A Software Maintenance Methodology: An Approach Applied to Software Aging

Jean Araujo*, Carlos Melo†, Felipe Oliveira†, Paulo Pereira†, and Rubens Matos‡
*Universidade Federal do Agreste de Pernambuco, Garanhuns, Brazil
†Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil
‡Instituto Federal de Sergipe, Lagarto, Brazil
jean.teixeira@ufape.edu.br, {casm3, fdo, prps}@cin.ufpe.br, rubens.junior@ifs.edu.br

***Abstract*—The increasing use of computational systems has highlighted concerns about attributes that may influence the quality of service, such as performance, availability, reliability, and maintenance capacity. Failures in the software development process may impact these attributes. Flawed code and overall software misdesign may cause internal errors, leading to system malfunction. Some errors might be identified and fixed during the software testing process. However, other errors may manifest only during the production stage. This is the case of the software aging phenomenon, which is related to the progressive degradation that a software performance or reliability suffers during its operational life. This paper proposes a methodology for software maintenance that is tailored to identify, correct, and mitigate the software aging effects. If the source code can be modified and a new version deployed with minimal impact, thus data from aging detection is used for corrective maintenance, i.e., for fixing the bug the causes the aging effects. If the software cannot be fixed nor its version updated without long system interruption or other bad consequences, then our approach can mitigate the aging effects, in a preventive maintenance to avoid service outages. The proposed methodology is validated through both Stochastic Petri Net (SPN) models and experiments in a controlled environment. The model evaluation considering a hybrid maintenance routine (preventive and corrective) yielded an availability of 99.82%, representing an annual downtime of 15.9 hours. By contrast, the baseline scenario containing only reactive maintenance (i.e., repairing only after failure) had more than 1342 hours of annual downtime—80 times higher than the proposed approach.**

***Index Terms*—Software aging and rejuvenation, software maintenance, methodology.**

I. Introduction

The increasing use of computer systems brings numerous benefits, such as convenience, safety, and time and resource savings. This trend has became even sharper with the adoption of software as a service model [1]. However, such benefits demand great responsibilities from service providers, who need to maintain high levels of quality for increasingly demanding audiences. Thus, attributes such as performance, availability, reliability, and maintenance capacity have gained a lot of attention. Failures throughout the software development process may impact negatively on these attributes, causing a system malfunction and even financial and reputation losses.

The testing phase is a very important part of software development lifecycle, as it allows for fault correction before the software is in use. However, some errors may manifest only in a production system. In this case, the adoption of corrective or preventive maintenance may avoid system malfunctions and even catastrophic failures. A commonly neglected phenomenon is the software aging, which is related to performance loss during its operational life [2]. Although, there is a countermeasure called software rejuvenation, a proactive action that can bring the system to its initial state. Software rejuvenation can be applied in several ways depending on the type of system, but it includes actions such as system reboot and application restart [3]. The identification of aging occurrence and proper scheduling of rejuvenation actions are essential for improving Quality of Service (QoS) metrics with minimum intervention.

This paper proposes a proactive maintenance methodology for computing systems affected by software aging. The proposed methodology may be applied with two maintenance policies: corrective and preventive. In order to demonstrate the effectiveness of the proposed methodology, we carry out experiments in a controlled environment and propose availability models to evaluate distinct scenarios. In addition, the experiments also served to validate the availability models.

This paper is structured as follows. Section II discusses some related works. Section III introduces basic concepts of software aging and rejuvenation, while also describes the availability evaluation background that is important to understand this paper. The proposed methodology is presented in detail in Section IV. The availability models are presented in Section V. Section VI outlines the experimental results and the models evaluation. Finally, Section VII draws some conclusions of the paper and presents some directions for future works.

II. Related Works

Although software aging is not a new research field, it still has many issues to be addressed. Many critical applications suffer from software aging yet; therefore, significant work must be done to overcome the challenges brought by this phenomenon. In this section, we discuss the work done by previous researchers on memory-related software aging and corresponding availability evaluation.

A hybrid approach was proposed by Li et al. [4], that combined measurements and models to build a probabilistic aging indicator. In their work, they proved that the proposed aging indicator was more effective than the traditional ones (e.g., resident memory consumption). However, differently

from our work, the authors did not conduct an availability evaluation in order to see how their approach affects the system uptime. Dang et al. [5] introduced a rejuvenation method with cyclic inspection for stand-alone systems. The authors stated that their proposal works according to the performance degradation aspects of the system. Although the authors proposed a Markov chain model, they did not perform a sensitivity analysis to indicate which component is the bottleneck of the system.

