Recovery from Software Failures Caused by Mandelbugs

Michael Grottke[1], *Member*, *IEEE*, Dong Seong Kim[2], *Member*, *IEEE*, Rajesh Mansharamani[3], Manoj Nambiar[4], *Senior Member*, *IEEE*, Roberto Natella[5], *Member*, *IEEE*, Kishor S. Trivedi[6], *Fellow*, *IEEE*
[1]*Department of Statistics and Econometrics, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany*
E-mail: Michael.Grottke@fau.de
[2]*Department of Computer Science and Software Engineering, University of Canterbury, New Zealand*
E-mail: dongseong.kim@canterbury.ac.nz
[3]*Freelance Consultant in Performance Engineering, India*
E-mail: rajesh.mansharamani@gmail.com
[4]*Tata Consultancy Services, India*
E-mail: m.nambiar@tcs.com
[5]*Federico II University of Naples, Italy*
E-mail: roberto.natella@unina.it
[6] *Duke High Availability Assurance Lab, Department of Electrical and Computer Engineering, Duke University, USA*
E-mail: ktrivedi@duke.edu

***Abstract*—Software failures are still a major concern in mission- and enterprise-critical contexts, despite significant efforts spent in software testing. In fact, while software testing is effective against easily-reproducible bugs (*Bohrbugs*), it is considerably less suitable for dealing with bugs that lead to hard-to-reproduce failures (*Mandelbugs*). On the positive side, the elusive nature of Mandelbugs provides opportunities for failure recovery, which are investigated in this paper. Based on real cases of Mandelbugs in eleven Information Technology (IT) systems running in production, the paper proposes a model that describes the recovery processes in IT systems. It then presents closed-form expressions, and a numerical analysis, of the mean time to recovery, and the software (un)availability. This analysis allows the designer to compare recovery strategies, as well as to determine the parameters having a high influence on the efficacy of recovery from failures caused by Mandelbugs.**

Index Terms –Mandelbugs, mean time to recovery, semi-Markov model, sensitivity analysis, software quality.

ACRONYMS AND ABBREVIATIONS

ATM	Automated teller machine
DCE	Data communication equipment
IT	Information Technology
JVM	Java Virtual Machine
MTTF	Mean time to failure
MTTR	Mean time to recovery
ODC	Orthogonal Defect Classification
OS	Operating system
SSUA	Steady-state unavailability
TCP	Transmission Control Protocol

NOTATION

$E[D_{ad}]$	Mean time to automatic failure detection
$E[D_{md}]$	Mean time for manual failure detection
$E[D_{dg}]$	Mean time for failure diagnosis
$E[D_{rs}]$	Mean time for restart of software component or service

$E[D_{rb}]$	Mean time for reboot of hardware server or virtual machine
$E[D_{rc}]$	Mean time for reconfiguration of software component or service
$E[D_{hf}]$	Mean time to carry out hot-fix
$E[D_{bf}]$	Mean time to carry out bug-fix
p_{ad}	Probability of automatic failure detection
p_i	Probability that a failure is caused by the bug type *i*
p_{4hf}	Probability of correct hot-fix
d_{ij}	Probability of diagnosing the bug type *j*, given a bug type *i*

I. INTRODUCTION

Reliability is a key quality attribute of Information Technology (IT) software systems; and weeks, or even months, are spent in the functional, integration, and performance testing of mission- and enterprise-critical applications [1]. Despite these efforts, a number of bugs remain in the software released to users, and result in IT software failures. Such failures affect end users and operations personnel, leading to prolonged outages, increased maintenance costs, and a low perceived quality: the recent issues with the *HealthCare.gov* portal [2], and several accidents in mission-critical systems [3], are examples of how severe the consequences of IT software failures can be.

For these reasons, recent research [4], [5], [6], [7] investigated the nature of software failures, and countermeasures against them. The authors found that bugs can be classified into two categories: Bohrbugs, and Mandelbugs. A Bohrbug is a fault that is easy to isolate, and whose manifestation is consistent under a well-defined set of conditions. Once a Bohrbug has caused a failure, it is thus a relatively simple task to track it down in the software code, and to remove it. In contrast, the behavior of a Mandelbug may appear non-deterministic, because the same set of input data seems to make it cause a failure at some times, but not at others. Therefore, failures caused by Mandelbugs are usually difficult to reproduce. These characteristics of a Mandelbug may be traced back to the complexity of its activation or error propagation or both. One possible cause can be a time lag between the fault activation and the failure occurrence. Another possible cause can be the influence of indirect factors, such as the interactions of the software application in which the Mandelbug is located with its system-internal environment (hardware, operating system, or other applications), the timing of inputs and operations (relative to each other), and the sequencing of inputs and operations. A race condition [8] is a typical example for a problem caused by a Mandelbug. Concurrent programs, which are currently developed at a high rate due to the advent of multi-core servers and distributed architectures, are highly prone to containing Mandelbugs [9], [10].

While Bohrbugs can be effectively counteracted through testing and debugging, it is a daunting task to find and remove Mandelbugs because of their elusive nature. Of course, once a Mandelbug has been understood, it can be fixed in the code. However, for a failure caused by a Mandelbug, it is usually difficult to determine what is causing the failure. For instance, to deal with a race condition, a developer should first identify the sequencing of concurrent processes that cause the race, and then fix it through the use of proper sequencing or locking techniques.

On the positive side, the elusive nature of Mandelbugs provides opportunities for fault-tolerance strategies: as the conditions that trigger Mandelbugs (often related to event timing or to the environment) are volatile, theses bugs usually do not manifest again after restarting the failed system or component, and retrying the failed operation a second time (e.g., when the environment is slightly different) [6], [11], [12], [13], [14]. For instance, retrying a failed operation will succeed with a high probability in the case of a race condition, because the process scheduling by the operating system (OS) is likely to be different during the second execution. Other kinds of failures can be tolerated by rebooting the failed machine, where the reboot cleans up erroneous data that were corrupted by a Mandelbug, or by reconfiguring the system with different parameter settings, such as timeouts and size of buffers. IT system designers know from experience that they can exploit this kind of *environmental diversity* to quickly recover from software failures, thus improving the availability of their systems in a cost-effective way, without resorting to costly design diversity, or to fixing the Mandelbug which caused the failure. By contrast, restarting, rebooting, or reconfiguring a system will not succeed against Bohrbugs, because these bugs will again produce a failure when the same input data are submitted to the system for a second time [6].

This paper presents an analytic model for evaluating and comparing recovery strategies in IT systems, by taking into account the different nature of Mandelbugs and Bohrbugs. The interest in building and analyzing such models is threefold. First, we obtain a systematic understanding of how IT systems behave just by composing the model. Second, we get to derive measures of interest out of the model, such as the mean time to recovery, and steady-state software (un)availability. Third, and most importantly, the

model allows us to (i) compare the relative benefits of different recovery actions, and of recovery strategies based on these actions; and (ii) find recovery bottlenecks so as to be able to pinpoint the most critical system parameters for improving the mean time to recovery, through numerical and parametric sensitivity analysis. We deliberately designed a model that can easily be understood, configured, and used by IT practitioners; and we strived to make realistic assumptions about IT systems and failure recovery strategies. The main contributions of this paper include concrete examples of Mandelbugs found in real IT systems, a detailed model of recovery from failures due to such bugs, and closed-form solutions of the mean time to recover from them, as well as the resulting software (un)availability.

This paper is organized as follows. In Section II, we discuss related work in the area of bug analysis and analytic modeling. Section III presents several examples of Mandelbugs that have been found in eleven IT systems during operations, as well as a classification of the Mandelbugs and the recovery actions taken. In Section IV, we then develop an analytic model for recovery from failures caused by Mandelbugs, based on the insight gained in the previous section. The mean time to recover implied by this model is derived in Section V, and a sensitivity analysis is carried out. Finally, Section VI concludes the paper.

II. RELATED WORK

Software bug analysis has been a common theme in a variety of IT project implementations. Its main goal has been to reduce the impact and costs of bugs in software development projects, by avoiding the introduction of bugs, and by improving bug removal in the early stages of development. These analyses resulted in guidelines and practices ranging from test case design to root cause analysis, checklists, as well as programming and debugging practices. Among the many studies on bug analysis presented in the past, the following studies and their applications are worthy of note.

- Basili and Perricone [15] classified software fault types into *initialization*, *control structure*, *interface*, *data*, and *computation* faults; and further divided those into *omission,* and *commission* faults; obtained their frequency during software development; and pointed out their relationships with a number of factors such as code complexity (e.g., cyclomatic complexity), code reuse, and developers' experience.
- Perry and Evangelist [16] focused their analysis on *interface faults* (i.e., faults associated with structures outside the module's local environment, but which the module used), as they represent a significant part of faults in large systems. They classified the interface faults of a real-time system in a detailed way (e.g., disagreements on module's functionalities, misuse of interfaces, violation of data constraints), and provided suggestions to mitigate each category of interface faults, including the improvement of the training of inexperienced developers, and of design and verification activities.
- Orthogonal Defect Classification (ODC), proposed by Chillarege *et al*. [17], is a classification scheme for obtaining insights into the development process from the distribution of *defect attributes*. Attributes include the defect type, which reflects the fix made by the programmer. The definition of defect types is based on the cause-effect relationship between the defect and the development phase in which it originated, enabling the identification of the stages of the process that require more attention: for example, an excessive number of *Function* defects (that is, missing or incorrect functionalities of the system) indicates that the high-level design phase should be improved.
- Several books and technical papers from software practitioners [18], [19], [20], [21] surveyed common bugs occurring in the coding phase (e.g., syntactic and semantic faults due to misunderstandings of programming language constructs), and suggested programming practices to prevent them.

In this paper, we utilize the classification of bugs into Mandelbugs (and their antonym, Bohrbugs) as in Grottke and Trivedi [5], [6], [7]. This classification is related to the *non-reproducibility* of bugs due to the transient nature of failures caused by them. In his seminal work, Gray [22] observed that such a behavior, once attributed to hardware faults [23], [24], was also prevalent among software faults. He even hypothesized that most software failures occurring in mature, high-availability systems are *transient*, as rigorous testing activities and years of production usage removes most bugs except those related to complex conditions (such as race conditions, overloads, and device faults), which manifest themselves at (apparently) random times. Such bugs are thus difficult to reproduce and to debug by developers [12]. This transient behavior is caused by the subtle relationships between the bugs and the program's environment, such as the timing of events, and the state of OS resources.

To account for these aspects, Grottke and Trivedi [5], [6], [7] defined Mandelbug as a bug where either (i) the propagation of the error (i.e., the incorrect internal state of the running system) generated by the bug involves several error states or several subsystems or both before turning into a failure (i.e., it is not simply reproducible by repeating only the latest events and inputs that occurred just before the failure), or (ii) the occurrence of a failure is influenced by indirect, difficult to control factors, such as the program's environment (e.g., the OS, other concurrently executing programs, or the hardware), the timing of inputs and operations (relative to each other), and the sequencing of inputs and operations. If one of these two conditions holds, then failures caused by the bug are typically difficult to reproduce. This classification scheme has been adopted and validated in several studies (discussed below), and has been refined over time by considering subtypes of Mandelbugs [25].

Differing from the bug classification studies mentioned above, the Mandelbugs-Bohrbugs classification has important implications on the design of fault-tolerant architectures, as discussed in the introduction. Therefore, recent research studies have focused on understanding the different types of Mandelbugs, on proposing and evaluating techniques and strategies to tolerate

failures caused by them, and on investigating the relationships between the type and the domain of a software system and the occurrence of Mandelbugs.

