Increasing System Availability with Local Recovery based on Fault Localization

Hasan Sözer*, Rui Abreu†, Mehmet Akşit* and Arjan J.C. van Gemund‡

*University of Twente, The Netherlands
{sozerh, m.aksit}@ewi.utwente.nl

†Faculty of Engineering, University of Porto, Portugal
rma@fe.up.pt

‡Delft University of Technology, The Netherlands
a.j.c.vangemund@tudelft.nl

***Abstract*—Due to the fact that software systems cannot be tested exhaustively, software systems must cope with residual defects at run-time. Local recovery is an approach for recovering from errors, in which only the defective parts of the system are recovered while the other parts are kept operational. To be efficient, local recovery must be aware of which component is at fault. In this paper, we combine a fault localization technique (spectrum-based fault localization, SFL) with local recovery techniques to achieve fully autonomous fault detection, isolation, and recovery. A framework is used for decomposing the system into separate units that can be recovered in isolation, while SFL is used for monitoring the activities of these units and diagnose the faulty one whenever an error is detected. We have applied our approach to `MPlayer`, a large open-source software. We have observed that SFL can increase the system availability by 23.4% on average.**

I. Introduction

Due to the sheer size and complexity of today's software systems, testing software exhaustively has become prohibitive. Therefore, to increase system availability, software systems must cope with residual defects at run-time. This requires the incorporation of fault tolerance in which it is accepted that faults exist and might get activated while the system can recover from the consequences (i.e., errors), if possible before a failure occurs.

Software failures are generally caused by design and coding faults, being typically permanent [1]. However, most of these faults are only activated by, e.g., timing issues, race conditions, resource leaks and peak conditions in workload that could not all have been anticipated before. As a matter of fact, errors that are caused by such faults are likely to be resolved when the software is re-executed after a clean-up and initialization [1]. Consequently, it is possible to design a system that can recover from a significant fraction of errors [2] at run-time.

To attain high system availability [3], recovery actions can be applied only to a fraction of the system (when possible), while the other parts remain operational. For example, only the failed components can be restarted rather than the whole system (i.e., microreboot [2]) so that the other components can remain available. Moreover, applying recovery to a subset of the system components rather than the whole system decreases the mean time to recover [2]. Hence, for faster recovery and better availability, it is necessary to reduce the granularity of the parts in the system that can be recovered and as such realize local recovery. However, local recovery is not always successful in cases where, the erroneous/failed components are not also the ones *at fault*. Even if a failed component is restarted locally, another faulty component can, e.g., keep sending messages that will crash the failing component again, soon after its recovery. As a result, progressively larger subsets, and eventually the whole system might need to be restarted [2]. Therefore, to achieve efficient local recovery, effective fault diagnosis is needed, localizing the *root causes* of errors (i.e., faults, defects).

The process of pinpointing the fault(s) that led to symptoms (i.e., failures/errors) is called *fault localization*, and has been an active area of research for the past decades. Based on a set of observations, automatic approaches to software fault localization yield a list of likely fault locations, which is subsequently used by the developer to focus the software debugging process [4], [5]. Depending on the amount of knowledge that is required about the system's internal component structure and behavior, the most predominant approaches can be classified as *i)* statistical approaches, of which spectrum-based fault localization (SFL) [6] is an example, or *ii)* reasoning approaches. The former approach uses an abstraction of program traces, dynamically collected at runtime, to produce a list of likely candidates to be at fault, whereas the latter combines a static *model* of the expected behavior with a set of observations to compute the diagnostic report. In the remainder of this paper we only consider SFL for fault localization as *i)* it entails low time and space complexity, and *ii)* it is amongst the best performing techniques [5].

To achieve effective local recovery and as such increase system availability, we have utilized two techniques, namely local recovery supported by FLORA [7], extended by an implementation of SFL [8]. FLORA is a framework that is used for decomposing a system into separate units that can be recovered in isolation. SFL [4], [9] is used for monitoring the activities of these units and diagnoses the faulty one whenever an error is detected. We have combined SFL and FLORA for providing an integrated fault tolerance approach. Our approach does not depend on a particular application domain or an architectural structure, and as such it is generic. Our integrated framework provides an implementation of the approach for software systems that are implemented in C for Linux platform. We have applied our approach to an open-source software media player, `MPlayer`.

