# ソフトウェアにおけるフォールト，エージング，若化の概念

## Software Faults, Software Aging and Software Rejuvenation

Michael GROTTKE* Kishor S. TRIVEDI**

要旨: 過去10年間，長期間稼動するソフトウェアシステムに対して，hang/crash 障害の発生率が増加したり，徐々にシステムの性能が劣化する現象であるソフトウェアエージング（経年劣化）について，数多くの研究がなされてきた．本論文では，例えプログラムコード上にフォールトが作り込まれていなかったとしても，ソフトウェアシステムが経年劣化を引き起こすことについて考察する．まず最初に，ソフトウェアバグの分類について議論し，それらの定義と相互関係について明らかにする．特に，ソフトウェアエージングに起因するバグがここで提案する分類方法に適合していることを示す．ソフトウェアエージングに関する問題を解決するために，ソフトウェア若化（レジュビネーション）*** と呼ばれる予防的な方法が提案されている．具体的には，稼働中のソフトウェアシステムを一旦停止し，累積エラーの原因を除去した後にシステムを再始動するといったものである．ソフトウェア若化によって生じるオーバーヘッドにより，システムの初期化を行う最適なタイミングを求める問題が考えられる．本論文では，上記のような重要な問題を取扱うために開発された種々のアプローチについて概説する．

キーワード: エージングに関連したバグ，ボーバグ，ハイゼンバグ，マンデルバグ，最適スケジュール，予防的フォールトマネジメント，ソフトウェアエージング，ソフトウェア若化

**Abstract:** During the last decade, many studies have examined “software aging”, the phenomenon that long-running software systems show an increasing occurrence rate of hang/crash failures, progressive performance degradation, or both. This paper deals with the question how software can age even if no additional faults are introduced into the code. We discuss several categories of software “bugs” and clarify their definitions as well as their mutual relationships. We then show how those bugs responsible for software aging fit into this framework. To counteract software aging, a proactive technique called “software rejuvenation” has been proposed, which essentially involves stopping the running software and restarting it after removing the accrued error conditions. Due to the overhead incurred by software rejuvenation, an optimal timing of its initiation should be sought. We give an overview of approaches developed for dealing with this important question.

**Keywords:** aging-related bugs, Bohrbugs, Heisenbugs, Mandelbugs, optimal scheduling, proactive fault management, software aging, software rejuvenation

## 1 Introduction

The term “software aging” has been used for two different phenomena. Parnas [34] employed it for referring to the fact that a piece of software may on the one hand become obsolete due to changing user requirements and on the other hand get more and more difficult to maintain after many modifications have been introduced into its source code, especially if these modifications are insufficiently documented and not carried out by the original designers. A different kind of “aging” can be observed even if both the user requirements and the source code remain stable:

* Corresponding author, on leave of absence from the Chair of Statistics and Econometrics, University of Erlangen-Nuremberg, Germany. This work was supported by a fellowship within the Postdoc Program of the German Academic Exchange Service (DAAD).
Department of Electrical and Computer Engineering Duke University, P.O. Box 90291, Durham, NC 27708-0291, USA grottke@ee.duke.edu
** Department of Electrical and Computer Engineering Duke University, P.O. Box 90291, Durham, NC 27708-0291, USA kst@ee.duke.edu
*** ‘rejuvenation’ の邦訳として ‘若化’ を用いた最初の文献は，土肥，海生，尾崎，電子情報通信学会論文誌 (A), vol. J85-A, no. 2, pp. 197-206 (2002) である．

Researchers and practitioners have reported that software systems running continuously for a long time tend to show a degraded performance, an increased occurrence rate of failures (i.e., deviations of the delivered service from the correct service) or both due to development faults in the software. Among the kinds of systems observed to age are telecommunication billing applications [29], telecommunication switching software [2], web servers [27] and safety-critical military equipment [31]. Although Huang et al. [29] originally called this phenomenon "process aging", the term "software aging" has meanwhile established; see for example [1], [15], [22] and [42]. This is also the way in which we will use "software aging" in this paper.

Since software aging focuses on problems caused by software development faults and explicitly deals with situations in which the source code is not modified, one may well wonder how it is possible that the failure occurrence rate and the performance of software change over time. While this question has often been discussed, the terminology previously used in the software aging literature differs from existing classification schemes for software faults. The main contribution of this paper is an effort to reconcile the terminology and to clarify the relationships between different fault concepts, which is contained in Section 2.

As our discussion will show, software faults responsible for software aging are difficult to find in the testing phase of software development. Therefore, software aging is mainly dealt with via a fault tolerance technique: Given the presence of aging-related software faults, the occurrence of failures is to be avoided by preventively and proactively stopping the running software, cleaning its internal state and/or its environment and then restarting it. Huang et al. [29] introduced this approach as "software rejuvenation". Unlike the downtime caused by sudden software failures, the downtime related to software rejuvenation can be scheduled at the discretion of the users or the system administrators, e.g., for periods in which the workload is predicted to be low. Hence, the costs of downtime due to software rejuvenation can be expected to be much lower than the costs of unplanned downtime caused by a sudden failure. Nevertheless, like all preventive maintenance policies, software rejuvenation incurs an overhead. Therefore, an optimal timing of software rejuvenation with respect to availability and costs should be sought. Many research papers on this topic can be found at [11]. In Section 3, we discuss some of the approaches used for analyzing and solving of this optimization problem. A brief summary in Section 4 concludes this paper.

## 2 Software Faults and Software Aging

Talking about the causes of software-aging and their relationship with other concepts requires a clear definition of the terminology employed. In this section, we begin with discussing a number of important concepts used in the field of dependability. We then review several different types of software "bugs" and show how the causes of software aging are related to them.

### 2.1 Basic Concepts in Dependability

Our definitions of basic dependability concepts largely follow the recent taxonomy by Avižienis et al. [1]. However, since we focus on aging-related problems of software applications, we introduce some more specific definitions.

