The Fundamentals of Software Aging

Michael Grottke[*], Rivalino Matias Jr.[‡], and Kishor S. Trivedi[‡]
[*]*University of Erlangen-Nuremberg, Germany; Michael.Grottke@wiso.uni-erlangen.de*
[‡]*Duke University, USA; {rivalino, kst}@ee.duke.edu*

Abstract

Since the notion of software aging was introduced thirteen years ago, the interest in this phenomenon has been increasing from both academia and industry. The majority of the research efforts in studying software aging have focused on understanding its effects theoretically and empirically. However, conceptual aspects related to the foundation of this phenomenon have not been covered in the literature. This paper discusses foundational aspects of the software aging phenomenon, introducing new concepts and interconnecting them with the current body of knowledge, in order to compose a base taxonomy for the software aging research. Three real case studies are presented with the purpose of exemplifying many of the concepts discussed.

1. Introduction

Since the notion of software aging was introduced thirteen years ago [7], the interest in this phenomenon has been increasing from both academia and industry. The occurrence of software aging in real systems has been documented in the literature [3], [5], [8]. Many approaches have been used to study this phenomenon. The majority of these research efforts have concentrated on understanding its effects theoretically [2], [10] and empirically [5], [8], [11]. Moreover, the search for mitigation resulted in the so-called software rejuvenation techniques. Because the main research efforts have concentrated on the aging effects and their mitigation, conceptual aspects related to the foundation of this phenomenon have not been covered in the literature so far.

This paper discusses the foundations of the software aging phenomenon. We focus on conceptual and practical aspects involved and formulate a set of definitions that we consider essential for composing a taxonomy for the software aging and rejuvenation (SAR) research.

2. Physics of software failures

Software aging is usually a consequence of software faults. This section therefore revisits the body of knowledge related to the taxonomy of faults and other dependability concepts. In Section 3, we will discuss the specific nature of these concepts in the context of software aging.

According to system theory, a **system** is a collection of inter-operating elements (or components); the system boundary separates the system from its **environment**. For example, a single software system includes the hardware, the operating system (OS) and the applications as its elements; however, the users and other software systems are part of its environment. Each system can itself be an element of another (higher-level) system.

A **service failure** (or simply **failure**) of a system is a deviation of the service delivered by this system from its specification. Such a deviation can be in the form of incorrect service, or no service at all. Moreover, the system can be in a degraded mode in which service is slow or limited; this case is referred to as a **partial failure**. In [1], the pathology of a failure is discussed in terms of the "**chain of threats**", showing the causal relationships between fault, error, and failure; this chain is illustrated in Figure 1.

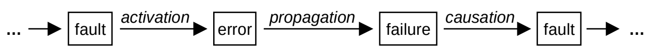

Figure 1. General "chain of threats" [1]

An **error** is that part of the internal system state which may lead to a failure occurrence. Errors can be transformed into other errors. For example, an error in component 1 can reach the service interface of this component, used by component 2, thus causing an error in this second component. This transformation of errors is referred to as **error propagation**. The error propagation leads to a system failure if the error is propagated to the service interface of the system, causing the service provided to deviate from its specification. Note that the term error propagation is used for the transformation of an error into another error both with and without the causation of a failure occurrence. The "chain of threats" in Figure 1 explicitly only shows the latter kind of error propagation.

Each error, in turn, is caused by the activation of a **fault**. A fault that (currently) does not produce an error is said to be dormant. Applying inputs containing an **activation pattern** to a dormant fault can make this fault cause an error; this is referred to as **fault activation**. The rate with which a dormant fault will become active therefore depends heavily on the intensity and the way in which a system is used; a quantitative characterization of the latter aspect is the **operational profile** [9].

The failure of a system causes a fault in those systems receiving service from it, as well as in those higher-level systems in which it is contained.