In this paper, we show that *i)* local recovery increases the system availability, and *ii)* fault localization significantly improves the recovery effectiveness. We have observed that for `MPlayer` local recovery takes up to 83.8% less time than restarting the whole system. Furthermore, for failure scenarios, in which the faulty location is different than the erroneous location, fault localization reduces the mean time to recover by 23.4% on average.

The remainder of this paper is organized as follows. The next two sections provide background information on the local recovery and fault localization techniques, respectively. In Section IV, we describe our integrated fault tolerance approach. Section V presents the case study and discusses the realization issues. Section VI presents an evaluation of the approach. In Section VII, related previous studies are summarized. Finally, in section VIII we discuss some future work issues and provide the conclusions.

II. Local Recovery

Local recovery is an effective approach for recovering from errors, in which the erroneous parts of a system are restarted while the other parts of the system are kept operational. Introducing local recovery to a system imposes certain requirements.

- *Isolation*: An error occurring in one part of the system can easily propagate and lead to errors in other parts. To prevent this error propagation we need to be able to decompose the system into a set of *Recoverable Units (RUs)* with clear boundaries and isolation between them.
- *Communication Control*: Although an RU is unavailable during its recovery, other RUs might still need to access it in the meantime. Therefore, the communication between RUs must be mediated and controlled (e.g., through blocking, queuing and retrying of messages), so that the recovery of an RU is transparent to the other RUs.
- *System-Recovery Coordination*: In case recovery actions need to take place while the system is still operational, interference with the normal system functions can occur. For this reason, the required recovery actions need to be coordinated.

FLORA [7] is a framework that supports the decomposition and implementation of software architecture for local recovery. A set of RU wrappers are defined to wrap system modules into separate RUs. FLORA partitions system modules as defined by RU wrappers and isolates these modules by assigning each RU to a separate process[1]. In addition to the specified RUs, FLORA introduces a *Communication Manager (CM)* and a *Recovery Manager (RM)*. The CM mediates and controls all inter-RU communication. The RM coordinates the recovery of RUs for killing and/or restarting them.

FLORA implements the detection of several type of errors, including deadlocks and fatal errors (e.g., illegal instruction, invalid data access), buffer overflows and null pointer exceptions. The RU that is associated with a detected error is assumed to be also the source of the error (i.e., the faulty RU). If the error is not recovered after the restart of the corresponding RU, the whole system is restarted. This is similar to the other local recovery approaches [2], where increasingly larger subsets of the system is restarted until the error is recovered. However, this can increase the time to recover. Also, if the fault and/or error is either in the CM or the RM, the framework then applies global recovery by restarting the whole system. In the following, we provide background information on fault localization techniques, and in particular the SFL technique that we have utilized in FLORA.

III. Fault Localization

The process of pinpointing the fault(s) that led to the observed symptoms (failures/errors) is called fault localization, and has been an active area of research for the past decades. Based on a set of observations, automatic approaches to software fault localization yield a list of likely fault locations, which is subsequently used either by the developer to focus the software debugging process, or as an input to automatic recovery mechanisms [10], [11]. Depending on the amount of knowledge that is required about the system's internal component structure and behavior, the most predominant approaches can be classified as *i)* statistical approaches or *ii)* reasoning approaches (for an overview of approaches, see [8]). The former approach uses an abstraction of program traces, dynamically collected at runtime (also known as program spectra [12]), to produce a list of likely candidates to be at fault [4], [9], [13], whereas the latter combines a static *model* of the expected behavior with a set of observations to compute the diagnostic report [14]. In this paper we use a statistical technique, in particular spectrum-based fault localization (SFL, [4], [9]), because it is not only amongst the best fault localization techniques, but also due to the fact that entails low time and space complexity [8].

Spectrum-based fault localization (SFL) is a dynamic program analysis technique that has shown that comparing the program behavior over multiple test runs can indicate which program components may be likely to contribute to an observed program failure.