A *system* in general is an entity interacting with other entities, which form the *environment* of the given system. For example, a *software system* interacts with humans (like users and system administrators) and the physical world. Most systems are composed of *components*. Among the components of a software system are the specific software application used or tested, other concurrently running applications, the operating system and the hardware. Since each component is an entity interacting with other components and/or the environment of the entire system, each component can be considered to be another system. A running software application (consisting of the actual program, its data and its files) therefore has an environment encompassing the environment of the software system as well as the other components of the software system. To the latter we will refer as the *(software) system-internal environment* of the application. The system-theoretic structure of a software system is sketched in Figure 1.

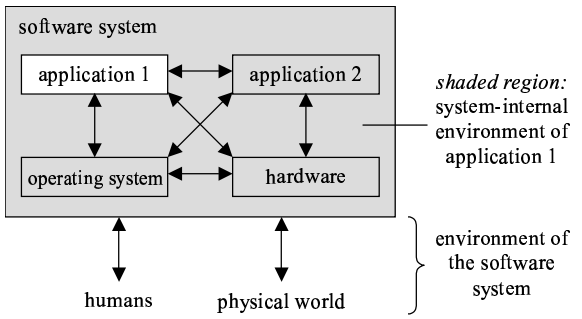


Fig. 1. System-theoretic structure of a software system

If the software system behaves the way it is intended to (laid down in its functional specification), it delivers *correct service*. A *(service) failure* occurs when the actual system behavior deviates from correct service. The failure of one or more services does not necessarily put the software into a state in which it does not anymore deliver any correct service. Rather, a subset of services may still be available, or the services may be offered at a lower performance. If the user can detect a degraded performance [2] or a reduced functionality, then a *partial failure* is said to have occurred.

Usually, before a software system suffers a failure it is already in a state in which there is a discrepancy between its actual and the correct internal condition, although the deviation is not perceivable. Such a discrepancy is called an *error*. An error may first be transformed into other errors before it finally leads to a failure. This functional chain is referred to as *error propagation*.

The causes of errors are *faults*. Typical examples of faults are wrong or missing lines of code. The event that occurs when a fault produces an error is called *fault activation*. The "chain of threats" linking faults, errors and failures, adapted from [1], is shown in Figure 2.

Avižienis et al. [1] classify faults according to eight criteria, including the phase of their creation, their persistence, their phenomenological cause and the fact whether they originate inside or outside the boundary of the software system.

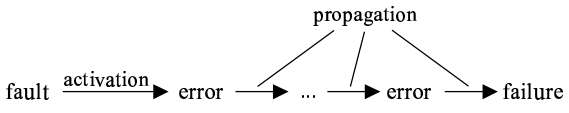


Fig. 2. "Chain of threats"

Our central question stated in Section 1 is: "How can wrong lines of codes and other faults in a software application lead to software aging, although the application is not modified?" According to the classification scheme by Avižienis et al. [1], this problem is concerned with human-made, internal, permanent software development faults. In the following, we will refer to this type of faults as "software faults" or "(software) bugs" for the sake of brevity.

### 2.2 Software Bugs – A Bestiary

Starting with the work of Huang et al. [29], there have been several attempts to classify software faults responsible for software aging according to the distinction between *Bohrbugs* and *Heisenbugs*; see for example [45] and [46]. All these references draw upon a paper by Gray, originally written as a technical report in 1985 [24] and published at a symposium one year later [25]. To the best of our knowledge, these are indeed the first occasions at which the two terms were used in print.

The neologism "Bohrbug" alludes to the rather simple Bohr atom model, in which the electrons move around the nucleus in orbit at different levels. In his characterization of Bohr's model, Gray also uses the word "solid" (although it would better fit with Dalton's atomic theory). The term "Bohrbug" therefore relates to *solid* or *hard* [1] software faults, i.e., faults that are easily detected and fixed and for which the failure occurrences are easily reproduced. To summarize it with Gray's pun: "Bohrbugs … are … boring" [24], [25]. Wherever the word is used, it is done in this vein – see e.g. [8], [36], [37], [48], [51].

For Gray, Heisenbugs are *soft* or *elusive* [1] software faults, for which the failure occurrences are not systematically reproducible. If an operation failed due to a Heisenbug, retrying the operation may not lead to a second failure – indeed it is possible that the very act of trying to observe the failure perturbs the situation and prevents the failure occurrence [24], [25]. This is the meaning with which the term is employed in the paper by Huang et al. [29] and in the subsequent work on software rejuvenation.

It seems that the word "Heisenbug" was mentioned in Gray's papers for the first time; for example, it is not listed

in *The Hacker's Dictionary* [40] from 1983. Nevertheless, the term is much older. Based on information they received from Gray [5], [9], both Birman [4] and Candea [8] attribute the invention of the word to Bruce Lindsay; Candea gives a colorful account of how Lindsay came up with the term at the University of Berkeley in the 1960s. In recent interviews [7], [55], Lindsay corroborated the claim that he was involved in creating the word "Heisenbug", which was coined while he and Gray were working on the CAL-TSS operating system project [55]. Lindsay also states that he thinks it was him who invented the term [30], which agrees with Gray's account. The authorship is of interest, because Lindsay's explanation of the meaning differs slightly from the one used by Gray: Lindsay originally used the word in order to refer to a software fault that "went away, because the measurement or the observation affected the phenomena you were trying to see" [55]. "So the real definition of a Heisenbug is when you look, it goes away" [7]. This explanation is reasonable. "Heisenbug" is obviously a pun on the name of the physicist Heisenberg, whose Uncertainty Principle is popularly believed to assert that the act of observing changes the quantity to be measured [49]. Outside the field of software rejuvenation, most references mentioning the word "Heisenbug" use it in this sense, sometimes including those software faults whose failure behavior alters (although the failure does not completely disappear) when it is researched; cf. for example [8], [17], [36], [37], [47], [49] and [52]. We will therefore stick to the original, more specific definition according to Lindsay instead of the one given by Gray.

