When legacy code turns into senescent code: Assessing software aging and its implications

Philip König (SBA Research), Kevin Mallinger (SBA Research), Alexander Schatten (SBA Research)

Abstract: In the last decades it became clear that, more often than not, software engineers were focused on delivering new features and in the process generated highly complex interacting layers and modules of software. Over long periods of times, very old legacy code interacts in complex ways with new code. Various studies imply that software aging is a phenomenon whereupon continuous execution of programs leads to a gradual build-up of errors and overall degradation of performance. Strange and unclear behaviour emerges from interactions of new and old modules, which in the worst case manifest itself in crashes, errors and other unwanted responses. This has especially drastic consequences for critical infrastructure networks like power grids or medical software, where conventional practices for error detection like controlled shutdowns and reboots are seldomly an option. Invaluable for early detection of such issues are non-invasive methods which would serve as detectors to assess when, for example, a software module begins to show first symptoms of developing aging related problems. Bio-inspired approaches are now a major player in software engineering, as processes like Darwinian evolution, mutation and learning are employed in more and more software systems. While first technology assessment methodologies concerning software aging have been developed none drew inspiration from the natural sciences, where, especially in the last few decades, biogerontology – the science of processes of aging and its consequences – has begun to pick up serious steam. By abstracting similar processes in biology and computer science the fundamental problems and their solutions can be analyzed and then transferred from one discipline to another. Recent discoveries showed that chronological age is not always an accurate variable to define a cell's or an organism's biological age, as different organisms and even different cell types within those organisms age at different rates. Similarly, it is important to not use chronological age as the only parameter to determine if and when code ages. Very old code that continuously gets maintained and cautiously ported to new platforms might exhibit less

signs of software aging than chronologically younger modules which were dragged along by techniques like wrapping. Thus, this paper aims to make a case that software aging could be the next success story in bio-inspired software engineering. To solidify this notion a first prototypical practical application will be presented that draws from analogies that connect biology and computer science.

1.Introduction

As software finds its way into more and more domains of human society, so do the problems and challenges that can be associated with it. Thus, it has become essential to monitor and assess the *health* status of software. This is especially important if older code is carried over from version to version, possibly becoming more essential to the whole construct with every new feature and module built upon it (Parnas 1994). Although Software Aging is a relatively obscure field, it is growing steadily in the last decades (Cotroneo 2014). On closer examination one notices that pretty much all this research is dedicated to the treatment and of symptoms, generally summed up under the term *software rejuvenation*. This field mostly concerns itself with techniques on how to mitigate damages and troubles caused by the various sources of aging, for the most part by finding out the optimal point in time to reboot the system. So far there were multiple factors identified that contribute to the phenomenon such as round-off errors, data corruption or memory leaks. What is still waiting to be discovered and formulated is a set of theories, which, when combined, explain the fundamental mechanisms that cause software aging (Grottke 2008). This would require fundamental research that concerns itself not only with the treatment of symptoms but examining the whole range of causes and crucially not restrict oneself to the inner workings of software but also incorporate socio-technological aspects of how such systems age.

When looking at recent software trends, many of them drew inspiration from biological systems like Genetic Algorithms (Kramer 2017), Ant Colony Optimization (Dorigo 2006) or Neural Networks (Abdi 1999). Although for that to be a valid method, the solutions to problems one seeks ideally are sufficiently similar to the problems and thus solutions nature has found. This paper argues that software aging lends itself to this

approach and is an excellent candidate to borrow future solutions from nature, by showing the parallels between certain aging mechanisms of both biological and man-made systems.

2.1 Software Aging

The first notion that might cross someone's mind while reading about software aging is ‘how is this even possible?’ as software per se is a mathematical construct without a physical body and thus unable to age. While this of course is true, software does indeed age, albeit in different ways than an embodied entity would. This does not mean that the mechanisms behind software aging are not comparable to those of physical aging. As there has been much research conducted in this field (Cotroneo 2014), there exist numerous definitions, which could be summarized as such: ‘With increasing running time, the software displays increasing performance decrease and elevated failure rates and even crashes.’ To make things worse such problems are hardly ever detected immediately and gradually build up in the background without being noticed until a tipping point is reached and manifested as incorrect service (or no service) or partial failures like increased response time (Grottke 2008).