In the following, we assume that a program $\mathcal{P}$ comprises a set of components $\mathcal{C}$ and is executed using a set of test cases $\mathcal{T}$ that either pass or fail, with $M = |\mathcal{C}|$ and $N = |\mathcal{T}|$, respectively. Program (component) activity is recorded in terms of

[1]Interaction among the RUs are redirected through Inter-Process Communication.

$$
\begin{array}{cc}
 & \begin{array}{ccccc} & M \text{ components} & & & \text{error vector} \end{array} \\
N \text{ spectra} & \left[\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1N} & e_1 \\ a_{21} & a_{22} & \dots & a_{2N} & e_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{M1} & a_{M2} & \dots & a_{MN} & e_N \end{array}\right] \\
 & \begin{array}{cccc} s_1 & s_2 & \dots & s_N \end{array}
\end{array}
$$

Figure 1. The ingredients of fault diagnosis

program spectra [4], [9], [13]. These data are collected at run-time and typically consist of a number of counters or flags for the different components of a program. We use the so-called *hit spectra* that indicate whether a component was involved in a (test) run or not.

Both spectra and pass/fail information is input to SFL. The combined information is expressed in terms of the $N \times (M+1)$ *activity matrix* A. An element a_{ij} is equal to 1 if component j took part in the execution of test run i, and 0 otherwise. The rightmost column of A, the error vector e, represents the test outcome. The element $e_i = a_{i,m+1}$ is equal to 1 if run i *failed*, and 0 if run i *passed*. For $j \leq M$ and $i \leq N$, the row A_{i*} indicates whether a component was executed in run i, whereas the column O_{*j} indicates in which runs component j was involved.

In SFL one measures the similarity between the error vector e and the activity profile vector A_{*j} for each component j (see Figure 1). This similarity is quantified by a *similarity coefficient*, s_j. In this paper, the Ochiai similarity coefficient, known from molecular biology, is used[2]. s_j associated with each component $C_j \in \mathcal{C}$ indicates the correlation between the executions of C_j and the observed incorrect program behavior. Applying the hypothesis that closely correlated components are more likely to be relevant to an observed misbehavior, s_j can be reinterpreted as "fault probability" and components can be listed in order of likelihood to be at fault.

The SFL technique can be applied to localize faults at different granularity levels (e.g., source code blocks, functions, modules). In this paper, we apply SFL at an architectural level and use this technique to identify faulty architectural components. As such, a local recovery mechanism can utilize the diagnosis information to focus on the faulty components in the software architecture. In the following, we explain our integrated approach.

IV. The Approach

The effectiveness of recovery strategies can be limited if they rely only on the information regarding the detected errors. Especially for local recovery approaches, such as FLORA, the effectiveness of recovery actions highly depends on the diagnosis i.e., to be able to know precisely which (recoverable) unit is at fault. In order for FLORA to be aware which component (RU) is faulty, it is augmented with a fault localization technique. In particular, we use the SFL technique, described in the previous section, to localize faults amongst the set of RUs. This integration allows FLORA to obtain educated guesses with respect to which RU(s) should be restarted first. Hence, the RU decomposition of the software architecture defines the granularity level (i.e., components) for the diagnosis. SFL is designed to be a part of the CM, which monitors all the messages exchanged among the RUs. Such monitoring offers the capabilities for the CM to record all the activity in the system in terms of a data structure to represent the information in Figure 1.

- the RU j was involved in execution i ($a_{ij} = 1$);
- the RU j was not involved in execution i ($a_{ij} = 0$).

The facilities provided by FLORA are used for automatic error detection. If an error is detected during a given run i, then the run is flagged as failed ($e_i = 1$). If no error is detected in run i, then the run is flagged as passed ($e_i = 0$). This recording process is repeated for every single execution (i.e., between key presses in the GUI, between completion of the processing of video frames) of the program. Successive executions are identified based on the detection of related messages as specified in a configuration file. The a_{ij} and e_i information are used as input for the SFL module whenever a failed run is observed. The SFL module computes a list of RU in order of likelihood to be faulty, which is used by the RM to decide which RU(s) should be recovered. The infrastructure and dependencies of its modules are depicted in Figure 2. We can see the main components of FLORA;