An interesting collection of field reports about Heisenbugs can be found at [53]. It reveals that there are two important categories of how trying to observe a failure can make it disappear:

1. Some debuggers **initialize unused memory** to default values. Failures related to improper initialization may therefore go away as soon as the debugger is turned on.
2. Trying to investigate a failure can influence **process scheduling** in such a way that the failure does not occur again. For example, scheduling-related failures in multi-threaded programs may disappear when a debugger is used to single-step through a process, or to set break-points. However a fault can become a Heisenbug even without application of any tool. Simple print or dump commands can sufficiently change the scheduling of threads. Sometimes even mere movements of the mouse make a failure go away by triggering the operating system to give certain processes a higher priority.

As for the class of elusive software faults, for which Gray employed the word "Heisenbugs", there seems to exist another term: "*Mandelbugs*". Again, the word alludes to the name of a scientist, this time the mathematician Mandelbrot, who is known best for his research in fractal geometry. The usual definition of a Mandelbug is "a bug whose underlying causes are so complex and obscure as to make its behavior appear chaotic and even non-deterministic" [37]; see the almost identical definitions in [8] and [50]. If the "behavior" of the bug is supposed to refer to the question whether it causes a failure or not, then "chaotic and even non-deterministic behavior" means that under seemingly identical conditions sometimes a failure occurs, while on other occasions no failure is experienced. This is another way of expressing the fact that a failure is not systematically reproducible, i.e., that it is caused by an elusive software fault. We should add that in [54] there are notable differences in how the term is defined: Firstly, a Mandelbug is said to have "a single simple cause" [54], which clearly disagrees with all other references cited above. Secondly, it is the system that shows a "chaotic and unpredictable behavior" [54], not the bug; therefore, this definition seems to have in mind the apparently non-deterministic nature of failure consequences (e.g., high fluctuations in wrong output values and varying severities of the failures caused by a specific fault) rather than the erratic occurrence/non-occurrence of failures itself. However, building on the definition consistently used in the other references cited above, we will essentially equate Mandelbugs with elusive software faults.

The confusion about the definitions on the one hand and the fuzziness of the definitions themselves on the other

hand have also led to discrepancies in statements about how the classes of Bohrbugs, Mandelbugs and Heisenbugs are related to each other. In general, Bohrbugs and Heisenbugs are considered to be antonyms [8] [40]; several definitions of Mandelbugs include the claim that they are Bohrbugs rather than Heisenbugs [40], [50]; some people think that all Heisenbugs are Mandelbugs [18]. Clearly, these three statements are incompatible. Moreover, if the distinction between Bohrbugs (i.e., hard faults) and Mandelbugs (i.e., elusive faults) relies on the fact that failures caused by the former are easily reproducible, while failures related to the latter cannot be reproduced systematically, the parting line between the two classes is essentially subjective: A fault is classified as a Mandelbug if the circumstances under which it leads to a failure are judged to be "too complex" for the human mind to grasp. It has therefore been argued that there is no real distinction between Bohrbugs and Mandelbugs; in the end, all software faults are the same [9], [39].

Therefore, we are confronted with the question, whether it is possible to restate the definitions such that the following three goals are attained:

1. Reducing (if not eliminating) the subjectivity in the definitions,
2. clarifying the relationships between the different categories, and
3. creating a framework into which aging-related bugs can easily be integrated.

Our suggested definitions are shown in Table 1. The main difference – from which other changes follow – lies in the additional explanation of what constitutes the complexity that makes a software fault a Mandelbug. We identify two possible cases that are not mutually exclusive: Firstly, a software fault in a specific application is a Mandelbug, if its activation and/or its error propagation depend on interactions with the system-internal environment of the application. (The idea for this classification criterion is due to Shetti [39].) Examples are faults causing failures due to side-effects of other applications and faults for which the scheduling done by the operating system is crucial for the occurrence of a failure.

Table 1. Proposed definitions of software fault types

| Category | Proposed definition |
|---|---|
| Bohrbug | A fault that manifests consistently under a well-defined set of conditions, because its activation and error propagation lack "complexity" as set out in the definition of Mandelbug. Complementary antonym of Mandelbug. |
| Mandelbug | A fault whose activation and/or error propagation are complex, where "complexity" can take two forms:<br>1. The activation and/or error propagation depend on interactions between conditions occurring inside the application and conditions that accrue within the system-internal environment of the application.<br>2. There is a time lag between fault activation and failure occurrence, e.g. because several different error states have to be traversed in the error propagation.<br>Typically, failures caused by a Mandelbug are not systematically reproducible. Complementary antonym of Bohrbug. |
| Heisenbug | A fault that stops causing a failure or that manifests differently when one attempts to probe or isolate it. Subtype of Mandelbug. |
| aging-related bug | A fault that leads to the accumulation of errors either inside the running application or in its system-internal environment, resulting in an increased failure rate and/or degraded performance. Sub-type of Mandelbug. |

Typically, such faults are difficult to detect, and the failures related to them are hard to reproduce, because the user/tester usually does not know how the system-internal environment influences the application.

We should point out that in our opinion failure occurrences that are *solely* induced by the system-internal environment of specific application are not related to a "Mandelbug" in this application. When the system crashes while testing an application because of a fault in the operating system, then this fault may be a Bohrbug in the operating system or a Mandelbug of the operating system, but it is

not a fault *in the application*.

Secondly, we classify a fault in an application as a Mandelbug if complexity of the error propagation results in a delay between the fault activation and the final failure occurrence. For example, an erroneous calculation due to a fault in the software code of an application may be kept in the memory without immediately causing the service delivered by the software system to deviate from correct service; only later, when the result of the calculation is accessed and used in a way that influences the system behavior perceivable by the user, a failure will be experienced. The reproduction of such failures is difficult since the time lag masks the cause-and-effect relationship between the execution of an operation and the related failure.

As a consequence of the difficulties in finding and reproducing Mandelbugs, one should expect that the residual faults in a thoroughly-tested piece of software mainly belong to this category. In fact, it has been claimed that most software faults in production systems are transient in nature [26].