Avižienis et al. [1] present a scheme for classifying faults according to eight criteria, e.g., the system boundaries (internal or external), the phenomenological cause (natural or human-made), the objective (malicious or non-malicious), the persistence (permanent or transient), the dimension (hardware or software), and the phase of creation or occurrence (development or operation). Based on this classification scheme, faults in the software code, referred to as **software flaws** by Avižienis et al. and often simply called **software faults** or **(software) bugs**, can be described as internal human-made non-malicious permanent software development faults.

Furthermore, Avižienis et al. mention a classification of faults according to their activation/propagation reproducibility: Faults whose activation and error propagation is reproducible are called **solid**, or hard, faults, whereas faults whose activation/propagation is not systematically reproducible are called **elusive**, or soft, faults. Especially for software bugs that permanently reside in the code (until they are detected and removed) it seems counterintuitive that repeating the actions (like user inputs) that previously caused a failure will not lead to another failure when repeated. It is therefore of interest to study the properties that a software fault needs to feature in order to have the potential to be non-reproducible. However, the classification into solid and elusive faults is subjective, because it also depends on the knowledge of the respective user about the fault activation and error propagation mechanism of the fault in question, as well as on the operational behavior of the user. A specific fault could be considered "solid" by one user, but "elusive" by another one.

The definitions of Mandelbug and Bohrbug [4], [6] classify the fault types using more objective criteria related to properties of the fault itself. A **Mandelbug** has the potential to be difficult to isolate and to cause failures that are not systematically reproducible. As an example, consider the code of an application in which the initialization of a variable is missing. If a debugger initializing all variables by default can prevent the fault from causing a failure, then this fault is Mandelbug, because the debugger, a part of the system-internal environment of the application, can affect fault activation. A **Bohrbug**, on the other hand, is an easily isolated fault that always manifests consistently under a well-defined set of conditions, because its activation and error propagation lack "complexity" as defined in [4], [6]. Bohrbug is the complementary antonym of Mandelbug. The Mandelbug definition [4], [6] uses the concept of the **(software-)system-internal environment** of an application. While the environment of an application consists of all the entities outside the system boundaries of the application (e.g., operating system, hardware, users, power supply network, office building), its system-internal environment only includes those entities outside the application that are located within the boundaries of the computer system. In particular, users and office infrastructure are excluded. The system-internal environment of an application thus contains the hardware, the OS, the other applications, etc.

3. Fundamental concepts of software aging

Software aging is the name given to a phenomenon empirically observed in many software systems: A general characteristic of this phenomenon is the fact that, as the runtime period of the system or process increases, its failure rate also increases. Again, a failure can take the form of incorrect service (e.g., erroneous outcomes), no service (e.g., halt and/or crash of the system), or partial failure (e.g., gradual increase in response time). For physical systems, aging is well-known to occur in the wear-out phase. In the bathtub curve [12] this behavior is illustrated by an increasing failure rate after a certain stable period of life. However, while hardware faults can come into existence due to wear and tear, it seems impossible, at first sight, that software bugs, which are permanent development faults, can be responsible for software aging. Nevertheless, many **aging-related failures** of software systems are indeed the consequence of software faults. To explain their pathology, Figure 2 shows a modified version of the general "chain of threats" specific to these aging-related (AR) failures.

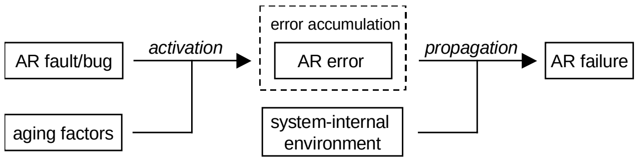

Figure 2. "Chain of threats" for an aging-related failure

The propagation of **aging-related errors** (i.e., errors that can cause AR failures) requires the state of the system-internal environment to meet certain criteria. Most AR errors that do not (yet) cause a failure are kept in the internal system state, where such **errors accumulate** if there are successive activations of the **aging-related fault**. It is usually exactly this accumulation of AR errors that leads the system-internal environment to a state in which AR errors are propagated causing AR failures.

