Self-Healing on the Cloud: State-of-the-art and Future Challenges

Nuno Cardoso and Rui Abreu

Department of Informatics Engineering
Faculty of Engineering, University of Porto
Porto, Portugal
nunopcardoso@gmail.com, rui@computer.org

***Abstract*—Despite the enhancement on software hardness induced by development-time automatic testing and debugging tools, it remains practically impossible to create fault-free applications. The research on self-healing systems emerged to enable applications to cope with unexpected events at run-time in order to maximize their availability, survivability, maintainability, and reliability, while minimizing human intervention. With the increase in software's complexity, in part triggered by the advent of the Internet and Cloud Computing, self-healing properties in applications are becoming a necessity. The main focus of this position paper is on presenting the issues that render unusable or ineffective the usage of the development-time Spectrum-based Fault Localization (SFL) algorithm in run-time environments. We concluded that, despite the issues found, it should be possible to devise an SFL algorithm for run-time environments that can also achieve the good results yielded by SFL at development-time.**

***Keywords*-Fault Localization, Cloud Computing, Spectrum-based Fault Localization, Self-healing**

I. Introduction

Society is increasingly dependent on technology and, with the appearance of computers as a mainstream good, this dependency experienced an exponential growth. One side effect is the increase in software's complexity which, almost unavoidably, leads to a growing amount of bugs as well as a harder process of fault detection, localization, and correction.

Efforts have already been made in order to alleviate the debugging stress from humans by relaying part of the process to automated mechanisms. Tools such as "Tarantula" [16], "EzUnit" [9], and "GZoltar" [20] already provide automated ways of pinpointing the most probable fault locations. In contrast to traditional debugging methods such as the use of tools like "GDB" [21] or mere log analysis, automated fault localization tools improve the development cycle by: (1) reducing the time needed to find the source of the error/failure, (2) enforcing a much more structured debugging process when compared to the ad-hoc, art-like traditional methods and (3) enabling regression testing.

Despite the great advance such tools represent, they largely rely on the provided test suite, both in terms of coverage and quality [1, 2]. It is common sense that it is not always easy to create a comprehensive test suite, either due to time, money, or even complexity constraints. Also, there are failures that are not triggered by a software bug but by a specific condition of the run-time environment, such as a hardware fault. Given that, it is clear that such tools, although they are of great help, are not able to ascertain fault-free applications. As a result the need to develop complimentary tools and techniques that are capable of autonomously "heal" failing applications during run-time becomes prominent. Despite the existence of bugs being an unavoidable issue, most applications cannot afford to have down times, and therefore it is very important to devise techniques to make them aware of their environment and health and be able to responde to changes effectively.

The goal of this position paper is to discuss how self-healing techniques may enhance Cloud applications, focusing our attention on the fault localization topic. We will address the issues that are inhibiting the application of Spectrum-based Fault Localization (SFL) [1, 4, 8], our algorithm of choice, to run-time fault localization.

The remainder of this paper is organized as follows. Section II will give an introduction on SFL applied to development-time debugging. Section III will discuss how self-healing techniques may improve Cloud applications. In Section IV we will cover the issues that render difficult or ineffective the usage of SFL in run-time scenarios. In Section V we will expose our evaluation methodology. Finally, in Section VI we will draw some conclusions on the issues addressed in this paper.

II. Spectrum-based Fault Localization for development-time fault diagnosis

SFL is a lightweight statistic based fault localization technique that takes as its input an execution trace abstraction, called Program Spectrum (PS) (Fig. 2), and produces a list of likely fault candidates, sorted according the probability of being the real failure explanation. The PS consists of matrix ("A"), known as Program Activity Matrix [1] and a vector ("e") in which are recorded all the transactions performed

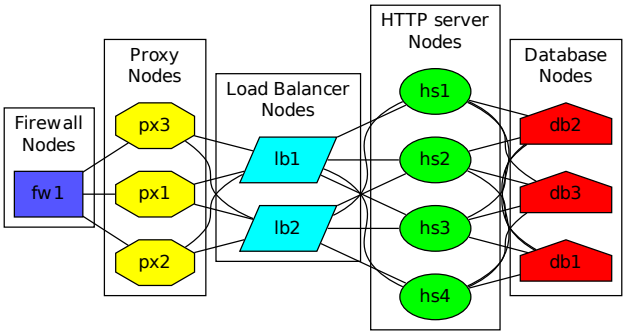

Figure 1. Web server architecture

by the system under analysis. "A" consists of a $T \times C$ binary matrix being "T" the number of recorded transactions and "C" the number of components in the system. The scope of each component is completely arbitrary and depends on the desired granularity, ranging from logical blocks to lines of code. Each element of matrix "A", "A_{ij}", is set to *True* if component j participated in transaction i and to *False* otherwise. "e" consist of a vector with "T" elements, being "e_i" set to *False* if the oracle's result for transaction i is pass and *True* otherwise.