According to our definitions, Mandelbugs and Bohrbugs are complementary antonyms; i.e., each software fault belongs to exactly one of the two categories. Since the "chain of threats" (see Figure 2) of a Bohrbug features none of the complexities explained above, reproducing a failure related to a Bohrbug will generally succeed if the operation that failed before is retried with the same direct input values.

The definition of a Heisenbug listed in Table 1 is essentially identical with the one due to Lindsay. The above discussion of Heisenbugs shows that the act of observing influences the failure behavior via factors belonging to the system-internal environment of the application in which the Heisenbug is located. Therefore, all Heisenbugs are Mandelbugs.

It should be noted that one can hardly state *a priori* which of the Mandelbugs are Heisenbugs. Whether a Mandelbug will stop manifesting or manifest differently depends on the method or tool employed for probing or isolating it. While a certain debugger initializing unused memory may make an initialization-related fault go away, this effect may not occur when using another debugger. As a consequence, a fault can only be classified as a *Heisenbug with respect to a specific observation method/tool*. In Figure 3, a Venn diagram depicting the relationships between the different types of software faults, this fact is indicated by the dashed line enveloping the category of Heisenbugs.

After discussing and defining the other three terms, we can finally return to our original question: "How can software faults in an application lead to software aging, even if the application is not modified?" Making use of the conceptual framework of the "chain of threats", we can explain the existence of aging-related bugs, i.e., software faults responsible for an increasing failure rate and/or a degraded performance, by the fact that those faults cause errors to accumulate over time. The error conditions may accrue either within the running application (e.g., round-off errors in program variables) or in the system-internal environment (e.g., unreleased physical memory due to memory leaks in the application); cf. our definition in Table 1. In either case, usually these error conditions do not lead to a (partial) failure right away – otherwise, there could be no aging –, but the failures occur with a delay; this fulfills one of the two criteria for classifying a software fault as a Mandelbug. Moreover, if the error conditions accumulate in the system-internal environment, then the other criterion for a Mandelbug is met as well, because the occurrence of a failure depends on conditions existing in the system-internal environment of the application. Therefore, aging-related bugs are a subtype of Mandelbugs, as shown in Figure 3. This explains why aging-related bugs are often not detected and removed during the testing phase of software development.

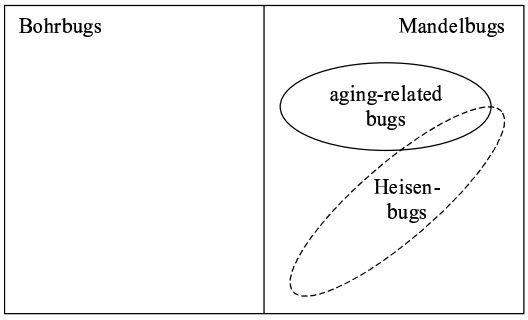


Fig. 3. Venn diagram of software fault types

The class of aging-related bugs of an application may or may not overlap with the class of those software faults that are Heisenbugs with respect to a certain observation tool or method.

## 3 Approaches to Analyzing Software Aging and Rejuvenation

There are two kinds of approaches to studying software aging and deriving optimal software rejuvenation schedules: model-based ones and measurement-based ones.

Model-based approaches are aimed at building analytic models that capture system degradation and software rejuvenation. These models are then solved in order to determine dependability measures (like the steady-state availability) under a given rejuvenation policy and to optimize the timing of software rejuvenation.

The basic idea of measurement-based approaches is to periodically monitor attributes of the software system that may show signs of software aging (e.g., the physical memory). The collected data is used for assessing the current "health" of the system and for predicting possible failures due to aging-related faults.

In the following subsections we review some of the literature on both types of approaches.

### 3.1 Model-based Approaches

The first, rather simple, model for software aging and rejuvenation can be found in the seminal paper by Huang et al. [29]. The authors propose that a software system starts out in the highly robust state $S_0$, in which failures due to aging are practically impossible. With a constant transition rate, it may switch to the failure probable state $S_P$. Based on our discussion in Section 2, we can identify this transition with the activation of an aging-related bug. The propagation of the internal error condition is prone to lead to a service failure; this possibility is modeled via a transition into the failure state $S_F$. Again, the transition rate is assumed to be constant; i.e., the failure time (from state $S_P$) follows an exponential distribution. If a failure occurs, then the software system has to be "repaired", e.g., restarted. In order to avoid failures, the system can be rejuvenated as soon as it enters the failure-probable state. While being rejuvenated the system is in state $S_R$. After both repair and rejuvenation the software system is as good as new, and it therefore returns to the highly robust state. Like the other time intervals, the time to invoke software rejuvenation (from state $S_P$), the time to complete software rejuvenation and the repair time are all assumed to follow an exponential distribution. The structure of this model is depicted in Figure 4. Huang et al. [29] determine those rates for triggering rejuvenation which maximize the steady-state availability and minimize the steady-state costs per time unit, respectively. They come to the conclusion that if the mean time needed for software rejuvenation and the average per unit costs of software rejuvenation are below certain thresholds, rejuvenation should be invoked as soon as the system switches to the failure-probable state. If the expected time or costs of rejuvenation are too high, then rejuvenation should never be carried out in order to optimize the respective criterion. The reason for this result is that due to the exponential failure time distribution the software does not age any further once it reaches the failure probable state. Therefore, there is no merit to delaying rejuvenation. If it is at all beneficial, it should be triggered right away; otherwise, it should be completely dismissed.

Dohi et al. [12], [13], [14] extend this original model by allowing for general transition distributions. In their semi-Markov model, rejuvenation is triggered $t_0$ time units after the system switches to state $S_P$. The constant $t_0$ is determined such that the steady-state system availability is maximized [12], [14] or such that the steady-state expected costs per unit time are minimized [13], [14].