Using the classification scheme due to Avižienis et al. [1], AR faults can be further classified: Internal AR faults are usually human-made non-malicious permanent software development faults, i.e., bugs in the program code; these are referred to as AR (software) bugs. An example of such an AR bug is the omission of commands that release memory that was dynamically allocated earlier. On the other hand, external AR faults are not static faults in the code of the system under study, but they are caused by external events. However, such events could in turn be caused by AR bugs in *other* systems.

For two reasons, all AR bugs are Mandelbugs: First, there can be a long time delay between fault activation and the final failure occurrence; it is exactly such a delay that allows the accumulation of errors. Second, the failure occurrence caused by error accumulation and/or error propagation can depend on the system-internal environment.

We refer to the activation patterns of an AR fault (i.e., the factors or the combinations of factors that activate this fault) as its **aging factors**. While **internal** aging factors are internal events (e.g., function calls triggering the execution of those parts of the code where the AR bug is located), **external** aging factors are triggers directly released by elements in the environment of the system, like its users.

The **time to aging-related failure** is the random time period from the moment of the system startup or process creation until a failure occurrence due to AR faults. The probability distribution of this random time (as well as its expected value, the mean time to aging-related failure) is mainly influenced by the intensity with which the system gets exposed to aging factors; it is therefore influenced by the quantity and type of work performed by the process, which we refer to as the **work journey** of the process, and thus by the operational profile and the usage intensity of the system.

In those cases in which the successive activation of AR faults causes the accumulation of AR errors, the **software aging effect** consists in the gradual shifting from a correct internal state to an erroneous and failure probable one. The extent with which each element in the cause-and-effect-chain (external aging factor/internal aging factor/fault activation/error occurrence) contributes to this effect can be measured in terms of the **error accumulation scale**, specifying the magnitude of the effect that each element in the chain has on the subsequent element.

Different types of aging effects have been observed; based on their common characteristics we have created an initial set of **aging effect classes**, which are shown in Table 1.

Table 1. Classes of aging effects

Basic class	Extension	Examples
Resource leakage	(1) OS-specific (2) App-specific	- Unreleased • *Memory* (1, 2) • *File handlers* (1) • *Sockets* (1) - Unterminated • *Processes* (1) • *Threads* (1, 2)
Fragmentation	(1) OS-specific (2) App-specific	- Phys. memory (1) - File system (1) - Database files (2)
Numerical error accrual	(1) OS-specific (2) App-specific	- Round-off (1, 2)
Data corruption accrual	(1) OS-specific (2) App-specific	- File system (1) - Database files (2)

Aging effects can also be classified into **volatile** and **non-volatile** effects. They are considered volatile if they are removed by re-initialization of the system or process affected, for example via a system reboot. In contrast, non-volatile aging effects still exist after reinitializing of the system/process. Physical memory fragmentation and OS resource leakage are examples for volatile aging effects. File system and database metadata fragmentation are examples for non-volatile aging effects. Another example of a non-volatile aging effect is numerical error accrual preserved between system reboots via checkpoint mechanism. Note that hibernation and similar mechanisms (e.g., standby), which preserve the system memory (and thus the aging effects present in it) between system reinitialization, allow the majority of intrinsically volatile aging effects to persist even after system/process reinitialization.

Aging effects in a system can only be detected while the system is running, by monitoring **aging indicators**. Aging indicators are markers for aging detection, like antigens are markers to detect cancer disease. In the simplest approach, system health is considered a latent binary variable distinguishing between a stable internal state on the one hand and a failure probable state on the other [7]. Aging indicators are then explanatory variables that individually or in combination can suggest whether or not the system is healthy. They can be considered at several levels, such as OS, application process, application component, middleware, virtual machine (VM), and VM monitor (VMM). We can classify aging indicators in two general classes according to their granularity:

1. **System-wide** indicators provide information related to subsystems shared by several running applications. Examples of shared subsystems are OS, middleware, VM and VMM, among others. Indicators in this category are often used to evaluate the aging effects on the system as a whole and not for a specific application, since the shared na-

ture of their environment may cause noise in the captured data. Examples of aging indicators in this category are free physical memory, used swap space, file table size, and system load.