A methodology to perform software aging and software rejuvenation experiments was introduced by Torquato et al. [6]. Their methodology was divided into three phases: the stressing phase, the waiting phase, and the rejuvenation phase. The stressing phase seeks to provoke failures in the system by increasing the workload. The waiting phase pauses the workload and sees the system state. The rejuvenation phase seeks to catch the problems caused by software rejuvenation actions. Differently from our work, the authors did not provide models. Pereira et al. [7] proposed a tool that predicts the best moment to perform software rejuvenation. The authors monitored the system's resources as a time series, then they used the time series to predict when it was the best moment to perform software rejuvenation activities. Their proposal used five well-known forecasting techniques: Drift, Exponential Smoothing, Holt, Holt-Winters, and ARIMA. However, the authors did not perform an availability evaluation.

Cotroneo et al. [8] investigated the software aging in the Android OS. Their work highlighted if smartphones from several vendors, under different usage situations and settings, are affected by software aging. In other work [9], they presented an aging detection and rejuvenation tool for Android. The authors did not provide availability models.

III. Background

In this section, we see the essential ideas of availability evaluation and software aging and rejuvenation. These theories are important to have a clear understanding of our proposal, including the features surrounding the case studies.

A. Availability Evaluation

System administrators aim at increasing the system's availability as much as possible while decreasing the related cost [10]. Availability evaluation measures the probability of finding the target system working properly. We assume that a random variable $X(t)$ is the system's state at a moment t. $X(t) = 1$ means the state up of the system; in other words, the system is working properly. On the other hand, $X(t) = 0$ means that the system is down (i.e., not working). If we assume a random variable T corresponding the necessary time to reach $X(t) = 0$, knowing that the system starts at $X(0) = 1$, therefore, T is the time that the system takes to fail. $F_T(t)$ is its cumulative distribution function, and $f_T(t)$ is its probability density function [10]. Equation 1 depicts it.

$$\begin{aligned} F_T = 0 \quad and \lim_{t\to\infty} f(t), \\ f_T(t) = \frac{dF_T(t)}{dt}, \\ f_T(t) \geq 0 \quad and \int_0^\infty f_T(t)dt. \end{aligned} \tag{1}$$

The availability of a system is represented by Equation 2, where up means the time that the system is working, and down means the time that the system is not working. The availability value is a number between 0 and 1. For example, if we calculate the availability of a system and its result is 0.9735, then it means that the system is working 97.35% and not working 2.65% of the time. Availability can also be represented by the relationship of the mean time to failure (MTTF) and the mean time to repair (MTTR), as expressed in Equation 5. Equation 3 and Equation 4 show the formulas of MTTF and MTTR, respectively [10].

$$A = \frac{up}{up + down} \tag{2}$$

$$MTTF = \int_0^\infty (1 - F_T(t))dt \tag{3}$$

$$MTTR = MTTF \times \frac{1 - A}{A} \tag{4}$$

$$A = \frac{MTTF}{MTTF + MTTR} \tag{5}$$

B. Petri Nets

Fault trees (FTs) and Reliability Block Diagrams (RBDs) are widely used to evaluate the availability of systems [10]. However, they cannot capture some dependencies that occur in real systems. On the other hand, Petri Nets are capable of capturing these dependencies and dynamic behaviors such as concurrency, synchronization, and so forth. Petri Nets allow us to model and analyze discrete events, which are way too complex to be analyzed using automata or queue modeling [10]. Petri Nets also allow us to represent useful information about the system's behavior in a rich but not too complex graphical model.

The structural features of a Petri Net correspond to a bipartite directed graph comprising some basic elements: places, transitions, arcs, and tokens. Figure 1 depicts these basic elements. The places represent the states of the system; the transitions represent the actions or events that may occur; on the other hand, the arcs link the places to transitions or transitions to places, which means that if the system is in some state, it can perform some action or some event may occur, which will lead to another state.

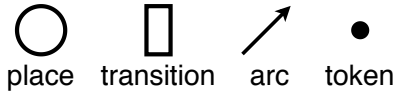

Fig. 1: Basic Elements

In order to execute an action in a Petri Net, the action needs to be associated with some condition; in other words, there is a relation between places and transitions that enables or not the execution of an action. After that, some places may have the number of tokens updated, which may enable or not other actions or events [10]. The arcs represent how the tokens flow in the system, and the tokens represent the state that the system is at some point in time. Stochastic Petri Nets (SPNs) have transitions that may be timed or immediate. Timed transitions have delays that are considered random variables with exponential distribution; on the other hand, immediate transitions have delay equal to zero.

C. Software Aging and Rejuvenation

In summary, software aging is a gradual phenomenon that compromises the performance of a system as it runs [7]. In other words, the software aging phenomenon increases the failure rate and/or performance degradation of a system as time goes by. It usually happens because there is an accumulation of programming errors that keeps absorbing the system's resources making it crash eventually [11].

This phenomenon is known by researchers for a long time; for instance, a sign of software aging was observed in the Safeguard military system in the 1960s, when the system crashed and the system error report presented that the buffers were full [12]. Services have increased in capacity and complexity as the time goes by, consequently, software aging has been observed in long-running systems, such as telecommunication switching, billing software, and so forth [13].