- Huang *et al.* [26] observed that a special type of Mandelbug, affecting *continuously-running* applications and called *aging-related (Mandel)bug* by Grottke *et al.* [7], can be mitigated by a *proactive* method that is applied before the occurrence of a failure, namely *software rejuvenation*. This is the case, for instance, of bugs causing memory leaks and bloating, which lead to the gradual exhaustion of memory, and to a sudden stop of the application. In this scenario, performing a restart of the application at a convenient time (e.g., during a scheduled maintenance period) frees the leaked memory, and potentially avoids the occurrence of a failure. Alonso *et al.* [33] presented an experimental comparison of software rejuvenation strategies, and Cotroneo *et al.* [27] conducted a comprehensive survey of software aging and rejuvenation studies. Aging-related bugs found in open-source software were analyzed in [28] and [29].
- Based on the definitions due to Grottke and Trivedi [6], subsequent studies analyzed the occurrence of Mandelbugs in real systems, by classifying and analyzing Mandelbugs found in the field and during pre-release testing. The analysis of software anomalies in NASA missions conducted by Grottke *et al.* [7] revealed that, although Bohrbugs represented the majority of faults, Mandelbugs accounted for a substantial share, in the 20% to 40% range. Moreover, these authors showed that the proportions of Bohrbugs (and, consequently, of Mandelbugs) for different missions seem to stabilize around almost the same value. They also found that aging-related bugs represent a non-negligible share of faults, even in long-running mission-critical software (4.4%). More recent studies analyzed the fraction of Mandelbugs in systems of different types, and from different domains, including a large, distributed defense system [30]; the Linux, MySQL, Apache HTTPD, and Apache AXIS open-source projects [25]; and two enterprise products [31]. Even if these studies made similar conclusions about the general trends of Mandelbugs and Bohrbugs, they pointed out that the proportion of Mandelbugs is influenced by the domain and the type of software, where embedded and operating system software tend to exhibit a higher proportion of Mandelbugs, while this proportion is lower for middleware and enterprise software. The stage of the software lifecycle also has an influence on the proportion of Mandelbugs, as software in the pre-release testing stage tends to exhibit a higher proportion of Bohrbugs than software in the post-release stage. Finally, empirical evidence [7], [25] showed that Bohrbugs and Mandelbugs are perceived by users and developers as similarly important according to a severity scale, while Chillarege [32] pointed out that Mandelbugs impact different product areas than Bohrbugs; in particular, Mandelbugs significantly affect the perceived availability (i.e., continuity of service), but they do not seem to have an impact with respect to the functional requirements of the system.
- Among the several fault tolerance techniques that have been proposed for tolerating failures caused by Mandelbugs, we mention the *microreboot* [11], which reacts to failures by restarting selected sub-components of an application, thus avoiding a full application restart, and reducing the time to recovery; NT-SwiFT [13], which provides several tools for developing fault-tolerant cluster systems, including tools for detecting process and nodes failures, and for the checkpoint and rollback of failed processes; and *Rx* [14], which re-executes a failed program under a *modified* environment (e.g., by changing the timing of asynchronous events, the thread scheduling, and the allocation of data structures in heap memory) to increase the likelihood of masking Mandelbugs. Because re-execution in a different environment is used in all of the above, we refer to these approaches as *environmental diversity* in contrast with the classical software fault tolerance techniques based on design diversity.

These studies pointed out the importance of Mandelbugs in mission-critical systems, highlighting that a large number of Mandelbugs affect such systems, and that availability and fault tolerance can be significantly improved by adopting fault recovery strategies against failures caused by Mandelbugs.

Many models for tuning software rejuvenation to counteract the effects of aging-related bugs have been published in [26], [34], [35], [36], [37], [38], and [39]. But only a few studies have been conducted on the modeling of the more general class of Mandelbugs, and on the analysis of related recovery strategies. Models for system availability have been proposed in the literature for specific instances of platforms. Garg *et al.* [40] proposed analytical models of cold and warm replication schemes provided by the SwiFT and DOORS fault-tolerance technologies, and derived closed-form expressions for availability, throughput, and probability of request loss in the presence of transient software failures. Trivedi *et al.* [41] analyzed a Session Initiation Protocol (SIP) for IBM Websphere; the paper took into account non-aging-related Mandelbugs, and recovery from failures due to such bugs. Grottke and Trivedi [42] carried out a detailed model-based study of systems that can be recovered using various techniques.

The recovery model presented in this paper is inspired by that in [41] and [42]. While the model in [42] features a very general topology, the model presented in this paper is tailored to the specifics of recovery from Mandelbugs in IT systems. Compared to our preliminary work [43], we provide an enhanced version of the recovery model. The new model uses a customized recovery strategy by tuning failure detection, failure diagnosis, and recovery actions in the model, thus making it useful for comparing alternative recovery strategies. Moreover, the new model is more suitable for including information from field failure data on Mandelbugs (in terms of type and frequency), thus making it easier to analyze recovery strategies under realistic scenarios. We

use the model to compare three recovery strategies in the presence of Mandelbugs, and provide insights about the effectiveness of recovery strategies and about the most critical factors for effective recovery. The analysis is performed by taking into account field failure data about Mandelbugs described in [25], from which we derive three realistic scenarios for the analysis of recovery strategies. Combining recovery strategies with software rejuvenation is not considered in this paper, and is left for future research.

III. ANALYSIS OF MANDELBUGS IN REAL IT SYSTEMS

For this study, we considered failures due to Mandelbugs that occurred in eleven IT systems. We analyzed Mandelbugs that were not found during testing, and that affected the IT systems in production even after several months after their release. Two of the authors of this paper, who were involved in the review and maintenance of these IT systems, had first-hand access to design documents, test suites, and data about IT system failures and bugs. With the support of the IT staff, we analyzed the processes that they adopted for failure detection, recovery, and fix.

The eleven IT systems span several domains, and include a stock exchange system, a foreign exchange trading system, a government's tax information system, a product for front office order routing, and an online quizzing application in an IT organization. The remaining projects involve a large bank, a clearing corporation, two brokerages, a large pharmacy, and a large telecom vendor. Except for the quizzing application, these IT systems are critical projects for the core business of their companies. All the IT systems had up to thousands of concurrent users, and were subject to peak loads. In almost all cases, no unplanned downtime was allowed, because it would result in substantial business losses.

Our analysis was focused on the technical aspects of the IT failures, and was aimed at identifying root causes and the actual recovery actions adopted by the IT operations staff. In the following subsections, we first provide examples of Mandelbugs, along with the respective recovery actions taken. We then provide a classification of the Mandelbugs found by our analysis, and the recovery methods employed to deal with them.

A. Examples of Mandelbugs

The following six examples are Mandelbugs found in our analysis, and are representative of the failures and recovery actions performed by the IT operations staff.

- **Bug #1: Out-of-sync request elaboration in a stock exchange system.**
  - *System description*: The system managed incoming requests from traders for entering, modifying, and cancelling orders. Requests were processed by a pipeline of two stages. First, requests were queued for validation, and a confirmation was sent to the user if the request was valid (e.g., the requested order was a valid one). Second, valid requests were then queued for processing.
  - *Bug description and manifestation*: The system was affected by a timing issue involving requests for order entry and requests for modification of the same order. The root cause of the failure was that new orders were inserted into an order book only at the end of the pipeline, instead of inserting them just after validation. Thus, if the order entry request had been validated but it was still in the processing queue, then modification requests for that order were erroneously rejected, even if the requested order had already been validated. The failure occurred sporadically, for instance under rare overload conditions, and manifested itself to traders as a failure of modification requests.
  - *Recovery action*: The failure could be recovered by re-issuing a modification request after a few seconds.

- **Bug #2: Fragmentation of a large logistics company database.**
  - *System description*: The system had a primary database for on-line transaction processing, and an archival database for old data. A nightly batch job moved data from the primary to the archival database in order to improve performance. To avoid unavailability, the data was archived without shutting down the database, by using delete queries on the primary database.
  - *Bug description and manifestation*: The delete queries performed at run-time caused the fragmentation of the primary database, thus affecting its performance. In some cases, the archival job did not complete on time, and interfered with jobs executed in the morning, which updated the status of shipments.
  - *Recovery action*: The system recovered from the performance degradation after a daily coalescence of database indexes was carried out to defragment them.

- **Bug #3: Front-end screen unresponsiveness in a large telecommunications system.**
  - *System description*: Users accessed the system through front-end screens. When a user initiated a new session, a small temporary file was created on a server.
  - *Bug description and manifestation*: The temporary file was never cleaned up at the end of a session, causing the accumulation of thousands of temporary files in the file system of the server. Therefore, the server became slower, and front-end screens started to freeze at random times. Screen freezes kept occurring even after a reboot of the server, at an increasing frequency.

- *Recovery action*: Developers introduced a background utility to periodically move temporary files away from the server, and to delete them later.

- **Bug #4: Memory exhaustion in a large government tax information system.**
  - *System description*: The system was adopted by companies to submit income records of their employees. Users uploaded files with records to a Java Web application.
  - *Bug description and manifestation*: The Java virtual machine (JVM) running the application crashed due to the exhaustion of JVM heap memory. As the heap memory consumption increases over time, the probability of crash increased as well, and a crash occurred when a company uploaded a large file.
  - *Recovery action*: System administrators had to increase the JVM heap size to allow the upload of large files.

- **Bug #5: Crash of an X.25 data communication equipment (DCE).**
  - *System description*: An X.25 DCE is a network interface to connect to X.25 networks. In this case, the DCE was produced by a large telecom equipment vendor, and was based on a general-purpose CPU running a proprietary real-time OS. The DCE was updated with a new software feature, and this feature was subject to high packet traffic.
  - *Bug description and manifestation*: The software running on the DCE crashed under certain load conditions that invoked the new feature many times. The new feature used a function for copying memory areas (similar to the *memcpy*() function in the C language), but was copying twice the amount of data necessary to be copied. As a result, a shared linked list was getting corrupted. There was no immediate segmentation fault as the process address space was not violated. A failure occurred when another unrelated process tried to access this list, causing a DCE crash.
  - *Recovery action*: The failure could be recovered by re-initializing the X.25 DCE, and was eliminated in production once the root cause was determined.

- **Bug #6: Network communication failure of automated teller machines (ATMs).**
  - *System description*: An ATM connected to a server using an X.25 network. The X.25 DCE interface exchanged acknowledgements with the server to control the data flow.
  - *Bug description and manifestation*: The DCE did not correctly send acknowledgements when a bit variable was erroneously set to one, causing a communication failure. The failure occurred randomly depending on the state of memory, because the bit variable was not explicitly initialized to zero by the ATM software.
  - *Recovery action*: The failure could be recovered by restarting the ATM software, and was ultimately fixed by initializing the bit variable.

B. Classification of Mandelbugs

In our analysis of the eleven IT projects, we found a total of 38 Mandelbugs. We recognized that these Mandelbugs can be grouped into classes, which are summarized in Table I. An example of a *Lag*-type Mandelbug is represented by Bug #1 described above, where a timing issue caused an inconsistent state of the pipeline, and the failed request could be performed successfully after a few retries. Bugs #2 through #5 are examples of *aging-related* Mandelbugs, where Bug #2 and Bug #3 were dealt with by cleaning up system resources, while Bug #4 required a reconfiguration of a system parameter to resume operations. Bug #5 and Bug #6 were resolved after the root cause had been identified, but only after an initial downtime.