Oversimplifying the problem, the process of defining the probability of some component being faulty consists in finding the similarity between each column of "A" and "e". Abreu et al. in [5], analyse several similarity coefficients and conclude that the coefficient used for SFL deeply impacts the diagnosis performance.

Fig. 1 shows and example architecture of a web server, which could possibly be deployed in a Infrastructure-as-a-Service (IaaS) Cloud. Fig. 2 presents a possible PS gathered from a running instance of the architecture in Fig. 1. For compactness sake, each component group is only represented by a single column and if some of the components in the group was involved in transaction "i", "A_{ij}" is set to the component number. In practice, each group column would be expanded to the number of components, as presented earlier. The most probable failing components, according to SFL would be "fw1" and "hs1". "fw1" is almost certainly faulty, due to Transaction 4 in which it is the only participant in a faulty transaction. However its that either the fault is transient or the oracle is inaccurate, because "fw1" participates in passed transactions. On the other hand, "hs1" has a permanent fault (at least during this observation), because it only appears in failed transactions.

The basic SFL algorithm has the limitation of only being capable of detecting single faults. However, more advanced methods based on SFL for localizing multiple faults have already been devised [8].

	Component Group					
Transaction	fw	px	lb	hs	db	Error
1	1	3	2	4	3	0
2	1	2	2	3	0	0
3	1	1	1	2	1	0
4	1	0	0	0	0	1
5	1	2	1	1	0	1
6	1	1	2	1	0	1

Figure 2. PS example

III. Self-Healing in Cloud Applications

The appearance of computing and storage as a service, also known as Cloud Computing (CC), dramatically changed the way people see and use computers. If before CC applications and hardware had a tendency to work isolated and for local purposes, now, triggered by the advent of the Internet and the appearance of CC, we are observing an increasing interoperability between users and applications. However, such high degree of interaction between all components of the Cloud (users, applications, hardware, etc.) also leads to a higher probability of experiencing failures. Considering the high dynamic qualities of CC, the creation of self healing techniques becomes naturally a necessity. On the one hand, we believe that Cloud applications can deeply benefit from self-healing methods by being able to (1) increase their reliability while (2) reducing operational costs. On the other hand, the dynamic properties of the Cloud and the interoperability services provided by applications, easily enable the implementation of healing techniques such as infrastructures scaling, component rejuvenation, etc. Also, as Cloud applications tend to be distributed, modularized and service based, they ease the creation of a self-healing service layer.

In order to an application being able to self heal, it must at least be able to: (1) detect errors/failures, (2) locate the faulty components and (3) target such components with an appropriate remedy. As stated earlier, our main target of research is run-time fault localization. The algorithm in which we will focus our attention is SFL. As said before, SFL is a statistic based algorithm that was created to localize faults during development time and little work has been done in applying SFL to run-time environments. However, from the experience of previous work ([6]), we think that it manifests a set of properties that may be of interest in the scope of run-time, dynamic environments, such as Cloud Computing. First, on the performance side, SFL is considered to be lightweight (at least in comparison to other methods), being able to deliver results in a timely fashion. In run-time environments, the information about the possible error sources may experience a very narrow utility window.

Second, it does not require a model of the system, which in dynamic systems can sometimes be hard to obtain. As

users pay per use in the Cloud paradigm, applications tend to self adapt to environment changes by increasing or decreasing the number of components in order to be able to deliver the expected performance while reducing expenses.

Furthermore, it is relatively resilient to low quality oracles. Previous work ([2, 5, 7]) showed that it is a useful technique for reducing the scope of debugging and, even with low quality test suites, over 80% of the initial code is considered not to be a probable cause to a particular failure.