So the longer software runs, the more aging-related bugs it might pick up which accumulate and over time cause the system to move from its intended state to a failure-prone one. A dramatic example of what can happen when this occurs in critical situations is the 1991 case of a patriot missile killing 28 US soldiers because the targeting system has been running for over 100 hours straight without a reboot. A bug that introduced a timing lag caused the missile to miss its intended target and thus kill the men. Tragically this could have been avoided if the system was restarted approximately every 8 hours (Marshall 1992). There are also less dramatic examples from everyday life, like operating systems needing to be restarted after long sessions to avoid complications because of various factors accumulating in the background.

Problems like these arise at the runtime environment, concerning the internal mechanisms and microenvironment of software, but a complicated and complex process

such as aging also occurs in a multitude of different ways. The authors propose that there are at least two meanings of *running*:

1) Runtime environment: Software is executed and running, thus runtime effects take place (e.g., memory leaks)
2) How people use software in general

The first interpretation would correspond to the missile example, but the second one would lead to other forms of deterioration that arise from the socio-technological setting in which the software is embedded and itself is characterized as two distinct types. First is the oversight of the program's authors and/or owners to modify it according to different, new needs. The second is in a sense the reverse, wherein changes are made without the needed care and insight (Parnas 1994). Such headaches are further complicated therein that they can not only happen to the software of interest at hand, but also external supportive structures like libraries. These are a collection of scripts, functions and routines that generally seek to ease and smoothen out the process of programming. If such aging-related bugs originate from one or more third party libraries or even the application execution environment such as the operating system, it becomes very hard to locate and fix bugs as developers might lack the source code or expertise on the internal intricacies of the code they are maintaining. Issues like these demand some kind of indicator that flags a functional unit in a program as a potential source for trouble, but must not rely on naive parameters like simple chronological age of a unit. Well maintained and cared for code can remain operational for a long time and even when ported to new platforms might exhibit less signs of software aging than chronologically younger modules which were written and added with less thought for long term sustainability.

2.2 Biological Aging

First and foremost it has to be said that we are just starting to understand biological aging as it is a highly complicated and complex set of processes which consist of heavily intertwined functional regulatory networks, dynamically reacting to internal and external stimuli. This is further obstructed by the fact that each species, individuals within said species and separate modules, like tissues and cells, of those individuals age

at different rates and speeds. Adding to this is the inherent difficulty of researching aging as getting first results could take years to decades (Goldsmith 2020). Generally speaking, aging has been described as a gradual decline of function over time, driven by local molecular and systemic alterations (Newgard and Sharpless 2013). For example, some species like the 'immortal jellyfish' *Turritopsis dohrnii* show negligible deteriorative changes in their bodies, also known as *negligible senescence*, implying that there seem to be repair mechanisms in action that actively work against forces that little by little degrade form and function on all levels of organismal organization. Under this point of view a more precise definition of biological aging has been conceived, namely it being 'the random, systemic accumulation of dysfunctional molecules that exceeds repair or replacement capacity' (Hayflick 2016).

This simply means that the structural integrity of organisms is continuously compromised, but the balance of damage and responsible systems working against that slowly shifts over time in the direction of accumulating damage. It is hypothesized that a major factor of this is the fact that the maintenance mechanisms themselves deteriorate and thus the whole system enters an ever-accelerating downwards spiral. This combination of accumulation of increasing damage plus dwindling servicing results in error-prone functional units. Many higher organisms developed strategies of flagging such withered modules, one of them being the so-called Hayflick limit, named after its discoverer. It serves to describe an aspect of assessing a cell's actual age in contrast to chronological age. It explains why normal human body cells (this does not apply to most stem and tumor cells) only divide a certain number of times before stopping to do so. After this the cell will enter the so-called senescent state, a most interesting condition that will be discussed in more detail. One reason for this limit is that at the end of the chromosomes, which are highly compact storage units of DNA, protective structures called telomeres reside, which shorten with each cell division. When a critical length of those protective structures is reached, the cell starts a stress response cascade, ultimately reprogramming itself and irreversibly becoming senescent (Hayflick 1961). Important here is that this should not be seen as a sort of chronometer, as not time, but replicative events are being measured, or more precisely, how often the respective chromosomes and thus strands of DNA are replicated. Hayflick (2016) himself called this not a clock

but a ‘replicometer’. This gives the cell the ability to effectively count the amount of times it has already divided, thus representing a measure for the effective age of a functional unit, in this case a cell.

