically employ a hierarchical approach where the top-level is a non-state-space model, such as a fault tree (Lanus et al. 2003; Smith et al. 2008) or a reliability block diagram (Trivedi 2000; Trivedi et al. 2006). Lower-level stochastic models for each subsystem in the fault tree model are then built. These submodels are usually continuous-time Markov chains, but if necessary non-Markovian models (Wang et al. 2003) can be employed. Weak interactions between submodels can be dealt with using fixed-point iteration (Mainkar and Trivedi 1996; Tomek and Trivedi 1991). The key advantage of such hierarchical approaches is that closed-form solution now appears feasible (Sato et al. 2007; Smith et al. 2008) as the Markov submodels are typically small enough to be solved by Mathematica and the fault tree can be solved in closed form using tools like our own SHARPE software package (Sahner et al. 1996). Once the closed-form solution is obtained, we can also carry out sensitivity analysis to determine bottlenecks and provide feedback for improvement to the designers (Sato et al. 2007). Errors in these approximate hierarchical models can be studied by comparison with discrete-event simulation and exact stochastic Petri net models solved numerically.

5 Proactive recovery and aging-related bugs

As noted in Sect. 2.1, aging-related bugs are such that they can cause an increasing failure rate and/or degraded performance while the system is up and running. For such bugs proactive software rejuvenation cleaning the system internal state and resetting the system runtime may effectively reduce the failure rate and improve performance. Many types of software systems, such as telecommunication software (Avritzer and Weyuker 1997; Bernstein and Kintala 2004; Huang et al. 1995), network devices (Cisco Systems (2001), Document ID 13618), web servers (Grottke et al. 2006; Matias and Freitas Filho 2006), and military systems (Marshall 1992), are known to experience aging. Rejuvenation has been implemented in several kinds of software systems, including telecommunication billing data collection systems (Huang et al. 1995), transaction processing systems (Cassidy et al. 2002), spacecraft flight systems (Tai et al. 1999), distributed CORBA-based applications (Pertet and Narasimhan 2004), and cluster servers (Castelli et al. 2001).

The main advantage of planned preemptive procedures such as rejuvenation is that the consequences of sudden failures (like loss of data and unavailability of the entire system) are postponed or prevented; moreover, administrative measures can be scheduled to take place when the workload is low. However, for each such preemptive action, costs are incurred in the form of scheduled downtime for at least some part of the system. Rejuvenation can be carried out at different granularities: restart a software module, restart an entire application, perform garbage collection in a node, or reboot a hardware node (Candea et al. 2004; Hong et al. 2002; Matias and Freitas Filho 2006; Xie et al. 2005). A key design question is finding the optimal rejuvenation schedule and granularity.

Rejuvenation scheduling can be time-based or condition-based. In the former, rejuvenation is done at fixed time intervals (Castelli et al. 2001; Dohi et al. 2000, 2001; Garg et al. 1995; Hong et al. 2002; Huang et al. 1995; Liu et al. 2005; Vaidyanathan et al. 2001), while, in the latter, the condition of system resources is monitored. A simple threshold-based rejuvenation is carried out or prediction algorithms are used to determine an adaptive rejuvenation schedule (Avritzer et al. 2006; Garg et al. 1998; Silva et al. 2006; Vaidyanathan and Trivedi 2005; Xie et al. 2005).

For time-based rejuvenation, the stochastic models discussed in Sect. 2 are enhanced to incorporate aging-related bugs and the rejuvenation triggers at various levels of granularity. The resulting models are no longer Markovian.