Table I
Classification of Mandelbugs (extended from Table I of [43])

Type of Mandelbugs	Explanation	No. of Bugs
Aging:	Failure rate increases or performance degrades over time, e.g., due to resources being depleted.	
• Memory Leak	Memory allocations and objects not released even if not required.	3
• Cursor Leak	Open database cursors not closed.	2
• TCP Aging	Degraded performance after a certain number of Transmission Control Protocol (TCP) connections have been opened.	1
• Numeric Overflow	Numeric quantity like a sequence number overflowing.	1
• Fragmentation	Performance degradation due to creation of holes in database files with frequent inserts and data purging.	1
• Memory Trampler	Shared data structures corrupted by a participant process.	1

• Network Equipment	Network switch or router malfunctions such as randomly garbling some bits or sending duplicate packets occur at an increasing rate.	2
Race Condition	Sequence of access changes due to concurrency and no proper synchronization primitives.	5
Lag	The state of a given component is out of sync with another for a small period of time, usually because of asynchronous processing.	7
Overload	Component failure or very poor performance due to a surge in workload for a small period of time.	2
Limit	Failure when a limit is reached such as the maximum amount of memory allocated or when load increases beyond a certain configured threshold without aging.	5
Timeout	A transaction or a session or a TCP connection times out causing a user request to be rejected.	4
Abort	Normal operations resume when a given request being processed is aborted. Reason for the request to hold up system processing may not be known.	2
Retry	Reason for failure unknown, but retrying the operation succeeds.	1
Uninitialized Bit	Malfunction due to assumption that un-initialized bit is set to zero.	1

C. *Recovery methods*

From the descriptions of the 38 Mandelbugs analyzed, we identified the recovery actions adopted to handle them. These recovery actions can be grouped into a few categories, which are representative of actions taken by IT operations staff to reduce unplanned downtime in business-critical applications. In our experience, these recovery actions are also adopted outside of these eleven IT systems. The four approaches most frequently employed to recover a system in the wake of a failure caused by Mandelbugs are as follows[1].

- **Restart:** This recovery action restarts a software component or service, which may require a few seconds, and which can only affect the transactions that the restarted component was processing. Moreover, this recovery action can be automated using IT management systems [44], [45]. In our experience, most failures due to aging- and non-aging-related Mandelbugs can be recovered with a software restart.
- **Reboot:** This recovery action involves the reboot of a hardware or virtual machine, and can require a few minutes to complete. This action is necessary when the failure involves system-wide resources (e.g., a shared data structure), for which a simple component or service restart does not suffice.
- **Reconfigure:** This recovery action changes a parameter of the hardware, virtual machine, or application before performing a restart or a reboot. The change of parameters often increases limits for system resources or timeout values, or involves a cleanup of system resources to improve performance. The *Abort*-type Mandelbug listed in Table I constitutes a special case: here, reconfiguration involves the removal of the currently-processed request. Reconfiguration can require up to several minutes, depending on whether the parameter tuning is done at the application (e.g., increasing timeouts of a software service), or at the system level (e.g., increasing memory and storage of a virtual machine, or migrating a service to another physical machine with a different hardware), and whether the recovery is supervised by automated software management tools.
- **Hot-fix:** A hot-fix is a minor change to the source code of the IT system in production, or to the system software (e.g., an update of the OS, of the run-time system, and of library code). A hot-fix is created and applied in a short time (up to a few hours) without extensive testing of the change. The former type of hot-fix in the source code of the IT system can remove the root cause of a failure (e.g., by fixing a bug), or can avoid its effects (e.g., by retrying failing operations); the latter kind of hot-fix in system software can also mitigate the effects of Mandelbugs (e.g., by introducing periodic garbage collection of OS resources).

In the case of failures that were not resolved by any of these four methods, a regular bug-fixing process with thorough testing and code debugging was needed to avoid the failures. Bug-fixing requires a few days, depending on the complexity of the bug underlying the failure. The bug-fix is included in the next software release, along with other changes and new features.

IV. MANDELBUG RECOVERY MODEL

In our recovery model, we consider an IT system in production, which is comprised of a set of software components (application or platform) deployed on underlying hardware or virtual machine technology. It is assumed that (i) software components can be restarted, (ii) software components can be reconfigured with different parameter settings such as buffer pool

[1] For the *retry* type of bug, the user behavior will come back to normal after a few retries, and nothing needs to be done to the IT system.

sizes or sort area sizes in database systems, and (iii) a server can be rebooted either automatically or manually. These assumptions are very realistic, and form the basis of the recovery model.

We depict the failure recovery behavior of an IT system during its operational phase by means of a flowchart, as shown in Fig. 1, and then explain the flowchart. The flowchart depicts the actions taken for recovery after a failure has occurred. The failure may either be detected automatically by the system (which we refer to as *automatic detection*), usually with enterprise system management tools, or it may be detected manually (which we refer to as *manual detection*). Detection is then followed by an investigation about the problem (which we refer to as *diagnosis*); and, finally, by one or more actions that are selected by diagnosis to recover from the failure (which we refer to as *recovery actions*). Depending on the type of bug that caused the failure (Bohrbug or Mandelbug), the recovery process follows one of four different branches of our model. The model distinguishes between four types of bugs, as follows.

- **Restart-maskable Mandelbugs**: Mandelbugs that can be masked by restarting the software component or service affected by the failure. Other recovery actions, such as reconfiguring or rebooting, will also mask the fault, because they involve a restart.
- **Reboot-maskable Mandelbugs**: Mandelbugs that require a hardware or virtual machine reboot to avoid the re-occurrence of failure.
- **Reconf-maskable Mandelbugs**: Mandelbugs that require a parameter tuning to avoid the re-occurrence of failure; in this case, a simple restart or reboot will not suffice.
- **Bohrbugs, and other types of Mandelbugs**: Bugs that cannot be masked by retrying the failed operation after a restart, reboot, or reconfiguration. They include Bohrbugs, which consistently manifest themselves when the operation is retried, and Mandelbugs influenced by environmental conditions that still persist after a restart, reboot, or reconfiguration. These bugs require an in-depth investigation of the issue, and a human intervention to remove the root cause of the failure, such as a bug-fix.

The model includes a distinct branch for each of these four types of bugs. In each branch, the recovery actions described in Subsection III.C are attempted (according to some recovery strategy, as discussed later in this section); and, depending on the type of bug underlying the failure, these recovery actions lead to different outcomes. For instance, in the case of restart-maskable Mandelbugs (leftmost nodes in the flowchart), every recovery action eventually brings the system into a correct state, because every action (restart, reboot, reconfiguration, fix) suffices to mask this kind of bug. In the case of Bohrbugs (rightmost nodes in the flowchart), only a fix can mask the failure, while the other recovery actions (restart, reboot, reconfiguration) are not beneficial for this type of bug, and should eventually be followed by a fix by developers.

Note that, in the cases of restart, reboot, or reconfiguration-maskable Mandelbugs, a hot-fix suffices to mask a failure, because a hot-fix should include a restart or reboot to apply the change, along with a change of parameters in the code for reconfigurations (such as resource thresholds and timeouts). A hot-fix thus includes a reconfiguration or a reboot to re-initialize the system and mask a failure. Instead, in the case of Bohrbugs, there is a chance that a hot-fix is not enough to correct the bug, and that the failure still persists (in the other cases, the restart, reboot, or reconfiguration performed by the hot-fix was enough to assure that the failure would not persist, even if the hot-fix is not able to remove the root cause of the problem). Thus, in the case of Bohrbugs, a regular bug-fix may need to be developed, tested, and deployed, which may require several days of work.

Before the recovery actions, we introduce a *diagnosis phase*, in which either developers or automated tools select a recovery action among restart, reconfiguration, reboot, or fix. Then, depending on the bug type, the action can successfully bring the system to a correct state, or the recovery action may not be successful, and the system may still exhibit a failure. In the latter case, we assume that another recovery action is attempted, by performing another action with both a higher duration and a higher likelihood to avoid a failure (e.g., if a restart fails, then a reboot is attempted; rebooting takes more time, but recovers from failures due to both restart-maskable and reboot-maskable Mandelbugs).

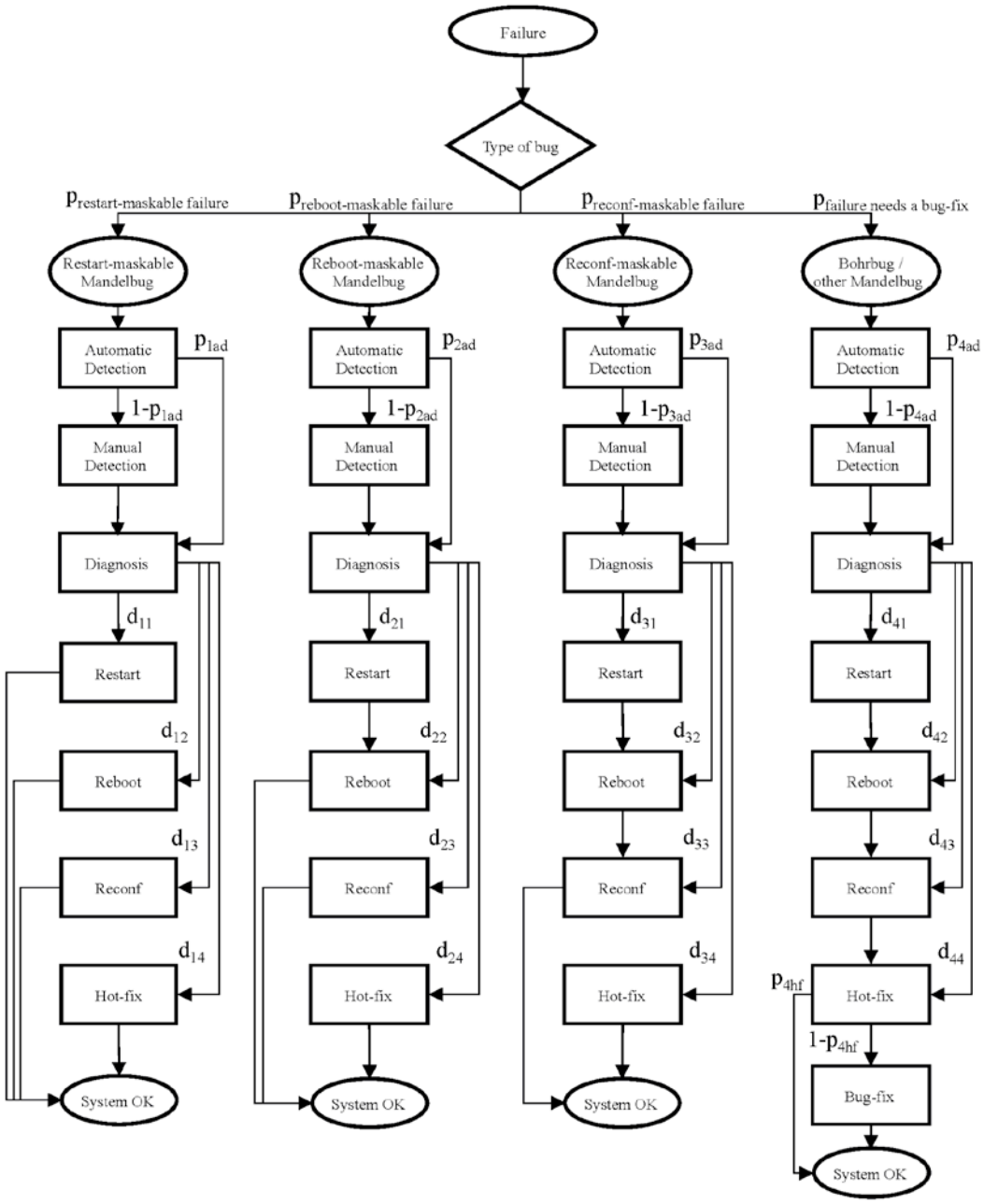


Fig. 1. Flowchart of Recovery after a Failure.