Software programming errors are generally responsible for causing the software aging phenomenon; these errors may cause failures as they change the quality of the service delivered, and sometimes these errors make the system stop, consequently, no service is provided at all [3]. The aging consequences may be discovered by observing the operating system and analyzing resource usage data [14]. The more complex the provided service is, the more programming errors it may have, consequently, it is impractical to fix them all at once [7].

Software rejuvenation is a cost-effective concept to overcome the software aging phenomenon [14]. It usually works by restarting an application to clean the internal state of the system [2]. However, performing software rejuvenation activities cause system unavailability during the process, affecting directly the uptime of a service [2], [14]. The cost of downtime caused by software rejuvenation is expected to be much lower than the costs caused by a failure. Nevertheless, as all preventive maintenance policies, the software rejuvenation action needs to be scheduled carefully, otherwise, it may cause more problems than it solves.

There are many software rejuvenation strategies; however, we highlight the one that is based on monitoring the software aging rate and real-time service requests. The real-time data about service requests is observed and compared to past data. By using it, we can determine promptly when to perform software rejuvenation. In other words, we can decide the best time to perform the rejuvenation, in such a way that fewer requests are lost [7].

IV. Methodology

This section presents the proposed methodology for software maintenance applied to software aging issues, dealing with its complexity and also addressing the impact of software aging-related failures and rejuvenation policies in system availability. The proposed approach adopts a hybrid maintenance policy, which integrates the corrective and preventive maintenance types, aiming at improving availability metrics. Figure 2 shows the steps of the proposed methodology.

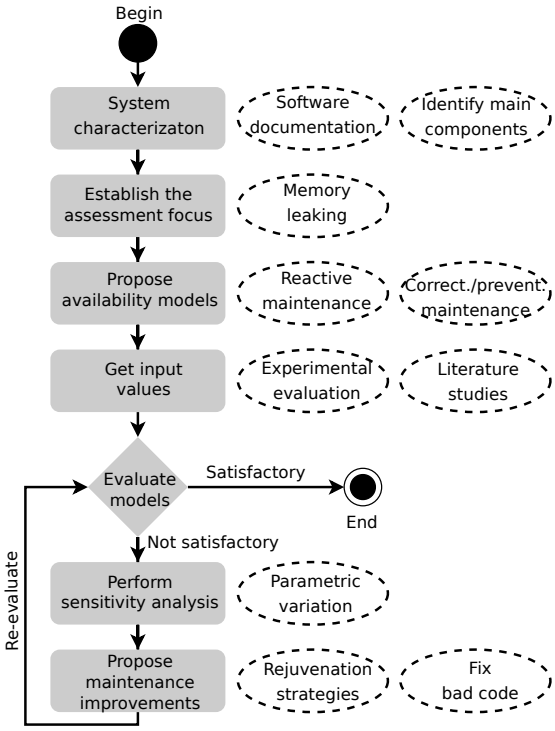

Fig. 2: Methodology

In order to implement an appropriate maintenance policy, it is necessary a **system characterization** to understand how the software works from its documentation and identifying its main components, aiming at delimiting the maintenance focus. Understanding the system behavior requires great attention and special care, in order to avoid rework, interpretation errors and compromise other methodology stages. This step enables to learn about the system details, which may influence the techniques that will be adopted in the next steps.

Next step is to **establish the system assessment focus**, as this will affect the models that will be created later. The maintenance team must define the main interest metrics, focusing mainly on those that have the greatest influence on the quality of service. Considering the focus of this paper, the maintenance team should check computing resources with possible signs of software aging, for example, the memory usage, because memory leaking is one of the most common aging effects.

Now that some gaps about the system functioning have been filled and the assessment focus has been established, we can **propose availability models** to obtain results of the system's behavior considering a scenario with memory leaking. Here we will propose two models: one for reactive maintenance and another one for corrective and preventive maintenance. The main objective is evaluating the system behavior with software aging, comparing the system's availability results with this two maintenance policies.

For the models to represent the system's behavior, some of the **input values** must be obtained through measurement experiments. An experiment is a series of tests that are performed in order to obtain the maximum information with the minimum number of observation samples or time [15]. The experiments are also used to study the behavior of processes and systems [16]. Running such experiments requires a testbed environment and may also be linked to the need to create support tools, if none of the available tools address the combination of analyzed metrics, observed scenario, investigated environment. However, for scenarios in which running an experiment is impracticable, such values must be obtained through studies in the literature.

Evaluate models is the activity that compares the output values with a predefined reference value through SLAs or the expectation of the end user or system administrators. When the computed value is satisfactory, the process is terminated, and it is restarted only when the system undergoes some modification. However, if at least one metric of interest has not reached a satisfactory level, the process is not finalized and proceeds to the next step. For the purpose of this paper, the results of reactive maintenance are computed first (baseline scenario). Then, we follows to the next step.