2. **Application-specific** indicators provide specific information about an individual application process, thus giving more accurate information about it than system-wide indicators. When the application process is running under a VM (e.g., Java programs), then aging indicators applied to the VM can also be used as a reference for the application being executed under the VM. Examples of aging indicators in this category are resident set size of the process, Java VM heap size, and response time.

An example for how an aging-related failure can be analyzed based on the concepts discussed above is shown in Table 2.

While in many cases software aging is due to AR bugs, even in the absence of such faults in the code aging effects can occur as a consequence of the natural dynamics of a system's behavior. This kind of aging is thus referred to as **natural aging**. Among the examples for natural aging are the fragmentation problems experienced by file systems, database index files, and main physical memory. Such aging effects are not related to a faulty code or design, but they are a consequence of the system/application usage over its lifetime. For example, in database servers the fragmentation class of aging effects can be captured via aging indicators such as the degree of index-related metadata fragmentation (e.g., Tablespace fragmentation value in the Oracle DBMS).

Considering not only the software system itself, but the higher-level system including its users, one could argue that natural aging *is* due to faults, namely to mistakes on the part of the operators; e.g., in the case of fragmentation problems the operator has made the mistake of not executing defragmentation routines. However, as such measures only mitigate the effects of natural aging, even "correct" behavior of the operators would not have solved the underlying problem. This is in contrast with software aging caused by AR bugs, discussed above, where fixing the software fault permanently removes the aging effect.

The notion of natural aging without existence of a fault should not give the impression that *any* service degradation or *any* increase in the failure rate of a software system is considered software aging. Otherwise, this concept would also include increases in the failure rate that are merely due to changes in the operational profile or due to an increase in the intensity with which the system is being used. We therefore propose the following characteristics of the **software aging** phenomenon:

1. The aging effect is not reversible without external intervention. For example, the accumulated internal error states caused by successive activations of aging-related faults do not disappear without external intervention; at the very best, no further errors may accumulate in the future, during periods in which the system is not exposed to any aging factors. Based on this characterization, an increasing failure rate due to the queuing of jobs in an overloaded system is not considered software aging, since the accumulated set of jobs not yet served will be reduced (and will finally disappear) once the workload falls below a certain threshold; see the Apache Web server example (Section 4.2).

Table 2. Example for analyzing the pathology of an AR failure

Failure as perceived by the user:	File server does not respond.
Failure as perceived by the troubleshooter:	Operating system halted.
State of system-internal environment required for error propagation:	Insufficiency of main physical memory (availability of < 350 kB)
Error that leads to the failure:	A memory leak inside the file server process.
Aging effect:	Loss (leakage) of 100 kB per activation of the AR fault.
AR fault that causes the error:	A wrong value of the parameter used in the free() function in the comm.c program file.
Location of the AR fault:	(x) Internal / ( ) External
Internal aging factor(s):	A call of the write_record() function.
External aging factor(s):	The arrival of a packet carrying out the command SAVE_FILE.
Failure mechanics: External aging factor => Internal aging factor => Fault activation => Error accumulation + Required state of system-internal environment => Failure	[Packet Save File] => write_record() => {free(), Line_200} => Leakage of 100 kB + Availability of < 350 kB of main physical memory => System crash
Error accumulation scale:	1 : 1 : 1 : 100 kB

2. The aging effect depends on the clock time (the time since system startup or process creation) only if this clock time constitutes a part of the system-internal environment influencing error accumulation and/or propagation; see the Patriot example (Section 4.3).
3. The CPU time influences the aging effect only if the process's work journey, during its runtime, triggers aging-related faults or causes natural aging effects, or if the CPU time constitutes a part of the system-internal environment influencing error accumulation and/or propagation.