We have solved such models using a combination of phase-type expansions and deterministic and stochastic Petri nets (DSPN) type techniques (Lindemann 1998; Wang and Trivedi 2009). To solve and optimize these models, besides the input parameters discussed in Sect. 2, we need parameters for various proactive recovery actions and the time to failure distribution due to aging-related failures. Before the system is deployed, a distribution and its parameters will have to be based on past experience. During the operation of the system, time to failure data can be collected and used to parameterize the optimization model. It is possible to directly use the measured time-to-failure data in the optimization of the rejuvenation schedule via the notion of total time on test (TTT) transform to avoid the error-prone process of fitting of this data to a distribution (Barlow and Campo 1975; Dohi et al. 2000). The scheme is then a closed-loop feedback control system (Hellerstein et al. 2004) where the "fixed-time" is adaptive in response to monitored time-to-failure data. A rejuvenation trigger interval, as computed in time-based rejuvenation, adapts to changing system conditions, but its adaptation rate is slow as it only responds to failure occurrences that are expected to be rare.

Condition-based rejuvenation instead does not need time to failure inputs; it computes the rejuvenation trigger interval by monitoring system resources and predicting the time to exhaustion of resources for the adaptive scheduling of software rejuvenation (Avritzer and Weyuker 1997; Castelli et al. 2001; Garg et al. 1998; Pertet and Narasimhan 2004; Vaidyanathan and Trivedi 2005). Garg et al. (1998) measured variables such as free main memory, used swap space, and file table size in a network of UNIX workstations. These measured variables showed a statistically significant (decreasing or increasing) trend over time. Using a non-parametric technique, Garg et al. determined the global aging trend and calculate the estimated time until complete exhaustion via linear extrapolation for each resource. In case some form of rejuvenation or periodicity is already implemented by the system, as in the Apache Web server (Grottke et al. 2006), piecewise linear (Castelli et al. 2001), autoregressive time series with deterministic seasonal component (Grottke et al. 2006), nonlinear statistical methods (Hoffman et al. 2006), and fractal-based methods (Shereshevsky et al. 2003) have also been used on such data. Regardless of the prediction method used, the resources selected for monitoring must be determined to minimize the monitoring overhead. Design of experiment (DOE) and analysis of variance (ANOVA) have been used to answer this question (Matias et al. 2010; Matias et al. 2010; Montgomery 2004). All the published methods predict the times to exhaustion of individual resources, but time to system failure is a complex combination of these times. Predicting time to failure is an open question.

Whatever schedule and granularity of rejuvenation is used, the important question is what improvement this implies on system availability, if any. Published results are based on either analytic models (Vaidyanathan et al. 2001) or simulations (Dohi et al. 2000). Early results of a measurement experiment at Tokyo Institute of Technology are very encouraging, where rejuvenation increased the MTTF by a factor of two (Kourai 2008, personal communication, January 9) on the system reported in (Kourai and Chiba 2007).

6 Case studies of analytic models

In this section, we discuss quantitative assurance via analytic models.

6.1 SIP on IBM WebSphere

In (Trivedi et al. 2008), we model a SIP (Session Initiation Protocol) service consisting of the WebSphere Application Server and a proxy server running on IBM BladeCenter hardware. Figure 4 presents the IBM SIP Application Server cluster configuration. It consists of two BladeCenters each having four blade servers (which we call nodes). Each blade server has WebSphere Application Server (WAS) Network Deployment v6.1 installed. In the cluster, two nodes (one in each chassis) are configured as proxy servers to balance SIP traffic and perform failover. The other nodes host application servers. The SIP application installed on the application servers is back-to-back user agent (B2BUA), which acts as a proxy for SIP messages in VoIP call sessions. Twelve application servers are paired in groups of two; each of these are known as replication domains.

This configuration provides high availability by using hardware and software redundancy and escalated levels of recovery. Besides, different types of fault detectors, detection delays, failover delays, restarts, reboots and repairs are considered. Imperfect coverages for detection, failover and recovery are incorporated. Software redundancy used is with identical replicas and not design diversity. Mitigation method for software failures is to first try automated restart and in case that does not work, manual restart followed by node reboot and only as a last resort manual repair is utilized. Clearly, designers of this system assume that failures due to Mandelbugs are much more likely than those due to Bohrbugs.