The flowchart allows us to model the different strategies that can be adopted in IT organizations to deal with software failures. A first strategy (which can be referred to as *diagnosis-based recovery*) is to investigate the failure causes, for instance by looking at application and OS logs, to diagnose the root cause of the failure, and to perform a recovery action according to diagnosis. The diagnosis phase can include the time to collect and analyze failure data, and to decide and approve a recovery action. It is important to note that, in our model, the underlying bug type is not known before diagnosis and recovery take place, and that it is possible that diagnosis does not select the recovery action best suited to counteract the failure. In other words, there can be cases in which diagnosis selects a recovery action that does not succeed, while in other cases the selected action may require more time than other, equally effective actions.

The diagnosis process (including the selection of correct and incorrect recovery actions) is modeled by the probabilities d_{ij}, which represent the conditional probability of selecting the *j*-th recovery action given that the failure was caused by the *i*-th type of bug, where *i*={ 1=*restart-maskable Mandelbug*, 2=*reboot-maskable Mandelbug*, 3=*reconf-maskable Mandelbug*, 4=*Bohrbug* } represents the type of bug underlying the failure, and *j*={ 1=*restart*, 2=*reboot*, 3=*reconfigure*, 4=*hot-fix* } represents the selected

recovery action. For example, consider i=2 (i.e., a reboot-maskable Mandelbug caused the failure): the diagnosis will select the optimal recovery action (i.e., a reboot, represented by j=2) with probability d_{22}. With probability d_{21}, the diagnosis will select a restart, which will not recover from the failure, and will need to be followed by a reboot. And with probability $d_{23}+d_{24}$, the diagnosis will select an action that recovers from the failure, but at an increased cost compared to rebooting.

This recovery strategy can be further divided into two strategies: *manual-diagnosis-based recovery*, and *automated-diagnosis-based recovery*. In the case of manual diagnosis, a developer or administrator manually examines failure evidence, and then selects the most appropriate recovery technique once the type of bug underlying the failure has been diagnosed. This strategy requires some time to perform recovery, because the failure process is investigated before starting recovery. However, given that the diagnosis is supported by careful analysis and failure evidence, the probability of incorrect diagnosis is negligible.

In the case of automated diagnosis, a management tool automates the collection and analysis of failure evidence (e.g., logs), and the decision of the recovery action to perform [46]. This approach is supported by IT management systems, such as IBM Tivoli [45], and HP Business Service Management software [44], which provide extensive monitoring for problem determination, and policy-based automation and recovery. It is reasonable to assume that this recovery strategy is quicker than manual diagnosis, but that recovery actions can sometimes be erroneous (i.e., for some failures, the management tool may not select an effective recovery action, in the case that a policy was not provided by the system administrator for that type of failure).

Another possible strategy, referred to as *escalated recovery***,** is to first attempt a restart every time a failure occurs, then to perform a reboot if restart does not succeed, then a reconfiguration if reboot does not succeed, and finally to debug the failure and develop a fix only when every other recovery action fails. With the escalated recovery strategy, there is no diagnosis phase: the effects and the root cause of the failure are not analyzed (thus disregarding the nature of the underlying bug), and this strategy deterministically selects the same sequence of recovery actions (restart, reboot, reconfigure, fix) at every failure.

The proposed flowchart can be adopted to derive a semi-Markov model [47] for the time to recovery from a failure, which is depicted in Fig. 2. Recovery actions can have a generally-distributed duration, with the only assumption that its mean value should be finite. The model represents the distribution of the time to recovery from a failure for a generic IT system. We adopted this model to derive a closed-form expression of the mean time to recovery (MTTR), which can be used to also compute the steady-state availability of a system. It is important to note that the model does not take into account the processing lost due to failures, and does not include Mandelbugs (such as the *Retry* type) for which a retry on the part of the user will make the failure disappear with a high probability; the model should not be interpreted from the user's viewpoint, but from the system's viewpoint. It must also be noted that the model is focused on failure recovery strategies, rather than failure prevention strategies, such as software rejuvenation; while the model takes into account aging-related failures, the interactions between failure recovery and software rejuvenation would make the model significantly more complex, shifting it from its main focus, and are thus left for future research.

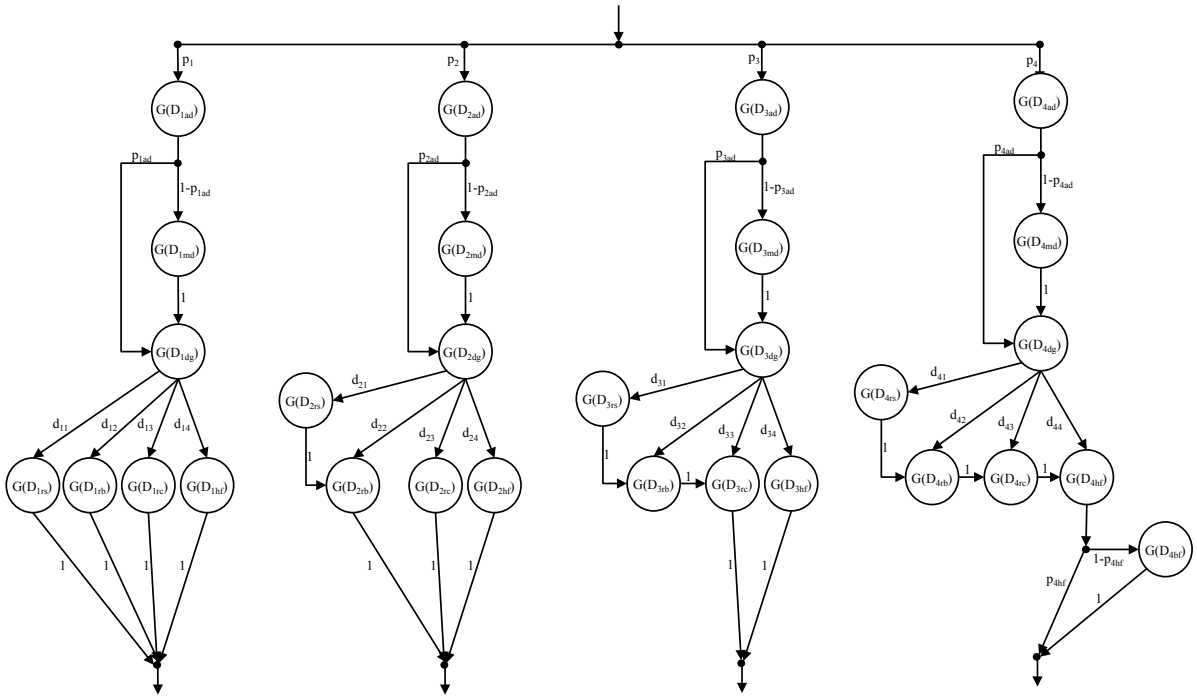


Figure 2. Time to Recovery Distribution of a Failure.

An important aspect of the model is that the proportion of bug types remains the same across failures. The two factors that may contribute to variations of the bug type proportions are (i) fixes that remove bugs underlying failures; and (ii) new releases of the software, which can contribute to new bugs. We model this phenomenon with proportions that do not vary with new releases. This assumption is supported by field failure studies [7], [25], which have observed that the proportion between Mandelbugs and Bohrbugs stabilizes around a constant value once a project has matured, due to the gradual removal of old bugs during the project lifecycle on the one hand, and the introduction of new bugs with new releases of the project on the other hand. While this choice may not be fully accurate for some systems, this being one of the first models dealing with the specifics of recovering from Mandelbugs, we wished to keep the model simple. Moreover, for large, complex systems, it would be reasonable to represent the model as one for steady-state operations that include software release management.

From the time to recovery distribution diagram shown in Fig. 2, we obtain the following closed-form solution of MTTR.

$$
\begin{aligned}
&MTTR = \\
&p_1\left[E[D_{1ad}]+(1-p_{1ad})E[D_{1md}]+E[D_{1dg}]+d_{11}E[D_{1rs}]+d_{12}E[D_{1rb}]+d_{13}E[D_{1rc}]+d_{14}E[D_{1hf}]\right] \\
&+p_2\left[E[D_{2ad}]+(1-p_{2ad})E[D_{2md}]+E[D_{2dg}]+d_{21}\left(E[D_{2rs}]+E[D_{2rb}]\right)+d_{22}E[D_{2rb}]+d_{23}E[D_{2rc}]+d_{24}E[D_{2hf}]\right] \\
&+p_3\left[E[D_{3ad}]+(1-p_{3ad})E[D_{3md}]+E[D_{3dg}]+d_{31}\left(E[D_{3rs}]+E[D_{3rb}]+E[D_{3rc}]\right)+d_{32}\left(E[\mathrm{D}_{3rb}]+\mathrm{E}[D_{3rc}]\right)+d_{33}E[D_{3rc}]+d_{34}E[D_{3hf}]\right] \\
&+p_4\left[\begin{array}{l}E[D_{4ad}]+(1-p_{4ad})E[D_{4md}]+E[D_{4dg}]+d_{41}\left(E[D_{4rs}]+E[D_{4rb}]+E[D_{4rc}]+E[D_{4hf}]\right) \\ +d_{42}\left(E[\mathrm{D}_{4rb}]+\mathrm{E}[D_{4rc}]+\mathrm{E}[D_{4hf}]\right)+d_{43}(\mathrm{E}[D_{4rc}]+\mathrm{E}[D_{4hf}])+d_{44}E[D_{4hf}]+(1-p_{4hf})E[D_{4bf}]\end{array}\right]
\end{aligned}
\tag{1}
$$

V. Model analysis and inferences

We adopt the proposed model to analyze and compare recovery strategies, and to evaluate trade-offs and key factors that impact on the effectiveness of failure recovery. Our analysis is organized according to the following research questions.

1) **Which recovery strategy is most effective?** We analyze mean time to recovery and steady-state (un)availability in several scenarios, by considering different recovery strategies, namely (i) manual-diagnosis-based recovery, (ii) automated-diagnosis-based recovery, and (iii) automated escalated recovery. Moreover, we perform the analysis on different types of systems with different bug proportions, including (i) high-complexity, low-level systems with a high percentage of Mandelbugs; (ii) intermediate-complexity systems; and (iii) lower-complexity systems with a high percentage of Bohrbugs.

2) **Which factors have the highest impact on failure recovery, and need to be optimized to improve availability?** Because several factors are involved in failure recovery, as represented by the parameters and steps in the proposed flowchart, we are interested in identifying the ones that are most influential on recovery. Knowing them will enable developers to tune failure detection, diagnosis, and recovery actions; moreover, setting up automated recovery actions (such as reconfiguration) can have a cost, and increase the complexity of a system, so practitioners could be interested in analyzing the trade-off between the cost and the effectiveness of these actions.

A. Model parameters

In this study, we analyze the recovery model using parameters that were selected to appropriately reflect IT systems in operation. Most of the parameters can be set by developers and IT operations staff to reflect their real-life experience, such as the average time to perform a restart and to reboot (depending on OS, middleware, and virtual machine technologies adopted in the system), and the average time needed for failure detection (depending on user-defined parameters such as timeout values). Instead, it is difficult for developers and IT staff to estimate parameters related to the probability and types of failures, given that failures are relatively rare events, and that the amount of measured data on failures is often limited. Unfortunately, in the IT industry, there are no standard values available due to a lack of publications, and a lack of data sharing. Therefore, we set the model parameters on the basis of quantitative data from the scientific literature, when available, and on our experience with failure recovery in IT systems, including the eleven projects discussed in Section III. The model parameters are shown in Tables II through IV, and are discussed in the following in this section. An advantage of our recovery model, and of the MTTR analysis, is that it *does not rely on the mean time to failure (MTTF) due to Mandelbugs*, which would be difficult to estimate. We also point out that, even if parameters may vary across projects, the model proposed in this paper is generic, and it allows practitioners to easily tune its parameters according to their specific projects.