According to the first characteristic, software aging can be dealt with by external intervention. The fault tolerance technique using environmental diversity to mitigate the aging effects of a system is known as **software rejuvenation**. It involves occasional resets of the internal system state, thus cleaning accumulated error conditions and constraining the possible domain of the system-internal environment. Software rejuvenation can be implemented at several granularity levels and applied to many types of elements in a system, such as the operating system, individual software processes, or persistent data objects (like file system metadata and database index files).

Software rejuvenation can be triggered at intervals derived from analytical system models, or based on aging indicators monitored. Discovering an efficient and effective set of system variables that are the best aging indicators is a variable selection problem. The quality of the aging indicators directly influences the accuracy of the timing with which rejuvenation is triggered. It thus determines the costs (e.g., downtime during rejuvenation) and benefits (e.g., avoided downtime by unexpected failures) of the rejuvenation mechanism.

It has been argued that software rejuvenation is not an engineering solution. Rather than dealing with the symptoms of aging, one should locate and fix the underlying aging-related fault, which solves the problem for good. However, this attitude neglects important advantages of the software rejuvenation technique: It can be employed to remove aging effects and avoid failure occurrences if the location of the AR fault, or even the very fact of its existence, is unknown. Moreover, software rejuvenation can be preferable due to technical, economical or time limitations. Finally, in the case of natural aging, where the aging effects are the consequence of normal system behavior, rejuvenation is indeed the only solution.

4. Real case analysis

4.1. Cisco Catalyst switch

Cisco Systems, Inc. [3] reported a software fault affecting several network switch products, including Catalyst 2900, 4000, 5000, and 6000. The fault was related to the telnet service provided by these products as well as to their web management interface enabling the network manager to access the switches through a remote virtual console. Based on the problem report analysis, this fault can clearly be categorized as an aging-related fault.

The effect of activating the fault was a memory leak in the *telnetd* processes that implemented these services. The accumulation of such errors gradually degraded the whole system state in terms of physical memory availability. If the period of uninterrupted execution was long enough to exhaust the physical memory, then the error propagation caused the failure of the switch not to execute any other processes, such as forwarding traffic or management. The aging-related fault was activated under the following two circumstances:

1. A telnet connection was closed due to failed authentication; or
2. A successful login had an extremely short duration.

These circumstances, directly related to the system usage, represent two confirmed external aging factors forming activation patterns. If two identical products were deployed in the same environment but only one of them was exposed to those activation patterns, then just that one suffered from the abovementioned aging effect.

The occurrence of a switch failure depended on the state of the system-internal environment, which in turn was influenced by the extent of previous error accumulation. In this example, the relevant part of the system-internal environment was the physical memory. Deployed configurations of the same switch model could have different RAM capacities; naturally, for systems with a larger amount of RAM the mean time to aging-related failure was longer than for system with less RAM.

4.2. Apache Web server

A Web server running Apache version 1.3.14 on a Linux platform was put in an overload condition by synthetic requests. Used swap space showed both a statistically significant increasing trend as well as a "seasonal" pattern repeating every seven days. The seasonal behavior was caused by the weekly log-rotation triggering Apache to kill all of its child processes. As a consequence of this triggering event, swap space was released for two reasons:

1. Swap space occupied by the Apache child processes, including unused memory not released earlier due to memory leaks, was released because these processes were killed.
2. Swap space occupied by low-priority processes not executed but directed to the swap space due to the

overload condition was released because these processes were able to execute and terminate as soon as resources were freed by killing the Apache child processes.

Although not planned as a software rejuvenation technique, the “external” intervention of the weekly log-rotation rejuvenated parts of the system. However, most or all of the used swap space mentioned under (2) and maybe some of the swap space mentioned under (1) could probably have been released even without this incidental rejuvenation, by simply decreasing the intensity with which the system was used; to this extent the decrease in available swap space experienced was not an aging effect. Nevertheless, that part of the increase in swap space usage that could only be counteracted with the partial rejuvenation, as well as the increase in swap space usage not affected by it (embodied in the statistically significant time trend) can be considered aging effects [5].