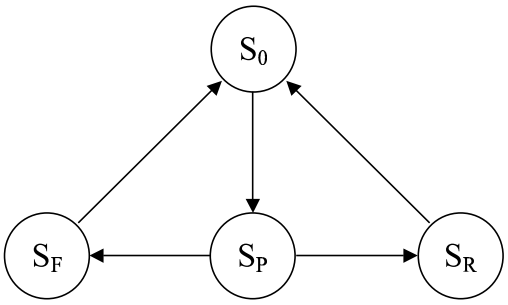


Fig. 4. State transition diagram of a simple model for software aging and rejuvenation

Moreover, Dohi et al. [12], [13] consider a variant of the model according to which software rejuvenation also needs to be carried out after the repair of the failed system is completed. For optimizing $t_0$, the authors of [12], [13], [14] apply a non-parametrical statistical technique which does not require specification of the underlying failure time distribution. This is a big advantage over other modeling approaches, because it is in general unknown from which type of distribution the failure times are sampled.

Garg et al. [21] also introduce a constant time to invoke rejuvenation into the basic model by Huang et al. However, they measure the rejuvenation interval beginning in the highly robust state $S_0$. For this end, they formulate the model as a Markov regenerative stochastic Petri net. Although Garg et al. assume that the transition time to the failure probable state as well as the time to fail after entering that state are exponentially distributed, they show that it is possible to derive optimal lengths of the rejuvenation interval minimizing the expected steady-state unavailability and the expected steady-state costs per time unit, respectively. At first sight, this seems to contradict the results by Huang et al., but it is in fact consistent with them: If we cannot observe the current state of the software system, for a software starting in the highly robust state, it becomes more and more probable that it has switched to the state $S_P$ the longer it has been operating; therefore, its failure rate tends to increase over time, and a rejuvenation interval (measured from the beginning like in the approach by Garg et al.) unequal to both zero and infinity can be optimal. The situation is different in the model by Huang et al., because it assumes that the transition to the failure probable state – after which the failure rate remains constant – can be observed.

In the Markov regenerative stochastic Petri net model by Garg et al. [21], the optimization has to be carried out numerically. However, Suzuki et al. [41] reformulate it as a semi-Markov model, which can be solved analytically. The optimal software rejuvenation schedule maximizing the steady-state availability depends on the distribution of the overall (i.e., measured from the beginning, when the software is still in state $S_0$) time to failure. Applying the same non-parametric technique as Dohi et al. [12], [13], [14], Suzuki et al. show that the optimal rejuvenation interval can be estimated based on data sampled from the overall time to failure distribution.

As an alternative to the steady-state system availability, Dohi et al. [16] introduce the expected up rate per unit cycle. For the semi-Markov models proposed in [12] and [41] they numerically calculate the software rejuvenation policies maximizing this new measure of dependability, and they contrast them with the results obtained based on the optimization of system availability.

In the models discussed so far, it is implicitly assumed that the time to complete rejuvenation is significantly shorter than the time to finish repair; that is why preventive rejuvenation may increase the steady-state availability although it incurs an overhead. However, for some systems recovery and rejuvenation involve the same steps, which suggests that the time required for both is about the same. Even if this should be the case, rejuvenation can still be beneficial. In fact, apart from a possible gain in availability, rejuvenation cost models tacitly include further aspects when claiming that the costs per time unit are smaller for rejuvenation than for repair: For example, in server-type systems a sudden failure may lead to the loss of the jobs currently in the system as well as unavailability in peak periods; planned rejuvenation, on the other hand, can be scheduled for periods with low predicted workload. The arrival and queuing of jobs in a system is explicitly included in the stochastic reward net model by Garg et al. [20], who study two different rejuvenation policies: a purely time-based one, (i.e., a fixed rejuvenation interval like in [21]) and a load and time-based one, in which rejuvenation is only carried out if the server is idle and the current arrival rate of jobs is low. As can be expected, the latter policy is superior in minimizing the number of lost jobs.

The basic structure of all the models mentioned until now is based on the four-stage model by Huang et al. The coarse nature of software aging in these models is criticized by Avritzer and Weyuker [2], who examine smooth performance degradation in telecommunication switching software.

Responding to this critique, Bobbio et al. [6] develop a fine-grained software degradation model in which

software aging is modeled as a sequence of additive random shocks. They study two rejuvenation policies: one based on a pre-assigned risk level, and one based on an alert threshold. Fujio et al. [19] extend this model by introducing three different rejuvenation policies. For each of them, they derive the optimal rejuvenation schedule maximizing system availability.

A different approach to accounting for smooth degradation is chosen by Pfening et al. [35], who model performance degradation in a server-type system via a time-dependent service rate.

The model of a transaction-based software system by Garg et al. [22] includes both hang/crash failures and performance degradation. Moreover, the times to failure as well as the service times may follow general distributions. In fact, the failure and service rates can depend on time, instantaneous load and mean accumulated load. Similar to [20], Garg et al. examine two rejuvenation policies: Under the purely time-based policy, rejuvenation is invoked after a constant waiting period has elapsed since the system was (re-)started. Under the instantaneous load and time-based policy, rejuvenation is triggered as soon as the system becomes idle for the first time after the constant waiting time has passed. For both policies, the authors derive expressions for the steady-state system availability, the probability of losing an incoming transaction and an upper bound on the response time of a transaction.

For the same server-type software system, Okamura et al. [32] introduce two completely workload-based rejuvenation schemes. Instead of a constant waiting period, these policies involve a specific number of transactions that have to be completed before the system can be rejuvenated.

Okamura et al. [33] extend the model in [22] by considering multiple servers, and they examine the same two rejuvenation policies as Garg et al.

An explicit connection between the existence of resource leaks and an increasing failure rate is established in the degradation model by Bao et al. [3]. On a higher level, the authors model proactive fault management via a semi-Markov process consisting of a working state, a failure state and a rejuvenation state.

The models mentioned so far only deal with one level of rejuvenation, usually a full system restart. Vaidyanathan et al. [44] consider two kinds of preventive maintenance in operational software systems. Xie et al. [56] generalize the semi-Markov model presented in [12], [14] by introducing the possibility of service-level rejuvenation.