An important aspect of our analysis is the proportion of the types of failures, because the effectiveness of a recovery action depends on the bug that caused a failure. Developers can obtain this kind of information by leveraging issue tracking and management tools (and other field failure data sources) for the system under analysis, or similar systems within the IT organization, from which developers can know which are the most common failures and their underlying root causes. In [25], we have presented an example of such an analysis based on problem reports, in which we examined Mandelbugs and Bohrbugs in popular open-source software projects that are often adopted in business-critical contexts.

Table II
Explanations of Common Input Parameters and Their Values

Parameter	Explanation	Value
$E[D_{ad}]$	Mean time to automatic failure detection	30 seconds
$E[D_{md}]$	Mean time for manual failure detection	5 minutes
$E[D_{rs}]$	Mean time for restart of software component or service	10 seconds
$E[D_{rb}]$	Mean time for reboot of hardware server or virtual machine	1 minute
$E[D_{rc}]$	Mean time for reconfiguration of software component or service	5 minutes
$E[D_{hf}]$	Mean time to carry out hot-fix	30 minutes
$E[D_{bf}]$	Mean time to carry out bug-fix	1 day
p_{ad}	Probability of automatic failure detection	0.9
p_{4hf}	Probability of correct hot-fix	0.9

We base the parameters of the recovery model on our previous empirical analysis of bugs [25]. In that study, we performed an in-depth analysis of the nature of bugs, including whether bugs were influenced by the timing or sequencing of inputs or operations, or by the environment, whether they involved resource leaks, etc. This information provided us with estimates of the relative proportions of Bohrbugs and of Mandelbugs in these projects. In particular, the empirical analysis showed that the presence of Mandelbugs is influenced by the size, the domain, and the technology of a given project [25]. We found (on the basis of the density of bugs per lines of code, of the software architecture, and of software complexity metrics) that three different scenarios are possible:

- **high-complexity** software, with a significant share of Mandelbugs (~40.2%), affecting hardware- and OS-related software such as the Linux kernel and the MySQL database management system;
- **medium-complexity** software, with ~17.7% Mandelbugs, such as the Apache HTTPD Server; and
- **low-complexity** software, with a relatively lower percentage of Mandelbugs (~7.5%), such as the Apache AXIS project.

Then, for each of these scenarios, we divided the proportion of Mandelbugs into three parts (respectively, the sub-proportions of restart-maskable, reboot-maskable, and reconf-maskable Mandelbugs). To this end, we analyzed the Mandelbugs in the eleven IT systems discussed in Section III, to identify how many Mandelbugs could be recovered using the three recovery actions (restart, reboot, and reconfiguration). From this analysis, we estimated the percentage of Mandelbugs that are restart-maskable, reboot-maskable, and reconf-maskable Mandelbugs. Table III summarizes the resulting proportions of bug types for each of the three scenarios. We observe a good proportion of cases where a simple restart suffices (usually true for several aging-related bugs), and a smaller proportion of bugs for which a reconfiguration is enough (like the overload, limit, timeout, and abort type of bugs), or which belong to the reboot category (true for some cases of aging, specially OS-related, and also true as a last resort before venturing out to a bug-fix).

Table III
Explanations of Input Parameters and Their Values for Three Complexity Scenarios

Param	Explanation	**High**-complexity (share of Mandelbugs: 40.2%)	**Medium**-complexity (share of Mandelbugs: 17.7%)	**Low**-complexity (share of Mandelbugs: 7.5%)
p_1	Probability of restart-maskable Mandelbug	0.2644	0.1166	0.0496
p_2	Probability of reconf-maskable Mandelbug	0.0635	0.0280	0.0119
p_3	Probability of reboot-maskable Mandelbug	0.0740	0.0327	0.0139
p_4	Probability of Bohrbug	0.5981	0.8227	0.9246

As for diagnosis, we selected model parameters to reflect the three recovery strategies discussed in Section IV. The parameters related to diagnosis, which are summarized in Table IV, were selected as follows.

- **Manual diagnosis-based recovery**: In this case, the recovery is overseen by a human operator, who always selects the best recovery option (with a negligible probability of wrong diagnosis), while taking a longer time. For this strategy, the probability d_{ij} of selecting the *j*-th recovery action, given that the failure is caused by the *i*-th bug type, is $d_{ij}=1$ iff $i=j$ (i.e., the selected recovery action *j* matches the bug type *i*), and $d_{ij}=0$ otherwise. For instance, for reboot-maskable bugs, $d_{22}=1$, while $d_{21}=d_{23}=d_{24}=0$.
- **Escalated recovery**: In this case, a restart is always used as the first recovery action on the occurrence of a failure, regardless of the type of bug actually causing the failure. Thus, we have $d_{i1}=1$ (i.e., the first recovery action, a restart, is always selected), and $d_{i2}=d_{i3}=d_{i4}=0$, for every bug type *i*.
- **Automated diagnosis-based recovery**: In this case, an automated tool attempts to determine the best recovery action depending on the type of bug causing the failure. The diagnosis automatically selects a recovery action in a short time, but in some cases the selected recovery action may not be the optimal one (i.e., either the action is not sufficient to recover from the failure, thus requiring a new action to recover; or the action is more costly than the one that would have sufficed to recover from the failure). For this strategy, we assume that the diagnosis of failures is accurate, but not perfect; according to experimental data from a previous study on automated diagnosis techniques [46], the correct recovery action is selected in 95% of the cases. Thus, we have $d_{ij}=0.95$ iff $i=j$, and $d_{ij}=0.05/3=0.01\overline{6}$ otherwise (this latter value is chosen to ensure that $d_{i1}+d_{i2}+d_{i3}+d_{i4}=1$, for every bug type *i*).

Table IV
Explanations of Input Parameters and Their Values for Three Diagnosis Scenarios

Param	Explanation	Manual diagnosis	Escalated recovery	Automated diagnosis
$E[D_{dg}]$	Mean time for failure diagnosis	30 minutes	1 second	1 second
d_{11}	probability of diagnosing *restart*, given a ***restart-maskable*** Mandelbug	1	1	0.95
d_{12}	probability of diagnosing *reboot*, given a ***restart-maskable*** Mandelbug	0	0	0.0167
d_{13}	probability of diagnosing *reconfiguration*, given a ***restart-maskable*** Mandelbug	0	0	0.0167
d_{14}	probability of diagnosing a *hot-fix*, given a ***restart-maskable*** Mandelbug	0	0	0.0167
d_{21}	probability of diagnosing *restart*, given a ***reboot-maskable*** Mandelbug	0	1	0.0167
d_{22}	probability of diagnosing *reboot*, given a ***reboot-maskable*** Mandelbug	1	0	0.95
d_{23}	probability of diagnosing *reconfiguration*, given a ***reboot-maskable*** Mandelbug	0	0	0.0167
d_{24}	probability of diagnosing a *hot-fix*, given a ***reboot-maskable*** Mandelbug	0	0	0.0167
d_{31}	probability of diagnosing *restart*, given a ***reconf-maskable*** Mandelbug	0	1	0.0167
d_{32}	probability of diagnosing *reboot*, given a ***reconf-maskable*** Mandelbug	0	0	0.0167
d_{33}	probability of diagnosing *reconfiguration*, given a ***reconf-maskable*** Mandelbug	1	0	0.95
d_{34}	probability of diagnosing a *hot-fix*, given a ***reconf-maskable*** Mandelbug	0	0	0.0167
d_{41}	probability of diagnosing *restart*, given a ***Bohrbug***	0	1	0.0167
d_{42}	probability of diagnosing *reboot*, given a ***Bohrbug***	0	0	0.0167
d_{43}	probability of diagnosing *reconfiguration*, given a ***Bohrbug***	0	0	0.0167
d_{44}	probability of diagnosing a *hot-fix*, given a ***Bohrbug***	1	0	0.95

Table V
Nine Cases Obtained from the Three Complexity Scenarios and the Three Diagnosis Scenarios

	Case 1	**Case 2**	**Case 3**	**Case 4**	**Case 5**	**Case 6**	**Case 7**	**Case 8**	**Case 9**
Complexity scenario	High	High	High	Medium	Medium	Medium	Low	Low	Low
Diagnosis scenario	Manual diagnosis	Escalated recovery	Automated diagnosis	Manual diagnosis	Escalated recovery	Automated diagnosis	Manual diagnosis	Escalated recovery	Automated diagnosis

Overall, the considered scenarios and recovery strategies lead to a total of nine cases: the combinations of three recovery strategies, and of three alternative bug type proportions. They are listed in Table V.

The manual diagnosis can take a significant amount of time, even in business-critical systems. We have seen diagnosis times of up to 30 minutes before arriving at a conclusion of what needs to be done. The diagnosis in some cases includes a delay for gathering experts from system, application, and technology areas together. Instead, automated diagnosis is based on recovery policies, and can be completed in a matter of seconds.

As for detection, the IT operations staff of business-critical systems must automate most of the failure detection and recovery procedures for all the known types of failures. Our conservative estimate (based on our experience with IT systems, as discussed in Section III, and on a previous experimental study [48]) is that 90% of the failures are detected automatically. Automatic failure detection would happen within seconds, using IT management tools that periodically check the system health. In the case that a failure is manually detected, the IT staff often notices failure symptoms by monitoring system resources, or are warned by customer complaints, and failure detection can take up to a few minutes.

As for automated recovery, restarting a software service or its components often takes a few seconds, while rebooting takes a few minutes. In our experience, rebooting Windows, Unix, and Linux servers requires from two to fifteen minutes. Reconfiguration requires from a few seconds to several minutes, depending on whether it is followed by a restart (with new parameter values) or by a reboot (possibly after a migration to a new machine). A hot-fix can take from a few minutes to two or three hours, while a regular bug-fix, which requires rigorous testing, can be applied within a few days if it is immediately released; or, depending on the severity of the bug, it can take a week or more if it is distributed with the next release of the software.