Availability model computations are based on a set of interacting submodels of all system components capturing their failure and recovery behavior. It is important to highlight that the parameter values used in the calculations are based on several sources, including field data, high

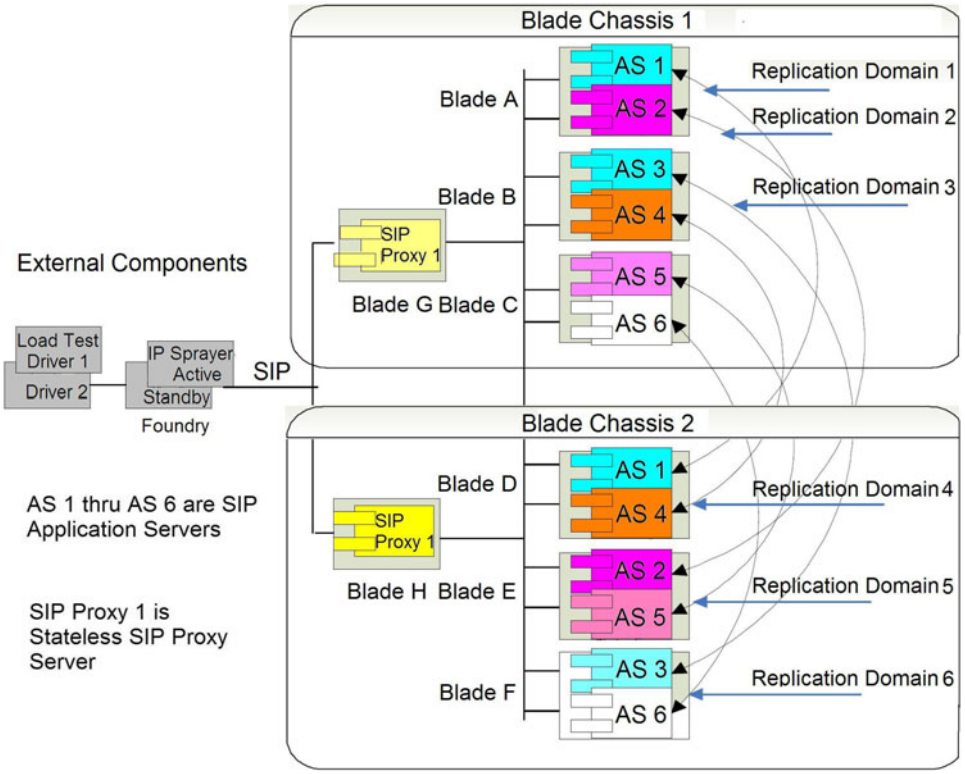

Fig. 4 IBM SIP Application Server clusterad. Adapted from Trivedi et al. (2008)

availability testing, and agreed-upon assumptions. The top level is a fault tree (shown in Fig. 5) whose nodes represent all of the software and hardware failures. At this level, it is possible to capture the system structure that tells whether the whole system is available given the state of the software and hardware subsystems.

Some of the hardware subsystems are broken down into more detailed fault trees. In (Trivedi et al. 2008), these are then included in the top-level fault tree model, but they could also have been separated into mid-level models. Which to choose depends on the nature of the system, the complexity of the models at each level, the modeler's