In [10] and [43], software aging and rejuvenation in a cluster system are studied via stochastic reward net models. For different rejuvenation intervals in a purely time-based rejuvenation policy, the authors calculate the expected costs and the expected downtime for various types of cluster systems. Moreover, they consider a prediction-based rejuvenation policy, in which a node is rejuvenated when it is predicted to transit into the failure probable state. Such forecasts can be made with the help of actual measurements of system parameters, which links the model-based approaches with the measurement-based approaches.

### 3.2 Measurement-based Approaches

As pointed out before, these approaches to studying software aging and rejuvenation use actual measurements of observable attributes in order to detect the existence of software aging, estimate its intensity and predict failure occurrences due to aging.

Garg et al. [23] analyze software aging in a network of UNIX workstations. Using an SNMP-based distributed monitoring tool, they collect operating resource usage information and system activity data like free real memory, used swap space and file table size. A non-parametric trend test reveals significant developments consistent with the software aging hypothesis for all the metrics measured on three computers. Garg et al. estimate these trends via another non-parametric procedure and use linear extrapolation for predicting the time to exhaustion for each resource. On all three machines, real memory and swap space are those resources for which exhaustion is predicted to occur earliest; the aging of other resources like file table and process table seems to be comparatively unimportant. Although the analysis by Garg et al. indicate daily or weekly periodicities for some of the metrics, they do not account for seasonality in their predictions.

While the analysis by Garg et al. assumes that memory

usage merely depends on the elapsed time, Vaidyanathan and Trivedi [46] consider the influence of system workload. For this end, they periodically monitor four variables characterizing the workload, like the number of system calls as well as the number of page-in and page-out operations made since the last observation, in addition to used swap space and free real memory. Using cluster analysis on the workload data, the authors identify eight clusters. Based on these clusters a state-transition model for the system workload is built. The sojourn time distributions of this model are derived by fitting either a two-stage hyperexponential or a two-stage hypoexponential distribution to the empirical data.

Slopes of the development of used swap space are estimated separately for each of the eight workload states. Attaching these slopes to the workload states leads to semi-Markov reward models for used swap space. Solving this model an overall estimate for the increase of used swap space per time unit is obtained; based on this slope, the time to exhaustion of swap space is estimated via linear extrapolation. The same procedure is repeated for the free real memory.

For both memory resources the estimated time to exhaustion produced by the workload-based approach is significantly lower than the one computed using the time-based approach employed by Garg et al. [23]. This result seems to be favorable, because it was indeed observed that failures due to resource exhaustion tend to happen earlier than predicted by the time-based method; therefore, the workload-based estimations are more realistic. The reason for this is that in the time-based approach significant changes in the data are "averaged out"; the workload-based method, in contrast, separately considers those "busy hours" in which memory depletes particularly fast.

However, although different workload states with individual slopes are identified, the predictions in the workload-based approach by Vaidynathan and Trivedi [46] are again linear functions of time.

In connection with the prediction-based rejuvenation policies for cluster servers mentioned at the end of the last section, Castelli et al. [10] fit (piecewise) linear trends to measurements of resource usage taken within a fitting window, or to the logarithm of these measurements.

Grottke et al. [27] study software aging in an Apache web server subjected to a synthetic load. Data collected during experiments in which the server was put in an overload condition indicate the presence of software aging. Furthermore, the data reveal how configuration settings related to the operating system as well as to the web server itself can influence the development of resource utilization. Both nonparametric techniques and parametric time series models are employed for data analysis. While previous research did not account for seasonality in the predictions of resource usage, the authors explicitly model the existing seasonal pattern and exploit it for forecasting the future behavior.

Hong et al. [28] suggest a closed-loop approach to software rejuvenation in which service-level rejuvenation or system-level rejuvenation are triggered based on the periodic measurements of the degrading resource. Under a linear degradation assumption, they compare closed-loop rejuvenation with a purely time-based (open-loop) policy as well as the situation without rejuvenation. In an experiment, they apply their closed-loop approach to an Apache web server with a simulated memory leak.

A completely different approach to the analysis of aging in memory resources is chosen by Shereshevsky et al. [38]. These authors do not model and predict memory utilization itself, but they monitor the local rate of fractality of the system parameters via the Hölder exponent, and they conclude that the second abrupt increase in this measure indicates an imminent system crash.

## 4 Conclusions

For one decade, the phenomenon that long-running software systems often exhibit an increasing failure rate and/or a degrading performance, has been studied under the name of "software aging". The underlying causes of this phenomenon, aging-related bugs, have often been discussed together with so-called "Heisenbugs". In this paper, we have shown that the way in which the term "Heisenbug" has been used in the software aging literature differs from its original meaning as well as the definition usually employed in software engineering and

testing. For several categories of bugs we have suggested revised definitions that help to clarify their mutual relationships. Moreover, we have given an overview over the research in both model-based an measurement-based approaches to software aging and rejuvenation.

## Acknowledgments

We would like to thank Ken Birman, George Candea, Christof Fetzer, Jim Gray, Bruce Lindsay, Rich Martin and Elaine Weyuker for providing us with information on the genesis and the meaning of the term "Heisenbug".

## References

[1] Avižienis, A., J.-C. Laprie, B. Randell, and C. Landwehr (2004): "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11-33.

[2] Avritzer, A. and E.J. Weyuker (1997): "Monitoring smoothly degrading systems for increased dependability," *Empirical Software Engineering*, vol. 2, pp. 59-77.

[3] Bao, Y., X. Sun, and K.S. Trivedi (2005): "A workload-based analysis of software aging and rejuvenation," *IEEE Transactions on Reliability*, vol. 54. (To appear).

[4] Birman, K. (2005): "Intermediate course in operating systems," Lecture notes CS514, Part 11, URL = http://www.cs.cornell.edu/Courses/cs514/2005sp/lec-11.ppt (Link verified on June 6, 2005).

[5] Birman, K. (2005): Personal communication with M. Grottke.

[6] Bobbio, A., M. Sereno, and C. Anglano (2001): "Fine grained software degradation models for optimal rejuvenation policies," *Performance Evaluation*, vol. 46, pp. 45-62.