B. Results

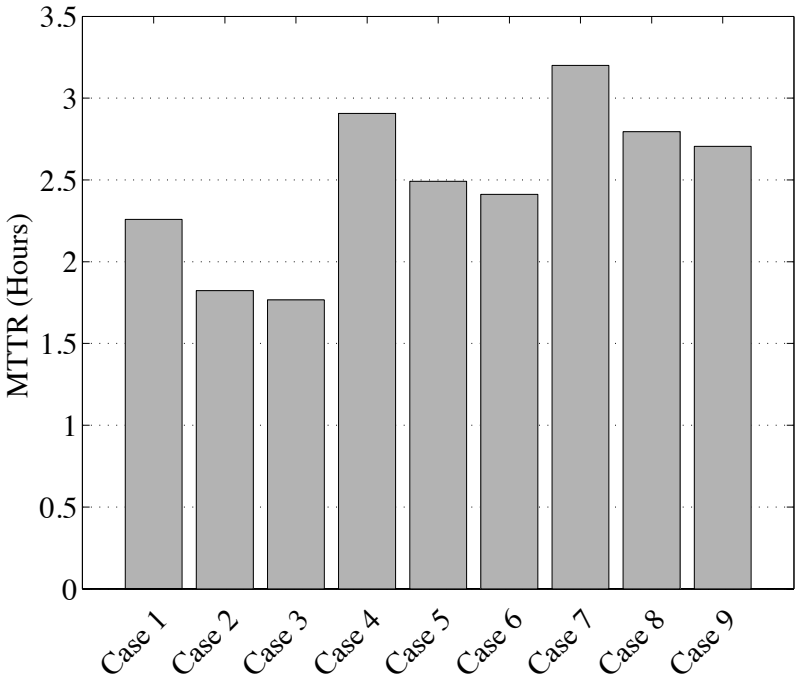
We compute the MTTR using the input parameters as shown in Tables II, III, and IV. Moreover, we computed steady-state unavailability (SSUA) using [49]

$$\text{SSUA} = \frac{\text{MTTR}}{\text{MTTF}+\text{MTTR}}. \qquad (2)$$

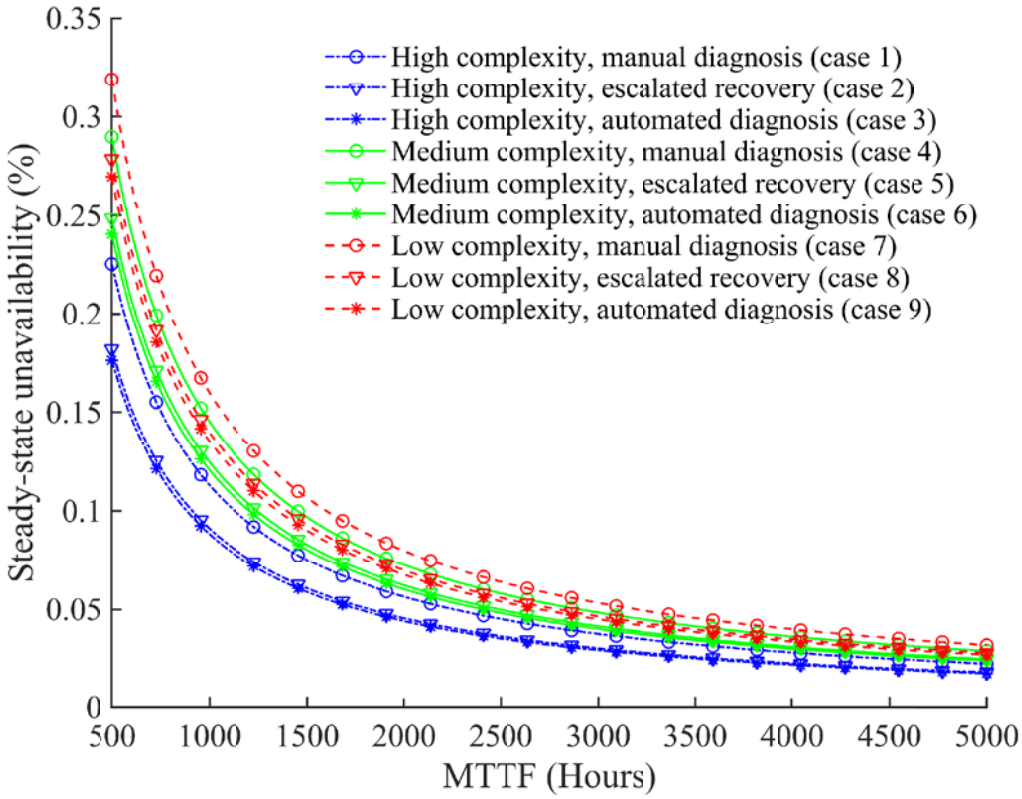
Table VI lists the mean times to recovery obtained for all nine cases based on our parameter settings; these values are also depicted in Fig. 3(a). As steady-state unavailability depends on the mean time to failure, which is difficult to estimate, Fig. 3(b) shows how SSUA varies for different values of MTTF. Of course, the respective steady-state availability can easily be calculated by subtracting SSUA from one. It is important to note that, in business-critical applications with high-availability requirements, a downtime of even a few minutes can have a significant cost and cause noticeable service disruptions. Therefore, although the values of MTTR in Table VI and Fig. 3(a) are in the same order of magnitude, the differences between the cases analyzed can be very important for IT system operators and users.

Table VI
Mean Time to Recovery for the nine cases

	High-complexity systems			**Medium**-complexity systems			**Low**-complexity systems		
	Manual diagnosis (Case 1)	**Escalated recovery (Case 2)**	**Automated diagnosis (Case 3)**	**Manual diagnosis (Case 4)**	**Escalated recovery (Case 5)**	**Automated diagnosis (Case 6)**	**Manual diagnosis (Case 7)**	**Escalated recovery (Case 8)**	**Automated diagnosis (Case 9)**
MTTR (Hours)	2.2591	1.8224	1.7660	2.9060	2.4915	2.4119	3.1996	2.7952	2.7050



(a) Mean time to recovery (MTTR).



(b) Steady-state unavailability (SSUA) as a function of MTTF.

Fig. 3. MTTR, and SSUA for the nine cases considered in this study.

Note that the impact of failures and of recovery strategies depends on the specific IT system considered, and may vary for different systems with different parameters (e.g., different proportions of Mandelbugs, and different MTTF). While the scope of the results is limited to the considered scenarios, the adoption of failure data from real-world projects makes these scenarios representative of many IT systems, and allows us to make the following observations.

Observation 1: Manual failure diagnosis is less effective than escalated recovery. In all the considered systems (high, medium, and low complexity), escalated recovery has a mean time to recovery lower than manual diagnosis. This difference indicates that the gain of automatically recovering from Mandelbugs through restarts, reboots, and reconfigurations is higher than the penalty due to useless restarts, reboots, and reconfigurations attempted in the presence of Bohrbugs. Even if Bohrbugs are the majority of bugs (~60% or more in the considered scenarios), the automated recovery actions can avoid, at least in some cases, to perform a full problem diagnosis and bug-fixing, thus saving significant effort while increasing the availability of the IT system. As shown in Fig. 4, the main factor impacting the manual diagnosis is the time required to perform the manual problem determination. To be as effective as escalated recovery, the manual problem determination should be performed in less than 5 minutes. However, in the experience of the authors, more than 5 minutes is often required to determine a problem, and the quickest solution is to attempt an escalated recovery and to mask the failure; only in the case that the problem persists is an in-depth investigation of the root cause of the problem necessitated.

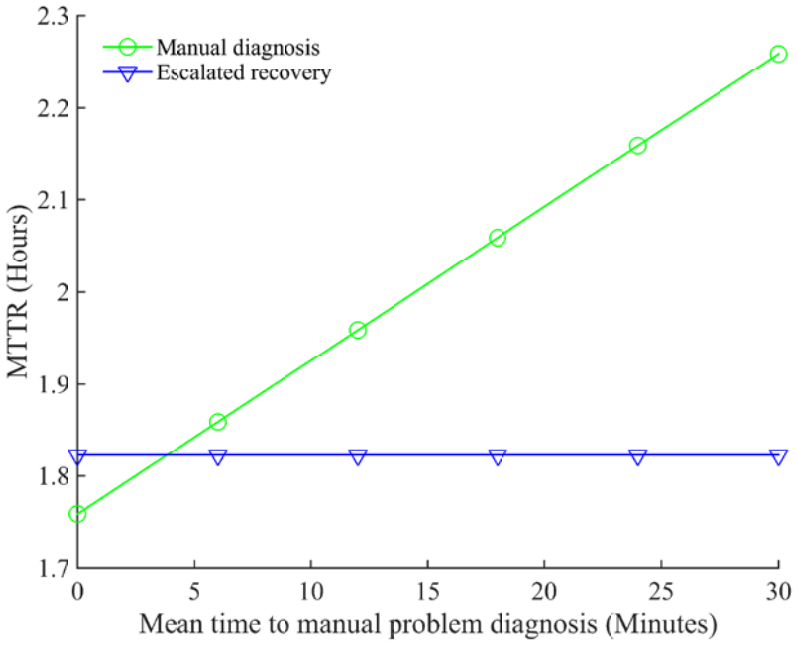


Fig. 4. MTTR of manual diagnosis compared to escalated recovery (in high-complexity software), for different values of the mean time to problem diagnosis.

Observation 2: Automated diagnosis is better than manual diagnosis, but it is not substantially more effective than escalated recovery. This result may appear not to be intuitive, because the reader could expect that automated diagnosis makes a quick and optimal use of automatic recovery actions, thus providing even better results than escalated recovery. Nevertheless, the gain from automated diagnosis tends to be limited. Even if the mean time to recovery of each strategy depends on the particular system on which it is applied, the relative effectiveness of the three recovery strategies seems consistent across all the considered systems. This result emphasizes that the adoption of automatic recovery strategies should be carefully evaluated before putting them in production. In particular, the factor that affects the usefulness of automated diagnosis is the probability of mistaken diagnosis. If the probability of mistakes is high enough, then automated diagnosis can trigger useless recovery actions (e.g., reboot a system when a reconfiguration is necessary); or, even worse, can trigger recovery actions more costly than necessary (e.g., require the intervention of IT administrators or developers when a restart would have sufficed to recover from the fault), increasing the overall cost of recovery and reducing availability. A high mistake rate can thus make automated diagnosis not worth the additional complexity that it would bring, or can even make it counterproductive. Therefore, it should be adopted only when developers can assure a high accuracy of diagnosis policies. For instance, as can be seen from Fig. 5, in the case of high-complexity systems, the gain from automated diagnosis becomes less substantial when the probability of correct diagnosis approaches 50%, and automated diagnosis is even counterproductive below that percentage. If accurate diagnosis cannot be assured, then escalated recovery is the best strategy due to the very quick speed of automated recovery actions such as restarts, reboots, and reconfigurations. This result suggests that IT administrators and developers should perform tests for the accuracy of diagnosis to gain such confidence, for instance by means of fault-injection tests.

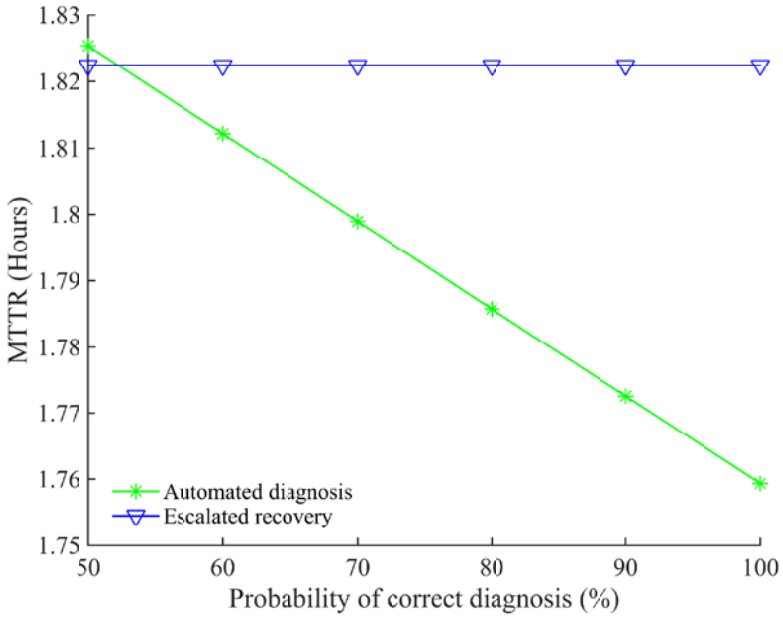


Fig. 5. MTTR of automated diagnosis compared to escalated recovery (in high-complexity software), for different values of the probability of correct diagnosis.