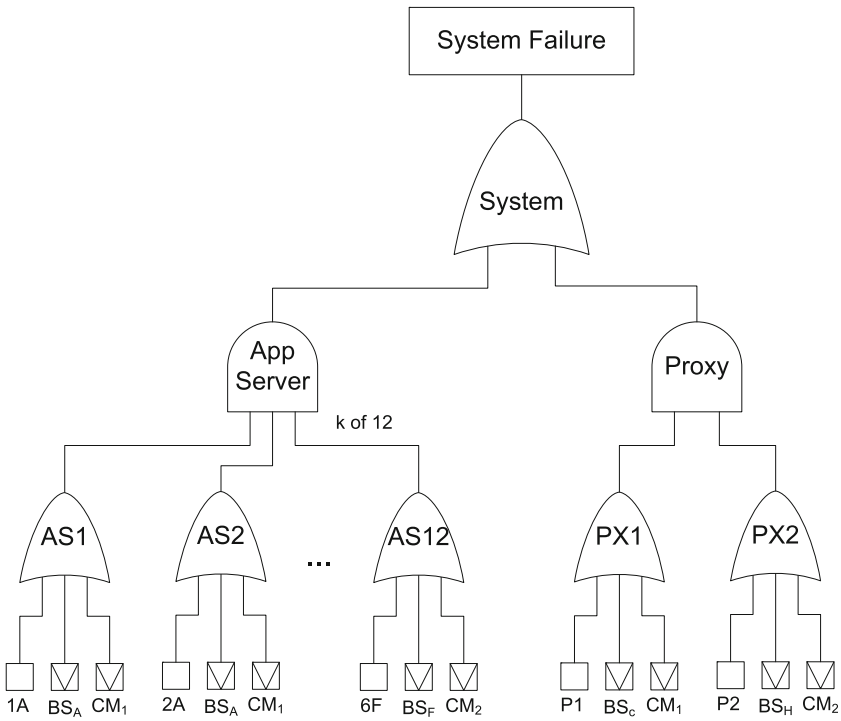

Fig. 5 Top-level fault tree model. Adapted from Trivedi et al. (2008)

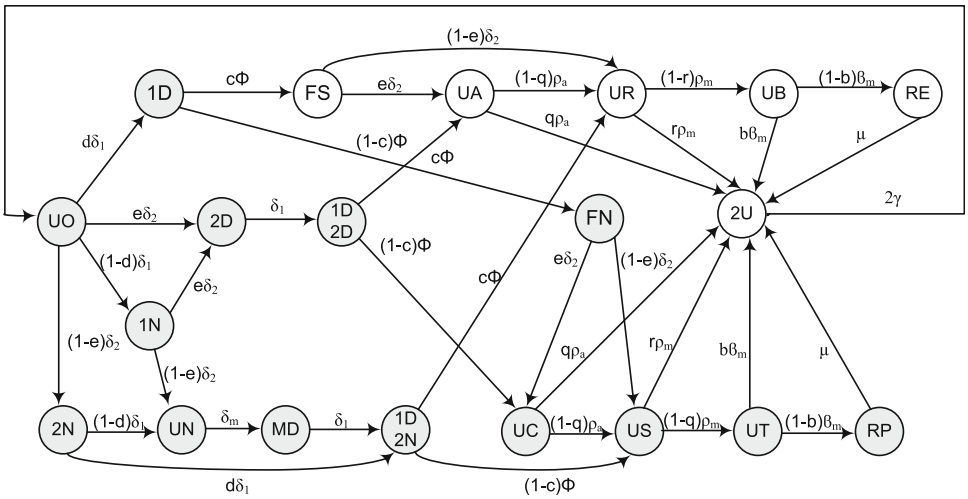

Fig. 6 Replication domain availability model. Adapted from Trivedi et al. (2008)

preference, and the behavior of the analysis algorithms. Some subsystems represented by fault tree leaves can have states where the system is up but with some components non-operational and undergoing repair. Failures are sometimes followed by successful reconfiguration ("covered") but sometimes bring the system down. Such systems are not easily modeled with fault tees, but can be modeled with Markov chains. The SIP system was modeled with several different Markov chain submodels of fault tree leaves. In other words, a hierarchical approach was used. For example, Fig. 6 shows the availability model for the two application servers in one replication domain (all six replication domains have the same Markov chain model). This model captures two different automated failure detectors, manual detector, automated and manual restarts, reboots and manual repair besides failover to the other application sever in the pair. Escalated levels of recovery and imperfect coverage for each phase of recovery are also modeled. In (Trivedi et al. 2008), sensitivity analysis of availability was carried out in order to indicate the failure types and recovery parameters that are most critical in their impact on overall system availability. Models in (Trivedi et al. 2008) and (Trivedi et al. 2010) were responsible for the sale of the system by IBM to a Telco customer.