The **sensitivity analysis** stage is responsible for identifying which components have the greatest impact on the metrics of interest. Different methods can also be used for each model to deal with specific solution constraints or analysis preferences. Different techniques such as availability importance, reliability importance, and parametric variation can be applied to create a ranking of the components importance. However, in this paper we adopted only the parametric variation, where the input parameters of the models are tested with different values. This analysis assists in the actions that will be taken in the next.

Finally, several **improvements** may be **proposed** to improve availability levels. The adoption of a hybrid maintenance policy, with both preventive and corrective actions, may be used to reach high quality of service indicators. A corrective maintenance has the potential to fix bad code and solve already identified aging issues. In addition, rejuvenation actions are performed to mitigate and prevent failures caused by software aging which cannot be fixed in source code at a short time.

V. Availability Models

Many service providers rely on a reactive maintenance routine: whereas an application enters into a failure state then someone calls to the maintenance team asking for help on the site. After the maintenance is finished the team goes back to idleness, either doing other stuff or waiting for a new call to extinguish the fire on service provisioning.

Figure 3 presents an SPN model that represents this general routine adopted by many companies. This model comprises 7 (seven) places (white circles), 4 (four) timed transitions (white rectangles), and 2 immediate transitions (black rectangles). A transition may fire and move a token (black points inside the places) from a place to another. The tokens indicate the current system state.

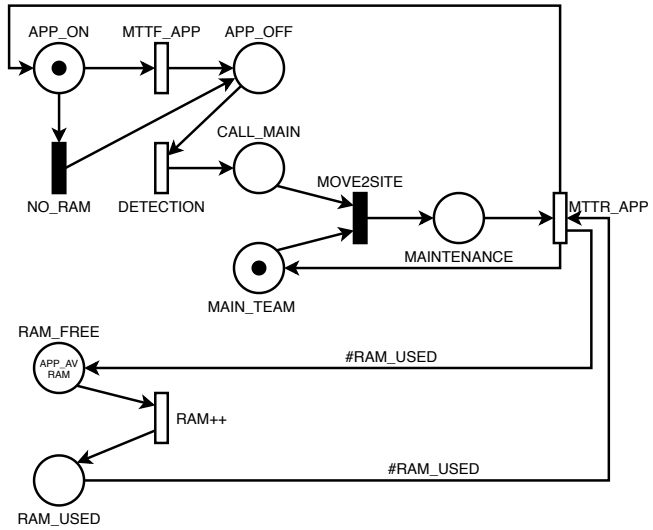

Fig. 3: General Baseline Scenario

By reaching criteria related to both specific times (timed transitions, represented by white rectangles) and required conditions or probabilities (immediate transitions represented by black rectangles) the system state can change. As an example, by firing the MTTF_APP transition, the system moves a token from the APP_ON place to the APP_OFF place, meaning that from the system has failed. A system failure is also represented by firing the NO_RAM immediate transition, which has a condition associated to full consumption of RAM, i.e., the exhaustion of tokens in the RAM_FREE place. It is important to mention that the application has a reserved RAM space to run, which increases over time in case of memory leaking.

In any case of failure (caused by software aging or not), the system malfunctioning is detected by the maintenance team after a time interval associated to the DETECTION transition. If the team is available for repairing the system, the maintenance action will last a time represented by the MTTR_APP transition.

As an improvement to the baseline general model, we propose a proactive, self-healing, routine based on the presence of a system monitor that will repair the application to a previous state based on the presence of software aging issues related to memory leaking. This approach considers the degradation of memory usage by the application over time and copies the application context before restarting it. Figure 4 presents the proposed model, which considers both, proactive and reactive maintenance routines.