3.1 Senescence and the SASP

As mentioned before, senescence is a stress induced cellular state that leads to particular behaviour and metabolic activity, including growth arrest. These stresses include telomere depletion, oxidative stress, general DNA damage and the onset of becoming a tumor cell. This is a gradual process that over time causes the cell to lose its designated function which happens at all levels of organismal organization, from single building blocks of the cell to whole cells and tissues and ultimately the whole organism. One must not mistake this for random order of events though. All bodily cells that undergo division have the option of inducing this particular state themselves and for the most part this process is executed orderly and uniformly, respective to the type of cell.

Generally this program is activated when a cell notices that it has been damaged beyond a level of repair. It is a well established hypothesis that senescence developed as a tumor suppressor mechanism. If a cell notices that it has been damaged and might become a tumor cell, stopping its own growth and thus spreading is a logical step. Thus in the short term, senescence is a net positive mechanism, sacrificing single cells to ensure the protection of the tissue and therefore whole organism. But if damage accumulates over time and thus more and more cells need to be sent into senescence, the function of the concerned tissue can be compromised if cells are not properly executing their designated task, which has been demonstrated to be a major driving factor behind many age-related diseases (cf. Regulski 2017). In contrast, there are some cell types that are able to continuously fend off this fate and lengthen their own telomeres like certain human stem cells or most successful tumor cells (Blackburn 1991). Those cancer cells that were able to avoid being sent into senescence, including growth arrest, and developed mechanisms to remain undetected by the immune system are then able to attain immortality, which of course is suboptimal for the host organism, but also led to immortal cell lines that can be grown in labs and used for research indefinitely.

It was established that senescent cells change their metabolism permanently, but this transformation is not restricted to purely internal affairs. To the contrary most senescent cells develop a specific secretion pattern that is called the senescence-associated secretory phenotype (SASP), which consists of a myriad of different factors (Coppé 2010). Most prominent are its pro-inflammatory characteristics, meaning that cells expressing the SASP induce permanent sterile inflammation (inflammation without present pathogens) in their immediate vicinity and thus recruit an immune response. This might indicate that tumor suppression is not the only purpose of senescence, but essentially flagging the affected area as in need of further inspection. Paradoxically and crucially, there is a dark side to this mechanism, as evidence is mounting up that this continuous inflammation in their tissue microenvironment leads to a plethora of pathologies including cancer (Courtois-Cox 2006). If neighbouring cells are exposed to these conditions and additional secretion of growth factors through the SASP, they themselves can, over time, become damaged and thus enter senescence, which leads to a positive feedback loop that gradually turns whole sectors of tissue into a zombie-like state and in the worst case even promotes tumorigenesis. Interestingly, the same mechanism can be beneficial and devastating, tumor suppressive and tumor forming. Although this might seem improbable or even impossible, processes with such opposed attributes are actually not unheard of and perfectly in line with evolutionary theory, summarized under the term antagonistic pleiotropy (Campisi 2007). To illustrate how such a trait might be propagated through generations one must only imagine an attribute that gives an individual a fitness advantage in its reproductive period, but a disadvantage in its later stages of life. By the time the negative influences come into effect, the individual has already had a high chance of generating offspring and thus passing his genes on to the next generation.

It seems that part of the arsenal of bodily repair mechanisms is the ability to remove senescent cells, in particular natural killer cells might play a role in this (Wijshake T., & Deursen 2016), but as every cog of the machine continuously rusts and ages, protective mechanisms also degrade in its function and more and more senescent cells survive for too long and induce senescence in their neighbouring cells. This could also be the reason why only a small absolute number (<20%) of those might be able to

contribute to systemic effects more gravely than one might expect (Campisi 2007). It is safe to say that wherever there are clusters of these cells, they serve as an indicator for anomalous tissue as they are at the same time cause and consequence of aging related diseases (Regulski 2017).