6.2 Software rejuvenation schedule

Dohi et al. (2000) analytically derive the optimal software rejuvenation schedules maximizing the system availability for time-based rejuvenation. The aging behavior of the software systems as well as rejuvenation and repair are modeled as a semi-Markov process; the transition diagram for one of the models is depicted in Fig. 7.

A non-parametric statistical algorithm is then developed to estimate the optimal software rejuvenation schedule, provided that the sample data to characterize the system failure times is given. It is important to highlight that the proposed statistical algorithm is useful in determining the optimal rejuvenation schedule early in the operational phase; in contrast, the probability distribution of the system failure time cannot easily be estimated from a few data samples. Moreover, the non-parametric approach does not even require any assumption concerning the underlying failure time distribution. Finally, the estimators of the optimal software rejuvenation schedule have nice convergence properties.

7 Conclusions

We have discussed models for quantifying availability of a software system. We have considered reactive recovery techniques for Mandelbugs and availability models that incorporate these recovery techniques. For aging-related bugs, a powerful proactive recovery technique is rejuvenation. We discussed rejuvenation scheduling and availability

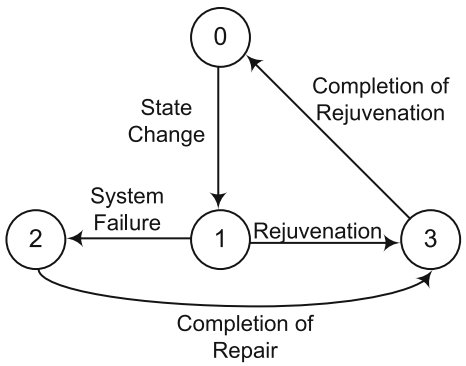

Fig. 7 Semi-Markov diagram. Adapted from Dohi et al. (2000)

models for software systems when rejuvenation is used to deal with aging.

References

Adams E (1984) Optimizing preventive service of the software products. IBM J Res Dev 28(1):2–14

Arlat J, Costes A, Crouzet Y, Laprie J-C, Powell D (1993) Fault injection and dependability evaluation of fault tolerant systems. IEEE Trans Comput 42(8):913–923

Avižienis A, Chen L (1977) On the implementation of N-version programming for software fault tolerance during execution. In: Proc. IEEE computer software and applications conference, Chicago, pp 149–155

Avritzer A, Weyuker EJ (1997) Monitoring smoothly degrading systems for increased dependability. Empir Softw Eng 2(1):59–77

Avritzer A, Bondi A, Grottke M, Trivedi KS, Weyuker EJ (2006) Performance assurance via software rejuvenation: monitoring, statistics and algorithms. In: Proc. international conference on dependable systems and networks 2006, Philadelphia, pp 435–444

Barlow RE, Campo R (1975) Total time on test processes and applications to failure data analysis. In: Barlow RE, Fussell J, Singpurwalla ND (eds) Reliability and fault tree analysis. SIAM, Philadelphia, pp 451–481

Bernstein L, Kintala C (2004) Software rejuvenation. CrossTalk 17(8):23–26

Bharadwaj R (2008) Verified software: the *real* grand challenge. In: Meyer B, Woodcock J (eds) Verified software: theories, tools, experiments. Lecture notes in computer science, vol 4171, Springer, Berlin, pp 318–324

Bolch G, Greiner S, de Meer H, Trivedi KS (2006) Queueing networks and Markov chains modeling and performance evaluation with computer science applications, 2nd edn. Wiley, New York

Candea G, Cutler J, Fox A (2004) Improving availability with recursive microreboots: a soft-state system case study. Perform Eval 56(1–4):213–248

Cassidy KJ, Gross KC, Malekpour A (2002) Advanced pattern recognition for detection of complex software aging in online transaction processing servers. In: Proc. international conference on dependable systems and networks, Washington, pp 478–482