Note that, even if the MTTR is higher for low-complexity systems, this condition does not imply that they are necessarily less available than high-complexity ones. The MTTR for low-complexity systems is higher because most of the failures are caused by Bohrbugs; thus, fewer failures can be masked by automated recovery actions (restart, reboot, reconfiguration), while a higher proportion of them require bug-fixes, which tend to take a much longer time. However, (un)availability is a function of both MTTR and MTTF. As can be seen from Fig. 3(b), low-complexity systems can have a higher availability than high-complexity ones if their MTTF is high enough. For instance, consider a high-complexity system with escalated recovery, and a MTTF of 1000 hours; from Fig. 3(b), it can be seen that a low-complexity system with escalated recovery attains a lower SSUA than this high-complexity system if its MTTF exceeds 1545 hours.

To get more insights into the recovery process, we performed a parametric sensitivity analysis of the model. Sensitivity analysis is a method to determine factors that are most influential on model results. It can be used to find recovery or availability bottlenecks in the system, and to thus guide improvement and optimization. Parametric sensitivity analysis is performed by computing the elasticity of each parameter. Here, elasticity represents the percentage change in MTTR (or SSUA) that results from a percentage change in the respective parameter. This measure provides a uniform way to compare the impact of different parameters with different measurement units. The elasticity of MTTR with respect to a parameter p can be computed from the partial derivative with respect to that parameter:

$$\text{Elasticity}_{\text{MTTR}}(p) = \frac{p}{\text{MTTR}(p)} \text{MTTR}'(p) \,. \qquad (3)$$

The elasticity of SSUA can be computed in a similar way to MTTR. Using the chain rule on the derivative of SSUA, its elasticity can be expressed as

$$\text{Elasticity}_{\text{SSUA}}(p) = \frac{p}{\text{SSUA}(p)} \text{SSUA}'(p) = \frac{p}{\text{MTTR}(p)} \text{MTTR}'(p)(1 - \text{SSUA}(p)). \quad (4)$$

To compute elasticity, we fix all parameters to their default values (shown in Tables II through IV), and compute the partial derivative (with respect to the parameter p under evaluation), and the MTTR and SSUA functions. We considered all parameters from Tables II through IV, except the bug proportions (for which we are considering three scenarios, based on quantitative field failure data) and the diagnosis parameters d_{ij} (because they are imposed by the recovery strategy, and we have discussed them previously).

Using the above equations, we compute the elasticity for each parameter as shown in Table VII. The ranking of parameters is ordered according to the absolute value of elasticity, and the ranking is shown in brackets. Because SSUA is very close to 0, the elasticity of MTTR is very similar to the elasticity of SSUA, and the rankings with respect to MTTR are always the same to the rankings with respect to SSUA, for all parameters and all nine cases. Therefore, Table VII only shows elasticity values for MTTR. A positive elasticity indicates that an increase of a parameter causes an increase in MTTR, whereas a negative elasticity indicates that, if a parameter value increases, MTTR decreases.

Table VII
Elasticity of MTTR with respect to model parameters (and their rankings in brackets)

(a) High-complexity software systems

Parameter	Case 1 (Automated diagnosis)	Case 2 (Escalated recovery)	Case 3 (Manual diagnosis)
p_{ad}	-3.3199e-02 (5)	-4.1154e-02 (4)	-4.2469e-02 (4)
p_{4hf}	-5.7186e+00 (1)	-7.0887e+00 (1)	-7.3152e+00 (1)
$E[D_{ad}]$	3.6888e-03 (6)	4.5726e-03 (7)	4.7188e-03 (6)
$E[D_{md}]$	3.6888e-03 (7)	4.5726e-03 (8)	4.7188e-03 (7)
$E[D_{dg}]$	2.2133e-01 (3)	1.5242e-04 (10)	1.5729e-04 (10)
$E[D_{rs}]$	3.2513e-04 (10)	1.5242e-03 (9)	4.1443e-04 (9)
$E[D_{rb}]$	4.6819e-04 (9)	6.7271e-03 (6)	8.3250e-04 (8)
$E[D_{rc}]$	2.7311e-03 (8)	3.0734e-02 (5)	5.1079e-03 (5)
$E[D_{hf}]$	1.3237e-01 (4)	1.6409e-01 (3)	1.7125e-01 (3)
$E[D_{bf}]$	6.3539e-01 (2)	7.8763e-01 (2)	8.1280e-01 (2)

(b) Medium-complexity software systems

Parameter	Case 4 (Automated diagnosis)	Case 5 (Escalated recovery)	Case 6 (Manual diagnosis)
p_{ad}	-2.5809e-02 (5)	-3.0102e-02 (4)	-3.1096e-02 (4)
p_{4hf}	-6.1150e+00 (1)	-7.1322e+00 (1)	-7.3677e+00 (1)
$E[D_{ad}]$	2.8676e-03 (6)	3.3447e-03 (7)	3.4551e-03 (5)
$E[D_{md}]$	2.8676e-03 (7)	3.3447e-03 (8)	3.4551e-03 (6)
$E[D_{dg}]$	1.7206e-01 (3)	1.1149e-04 (10)	1.1517e-04 (10)
$E[D_{rs}]$	1.1150e-04 (10)	1.1149e-03 (9)	1.4462e-04 (9)
$E[D_{rb}]$	1.6056e-04 (9)	5.9090e-03 (6)	3.9789e-04 (8)
$E[D_{rc}]$	9.3661e-04 (8)	2.8609e-02 (5)	2.6173e-03 (7)
$E[D_{hf}]$	1.4155e-01 (4)	1.6510e-01 (3)	1.7118e-01 (3)
$E[D_{bf}]$	6.7945e-01 (2)	7.9247e-01 (2)	8.1864e-01 (2)

(c) Low-complexity software systems

Parameter	Case 7 (Automated diagnosis)	Case 8 (Escalated recovery)	Case 9 (Manual diagnosis)
p_{ad}	-2.3441e-02 (5)	-2.6832e-02 (5)	-2.7726e-02 (4)
p_{4hf}	-6.2421e+00 (1)	-7.1451e+00 (1)	-7.3833e+00 (1)
$E[D_{ad}]$	2.6045e-03 (6)	2.9813e-03 (7)	3.0807e-03 (5)
$E[D_{md}]$	2.6045e-03 (7)	2.9813e-03 (8)	3.0807e-03 (6)
$E[D_{dg}]$	1.5627e-01 (3)	9.9377e-05 (10)	1.0269e-04 (9)
$E[D_{rs}]$	4.3053e-05 (10)	9.9377e-04 (9)	6.4677e-05 (10)
$E[D_{rb}]$	6.1996e-05 (9)	5.6670e-03 (6)	2.6913e-04 (8)
$E[D_{rc}]$	3.6164e-04 (8)	2.7980e-02 (4)	1.8794e-03 (7)
$E[D_{hf}]$	1.4449e-01 (4)	1.6540e-01 (3)	1.7116e-01 (3)
$E[D_{bf}]$	6.9356e-01 (2)	7.9390e-01 (2)	8.2036e-01 (2)

The ranking of elasticities shows the following observations.

Observation 3: The probability of correct hot-fix is the most influential parameter. It is critical to avoid an incorrect hot-fix that requires a subsequent bug-fix. This result is true because, if an IT system goes to the bug-fix step, then it takes much more time for it to recover. In all the nine cases, the probability of correct hot-fix was ranked first, with elasticity higher than for the other parameters by orders of magnitude. Thus, it is important to optimize this parameter, by paying as much attention as possible when applying hot-fixes to recover from failures. Even if improving this probability increases the duration of hot-fixing, the impact of a wrong hot-fix on MTTR and SSUA would be significantly higher. Thus, it is worth spending additional time to assure the correctness of hot-fixes. Of course, full bug-fixes cannot be avoided in all cases, because this case also depends on the complexity and on the severity of bugs, but avoiding bug-fixes when not strictly required is beneficial to availability.

Observation 4: The probability of automated failure detection has a noticeable influence. After hot-fixing and bug-fixing time, this parameter was consistently ranked as one of the most influential. This parameter is important because failure detection is a prerequisite for all types of recovery actions, and has to be performed regardless of the actions that will be undertaken once a failure has been noticed. Thus, avoiding delays between the occurrence of a failure and its discovery enables a quick recovery, and reduces unavailability. This parameter can be optimized by adopting advanced IT management systems that provide facilities for precise monitoring of system resources, processes, and services; and can automatically discover anomalies through failure detection policies provided by system administrators.

VI. Conclusion

In this paper, we have first introduced the practical cases, types of Mandelbugs, and methods for recovering from failures caused by Mandelbugs. Then, we have presented a recovery model using a flowchart, and we have developed a semi-Markov model based on the flowchart to derive the closed-form solution of MTTR for an IT system. The model is meant to be simple, and easy to use by practitioners. Finally, we adopted the model to evaluate the relative value of recovery actions and strategies, by considering nine representative scenarios, based on failure data from this study, and from previous ones. The analysis of the model showed that the model is useful for comparing different recovery strategies, for obtaining insights about when a strategy is most effective, and for identifying critical parameters. Parametric sensitivity analysis was performed by computing the elasticity with respect to each parameter, to determine the parameters that have the highest impact on recovery from failures due to Mandelbugs, and that should thus be optimized by IT administrators and developers. Future work in this area includes investigating the possible interactions between reactive approaches (e.g., the failure recovery strategies analysed in this work) and proactive approaches (e.g., software rejuvenation), which are aimed at preventing failures before their occurrence, to support the tuning of such proactive approaches towards optimizing system availability.

Acknowledgments

This work has been supported by the Dr. Theo and Friedl Schoeller Research Center for Business and Society, and by the MIUR project DISPLAY (PON02_00485_3487784).

References

[1] J. D. Musa, *Software Reliability Engineering: More Reliable Software, Faster and Cheaper*. Tata McGraw-Hill Education, 2nd edition, 2004.

[2] Gizmodo.com, "A comprehensive review of what went down with HealthCare.gov," available at http://gizmodo.com/a-comprehensive-review-of-what-went-down-with-healthcar-1479224003, 2013 (last visited on 2014-08-29).

[3] W. E. Wong, V. Debroy, A. Surampudi, K. HyeonJeong, and M. F. Siok, "Recent catastrophic accidents: Investigating how software was responsible," in *Proc. IEEE Intl. Conf. on Secure Software Integration and Reliability Improvement (SSIRI)*, 2010, pp. 14-22.

[4] M. Grottke and K. S. Trivedi, "Software faults, software aging and software rejuvenation," *J. Reliability Engineering Association of Japan*, vol. 27, no. 7, pp. 425-438, 2005.

[5] M. Grottke and K. S. Trivedi, "A classification of software faults," in *Supplemental Proc. IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2005, pp. 4.19-4.20.

[6] M. Grottke and K. S. Trivedi, "Fighting bugs: Remove, retry, replicate, and rejuvenate," *IEEE Computer*, vol. 40, no. 2, pp. 107-109, 2007.

[7] M. Grottke, A. P. Nikora, and K. S. Trivedi. "An empirical investigation of fault types in space mission system software," in *Proc. Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 447-456.

[8] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. John Wiley & Sons, 7th edition, 2004.

[9] M. Cavage, "There is no getting around it: you are building a distributed system," *Communications of the ACM*, vol. 56, no. 6, pp. 63-70, 2013.

[10] S. Moore, "Multicore is bad news for supercomputers," *IEEE Spectrum*, vol. 45, no. 11, p. 15, 2008.

[11] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot-A technique for cheap recovery," in *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004, pp. 31-44.

[12] D. Cavezza, R. Pietrantuono, J. Alonso, S. Russo, and K. Trivedi, "A study of the reproducibility of environment-dependent software failures," in *Proc. IEEE Intl. Symposium on Software Reliability Engineering (ISSRE)*, 2014, pp. 267-276.