3.2 Deprecation

Code, maybe counterintuitively, is in itself treated as a liability and not an asset by many professional developers. What this means is that one wants to build their systems as lean and unconvoluted as possible. This is one aspect behind wanting to remove obsolete systems, every line of code brings with it not only upfront costs but also lifelong maintenance expenses. As the environment around code changes and evolves, it becomes more and more cumbersome to integrate it with new functionalities whereby it also becomes more expensive as programmers who are adept with languages of previous generations are getting more sparce. This and more are reasons as to why it is often more efficient, money and time wise, to remove obsolete parts than to let them chug along in the background (Winters et al. 2020).

Code that is deliberately deemed no longer important or adequate must be marked to be replaced in the future. When software classes evolve, their API (application programming interface) and therefore integrated methods and functions must keep up. This can happen in many different ways, methods can be renamed, replaced and fields can change (Oracle 2020). However, one can not just make such changes overnight and without heads-up. In many industries antiquated code is still heavily used to the extent that modern programs often need to be backwards compatible with many different and older environments like outdated operating systems. This means that abruptly removing superseded but critical supporting software potentially renders users with suddenly non-functional systems. This is especially crucial if there are programs that must run without interruption, e.g., applications observing critical infrastructure like power grids. The method of flagging outdated components as *deprecated* is a compromise to keep the old parts intact, but at the same time incentivize programmers to use the revised version. In practice this means that when programmers now use flagged elements, they get a warning that those will not be further supported or be removed in the future and thus

should not be used. This introduces a time buffer visible for everyone that uses these parts, recommending that users switch over to the updated versions. Because this is overall a rather involved process, there must be good reasons to deprecate something, which include the API being insecure, buggy, or encouraging bad programming practices (Oracle 2020). It needs to be emphasized that this flagging is a manual process, done by the respective programmer. This introduces a source for potential errors, such as simply forgetting to do it or even flagging the wrong sections.

To avoid this rather messy process of needing to manually go through sections and determine what can stay and what needs to go long after the system was built, one could look at how other disciplines cope with similar problems. Here it is important to mention that old does not equal obsolete and chronological age alone is not an indicator for obsoleteness. If a system is from the beginning on built with care and maintenance in mind, it might actually become better over time. But here lies the crux, the overwhelming majority of software systems are not built under this premise. Even the best development teams report that they have to work with tight deadlines and thus the idea of designing the product with easy removability in mind is often the first one to go (Winters et al. 2020). An illustrative example comes from Winters (2020), where he describes this on the basis of a nuclear power plant which shares attributes with software like high complexity and potential high criticality. It is a given that a nuclear plant has a natural decommission date, which of course influences many design choices, as it needs to be established beforehand exactly how, after its life span ends, each and every part is to be disassembled and disposed of in a safe manner. Such procedures stand in contrast to the practices of software engineering where this mentality has yet to be adopted widely. Winters (2020) also mentions that even in industry-spearheading companies like Google, systems are rarely built with such principles in mind mainly because of the reason mentioned before.

Just like senescence which operates under the principles of antagonistic pleiotropy, developing software systems that do not incorporate easy options of maintenance and removability from the beginning on gradually lead to *technical debt* (Alves et al. 2016). This term describes deficits in internal structures of a system: Long-term health of the whole construct is sacrificed for short term benefits, like lower development costs and

shorter development time, but eventually the amassed debt catches up. In most cases retroactively trying to fix problems that originated that way will cost way more and take much more time to correct compared to investing that time and money upfront. But even if rushed development means that some future proofing aspects of a project will be neglected, earlier introduction to the market will result in a short term ‘fitness gain’ simply by being first.