Castelli V, Harper RE, Heidelberger P, Hunter SW, Trivedi KS, Vaidyanathan K, Zeggert WP (2001) Proactive management of software aging. IBM J Res Dev 45(2):311–332

Chen D, Selvamuthu D, Chen D, Li L, Some RR, Nikora AP, Trivedi KS (2002) Reliability and availability analysis for the JPL remote exploration and experimentation system. In: Proc international conference on dependable systems and networks, Bethesda, pp 337–344

Cisco Systems (2001) Cisco catalyst memory leak vulnerability. Document ID:13618, Cisco Security Advisory. http://www.cisco.com/warp/public/707/cisco-sa-20001206-catalyst-memleak.shtml. Accessed 22 Dec 2010

Devraj A, Mishra K, Trivedi KS (2010) Uncertainty propagation in analytic availability models. In: Proc. IEEE symposium on reliable distributed systems, New Delhi

Dohi T, Goševa-Popstojanova K, Trivedi KS (2000) Statistical non-parametric algorithms to estimate the optimal software rejuvenation schedule. In: Proc. 2000 Pacific rim international symposium on dependable computing, Los Angeles, pp 77–84

Dohi T, Goševa-Popstojanova K, Trivedi KS (2001) Estimating software rejuvenation schedule in high assurance systems. Comput J 44(6):473–485

Dumitras T, Srivastava D, Narasimhan P (2005) Architecting and implementing versatile depend-ability. In: Gacek C, Romanovsky A, de Lemos R (eds) Architecting dependable systems, vol III. Lecture notes in computer science, vol 3549, Springer, Berlin, pp 212–231

Garg S, Puliafito A, Telek M, Trivedi KS (1995) Analysis of software rejuvenation using Markov regenerative stochastic Petri net. In: Proc. sixth international symposium on software reliability engineering, Toulouse, pp 24–27

Garg S, van Moorsel A, Vaidyanathan K, Trivedi KS (1998) A methodology for detection and estimation of software aging. In: Proc. ninth international symposium on software reliability engineering, Paderborn, pp 283–292

Garg S, Huang Y, Kintala CMR, Trivedi KS, Yajnik S (1999) Performance and reliability evaluation of passive replication schemes in application level fault tolerance. In: Proc. 29th annual international symposium on fault tolerant computing, Madison, pp 15–18

Gray J (1986) Why do computers stop and what can be done about it? In: Proc. 5th symposium on reliability in distributed systems, Los Angeles, pp 3–12

Grottke M, Trivedi KS (2005a) Software faults, software aging and software rejuvenation. J Reliab Eng Assoc Jpn 27(7):425–438

Grottke M, Trivedi KS (2005b) A classification of software faults. In: Supplemental proc. sixteenth international IEEE symposium on software reliability engineering, Chicago, USA, pp 4.19–4.20

Grottke M, Trivedi KS (2007) Fighting bugs: remove, retry, replicate and rejuvenate. IEEE Comput 40(2):107–109

Grottke M, Trivedi KS (2008) Analysis of the escalated levels of failure recovery approach. Working paper, University of Erlangen-Nuremberg, Nuremberg

Grottke M, Li L, Vaidyanathan K, Trivedi KS (2006) Analysis of software aging in a web server. IEEE Trans Reliab 55(3):411–420

Grottke M, Matias R Jr, Trivedi KS (2008) The fundamentals of software aging. In: Proc. first IEEE workshop on software aging and rejuvenation, Seattle

Grottke M, Nikora A, Trivedi KS (2010) An empirical investigation of fault types in space mission system software. In: Proc. 2010 IEEE/IFIP international conference on dependable systems and networks, Chicago, pp 447–456

Hellerstein J, Diao Y, Parekh S, Tilbury DM (2004) Feedback control of computer systems. Wiley, New York

Hoffman G, Malek M, Trivedi KS (2006) A best practice guide to resource forecasting for the Apache webserver. In: Proc. Pacific rim dependability conference, Riverside, pp 183–193