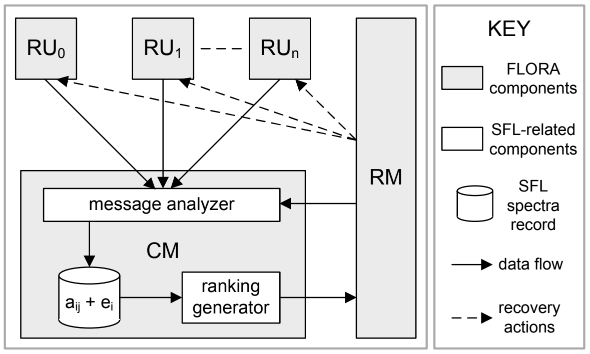

Figure 2. The SFL-based recovery infrastructure

the RM, the CM and a set of RUs. SFL is realized by a set of components within the CM. All the messages exchanged among different RUs are analyzed and the SFL spectra record is updated accordingly. Detected errors are notified by RUs and the RM with notification messages. A ranking of possibly faulty RUs is generated and provided to the RM based on the collected spectra record. This ranking is used for deciding on the recovery actions to be applied to a set of RUs or to the whole system.

In the following sections, we explain how our integrated approach is applied to adapt a given architecture for local recovery based on fault localization.

[2]Previous investigations have identified it as the best coefficient to be used for SFL [9].

V. Case Study: MPlayer

MPlayer [15] is a well-known media player, which supports many input formats, codecs and output drivers. It embodies approximately 700K lines of code and it is available under the GNU General Public License. In our case study, we have used version v1.0rc1 of this software that is complied on Linux Platform (Ubuntu version 7.04). Figure 3 presents basic modules of the MPlayer software architecture with dependencies among them. In the following, we briefly explain the important modules that are shown in this view.

Stream reads the input media and provides buffering, seek and skip functions. *Demuxer* demultiplexes (separates) the input to audio and video channels, and reads them from buffered packages. *Mplayer* connects all the other modules, and maintains the synchronization of audio and video. *Libmpcodecs* embodies the set of available codecs. *Libvo* displays video frames. *Libao* controls the playing of audio. *Gui* provides the graphical user interface of MPlayer.

We have realized our approach within the context of the MPlayer case study. We have used our framework to decompose the system intro 3 RUs; *i)* RU AUDIO, which provides the functionality of Libao *ii)* RU GUI, which encapsulates the Gui functionality and *iii)* RU MPCORE which comprises the rest of the system. Figure 3 depicts the boundaries of these RUs, which are overlayed on the module decomposition of the MPlayer software architecture.

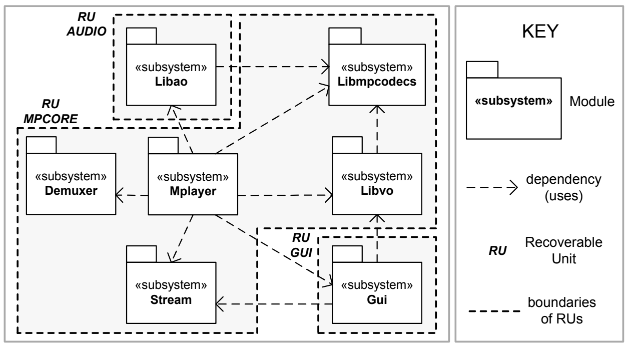

Figure 3. Basic modules of MPlayer and boundaries of the RUs, which are overlayed on the module decomposition

In Figure 4 the recovery design corresponding to this RU selection is shown. Here, we can see the CM, the RM and the three RUs, RU MPCORE, RU GUI and RU AUDIO. Each RU can detect deadlock errors. The recovery manager can detect fatal errors. All error notifications are send to the CM, which comprises the diagnosis facility. Diagnosis information is conveyed to the RM, which kills a set of RUs and/or restarts a dead RU. Messages that are sent from RUs to the communication manager are stored (e.g., queued) by RUs in case the destination RU is not available and they are forwarded when the RU becomes operational again.

In our approach (Section IV), we have applied SFL at a granularity level determined by the RU decomposition to provide recovery-oriented diagnosis information. Essentially, this means that each RU is a component for SFL. Thus, the CM reserves space to store the information needed by SFL (i.e., a_{ij} and e_i) for each RU and certain framework elements. In case of the MPlayer case study, 6 components were considered (i.e., $j \in \{1, \ldots, 6\}$): the RM, the CM, each RU (i.e., RU MPCORE, RU GUI and RU AUDIO), and the framework facilities that are used for state preservation. The a_{ij}s are updated at every message exchange. The passed runs are identified based on the reception of particular messages that are defined in a configuration file. In the MPlayer case study, we have specified the messages received from RU MPCORE for this purpose. These are the messages that are associated with a completion of processing of a video frame and completion actions related to user events (e.g., volume change, opening a video file, skipping). Whenever the specified messages are received from RU MPCORE, the activity information collected so far is associated with an execution i with $e_i = 0$. Whenever there is an error, the activity information collected since the last execution $(i - 1)$ is associated with an execution i with $e_i = 1$.