4.3. Patriot missile defense system

To project a target’s trajectory, the weapons-control computer of the Patriot Missile System required its velocity and the time as real values. The system kept time internally as an integer, counting tenths of seconds and stored in a 24-bit register. This register could have been used for counting 16777217 tenths of seconds, or roughly 19.4 days, without causing an overflow. However, whenever a target was spotted, the weapons-control computer needed to convert the integer value into a real value. This conversion caused an inaccuracy (error) whose magnitude was proportional to the length of time that the system had been continuously running. This inaccuracy immediately led to an imprecision in the calculated range where the detected target was expected next (i.e., the error was transformed into another error). If this imprecision was small enough, then the target was tracked, classified and – if necessary – intercepted; i.e., the error propagation did not lead to a failure occurrence, and the errors related to the respective target became irrelevant after the fact (they did not accumulate, and they did not propagate any further). The aspect of the system-internal environment determining whether or not the error resulted in a failure was the system runtime: After a system runtime of about eight hours, the imprecision in converting the integer value became so large that the target range calculated for a detected Scud missile was too far off the real location for tracking and intercepting the missile. On 25 February 1991, the Patriot system located at Dhahran, Saudi Arabia, failed to intercept an arriving Scud missile after a runtime of more than 20 hours since last reboot. [6]

Even after a runtime of more than eight hours, changes in the future work journey could have reduced the failure rate due to the aging-related fault. For example, if no further target (Scud missile, etc.) had ever arrived, then no further target range calculation would have been necessary, and the rate of missing any target would have been zero. However, this change in the future work journey would not have affected the factor in the system-internal environment influencing error propagation, i.e., the system time kept in the 24-bit register. Only external intervention (system reboot) was able to influence this factor.

5. References

[1] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[2] Y. Bao, X. Sun, and K. S. Trivedi. “A workload-based analysis of software aging and rejuvenation,” *IEEE Transactions on Reliability*, vol. 55, no. 3, pp. 541–548, 2005.

[3] Cisco Systems, Inc. “Cisco security advisory: Cisco Catalyst memory leak vulnerability,” Document ID: 13618, 2001. URL = http://www.cisco.com/warp/public/707/cisco-sa-20001206-catalyst-memleak.shtml.

[4] M. Grottke and K. S. Trivedi. “Software faults, software aging, and software rejuvenation,” *Journal of the Reliability Association of Japan*, vol. 27, no. 7, pp. 425–438, 2005.

[5] M. Grottke, L. Li, K. Vaidyanathan, and K. S. Trivedi. “Analysis of software aging in a web server,” *IEEE Transactions on Reliability,* vol. 55, no. 3, pp. 411–420, 2006.

[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] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. ”Software rejuvenation: analysis, module and applications,” In *Proc. Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pp. 381–390, 1995.

[8] R. Matias and P. J. Freitas. “An experimental study on software aging and rejuvenation in web servers,” In *Proc. 30th Annual International Computer Software and Applications Conference*, vol. 1, pp. 189–196, 2006.

[9] J. D. Musa. “Operational profiles in software reliability engineering,” *IEEE Software*, vol. 10, no. 2, pp. 14–32, 1993.

[10] M. Shereshevsky, J. Crowell, B. Cukic, V. Gandikota, and Y. Liu. “Software aging and multifractality of memory resources,” In *Proc. IEEE International Conference on Dependable Systems and Networks*, pp. 721–730, 2003.

[11] L. Silva, H. Madeira, and J. G. Silva. “Software aging and rejuvenation in a SOAP-based server,” In *Proc. Fifth IEEE International Symposium on Network Computing and Applications*, pp. 56–65, 2006

[12] P. Tobias and D. Trindade. *Applied Reliability*, 2nd edition. Kluwer Academic Publishers, Boston, 1995.