Hong Y, Chen D, Li L, Trivedi KS (2002) Closed loop design for software rejuvenation. In: Proc. workshop on self-healing, adaptive and self-managed systems, New York

Horning JJ, Lauer HC, Melliar-Smith PM, Randell B (1974) A program structure for error detection and recovery. In: Lecture notes in computer science, vol 16, Springer, Berlin, pp 177–193

Hsueh M-C, Tsai TK, Iyer RK (1997) Fault injection techniques and tools. IEEE Comput 30(4):75–82

Huang Y, Kintala C, Kolettis N, Fulton N (1995) Software rejuvenation: analysis, module and applications. In: Proc. twenty-fifth international symposium on fault-tolerant computing, Pasadena, pp 381–390

Hunter SW, Smith WE (1999) Availability modeling and analysis of a two node cluster. In: Proc. 5th international conference on information systems, analysis and synthesis, Orlando

Kourai K, Chiba S (2007) A fast rejuvenation technique for server consolidation with virtual machines. In: Proc. international conference on dependable systems and networks 2007, Edinburgh, pp 245–255

Lanus M, Liang Yin, Trivedi KS (2003) Hierarchical composition and aggregation of state-based availability and performability models. IEEE Trans Reliab 52(1):44–52

Laprie J-C (ed) (1992) Dependability, basic concepts and terminology. Springer, New York

Laprie J-C, Arlat J, Béounes C, Kanoun K, Hourtolle C (1987) Hardware and software fault tolerance: definition and analysis of architectural solutions. In: Proc. 17th international symposium on fault-tolerant computing, Pittsburgh, pp 116–121

Lee I, Iyer RK (1995) Software dependability in the Tandem GUARDIAN system. IEEE Trans Softw Eng 21(5):455–467

Lindemann C (1998) Performance modelling with deterministic and stochastic Petri nets. Wiley, New York

Liu Y, Ma Y, Han J, Levendel H, Trivedi KS (2005) A proactive approach towards always-on availability in broadband cable networks. Comput Commun 28(1):51–64

Mainkar V, Trivedi KS (1996) Sufficient conditions for existence of a fixed point in stochastic reward net-based iterative methods. IEEE Trans Softw Eng 22(9):640–653

Marshall E (1992) Fatal error: how Patriot overlooked a Scud. Science 255:1347

Matias R Jr, Freitas Filho PJ (2006) An experimental study on software aging and rejuvenation in web servers. In: Proc. 30th IEEE annual international computer software and applications conference, Chicago, vol 1, pp 189–196

Matias R Jr, Trivedi KS, Maciel P (2010) Using accelerated life tests to estimate time to software aging failure. In Proc. IEEE international symposium on software reliability engineering, San Jose, pp 211–219

Matias R Jr, Barbetta PA, Trivedi KS (2010) Accelerated degradation tests applied to software aging experiments. IEEE Trans Reliab 59(1):102–114

Meeker WQ, Escobar LA (1998) Statistical methods for reliability data. Wiley, New York

Mendiratta VB (1999) Reliability analysis of clustered computing systems. In: Proc. ninth international symposium on software reliability engineering, Paderborn, pp 268–272

Mendiratta VB, Souza JM, Zimmerman G (2007) Using software failure data for availability evaluation. In: Designer and developer forum, GLOBECOM 2007, Washington

Montgomery DC (2004) Design and analysis of experiments, 6th edn. Wiley, New York

Narasimhan P, Dumitras T, Pertet S, Reverte CF, Slember J, Srivastava D (2005) MEAD: support for real-time fault tolerant CORBA. Concurr Comput Pract Exp 17(12):1527–1545

Nelson W (1982) Applied life data analysis. Wiley, New York

Nicol D, Sanders W, Trivedi KS (2004) Model-based evaluation: from dependability to security. IEEE Trans Dependable Secur Comput 1(1):48–65

Pertet S, Narasimhan P (2004) Proactive recovery in distributed CORBA applications. In: Proc. international conference on dependable systems and networks, Florence, pp 357–366