[13] Y. Huang, P.E. Chung, C. Kintala, D. Liang, and C. Wang, "NT-SwiFT: Software implemented fault tolerance on Windows NT," in *Proc. USENIX Windows NT Symposium*, 1998.

[14] F. Qin, J. Tucek, J. Sundaresan, and Y.Y. Zhou, "Rx: Treating bugs as allergies–A safe method to survive software failures," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 235-248, 2005.

[15] V.R. Basili, and B.T. Perricone, "Software errors and complexity: an empirical investigation," *Communications of the ACM*, vol. 27, no. 1, pp. 42-52, 1984.

[16] D. Perry and W. Evangelist, "An empirical study of software interface faults," in *Proc. of New Directions in Computing Conference*, 1985, pp. 32-38.

[17] R. Chillarege, I.S., Bhandari, J.K., Chaar, M.J., Halliday, D.S., Moebus, B.K., Ray, and M.Y. Wong, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 943-956, 1992.

[18] B. Beizer, *Software Testing Techniques*. Van Nostrand Reinhold Co, 2nd edition, 1990.

[19] P. Carter, “Common C errors,”, available at http://www.drpaulcarter.com/cs/common-c-errors.php (last visited on 2014-08-29).

[20] A. Koenig, *C Traps and Pitfalls*. Addison-Wesley, 1989.

[21] V. Vipindeep and P. Jalote, “List of common bugs and programming practices to avoid them,” Technical Report, IIT Kanpur, 2005, available at http://www.cse.iitk.ac.in/users/jalote/papers/CommonBugs.pdf, (last visited on 2014-08-29).

[22] J. Gray, “Why do computers stop and what can be done about it?,” Tandem Computers Tech. Rep. 85.7, 1985.

[23] C. C. Liaw, S. Y. H. Su, and Y. K. Malaiya. “Test generation for delay faults using stuck-at-fault test set,” in *Proc. International Test Conf.*, 1980, pp. 167-175.

[24] S. Y. H. Su, I. Koren, and Y. K. Malaiya, “A continuous-parameter Markov model and detection procedures for intermittent faults,” *IEEE Trans. Computers*, vol. 27, no. 6, pp. 567-570, 1978.

[25] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K.S. Trivedi, “Fault triggers in open-source software: An experience report,” in *Proc. IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2013, pp. 178-187.

[26] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, “Software rejuvenation: Analysis, module and applications,” in *Proc. Annual International Symposium on Fault-Tolerant Computing (FTCS)*, 1995, pp. 381-390.

[27] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, “A survey of software aging and rejuvenation studies,” *ACM Journal on Emerging Technologies in Computing Systems*, vol. 10, no. 1, 2014.

[28] D. Cotroneo, R. Natella, and R. Pietrantuono, “Predicting aging-related bugs using software complexity metrics,” *Performance Evaluation*, vol. 70, no. 3, pp. 163-178, 2013.

[29] F. Machida, J. Xiang, K. Tadano, and Y. Maeno, “Aging-related bugs in cloud computing software,” in *Proc. Intl. Workshop on Software Aging and Rejuvenation (WoSAR)*, 2012.

[30] G. Carrozza, D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, “Analysis and prediction of Mandelbugs in an industrial software system,” in *Proc. IEEE Intl. Conference on Software Testing, Verification and Validation (ICST)*, 2013, pp. 262-271.

[31] R. Chillarege, “Comparing four case studies on Bohr-Mandel characteristics using ODC,” in *Proc. Intl. Workshop on Software Aging and Rejuvenation (WoSAR)*, 2013.

[32] R. Chillarege, “Understanding Bohr-Mandel bugs through ODC triggers and a case study with empirical estimations of their field proportion,” in *Proc. Intl. Workshop on Software Aging and Rejuvenation (WoSAR)*, 2011.

[33] J. Alonso, R. Matias, E. Vicente, A. Maria, and K.S. Trivedi, “A comparative experimental study of software rejuvenation overhead,” *Performance Evaluation*, vol. 70, no. 39, pp. 231-250, 2012.

[34] S. Distefano and K. S. Trivedi, “Non-Markovian state-space models in dependability evaluation,” *Quality and Reliability Engineering International*, vol. 29, no. 2, pp. 225-239, 2013.

[35] T. Dohi, K. Goševa-Popstojanova, and K. S. Trivedi, “Statistical non-parametric algorithms to estimate the optimal software rejuvenation schedule,” in *Proc. IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2000, pp. 77-84.

[36] M. Grottke and B. Schleich, “How does testing affect the availability of aging software systems?,” *Performance Evaluation*, vol. 70, no. 3, pp. 179-196, 2013.

[37] Y. Liu, K. S. Trivedi, Y. Ma, J. J. Han, and H. Levendel, “Modeling and analysis of software rejuvenation in cable modem termination systems,” in *Proc. IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2002, pp. 159-170.

[38] K. S. Trivedi, G. Ciardo, B. Dasarathy, M. Grottke, A. Rindos, and B. Vashaw, “Achieving and assuring high availability,” in *Proc. IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS)*, 2008, pp. 1-7.

[39] K. Vaidyanathan and K. S. Trivedi, “A comprehensive model for software rejuvenation,” *IEEE Trans. Dependable and Secure Computing*, vol. 2, no. 2, pp. 124-137, 2005.

[40] S. Garg, Y. Huang, C.M. Kintala, K.S. Trivedi, and S. Yajnik, “Performance and reliability evaluation of passive replication schemes in application level fault tolerance,” in *Proc. Annual International Symposium on Fault-Tolerant Computing (FTCS)*, 1999, pp. 322-329.

[41] K. S. Trivedi, D. Wang, D. J. Hunt, A. Rindos, W. E. Smith, and B. Vashaw, “Availability modeling of SIP protocol on IBM© Websphere©,” in *Proc. IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2008, pp. 323-330.

[42] M. Grottke and K. S. Trivedi, “Analysis of the escalated levels of failure recovery approach,” Technical Report, 2011.

[43] K. S. Trivedi, R. Mansharamani, D.S. Kim, M. Grottke, M. Nambiar, “Recovery from failures due to Mandelbugs in IT systems,” in *Proc. IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2011, pp. 224-233.

[44] Hewlett-Packard Company. *Business Service Management (BSM)*, available at http://www8.hp.com/us/en/software-solutions/business-service-management-overview.html (last visited on 2014-08-29).

[45] IBM Corporation, Tivoli System Automation for Multiplatforms,, available at http://www.ibm.com/software/products/en/tivosystautoformult (last visited on 2014-08-29).

[46] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons, “Correlating instrumentation data to system states: A building block for automated diagnosis and control,” in *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004, pp. 231-244.

[47] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley & Sons, 2nd edition, 2001.

[48] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox, “JAGR: An autonomous self-recovering application server,” in *Proc. Autonomic Computing Workshop*, 2003, pp. 168-177.

[49] R. Sahner, K. Trivedi, and A. Puliafito, *Performance and Reliability Analysis of Computer Systems*. Kluwer Academic Publishers, 1996.

Michael Grottke received an M.A. in economics from Wayne State University, USA; and a Diploma degree in business administration; and a Ph.D. from the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany. From 2004 to 2007, he was a Research Associate and Assistant Research Professor in the Department of Electrical and Computer Engineering at Duke University, USA. In 2010, he received the Habilitation degree from the FAU. His research interests include intermediate fields between statistics, computer science, and business administration, such as software aging and rejuvenation, software performance, software engineering economics, and stochastic modeling. He has published papers on these topics at international conferences as well as in international journals, including IEEE Computer, IEEE Transactions on Dependable and Secure Computing, IEEE Transactions on Reliability, Journal of Systems and Software, and Performance Evaluation. He is a member of the IEEE, and the German Statistical Society.

Dong Seong Kim is currently a lecturer (equivalent to an assistant professor in the North America system) in the Department of Computer Science and Software Engineering at the University of Canterbury, Christchurch, New Zealand since August 2011. He received Ph.D. degree in Computer Engineering from the Korea Aerospace University, South Korea in 2008. He was a visiting researcher at the University of Maryland at College Park, MD, USA, in 2007. From June 2008 to July 2011, he was a postdoctoral researcher at Duke University, Durham, NC, USA. His research interests are in dependability and security for systems and networks. In particular, anomaly intrusion detection systems, security for wireless ad hoc and sensor networks, Internet of things and cloud computing system dependability, and cyber security modelling and analysis. He is a member of the IEEE.

Rajesh Mansharamani is a freelance performance engineering consultant, since 2011, to IT service companies and stock exchanges. He is currently also the president of Computer Measurement Group, India, which is a community of over 1500 performance engineering and capacity planning professionals. From 2006 to 2011, Rajesh was Vice President and Chief Scientist of the Performance Engineering Research Centre at Tata Consultancy Services (TCS). From 1999 to 2006, Rajesh was heading the Corporate Performance Engineering Group at TCS, and prior to that he was a researcher at Tata Research Design and Development Centre from 1994. Throughout his career, he has worked on architecture, design, and performance engineering of a number of nationwide systems across banking, financial services, government, and pharmacy sectors. Rajesh has an M.S. and Ph.D. in Computer Science from University of Wisconsin Madison, and a BTech from IIT-Bombay.

Manoj Nambiar is currently working with TCS as a Principal Scientist, heading the Performance Engineering Research Center (PERC). He also leads the Parallelization and Optimization Centre of excellence as a part of the company's HPC Initiative. Until 2011, Manoj has been research lead in High Performance Messaging, Networking and Operating Systems in PERC. Prior to this he has been consultant in the performance engineering area specializing in network and systems performance. He has published papers in international conferences and journals on measurements, modeling, and performance analysis; and has also served on technical program committees of conferences. His research interests include performance and availability modeling, high performance computing, fault tolerant computing, and design and analysis of algorithms. Manoj has a B.E. (Computer Engineering) from the University of Bombay (1994), and a post graduate diploma in VLSI design from C-DAC, India (2001). He is a senior member of the IEEE.

Roberto Natella is currently a postdoctoral researcher at the Federico II University of Naples, Italy, where he received the Ph.D. degree in computer engineering in 2011, and co-founder of the Critiware s.r.l. university spin-off company. His research is on methods for dependability assurance of software for mission-critical systems, including dependability benchmarking, software fault injection, and software aging and rejuvenation, with emphasis on the practical application of these methods in industrial projects. He has authored papers, and has been reviewer, for leading conferences and journals in the fields of software engineering and fault-tolerant computing. He is a member of the IEEE.

Kishor S. Trivedi holds the Hudson Chair in the Department of Electrical and Computer Engineering at Duke University, Durham, NC. He has a B.Tech. (EE) from IIT Mumbai, and M.S. and Ph.D. (CS) degrees from the University of Illinois at Urbana-Champaign. He has been on the Duke faculty since 1975. He is the author of a well-known text entitled, *Probability and Statistics with Reliability, Queuing and Computer Science Applications*; a revised second edition (including its Indian edition) has been published by John Wiley. He has also published two other books entitled, *Performance and Reliability Analysis of Computer Systems*, and *Queueing Networks and Markov Chains*. He is a Fellow of the IEEE, and a Golden Core Member of the IEEE Computer Society. He has published over 500 articles, and has supervised 45 Ph.D. dissertations. He is the recipient of the IEEE Computer Society's Technical Achievement Award for his research on Software Aging and Rejuvenation. His research interests are in reliability, availability, performance, and survivability of computer and communication systems and in software dependability. He works closely with industry in carrying our reliability and

availability analysis, providing short courses, and in the development and dissemination of software packages such as HARP, SHARPE, SREPT, and SPNP.