By having a reliable fault localization technique, it should be easier to determine whether or not and to what extent some corrective action really solved the original problem. This information enables more complex reasoning when selecting healing strategies as the information gathered from previous healing actions may impact the decision on future troubleshooting. An example of this event would be the scenario where healing actions were performed recurrently without really solving some problem or solving it in a mediocre way. With this feedback loop, the application should be able to detect such type of events and either plan another method to solve the problem or fall-back to a human operator.

Also, with a sound run-time fault localization technique, it should be easier to perform proactive maintenance in order to enhance survivability. Most of the time, and as we point out in Section IV-A, there normally is a gradual transition from a healthy to a faulty state. We expect to have the possibility to detect which components are most likely to be responsible for the system's degradation and with such information trigger proactive maintenance tasks to restore the system's normality. Proactive maintenance techniques are expected to be more cost effective than the reactive techniques [18] as the former ones plan and execute maintenance tasks while the system is still responsive (preferably without downtimes) while on the latter ones the planning and the execution of maintenance tasks is done while the system is unresponsive, representing further downtime. Moreover, proactive maintenance actions are applied to functional system and empirically it should be plausible to assume that less effort is needed to fix the it when compared to a broken system.

Finally, concerning preventive strategies, which are based on heuristics such as age or mean time between failures of each component, they can extract from SFL the information of which components failed and more accurately perform their healing actions.

Having in mind one of the main goals of Cloud Computing, cost reduction, we believe that a correct usage of self-healing techniques may effectively reduce expenses. First, applications should experience a higher degree of adaptiveness, enabling a thinner baseline usage of resources while being able to scale on demand. Second, they should become more robust to unexpected events and respond to those in an effective manner, assuring survivability. Finally, a more comprehensive view applications' health with much more clear and verbose bug reports. By being able to recognize and locate problems within the application, it should be possible to correlate faults, automatically report bugs and improve human supervised maintenance's efficiency.

IV. Spectrum-based Fault Localization for run-time fault diagnosis

While impressive diagnostic results have been achieved during development-time, little work has been performed in applying SFL to run-time fault localization. This comes from the fact that there is a huge difference between run-time and development-time environments. In this section we will explain how such differences impact in the creation of an SFL algorithm for run-time environments. Some of the issues addressed in this section have already been reported in other works such as [10] or [19].

A. State Fuzziness

When compared to development-time debugging, components in a run-time environment tend to suffer the effects of age [15] and some start to deliver the correct responses to the requests made while breaking certain constrains (temporal, resource usage, etc.). This type of behavior (soft-fail) cannot be completely described using a binary pass/fail flag as the component is not failing in the sense that it delivers a wrong response but, at the same time, is failing due to breaking certain constraints. Also, issues such as performance degradation or resource hogging which may not represent a system failure (hard-fail) within certain thresholds but, may trigger a hard-failure, leading to service deliverance interruption. A close monitoring and analysis of such artifacts may enhance of the system's behavior by possibly reducing the number of hard-failures [18], improving the usage experience and reducing operational costs.

The generic state diagram of a run-time system [14] is depicted in Fig. 3. It is clear that it is somewhat difficult to define the limits of some system's degraded state as it heavily depends on thresholds that are not normally agreed among all users (e.g. threshold latency of response from some component). An appropriate definition of the degraded state boundaries is of great importance as, on the one hand small boundaries may cause a direct transition from a normal to a broken state and, on the other hand large boundaries may trigger a large amount of corrective actions, which in turn may have an adverse effect on the system's responsiveness.

This kind of failures must be taken into consideration when designing the run-time SFL algorithm as the traditional binary pass/fail flag is too simplistic and may hide certain problems. Considering for instance, a component that either takes too long or uses too much memory (at least more than it would be expected to) but outputs the correct data, may be considered healthy if no constraints were enforced. On the other hand, if some transaction would be considered failed

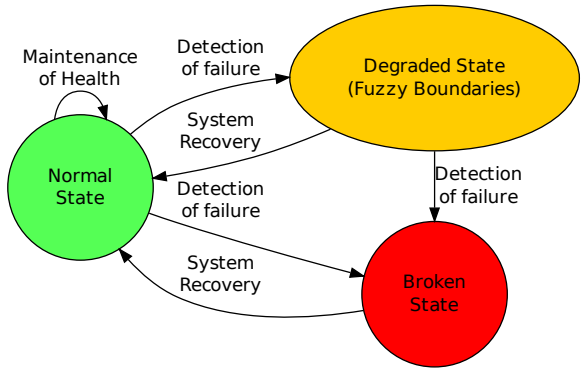

Figure 3. Run-time environment state diagram

due to resource overuse, assuming that no more data was available, the real reason of failure would be masked by the binary pass/fail information.