Pertet S, Narasimhan P (2005) Causes of failure in web applications. Carnegie Mellon University Parallel Data Lab Technical Report, CMU-PDL-05-109

Pietrantuono R, Russo S, Trivedi KS (2010) Online monitoring of software system reliability. In: Proc. dependable computing conference, Tokyo, pp 209–218

Raymond ES (1991) The new hacker's dictionary. MIT, Cambridge

Sahner RA, Trivedi KS, Puliafito A (1996) Performance and reliability analysis of computer systems. Kluwer, Boston

Sato N, Nakamura H, Trivedi KS (2007) Detecting performance and reliability bottlenecks of composite web services. In: Proc. ICSOC, Vienna

Shereshevsky M, Crowell J, Cukic B, Gandikota V, Liu Y (2003) Software aging and multifractality of memory resources. In: Proc. international conference on dependable systems and networks, San Francisco, pp 721–730

Silva L, Madeira H, Silva JG (2006) Software aging and rejuvenation in a SOAP-based server. In: Proc. fifth IEEE international symposium on network computing and applications, Cambridge, pp 56–65

Smith WE, Trivedi KS, Tomek L, Ackeret J (2008) Availability analysis of multi-component blade server systems. IBM Syst J 47(4):621–640

Tai A, Chau S, Alkalaj L, Hecht H (1999) On-board preventive maintenance: a design-oriented analytic study for long-life applications. Perform Eval 35(3–4):215–232

Tobias P, Trindade D (1995) Applied reliability, 2nd edn. Kluwer, Boston

Tomek L, Trivedi KS (1991) Fixed-point iteration in availability modeling. In: Dal Cin M (ed) Proc. fifth international GI/ITG/GMA conference on fault-tolerant computing systems, Springer, Berlin, pp 229–240

Trivedi KS (2000) Availability analysis of Cisco GSR 12000 and Juniper M20/M40. Cisco Technical Report

Trivedi KS (2001) Probability & statistics with reliability, queueing and computer science applications, 2nd edn. Wiley, New York

Trivedi KS, Vasireddy R, Trindade D, Nathan S, Castro R (2006) Modeling high availability systems. In: Proc. Pacific rim dependability conference, Riverside, pp 11–20

Trivedi KS, Wang D, Hunt DJ, Rindos A, Smith WE, Vashaw B (2008) Availability modeling of SIP protocol on IBM Websphere. In: Proc. pacific rim dependability conference, Taipei, pp 323–330

Trivedi KS, Wang D, Hunt J (2010) Computing the number of calls dropped due to failures. In: Proc. IEEE international symposium on software reliability engineering, San Jose, pp 11–20

Vaidyanathan K, Trivedi KS (2005) A comprehensive model for software rejuvenation. IEEE Trans Dependable Secur Comput 2(2):124–137

Vaidyanathan K, Harper RE, Hunter SW, Trivedi KS (2001) Analysis and implementation of software rejuvenation in cluster systems. In: ACM SIGMETRICS conference on measurement and modeling of computer systems, Cambridge, USA, pp 62–71

Vilkomir SA, Parnas DL, Mendiratta VB, Murphy E (2005) Availability evaluation of hardware/software systems with several recovery procedures. In: Proc. twenty-ninth annual international computer software and applications conference, Edinburgh, UK, pp 473–478

Wang D, Trivedi KS (2009) Modeling user-perceived reliability based on user behavior graphs. Int J Reliab Qual Saf Eng 16(4):303–330

Wang D, Fricks R, Trivedi KS (2003) Dealing with non-exponential distributions in dependability models. In: Kotsis G (ed), Performance evaluation—stories and perspectives, Österreichische Computer Gesellschaft, Wien, pp 273–302

Winslett M (2005) Bruce Lindsay speaks out. In: ACM SIGMOD Record, June 2005, pp 71–79

Xie W, Hong Y, Trivedi KS (2005) Analysis of a two-level software rejuvenation policy. Reliab Eng Syst Saf 87(1):13–22