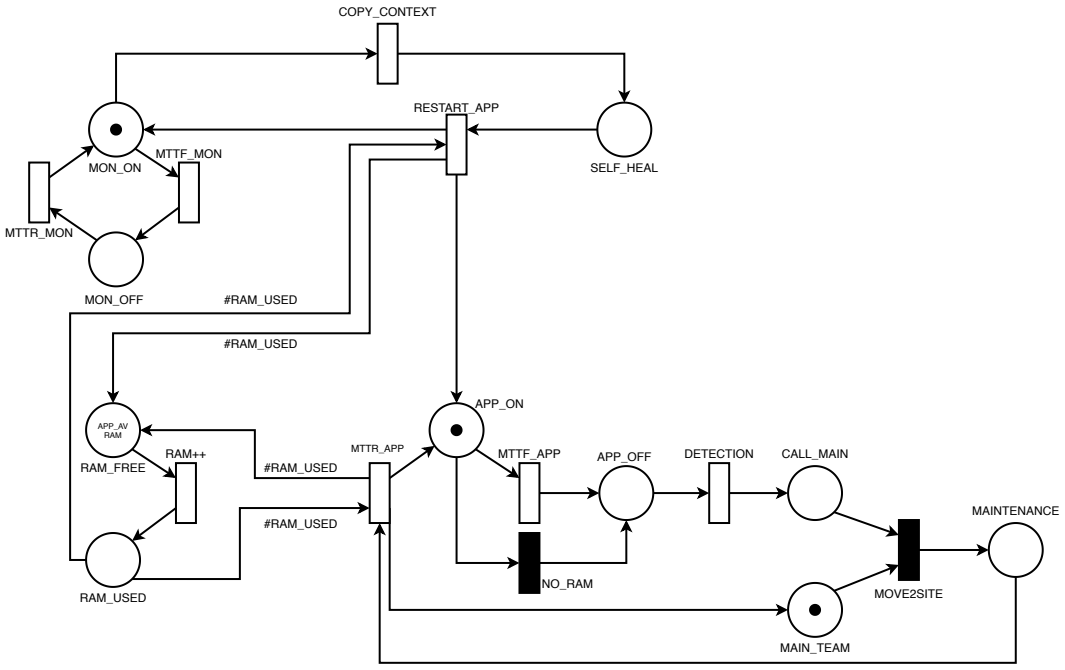

Fig. 4: Proposed Maintenance Model

The presence of a token on place APP_ON indicates that the application is operational, the same applies to place MON_ON, which indicates that the application monitor is functional. The same applies to MAIN_TEAM place, indicating that a maintenance team is available, and it will repair the system if it enters into a failure state. These three places, each one with a token, plus the number of available memory resources (RAM_FREE), indicates the initial system state: the monitor and the application are operational, and the maintenance team is ready to perform any required repair based on a call from system users or other stakeholders.

As for the initial system state, both application and monitor may enter into a failure condition, their corresponding transitions (MTTF_APP and MTTF_MON) indicate that an amount of time equivalent to the product life expectancy has passed, i.e., degradation of hardware, operating system bugs, or an error in any other kind of component may cause either monitor or application to fail.

On the presence of aging issues related to memory leaking, the number of available resources may decrease. The transition RAM++ indicates that the consumption of RAM increased, and the higher the number of consumed resources the higher the probability that the application enters into a failure state. In order to improve the life expectancy and reduce the time of application unavailability we may perform a self-healing maintenance, based on monitoring the number of available resources. The monitor is considered another application and can fail as well, but if it still performs flawlessly, it can copy the application context (COPY_CONTEXT transition) and restart it to a previous state. This self-healing mechanism also points out the possible memory leaking cause. That information can be used to improve the system and correct related bugs on next versions of the application.

Table I shows a representation of the "next place" function for each place in the proposed model. This function indicates which are the places that may receive tokens from other place, by firing a specific transition. Hence, the "source" places are listed in the first column, whereas the transitions are listed in the first row of the table. It is important to mention that we needed to use some acronyms on Table I, due to space issues. Trans. is an acronym to Transition, COPY_CONT. is an acronym to COPY_CONTEXT, MAINTEN. for MAINTENANCE, and MTTF_A and MTTR_A stands for the Application respective MTTF and MTTR, while MTTF_M and MTTR_M stands for the Monitor MTTF and MTTR respectively.

VI. Result Analysis

This section provides the case studies that demonstrate how feasible is the proposed approach. First, we provide the experimental methodology, which shows how we obtained some of the values that will be later used as input parameters for the proposed models.

A. Experimental Results

The first part of the proposed approach consists in debugging the application, finding the problematic code and solving them. For that we adopt Valgrind, a tool that "can automatically detect many memory management issues" [17]. Valgrind shows which parts of the code are causing the memory leaks, so we can fix it and deploy the new application. This is a definitive solution, that solves the bug and after

TABLE I: Next place function for each place on proposed model

Place/Trans.	MTTF_A	DETECTION	MOVE2ST	MTTR_A	RESTART	RAM++	COPY_CONT.	MTTF_M	MTTR_M	NO_RAM
APP_ON	APP_OFF	-	-	-	APP_ON	-	-	-	-	APP_OFF
APP_OFF	-	CALL_MAIN	-	-	-	-	-	-	-	-
CALL_MAIN	-	-	MAINTEN.	-	-	-	-	-	-	-
MAIN_TEAM	-	-	MAINTEN.	-	-	-	-	-	-	-
MAINTEN.	-	-	-	APP_ON MAIN_TEAM	-	-	-	-	-	-
RAM_FREE	-	-	-	-	-	RAM_USED	-	-	-	-
RAM_USED	-	-	-	-	RAM_FREE	-	-	-	-	-
MON_ON	-	-	-	-	MON_ON	-	SELF_HEAL	MON_OFF	-	-
MON_OFF	-	-	-	-	-	-	-	-	MON_ON	-
SELF_HEAL	-	-	-	-	APP_ON	-	-	-	-	-

that the application shows no more memory-related aging indicators.