4. Senescent Software

Considerable parallels can be drawn concerning certain aspects of aging in both naturally evolved and man-made systems. For example, the fact that biological maintenance mechanisms deteriorate in themselves can also be seen in the socio-technical aspect of software aging or what was also called software use in general. As programming languages and thus systems written in those become older, the people that are proficient in those and/or wrote the programs also age. In many cases those languages do not get updated indefinitely but superseded by new ones. The consequences of such negative deterioration spirals can be observed in many software companies, the main product of which is a software application of some sort. As this service is at the core of the business, it also must run constantly and thus as long as everything operates smoothly, it is not undergoing major updates that would disrupt the service for prolonged time. Some years later, the language in which it was written slowly becomes outdated and newer additional functions and addons are written in newer languages. This might be inconvenient but not a big problem at first, as long as the people that originally worked on it are still present and know the ins-and-outs of the system. But when these programmers eventually retire or leave the company for any other reason and the newer employees are not as familiar with the whole, increasingly complex system, this becomes a problem. The original team that was responsible for maintenance is now fractured or even replaced entirely and the new one simply is not as versed in the outdated programming language and thus the quality of maintenance degrades steadily as less and less people know how to deal with it and a downwards spiral is initiated. In such or similar cases, old code begins to express SASP like

behavior: one can barely modify and maintain the old parts any longer, so the problem spreads all over the system: functionality that was modular and located before is now spreading over more and more domains of the system in ever-increasing workarounds that no one really has under control any longer.

5. Practical Application

In many cases code that *needs* to go is also code that *can not* go due to different reasons like some users still relying on it or hidden critical dependencies linking them to newer modules, as even for experienced programmers it is non-trivial to discover such connections in systems with high complexity. These are symptoms of negligence in development that were allowed to fester to the point that suddenly a tipping point is reached, and the problem becomes acute and very hard to solve. If one takes a look on how in biological systems malicious and/or obsolete parts are identified, the crucial difference to man-made ones lies in *who* identifies the problematic parts. Instead of a third party making subjective decisions, it is indeed programmed into each and every cell to continuously run service programs that lets it check its own health status and flag itself if there is something wrong.

So, the first step in being able to remove such parts is identifying them. For this purpose, a first proof of concept prototype was devised, the goal of which is to analyse the history of a given software project and provide data concerning accessing and changes in the code made by programmers over time. The primary application was written in Java and uses the Git (a widely used software repository) history as its main data source. Currently it can be configured to run on three different levels of depth: Functions, Classes, and Files, which abstractly will be called scopes. It should be noted that depending on the adherence to programming standards, results from Classes and Files should be identic in most cases. In the first step a path to a directory is used as input to determine the project to analyse. From this path on all subfolders and relevant files are indexed recursively. A file is considered relevant if its extension is in a list of allowed and parsable extensions. From this a tree is constructed and changes are extracted by a command line call to Git at the level of the given scope. The text output of the command is then iterated over line by line with a state machine keeping track of which information

is expected next and extracting the author, date, commit hash, and how many lines were added, removed, or modified and the total number of lines in the given scope. These numbers are used to calculate an overall severity score of the change. Those values are then exported as a .csv file, containing the following information:

- File path (including name)
- Date and time of change
- Author of change
- Severity of change (measured in % of changed code)

This data is then fed into the programming environment Rstudio, where it can then be computed into statistical values and visualized as graphs. From these the relationship between different scopes, authors, and dates can be derived. Further it can also be observed which scopes change how often and how intensely allowing among other things the location of hotspots. Those enable to automatically detect dependencies between scopes by analysing which files are regularly modified together and thus giving information on potential hidden dependencies. Corresponding to how cells constantly check their health status, such a tool could be used throughout a development process to continuously check if newly written lines of code interfere with already existing ones. In addition, this can also be applied to already existing systems to identify senescent parts of code, as scopes that often get changed in tandem might have obscure links that are not obvious to programmers. This would serve to reduce systemic effects that could go unnoticed in the short run but lead to catastrophic consequences when discovered several years down the road.

Conclusion:

When one accepts that similar problems can help finding similar solutions, investigating approaches stemming from Darwinian evolution can serve as useful analogies for designing complex computer environments that also constantly change.