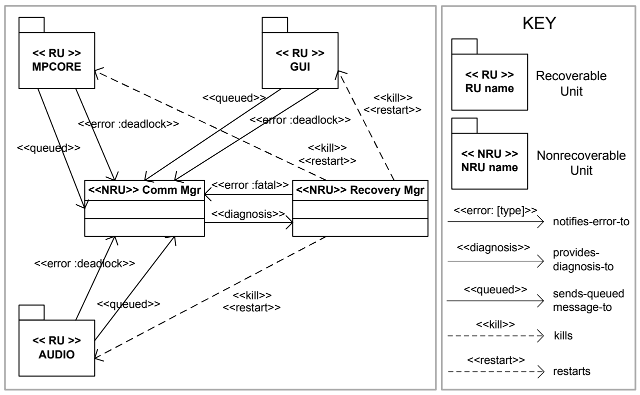

Figure 4. Realized MPlayer Software Architecture with 3 RUs

Diagnosis information is conveyed to the RM, which kills a set of RUs and/or restarts a dead RU. If the faulty RU is different than the erroneous RU, both RUs are restarted. If the RM, CM or framework facilities are subject to faults or errors, the whole system is restarted (global recovery). Messages that are sent from RUs to the CM are stored (queued) by RUs in case the destination RU is not available and they are forwarded when the RU becomes operational again.

VI. Evaluation

To test our framework, we have injected several faults within the modules of RU MPCORE, RU GUI and RU AUDIO. We have injected 3 types of faults: *i)* illegal memory operation that causes a fatal error *ii)* busy waiting or skipping a message that causes a deadlock *iii)* sending of messages that causes a buffer overflow and in turn, a fatal error at the destination of the messages. These faults are activated with certain (combination of) button presses at the user interface panel.

We have run the modified MPlayer several times, systematically activated the faults and observed the results and log files. The SFL-based diagnosis mechanism was able to point out the location of the fault correctly each time (i.e., the corresponding RU was ranked 1^{st} among the others). Each time a fault in RU MPCORE is activated, our framework initiated global recovery and restarted the whole system. Whenever the faults in RU GUI or RU AUDIO were activated, our framework was able to recover from the error locally by restarting these RUs, while the other parts of the system remained operational.

From the user perspective, local recovery increases the availability of the system significantly. During the recovery of RU GUI, the user interface panel vanishes and comes back within a second. However, audio/video streaming continues. Similarly, during the recovery of RU AUDIO, the sound is muted for about a second. However, all the screens remain intact and video streaming continues. After the recovery of RU AUDIO, audio streaming starts and synchronizes with the video. In addition, restarting RUs individually takes 83.8% less time compared to restarting the whole system[3].

We have observed that fault localization information leads to a faster recovery. When the erroneous RU is the same as the faulty RU, the corresponding RU is restarted. If the error is detected in the RM[4], CM, RU MPCORE or framework facilities, a global recovery is applied. In these cases, the first recovery attempt is successful. However, if the erroneous RU is not the faulty one, local recovery is not always successful. The faulty component keeps sending messages that crashes the same RU upon its recovery. The RM has to apply several local recovery attempts and finally a global recovery to recover from the failure successfully. However, the availability of diagnosis information enables the RM to *i)* restart both the faulty and erroneous RUs at once, or *ii)* restart the whole system directly rather than trying (unsuccessfully) local recovery first. In the first case, we have measured on average 32.8% reduction in mean time to recovery. In the second case, the reduction was 13.9% on average. The average reduction considering both cases is 23.4%. The impact is lower for the second case because of the significant difference (83.8%) of time to recover between the global recovery and local recovery. Thus, the time lost during the (unsuccessful) local recovery attempt becomes less significant.