However, sometimes a corrective maintenance is not possible due to software complexity or expensive costs. Thus, a preventive maintenance can be adopted in order to reduce the downtime and maximize service availability. In this situation, a preventive maintenance can temporarily mitigate that issue and provide a recovery to a state that is less error-prone. This is not a definitive solution, but can provide a good cost-benefit trade-off when compared to other solutions.

An experimental approach for testing this strategy has been developed. The experiments served to acquire data to validate the proposed model. For that purpose, a software with memory leaks has been implemented, with a mechanism to save and restore the context of the application. The implementation was made in the C programming language, and the aging monitor has been created with Bash scripts. The experimental setup comprises a single physical machine, with an Intel Core i7 CPU, 8 GB of DDR4 memory and a 240 GB SSD. The operating system was Debian 10 Buster GNU/Linux, with kernel image 4.19.0-9-amd64 and Valgrind software version 3.14.0. Figure 5 depicts how this experimental approach works.

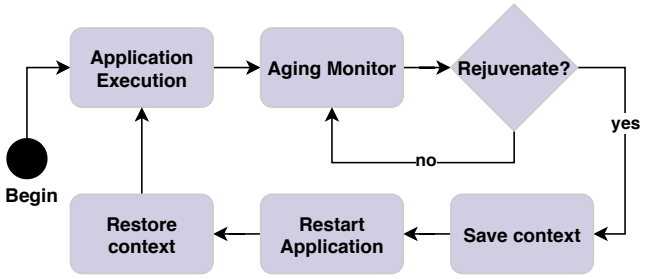

Fig. 5: Experimental approach for testing software rejuvenation

The stage of "application execution" represents the application running in a healthy way. Parallel to the application execution, the "aging monitor" checks the application health. As software aging manifests throughout execution time in the form of memory leak, the monitor periodically verifies if a critical amount of memory has been consumed. As we mentioned previously, the monitor script was implemented in Bash scripts using Linux utilities commands, like 'ps' and 'date'. The amount of memory used by the application has been logged each 15 seconds and in the same period the monitor verifies if the threshold is reached. Algorithm 1 shows the code of the aging monitor.

```
initiate_app();
while True do
    if log_memory > 2048 then
        send_sigterm();
        initiate_app(restore=True);
    end
    wait(15);
end
```

Algorithm 1: Aging Monitor

If the resource has leaked beyond the threshold, the monitor sends the signal SIGTERM to the application. This signal is captured by the application, which executes a routine to save the context (i.e., the values of its variables) to a text file and terminates the application. After that, the application is initialized again with a parameter 'restore', which enables it to load the previous state with the data that has been saved to a file before application had been closed. At this point, a new instance of the application is running in a healthy state again. At first, we wanted know if the Valgrind solution is effective to debug software aging issues. Before debug the application sometimes, we can say that tool can effectively point out the region of code that contains the issue, being therefore useful for implementing a corrective solution further. Figure 6 shows the results of memory utilization in a solution that uses a rejuvenation technique.

The memory utilization has grown up as can be expected in a software with aging issues. In the first six hours, the memory consumption reaches the threshold. After that, the rejuvenation is triggered by the aging monitor and the memory utilization returns to the level of the experiment beginning. This behavior is observed in the next 174 hours, when the memory is periodically leaked and rejuvenated.

B. Models Evaluation

In order to evaluate the proposed models, we need to obtain the system's parameters and metrics values to use them in the models. Some of these values were obtained from a literature review [18]–[20], while other ones were extracted from the experimentation presented on Subsection VI-A, which is the

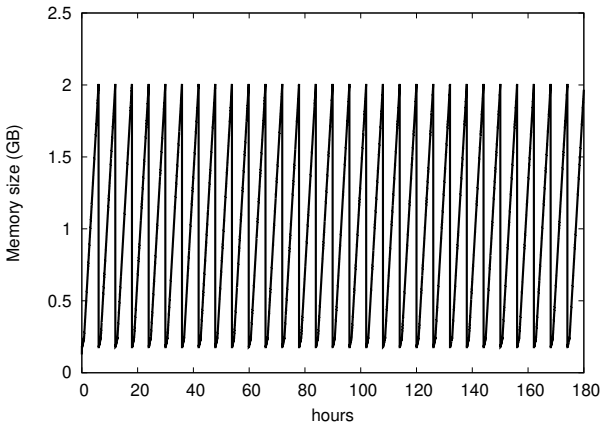

Fig. 6: Memory Utilization of application with rejuvenation mechanism

case for the time to restart the application, as well as for the amount of time required to copy the application context and the time interval between increases of RAM consumption. The values used to feed the models presented in this paper can be seen in Table II.