This paper showed that there are intriguing parallels in the ways that both biological and technological systems age, which might very well implicate that naturally evolved solutions are a valuable source of inspiration, especially when considering how often that approach already led to technological breakthroughs. This method is especially tempting for exploring the phenomenon of software aging, as research on the fundamental causes is still lacking and there are clearly analogous processes in nature that serve as a cornucopia of potential answers. To check if the results from fundamental research are valid and applicable to real world problems, they were and will be continuously tested in parallel by devising fitting tools and procedures to strengthen the overall endeavour.

References:

Abdi, H., Valentin, D., & Edelman, B. (1999). Neural networks (No. 124). Sage.

Alves, N. S., Mendes, T. S., de Mendonça, M. G., Spínola, R. O., Shull, F., & Seaman, C. (2016). Identification and management of technical debt: A systematic mapping study. Information and Software Technology, 70, 100-121.

Blackburn E. H. (1991). Structure and function of telomeres. Nature, 350(6319), 569–573. https://doi.org/10.1038/350569a0

Campisi, J., & d'Adda di Fagagna, F. (2007). Cellular senescence: when bad things happen to good cells. Nature reviews. Molecular cell biology, 8(9), 729–740. https://doi.org/10.1038/nrm2233

Coppé, J. P., Desprez, P. Y., Krtolica, A., & Campisi, J. (2010). The senescence-associated secretory phenotype: the dark side of tumor suppression. Annual review of pathology, 5, 99–118. https://doi.org/10.1146/annurev-pathol-121808-102144

Cotroneo, Domenico & Natella, Roberto & Pietrantuono, Roberto & Russo, Stefano. (2014). A Survey of Software Aging and Rejuvenation Studies. ACM Journal on Emerging Technologies in Computing Systems. 10. 10.1145/2539117.

Courtois-Cox, S., Genther Williams, S. M., Reczek, E. E., Johnson, B. W., McGillicuddy, L. T., Johannessen, C. M., Hollstein, P. E., MacCollin, M., & Cichowski, K. (2006). A negative feedback signaling network underlies oncogene-induced senescence. Cancer cell, 10(6), 459–472. https://doi.org/10.1016

Dorigo, M., Birattari, M., & Stutzle, T. (2006). Ant colony optimization. IEEE computational intelligence magazine, 1(4), 28-39.

Grottke, Michael & Matias Jr, Rivalino & Trivedi, Kishor. (2008). The fundamentals of software aging. Software Reliability Engineering Workshops, 2008. ISSRE Wksp 2008. IEEE International Conference on. 1 - 6. 10.1109/ISSREW.2008.5355512.

Goldsmith, Theodore. (2020). Introduction to Biological Aging Theory 2nd Ed -Rev 2.

Hayflick L. (2016) Unlike the Stochastic Events That Determine Ageing, Sex Determines Longevity. In: Rattan S., Hayflick L. (eds) Cellular Ageing and Replicative Senescence. Healthy Ageing and Longevity. Springer, Cham. https://doi.org/10.1007/978-3-319-26239-0_17

Hayflick L, Moorhead PS (1961). "The serial cultivation of human diploid cell strains". Exp Cell Res. 25 (3): 585–621

Marshall, E. (1992). Fatal error: how Patriot overlooked a Scud. Science, 255(5050), 1347. https://link.gale.com/apps/doc/A12050574/AONE?u=anon~9f2bb57b&sid=googleScholar&xid=1e6be5b8

Newgard CB, Sharpless NE (2013) Coming of age: molecular drivers of aging and therapeutic opportunities. J Clin Invest 123:946–950. doi:10.1172/JCI68833

Kramer, O. (2017). Genetic algorithms. In Genetic algorithm essentials (pp. 11-19). Springer, Cham.

How and When To Deprecate APIs. (2020). Oracle and/or its affiliates. https://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/deprecation/deprecation.html

Parnas, David. (1994). Software aging. 279-287. 10.1109/ICSE.1994.296790.

Regulski M. J. (2017). Cellular Senescence: What, Why, and How. Wounds : a compendium of clinical research and practice, 29(6), 168–174.

Wijshake, Tobias & Deursen, Jan. (2016). Targeting Senescent Cells to Improve Human Health. 10.1007/978-3-319-26239-0_16.

Winters, T., Manshreck, T., & Wright, H. (2020). Software engineering at Google: Lessons learned from programming over time.