[7] Bourne, S. (2004): "Interview: A conversation with Bruce Lindsay," *ACM Queue*, vol. 2, pp. 22-33.

[8] Candea, G. (2003): "The enemies of dependability I : Software," Lecture notes CS444a, URL = http://www.cs.stanford.edu/~candea/teaching/cs444a-fall-2003/notes/software.pdf (Link verified on May 26,2005).

[9] Candea, G. (2005): Personal communication with M. Grottke.

[10] Castelli, V., R.E. Harper, P. Heidelberger, S.W. Hunter, K.S. Trivedi, K. Vaidyanathan and W. Zeggert (2001): "Proactive management of software aging," *IBM Journal of Research and Development*, vol. 45, pp. 311-332.

[11] Department of Electrical and Computer Engineering, Duke University : "Software rejuvenation," URL = http://www.software-rejuvenation.com/ (Link verified on June 20, 2005).

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

[13] Dohi, T., K. Goševa-Popstojanova, and K.S. Trivedi (2000): "Analysis of software cost models with rejuvenation," *Proc. International Symposium on High Assurance Systems Engineering*, pp. 25-34.

[14] Dohi, T., K. Goševa-Popstojanova, and K.S. Trivedi (2001): "Estimating software rejuvenation schedules in high assurance systems," *Computer Journal*, vol. 44, pp. 473-482.

[15] Dohi, T., K. Goševa-Popstojanova, K. Vaidyanathan, K.S. Trivedi, and S. Osaki (2003): "Software rejuvenation: Modeling and Applications," *Handbook of Reliability Engineering* , H. Pham, ed., Springer, London, pp. 245-263.

[16] Dohi, T, H. Suzuki, and K.S. Trivedi (2004): "Comparing software rejuvenation policies under different dependability measures," *IEICE Transactions on Information and Systems*, vol. E87-D, pp. 2078-2085.

[17] Eisenstadt, M. (1997): "My hairiest bug war stories," *Communications of the ACM*, vol. 40, pp. 30-37.

[18] Fetzer, C. (2005): "Software fault tolerance – Robust programming," Lecture notes, URL = http://wwwse.inf.tu-dresden.de/~suesskraut/ss05/sft/SFT-Lecture-RobustProgramming.pdf (Link verified on May 6, 2005).

[19] Fujio, H., H. Okamura, and T. Dohi (2003): "Fine-grained shock models to rejuvenate software systems," *IEICE Transactions on Information and Systems*, vol. E86-D, pp. 2165-2171.

[20] Garg, S., Y. Huang, C. Kintala, and K.S. Trivedi (1995): "Time and load based software rejuvenation: Policy, evaluation and optimality," *Proc. First Conference on Fault-Tolerant Systems and Software,* URL = http://shannon.ee.duke.edu/ Rejuv/fts_sachin.ps.

[21] Garg, S., A. Puliafito, M. Telek, and K.S. Trivedi (1995): "Analysis of software rejuvenation using Markov regenerative stochastic Petri net," *Proc. Sixth International Symposium on Software Reliability Engineering*, pp. 180-187.

[22] Garg, S., A. Puliafito, M. Telek, and K.S. Trivedi (1998): "Analysis of preventive maintenance in transactions based software systems," *IEEE Transactions on Computers*, vol. 47, pp. 96-107.

[23] Garg, S., A. van Moorsel, K. Vaidyanathan, and K.S. Trivedi (1998): "A methodology for detection and estimation of software aging," *Proc. Ninth International Symposium on Software Reliability Engineering*, pp. 282-292.

[24] Gray, J. (1985): "Why do computers stop and what can be done about it?" Technical Report 85.7, PN87614, Tandem Computers, Cupertino, URL = http://www.hpl.hp.com/techreports/tandem/ TR-85.7.pdf (Link verified on June 23,2005).

[25] Gray, J. (1986): "Why do computers stop and what can be done about it?" *Proc. Fifth Symposium on Reliability in Distributed Systems*, pp. 3-12.

[26] Gray, J. and D.P. Siewiorek (1991): "High-availability computer systems," *IEEE Computer*, vol. 24, No. 9, pp. 39-48.

[27] Grottke, M., L. Li, K. Vaidyanathan, and K.S. Trivedi (2005): "Analysis of software aging in a web server," Technical Report 69/2005, Chairs of Statistics, University of Erlangen-Nuremberg, Nürnberg.

[28] Hong, Y., D. Chen, L. Li, and K.S. Trivedi (2002): "Closed loop design for software rejuvenation," *Proc. Workshop on Self-Healing, Adaptive, and Self-Managed Systems,* URL = http://www.cse.psu.edu/ ~yyzhang/shaman/ submission/shaman27.pdf (Link verified on June 27, 2005).

[29] Huang, Y., C. Kintala, N. Kolettis, and N.D. Fulton (1995): "Software rejuvenation: Analysis, module and applications," *Proc. Twenty-fifth Symposium on Fault Tolerant Computing*, pp. 381-390.

[30] Lindsay, B. (2005): Personal communication with M. Grottke.

[31] Marshall, E. (1992): "Fatal error: How Patriot overlooked a Scud," *Science*, vol. 255, p. 1347.

[32] Okamura, H., S. Miyahara, T. Dohi, and S. Osaki (2001): "Performance evaluation of workload-based software rejuvenation scheme," *IEICE Transactions on Information and Systems*, vol. E84-D, pp. 1368-1375.

[33] Okamura, H., S. Miyahara, and T. Dohi (2003): "Dependability analysis of a transaction-based multi server with rejuvenation," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E86-A, pp. 2081-2090.

[34] Parnas, D.L. (1994) : "Software Aging," *Proc. Sixteenth International Conference on Software Engineering*, pp. 279-287.

[35] Pfening, A., S. Garg, A. Puliafito, M. Telek, and K.S. Trivedi (1996): "Optimal rejuvenation for tolerating soft failures," *Performance Evaluation*, vol. 27-28, pp. 491-506.