If local recovery mechanisms are not supported with diagnosis information, several attempts for local recovery will be eventually followed by a global recovery. In such cases, time to recover increases, even compared to a naive global recovery approach. There have been also recursive approaches proposed, in which the system components are organized in a hierarchy, namely a reboot-tree [16]. Whenever an error is detected, a minimal subset of components is recovered first; if that does not work, progressively larger subsets are recovered, moving upwards in the reboot-tree hierarchy. This approach, however, is based on the structuring of the system at design-time. SFL does not depend on the structure of the system and it is a dynamic approach based on the information collected at run-time. The ranking provided by SFL can also be utilized by a recursive approach, where progressively larger subsets of the system are recovered based on their ranks.

[3]Mean time to recover is measured by calculating the mean time it takes to restart a process and the corresponding modules over 100 runs.

[4]The RM forks another process for monitoring and restarting itself.

VII. Related Work

Techniques for self-healing/recovering [17] from failures in software systems have been subject of research for decades. In the following, we summarize the techniques that are proposed so far and discuss the position of our work, especially related to the techniques for fault localization and local recovery.

In the fault localization domain, many approaches exist. Depending on the amount of knowledge that is required about the system's internal component structure and behavior, the most predominant approaches can be classified as *i)* statistical approaches (e.g., [4], [9], [13], [20], [21]) or *ii)* reasoning approaches [5], [14]. The former approach uses an abstraction of program traces, dynamically collected at runtime, to produce a list of likely candidates to be at fault, whereas the latter combines a static *model* of the expected behavior with a set of observations to compute the diagnostic report. For run-time usage, statistical techniques, of which SFL is an example (see Section III for details), are especially interesting because *i)* are amongst the best performing techniques [5], and *ii)* entail low time and space complexity [6].

There exist several micro-kernel architectures [22]–[24], programming environments [25], software libraries [1], middlewares [26] and workbenches [27] that support local recovery. However, they do not support the restructuring and partitioning of legacy software to introduce local recovery and they do not employ fault localization techniques.

To the best of our knowledge, the most similar work to the one presented in this paper is the Recovery Oriented Computing project [10]. In the context of this project, local recovery i.e., *microreboot* [2] is applied to increase the availability of Java-based Internet systems. To employ microreboot, a system has to meet a set of architectural requirements (i.e., *crash-only design* [2]), where components are isolated from each other and their state information is kept in state repositories. Designs of many existing systems do not have these properties and it might be too costly to redesign and implement the whole system from the start. Our framework provides a set of reusable abstractions and mechanisms to support the refactoring of existing systems to introduce local recovery.

VIII. Conclusion and Future Work

Coping with software failures at run-time entails the capacity of being able to detect and recover from them. However, the complementary domains of automatic fault localization and recovery of software failures have thus far been been investigated separately.

In this paper we have integrated a fault localization technique, namely spectrum-based fault localization (SFL), with the FLORA framework to achieve local recovery of errors, and fully automate fault tolerance. FLORA is used for decomposing the system into separate recoverable units, and SFL is used for monitoring the activities of these recoverable units and diagnose the faulty one whenever an error is detected. We have applied our integrated approach on a large open-source media player software, called `MPlayer`.

We have observed promising results based on the applied failure scenarios. SFL-based diagnosis mechanism was able to correctly localize the real faults and as such support the recovery mechanism. Especially when the faulty location is different than the erroneous location, diagnosis information leads to a faster recovery. Avoiding unsuccessful local recovery attempts reduces the mean time to recover by 23.4% on average. We have also observed that local recovery, when possible, increases the availability of the system significantly. In our case study, system components related to the graphical user interface and audio streaming can be recovered locally, while video streaming continues. This results in *i)* a faster recovery (83.8%), and *ii)* availability of video to the user during recovery.

For future work we would like to apply the SFL-FLORA framework to other software programs, so that we can generalize the findings reported on this paper. In addition, we would also like to apply the technique to a broader range of faults. Finally, we also intend to extend the framework with generic, automatic error detection techniques, e.g., [28], to be able to detect more errors.

Acknowledgments

We acknowledge the feedback from the discussions with our TRADER project partners. This work has been carried out as part of the TRADER project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program.

References

[1] Y. Huang and C. Kintala, "Software fault tolerance in the application layer," in *Software Fault Tolerance*, M. R. Lyu, Ed. John Wiley & Sons, 1995, chapter 10, pp. 231–248.