The distinction between soft-failures and hard-failures enables a better selection of the healing actions to use. The software ageing problem has already been acknowledged and several methods have been developed to tackle this problem (for instance software rejuvenation [17, 22], in which components are restarted in order to improve the system's robustness).

B. System Dynamics

Another issue on applying SFL to run-time environments is the assumption that, provided with the same input, an arbitrary chain of components should always deliver the same results. While in development-time environments this premise almost always observed, in run-time, as the system is not normally reset before each transaction, this condition is broken. This results from the fact that a particular set of operations may alter the component health and influence the result of the following transactions. This effect can possibly mangle the conclusions taken from the PS (see Section II).

In order to minimize the "memory effect" of the run-time operation it is necessary to have some notion of time when analysing the PS. One approach to the problem is by defining an analysis window and correctly dimension it. The sub-problem that arises is how to determine the time span in which an observation is valid. We believe that it should be possible to determine how dynamic the system under analysis and predict an appropriate window size for each component. Provided that a good observation window is used, SFL should deliver better results, when comparing to analysing all available data, as obsolete data is purged.

Another approach that can be used isolated or in addition to the analysis window, is by taking into account the time diference between each transaction and the analysis point. Empirically, it should be acceptable that an older observation as less impact in some diagnostic than a more recent one. By giving weight to each observation, it is possible to reduce an eventual negative impact induced by obsolete data. However, an similarly to the previous approach, we must determine how to weigh each observation. The weighing curve is once again deeply related with the system's dynamic behavior and should be possible to deduce an appropriate one automatically and dynamically.

Still regarding system dynamics, the effects of altering the system's configuration on the diagnosis performance of SFL, to the best of our knowledge, have not yet been studied. However we foresee that it may pose a problem to the classic SFL. When the system is altered without adding or removing any component, two scenarios exist: the components subjected to the corrective action (1) maintain their columns in matrix A and inherit the previous history or (2) a new columns are created and a new history branch is created. For (1), older data may confuse the algorithm as the components will change their behaviors. This effect may be attenuated by the window displacement and age weighing techniques previously presented. As for (2), there is the advantage that the corrected components are treated as a new components and little adaptation should be needed to the classic algorithm.

Still regarding the effects of triggering healing actions, it is very important to keep track of such events in order to acknowledge the effectiveness of some corrective action when applied to a failing application with a given set of symptoms. For instance, if some component was subjected to some healing action and is considered to be broken again, it should be possible to analyse its history and from this analysis determine the remedy effectiveness.

C. Performance

When applying SFL to a high dimension application, and considering a high granularity instrumentation, each transaction will have a significant impact in terms of resource utilization. For instance the Linux kernel version 3.2 has around 11.5 million lines of code. If instrumented to the statement level, each transaction record would have a size of approximately 1.3 MB (in a dense binary matrix).

Another related problem that results from the instrumentation of the target application is performance loss. Even though the performance impact derived from the instrumentation was reported to be low, averaging 6% loss [1], in run-time systems such loss may represent an issue. Also, as Cloud users pay based on the infrastructure usage, performance losses represent additional costs.

Such issues may be solved by creating zones within the application with different levels of granularity. One challenge to this solution would be how to decide the inspection granularity of each component. This decision could be any combination of either static or dynamic and human or automatic. The automatic dynamic mode, which is the most challenging, could be based on the knowledge gathered about the system. For instance, if a component only

appears in successful and high quality transactions, it should be more or less safe to discard the high granularity details about its inner processes. On the other hand, components with a high probability of being faulty should be inspected with a higher degree of detail in order to enable a better reasoning on the fault cause. The dynamic probe placement can be implemented by using just-int-time (JIT)frameworks such as LLVM . While the dynamic approaches present transient performance reductions due to the warm-up effect of JIT, they provide a higher degree of flexibility.

D. Test oracles

A requirement to the SFL algorithm is to have some mechanism that evaluates the status of the transactions. Those mechanisms are commonly called test oracles and, in development-time debugging, they are commonly implemented either by manually creating test cases or by having some reference implementation that is known to be faultless. Some work ([2, 3, 11, 12, 13]) has already been done in order to creating low-cost oracles by using program invariants to determine the transactions' statuses.