TABLE II: Input Parameters for Proposed Models

Transition	Time (h)
MTTF_APP	788.4
MTTR_APP	0.5
MTTF_MON	788.4
MTTR_MON	1
DETECTION	0.5
RESTART_APP	4.27×10^{-4}
COPY_CONTEXT	2.12×10^{-6}
RAM++	0.5×10^{-5}

The first evaluated model was the general baseline one defined in Figure 3, and represents a general application under a reactive maintenance routine and an application containing aging issues. By using the previously presented input values, we reached an availability of 84.56%, which corresponds to an annual downtime of 1342 hours – almost two months per year where a user is unable to access the system. After evaluating the availability of this model, we performed a sensitivity analysis to identify the components that impact the most on this metric. The times for each transition were varied from -50% to +50% of their initial value, and Table III presents the rank and sensitivity index for each transition.

The component that impacts the most on the application availability is the RAM increment, or the RAM++ transition, which is directed affected by the presence of memory leaking. Figure 7 presents the general results graphically. The RAM++ is followed by the Detection time, which is directed related to the efficiency of tools and methods available for the maintenance team to identify that a failure occurred. MTTR_APP and MTTF_APP are the other two transitions with highest impact

TABLE III: Percentage Difference Sensitivity Analysis Ranking

Transition	Rank	Sensitivity Index
RAM++	1st	0.17057
DETECTION	2nd	0.06659
MTTR_APP	3rd	0.06659
MTTF_APP	4th	0.00078

on the application availability. The faster the application fails, the worse the availability, as well as the quicker is the repair, the better is the availability.

The evaluation of the model for the proposed maintenance methodology (see Figure 4), considering two maintenance routines, presented as result an availability of **99.82**%, meaning an annual downtime of 15.9 hours, which is almost **85 times lower than the baseline** evaluated scenario containing only reactive maintenance. Depending on the application, this value may still be considered a bad thing, and this is why the methodology that points out where the leakage is happening is necessary for general system improvement. By removing the memory leaking, the application may reach three or more nines of availability, thus reducing substantially the system downtime to less than an hour per year. It is also worth to remember that our analysis is focused on a single application instance, without hardware or other infrastructure details. In a larger setup, the proposed models should be adjusted to represent possible interactions with other components and instances, as well as fault tolerance mechanisms such as redundancy.

VII. Final Remarks

This paper proposed a methodology for software maintenance that is tailored for dealing with software aging issues, considering also the rejuvenation policies to achieve better system availability. In order to overcome the challenges that are specific to the software aging phenomenon, we adopt a hybrid maintenance policy, which carries out corrective and preventive maintenance based on the assessment of some indicators of application and system-wide health status.

The case study results, obtained with SPN models and testbed experiments in a controlled environment, showed significant improvement when both maintenance routines are applied to mitigate and correct the software aging effects. The availability of this hybrid approach might reduce the annual downtime of the service to approximately 0.1% of the time of unavailability measured in a scenario with only reactive maintenance (15.9 hours compared to 1342 hours).

Future works can extend the proposed methodology to consider workload characteristics that may change the behavior of software aging, speeding it up or slowing it down, and therefore requiring real-time adjustments on the planned maintenance actions. Other important research direction is to integrate the automated detection of software aging-related flaws during testing stage of software development, which might avoid some work in the production stage.