[2] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot: A technique for cheap recovery," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, 2004, pp. 31–44.

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

[4] J. A. Jones and M. J. 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)*, D. F. Redmiles, T. Ellman, and A. Zisman, Eds. Long Beach, California, USA: IEEE Computer Society, 7 – 11 November 2005, pp. 273–282.

[5] R. Abreu, P.Zoeteweij, and A. van Gemund, "Spectrum-based multiple fault localization," in *24th International Conference on Automated Software Engeneering (ASE'09)*, G. Taentzer and M. Heimdahl, Eds. IEEE Computer Science, November 2009, pp. 88 – 99.

[6] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, November 2009.

[7] H. Sozer, B. Tekinerdogan, and M. Aksit, "FLORA: A framework for decomposing software architecture to introduce local recovery," *Software: Practice and Experience*, vol. 39, no. 10, pp. 869–889, 2009.

[8] R. Abreu, "Spectrum-based fault localization in embedded software," Ph.D. dissertation, Delft University of Technology, November 2009.

[9] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART'07)*, P. McMinn, Ed. Windsor, United Kingdom: IEEE Computer Society, September 2007, pp. 89–98.

[10] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery Oriented Computing (ROC): Motivation, definition, techniques, and case studies," University of California at Berkeley, Tech. Rep. UCB/CSD-02-1175, 2002.

[11] H. Sözer, "Architecting fault-tolerant software systems," Ph.D. dissertation, University of Twente, January 2009.

[12] M. Harrold, G. Rothermel, R. Wu, and L. Yi, "An empirical investigation of program spectra," *ACM SIGPLAN Notices*, vol. 33, no. 7, 1998.

[13] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, V. Sarkar and M. W. Hall, Eds. Chicago, Illinois, USA: ACM Press, 12 – 15 June 2005, pp. 15–26.

[14] W. Mayer and M. Stumptner, "Evaluating models for model-based debugging," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, A. Ireland and W. Visser, Eds. L'Aquila, Italy: ACM Press, 15 – 19 September 2008, pp. 128–137.

[15] "MPlayer official website," 2010, http://www.mplayerhq.hu/.

[16] G. Candea, J. Cutler, and A. Fox, "Improving availability with recursive micro-reboots: A soft-state system case study," *Performance Evaluation*, vol. 56, no. 1-4, pp. 213–248, 2004.

[17] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[18] N. Kolettis and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*. Washington, DC, USA: IEEE Computer Society, 1995, p. 381.

[19] K. S. Trivedi and K. Vaidyanathan, "Software aging and rejuvenation," in *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc., 2008.

[20] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, J. Grundy and J. Penix, Eds. Montreal, Canada: IEEE Computer Society, 6 – 10 October 2003, pp. 30–39.

[21] E. Wong, T. Wei, Y. Qi, and L. Zhao, "A crosstab-based statistical method for effective fault localization," in *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST'08)*, R. Hierons and A. Mathur, Eds. Lillehammer, Norway: IEEE Computer Society, 9 – 11 April 2008, pp. 42–51.

[22] J. Herder, H. Bos, B. Gras, P. Homburg, and A. Tanenbaum, "Failure resilience for device drivers," in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Edinburgh, UK, 2007, pp. 41–50.

[23] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Colub, and M. Jones, "Mach: A system software kernel," in *Proceedings of the 34th Computer Society International Conference (COMPCON)*, San Francisco, CA, USA, 1989, pp. 176–178.

[24] G. C. Hunt et al., "Sealing OS processes to improve dependability and safety," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 341–354, 2007.

[25] "Erlang/OTP design principles," 2010, http://www.erlang.org/doc/.

[26] Object Management Group, "Fault tolerant CORBA," Object Management Group, Tech. Rep. OMG Document formal/2001-09-29, 2001.

[27] R. Buskens and O. Gonzalez, "Model-centric development of highly available software systems," in *Architecting Dependable Systems IV*, ser. Lecture Notes in Computer Science, R. de Lemos, C. Gacek, and A. Romanovsky, Eds. Springer-Verlag, 2007, vol. 4615, pp. 409–433.

[28] R. Abreu, A. González, P. Zoeteweij, and A. J. C. van Gemund, "Automatic software fault localization using generic program invariants," in *Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC'08)*. ACM Press, 2008, pp. 712–717.