Program invariants are conditions that when broken the program is guaranteed to be in a faulty state. Such conditions may be user defined, naive or machine learned. User defined invariants may be defined in the code or in a model definition of the application and may manifest themselves in the form of exceptions or through specific framework calls. Naive invariants are those that are relatively easy to detect such as invalid pointer addresses or deadlocks. Machine learned invariants must be subjected to a prior training in order to perform adequately. Examples of machine learned invariants are array bounds or range detection. Once again, the effectiveness of machine learned invariants deeply relies on the quality and quantity of training done. Also, the use of machine learned invariants may introduce the existence of false negatives and, even worse, false positives, which are known to mangle SFL [6].

Given the difficulty of producing accurate oracles, it is very important to enable SFL to weigh each oracle output in order to produce accurate results out from inaccurate data. The weight of each output could be either determined by the oracle itself, with some score reflecting its confidence on the result, or based on the reputation, which could be generated during run-time.

Another issue that has to be solved is the necessity to define the limits in which each oracle will operate. Throughout this paper we used the term "transaction" to define such limits.

In the scope of SFL, transactions (lines in the PS, see Section II) are one of the building blocks of the algorithm. In development-time, transactions are delimited by extent of each unit test. However, when it comes to run-time systems, it is not always trivial to define such boundaries. If, on one hand there are systems in which it is easy to define the boundaries of a transaction, as for instance in the example of Fig. 1, on the other hand there are systems in which it is not clear to define transactions, for instance a word processor.

Additionally we foresee the possibility of having multiple oracles evaluating overlapping scopes as a way of improving the accuracy of the fault localization technique. This method follows a philosophy similar to the modular redundancy techniques and should enable a more effective usage of low-cost oracles. We also expect to be able to use the system test suite in order to gather accurate PS. By previously analyzing the coverage of each unit test in the system test suite, it is possible to target suspicious components, with a relevant set of tests. Those tests should be performed without reseting the system in order provide a representative picture of the system's status.

Regarding the performance degradation problems, the test suite should be first ran in an healthy system to create a set of reference values which can be compared to the results obtained in run-time.

In order not to disrupt the system's responsiveness, such tests should only be evaluated under a certain load threshold. However it may be worthwhile to perform some testing, when the expected accuracy of the PS available within the analysis window (see Section IV-B) falls under a certain threshold.

V. Evaluating Results

In order to evaluate the efficiency of our run-time SFL algorithm, we are considering a two pronged approach. On the one hand we are creating a simulator which from some arbitrary system's model and behavior description can generate Program Spectrum. On the other hand we will apply SFL to representative live systems while injecting faults in it. By comparing the injected faults with the results from SFL we can estimate how SFL behaves in different application types.

The first approach enables: (1) a more objective benchmarking by being possible to easily creating a simulated system that tests a set of specific artifacts; (2) an easier process of testing, as normally "normal" applications must be configured and altered to interact with a monitoring entity; (3) test reproducibility as the system's model and behavior are contained in a single file, which is easily distributable.

The second approach provides: (1) real world validation and (2) usefulness proof.

Provided that good results are found from the aforementioned methodology, we also plan integrating SFL algorithm with some correction mechanisms and evaluate SFL's performance when cooperating in a self-healing loop. For that we plan on enhancing the simulator to provide actuators on the simulation in order to alter the simulation path as if some healing task was triggered. Further, we expect to also test SFL in real self-healing applications.

VI. Conclusions

In this paper we discussed how the advent of the Internet and the Cloud aggravated the old software bug problem. We started by introducing SFL, an fault localization algorithm used in development time environments, which we believe that has several characteristics that are well suited for run-time environments. Following we explained why Cloud Computing both can benefit from and favor self-healing techniques. Self-healing techniques can take advantage of the modularity and flexibility of the Cloud to make it more robust and trustworthy. Our main topic, the issues that must be solved to create an SFL algorithm for run-time, was then addressed. We discovered at least four groups of issues that at best render ineffective the usage of development-time SFL in runtime environments. The first is related with state fuzziness where it is not always possible to categorize all transactions as having either passed or failed. The second issue is the fact of systems having a dynamic behavior when compared to development-time testing. In run-time the result or issues of previous transactions may have effects on future ones. The third is performance. On the one hand, the algorithm performance must be high enough to deliver results in a timely manner. On the other hand, the target application may suffer from performance losses due to instrumentation. Finally, and the most common issue in fault localization, is the difficulty of creating accurate oracles for runtime. We proposed several ideas on how to deal with such issues, which will be target of future work.