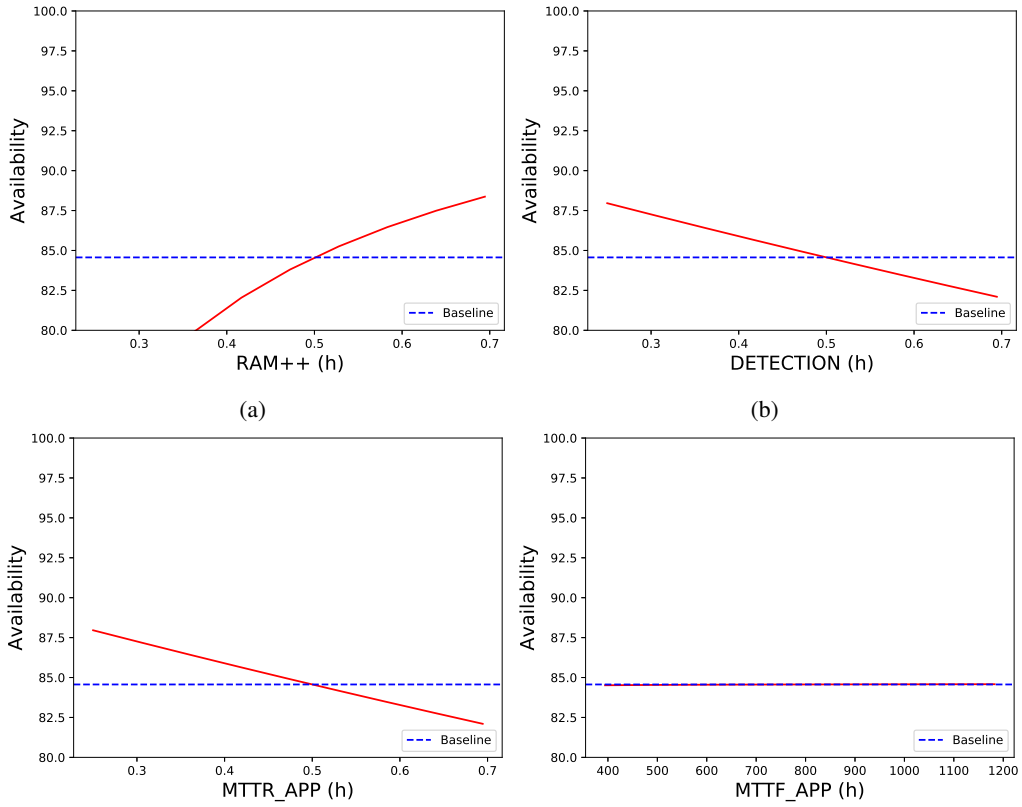

Fig. 7: Sensitivity Analysis

REFERENCES

[1] S. Andreo and J. Bosch, "API management challenges in ecosystems," in *Software Business - 10th International Conference, ICSOB 2019, Jyväskylä, Finland, November 18-20, 2019, Proceedings*, ser. Lecture Notes in Business Information Processing, S. Hyrynsalmi, M. Suoranta, A. Nguyen-Duc, P. Tyrväinen, and P. Abrahamsson, Eds., vol. 370. Springer, 2019, pp. 86–93.

[2] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*. IEEE, 1995, pp. 381–390.

[3] M. Grottke, R. Matias, and K. S. Trivedi, "The fundamentals of software aging," in *Software Reliability Engineering Workshops, 2008. ISSRE Wksp 2008. IEEE International Conference on*. Ieee, 2008, pp. 1–6.

[4] J. Li, Y. Qi, and L. Cai, "A hybrid approach for predicting aging-related failures of software systems," in *Service-Oriented System Engineering (SOSE), 2018 IEEE Symposium on*. IEEE, 2018, pp. 96–105.

[5] W. Dang and J. Zeng, "Optimization of software rejuvenation policy based on state-control-limit," *International Journal of Performability Engineering*, vol. 14, no. 2, p. 210, 2018.

[6] M. Torquato, J. Araujo, I. Umesh, and P. Maciel, "Sware: A methodology for software aging and rejuvenation experiments," *J. of Information Systems Engineering & Management*, vol. 3, no. 2, p. 15, 2018.

[7] P. Pereira, J. Araujo, R. Matos, N. Preguiça, and P. Maciel, "Software rejuvenation in computer systems: An automatic forecasting approach based on time series," in *2018 IEEE 37th International Performance Computing and Communications Conference*. IEEE, 2018, pp. 1–8.

[8] D. Cotroneo, A. K. Iannillo, R. Natella, and R. Pietrantuono, "A comprehensive study on software aging across android versions and vendors," *Empirical Software Engineering*, 2020.

[9] D. Cotroneo, L. De Simone, R. Natella, R. Pietrantuono, and S. Russo, "A configurable software aging detection and rejuvenation agent for android," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2019, pp. 239–245.

[10] P. Maciel, K. S. Trivedi, R. Matias, and D. S. Kim, "Dependability modeling," in *Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions*. IGI Global, 2012, pp. 53–97.

[11] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Software aging and rejuvenation: Where we are and where we are going," in *Software Aging and Rejuvenation, 2011 IEEE Third International Workshop on*. IEEE, 2011, pp. 1–6.

[12] L. Bernstein and C. Kintala, "Software rejuvenation and self-healing," *Software Tech News*, vol. 7, no. 2, p. 15, 2004.

[13] A. Avritzer and E. J. Weyuker, "Monitoring smoothly degrading systems for increased dependability," *Empirical Software Engineering*, vol. 2, no. 1, pp. 59–77, 1997.

[14] 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 (JETC)*, vol. 10, no. 1, p. 8, 2014.

[15] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. New York: Wiley Computer Pub., John Wiley & Sons, Inc., May 1991.

[16] D. C. Montgomery, *Design and Analysis of Experiments*. John Wiley & Sons, 2006.

[17] "Valgrind," 2020, https://www.valgrind.org.

[18] J. Dantas, R. Matos, C. Melo, and P. Maciel, "Cloud infrastructure planning: Models considering an optimization method, cost and performance requirements," *Int. J. of Grid and Utility Computing*, 2020.

[19] C. Melo, J. Dantas, D. Oliveira, I. Fé, R. Matos, R. Dantas, R. Maciel, and P. Maciel, "Dependability evaluation of a blockchain-as-a-service environment," in *2018 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2018, pp. 909–914.

[20] S. Sebastio, R. Ghosh, and T. Mukherjee, "An availability analysis approach for deployment configurations of containers," *IEEE Transactions on Services Computing*, 2018.