[36] Pitts, C.E. (1989): "Parallel processing support: So what is a 'Heisenbug' anyway?" *Proc. Seventeenth ACM SIGUCCS Conference on User Services*, pp. 237-242.

[37] Raymond. E.S. (1991): *The New Hacker's Dictionary,* MIT Press, Cambridge.

[38] Shereshevsky, M., J. Crowell, B. Cukic, V. Gandikota, and Y. Liu (2003): "Software aging and multifractality of memory resources," *Proc. International Conference on Dependable Systems and Networks 2003*, pp. 721-730.

[39] Shetti, N.M. (2005): "Heisenbugs and Bohrbugs: Why are they different?" Technical Report DCS-TR-580, Department of Computer Science, Rutgers University, Piscataway.

[40] Steele, G.L., D.R. Woods, R.A. Finkel, M.R. Crispin, R.M. Stallman, and G.S. Goodfellow (1983): *The Hacker's Dictionary – A Guide to the World of Computer Wizards*, Harper & Row, New York.

[41] Suzuki, H., T. Dohi, K. Goševa-Popstojanova, and K.S. Trivedi (2002): "Analysis of multistep failure models with periodic software rejuvenation," *Advances in Stochastic Modelling*, J.R. Artalejo and A. Krishnamoorthy, eds., Notable Publications, Neshanic Station, pp. 85-108.

[42] Trivedi, K.S. and K. Vaidyanathan (2006): "Software aging and rejuvenation," *Wiley Encyclopedia of Computer Science and Engineering*, B. Wah, ed., Wiley, New York. (To appear).

[43] Vaidyanathan, K., R.E. Harper, S.W. Hunter, and K.S. Trivedi (2001): "Analysis and implementation of software rejuvenation in cluster systems," *ACM Performance Evaluation Review*, vol. 29, pp. 62-71.

[44] Vaidyanathan, K., D. Selvamuthu, and K.S. Trivedi (2002): "Analysis of inspection-based preventive maintenance in operational software systems," *Proc. Twenty-first International Symposium on Reliable Distributed Systems*, pp. 286-295.

[45] Vaidyanathan, K. and K.S. Trivedi (2001): "Extended classification of software faults based on aging," *Supplemental Proc. Twelfth International Symposium on Software Reliability Engineering*, pp. 27-28.

[46] Vaidyanathan, K. and K.S. Trivedi (2005): "A comprehensive model for software rejuvenation," *IEEE Transactions on Dependable and Secure Computing*, Vol. 2, pp. 124-137.

[47] van Tilborg, A.M. (1988): "Instrumentation for distributed computing systems," *ACM SIGARCH Computer Architecture News*, vol. 16, pp. 20-25.

[48] Wikipedia (2005): "Bohr bug," Last modified May 2, 2005, URL = http://en.wikipedia.org/wiki/Bohr_bug (Link verified on May 26, 2005).

[49] Wikipedia (2005): "Heisenbug," Last modified May 23, 2005, URL = http://en.wikipedia.org/wiki/Heisenbug (Link verified on May 26, 2005).

[50] Wikipedia (2005): "Mandelbug," Last modified April 21, 2005, URL = http://en.wikipedia.org/wiki/Mandelbug (Link verified on May 26, 2005).

[51] WikiWikiWeb (2004): "Bohr bug," Last modified November 2, 2004, URL = http://c2.com/cgi/wiki?BohrBug (Link verified on May 26, 2005).

[52] WikiWikiWeb (2004): "Heisen bug," Last modified January 25, 2004, URL = http://c2.com/cgi/wiki?HeisenBug (Link verified on May 26, 2005).

[53] WikiWikiWeb (2004): "Heisen bug examples," Last modified January 21, 2004, URL = http://c2.com/cgi/wiki?HeisenBugExamples (Link verified on May 26, 2005).

[54] WikiWikiWeb (2004): "Mandelbug," Last modified December 21, 2004, URL = http://c2.com/cgi/wiki?MandelBug (Link verified on May 26, 2005).

[55] Winslett, M. (2005): "Bruce Lindsay speaks out," *ACM SIGMOD Record*, vol. 34, pp. 71-79.

[56] Xie, W., Y. Hong, and K.S. Trivedi (2005): "Analysis of a two-level rejuvenation policy," *Reliability Engineering & System Safety*, vol. 87, pp. 13-22.

(Michael GROTTKE / Kishor S. TRIVEDI)

Michael GROTTKE

Michael Grottke received the M.A. degree in economics (1999) from Wayne State University, Detroit, USA and the Diplom degree in business administration (2000) as well as the Dr. rer. pol. degree in statistics (2003) from the University of Erlangen-Nuremberg, Nürnberg, Germany. For his dissertation, based on the research he conducted for the EU-funded project PETS (Prediction of software Error rates based on Test and Software maturity results), he was awarded the Promotional Award for Science of the State Bank of Bavaria. He is currently working as a

research associate in the Department of Electrical and Computer Engineering at Duke University, Durham, USA, on a Fellowship from the German Academic Exchange Service (DAAD). His research interests include software reliability, software process maturity and software rejuvenation as well as stochastic point processes and combinatorial problems.

Kishor S. TRIVEDI

Kishor S. Trivedi holds the Hudson Chair in the Department of Electrical and Computer Engineering at Duke University, Durham, USA. 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, with a thoroughly revised second edition being published by John Wiley. He has also published two other books entitled, Performance and Reliability Analysis of Computer Systems, published by Kluwer Academic Publishers and Queueing Networks and Markov Chains, John Wiley. His research interests are in reliability and performance assessment of computer and communication systems. He has published over 300 articles and lectured extensively on these topics. He has supervised 39 Ph.D. dissertations. He has made seminal contributions in software rejuvenation, solution techniques for Markov chains, fault trees, stochastic Petri nets and performability models. He has actively contributed to the quantification of security and survivability. He is a Fellow of the Institute of Electrical and Electronics Engineers. He is a Golden Core Member of IEEE Computer Society. He is a co-designer of HARP, SAVE, SHARPE and SPNP software packages that have been well circulated.