Besides, we presented our evaluation methodology which will use to assess the diagnosis performance of the run-time SFL. Two approaches will be used. The first will use simulations in order to have a more fine tuned evaluation. The second will use real applications to have a proof that the method is implementable in the real world.

Finally, SFL has already shown its potential on detecting the possible fault sources in development-time debugging and, despite the amount of adaptations needed, we expect similar results when applied to run-time.

References

[1] Rui Abreu. *Spectrum-based Fault Localization in Embedded Software*. PhD thesis, Delft University of Technology, November 2009.

[2] Rui Abreu, Alberto Gonzalez, Peter Zoeteweij, and Arjan J.C. van Gemund. Automatic software fault localization using generic program invariants. In Roger L. Wainwright and Hisham Haddad, editors, *Proceedings of the 23rd Annual ACM Symposium on Applied Computing*, SAC'08, pages 712–717. ACM Press, March 2008.

[3] Rui Abreu, Alberto Gonzalez, Peter Zoeteweij, and Arjan J.C. van Gemund. On the performance of fault screeners in software development and deployment. In Cesar Gonzalez-Perez and Stefan Jablonski, editors, *Proceedings of the 3rd International Conference on Evaluation of Novel Approaches to Software Engineering*, ENASE'08, pages 123–130. INSTICC Press, May 2008.

[4] Rui Abreu and Arjan J. C. van Gemund. Diagnosing multiple intermittent failures using maximum likelihood estimation. *Artif. Intell.*, 174(18):1481–1497, 2010.

[5] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC '06, pages 39–46, Washington, DC, USA, 2006. IEEE Computer Society.

[6] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[7] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, TAIC PART'07, pages 89–98, Washington, DC, USA, 2007. IEEE Computer Society.

[8] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, ASE'09, pages 88–99, 2009.

[9] Philipp Bouillon, Jens Krinke, Nils Meyer, and Friedrich Steimann. EzUnit: A Framework for Associating Failed Unit Tests With Potential Programming Errors. *Proceedings of the 8th International Conference on Agile Processes in Software Engineering and eXtreme Programming (XP 2007), Como, Italy 2007*, 2007.

[10] Paulo Casanova, Bradley R. Schmerl, David Garlan, and Rui Abreu. Architecture-based run-time fault diagnosis. In *Proceedings of the 5th European Conference on Software Architecture*, ECSA'11, pages 261–277, 2011.

[11] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[12] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 449–458, New York, NY, USA, 2000. ACM.

[13] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69:35–45, December 2007.

[14] Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. Self-healing systems - survey and synthesis. *Decis. Support Syst.*, 42(4):2164–2185, January 2007.

[15] Michael Grottke, Rivalino Matias Jr, and Kishor S Trivedi. The Fundamentals of Software Aging. *Symposium A Quarterly Journal In Modern Foreign Literatures*, pages 1–6, 2008.

[16] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[17] Nick Kolettis and N. Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, FTCS '95, pages 381–, Washington, DC, USA, 1995. IEEE Computer Society.

[18] Ranganath Kothamasu, Samuel H Huang, and William H VerDuin. System health monitoring and prognostics - a review of current paradigms and practices. *The International Journal of Advanced Manufacturing Technology*, 28(9–10):1012–1024, 2006.

[19] Éric Piel, Alberto González-Sanchez, Hans-Gerhard Groß, and Arjan J. C. van Gemund. Spectrum-based health monitoring for self-adaptive systems. In *Proceedings of the 5th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, SASO'11, pages 99–108, 2011.

[20] André Riboira, Rui Abreu, and Rui Rodrigues. An opengl-based eclipse plug-in for visual debugging. In *Proceedings of the 1st Workshop on Developing Tools as Plug-ins*, TOPI '11, pages 60–60, New York, NY, USA, 2011. ACM.

[21] R. Stallman, R. Pesch, and S. Shebs. *Debugging with GDB: the GNU source-level debugger*. A GNU manual. GNU Press, 2002.

[22] Kishor S. Trivedi and Kalyanaraman Vaidyanathan. Software aging and rejuvenation. In *Wiley Encyclopedia of Computer Science and Engineering*. 2008.