AVA: Supporting Debugging With Failure Interpretations

Fabrizio Pastore and Leonardo Mariani
University of Milano Bicocca
Milan, Italy
{*pastore,mariani*}*@disco.unimib.it*

***Abstract*—Several debugging techniques can be used to automatically identify the code fragments or the runtime events likely responsible of a failure. These techniques are useful, but can help reducing the debugging effort only to a given extent. In fact, even when these techniques are successful, software developers still have to invest a lot of effort in understanding *if* and *why* something detected as suspicious is really wrong.**

In this paper we present the tool implementing the AVA technique. AVA, compared to other approaches dedicated to automatic debugging, in addition to automatically identifying the events likely responsible of a failure, generates an explanation about why these events have been considered suspicious. This explanation can be used by developers to quickly discard imprecise outputs and more effectively work on the relevant anomalies.

***Keywords*-debugging, failure interpretation, anomalies, log file**

I. Introduction

Debugging is a difficult and time consuming task that significantly affects the development cost of software projects. A number of techniques have been investigated to semi-automate and ease debugging. For instance, fault localization techniques exploit the differences between the program spectra of failing and successful executions to localize likely faulty statements [1], [2]. Anomaly detection techniques identify the suspicious events that have been produced in a failed execution to capture the rationale of the problem under investigation [3]–[5].

Regardless the debugging aid that is used, the output of these techniques still requires to be interpreted by software developers before a fault can be fixed. In particular, the developers must still understand if and why some selected statements should be considered faulty and if and why some selected events should be considered anomalous. Answering these questions is extremely time consuming and the effort necessary to produce these answers mitigates the benefit provided by automatic debugging techniques.

To simplify answering these critical why questions, debugging solutions should go in the direction of not only identifying the suspicious items (e.g., statements, events, etc.) but also decorating the identified items with information explaining why a selected element should be considered as suspicious. This description should be both easy to check, so that the developer can quickly discard irrelevant items, and easy to correlate with the failure under investigation, so that the time necessary to debug the problem can be significantly reduced.

In this paper we present AVA [6], a tool that implements a technique that can automatically identify the suspicious events produced by an execution and derive descriptions that can explain why an event should be considered suspicious. The set of suspicious events, together with the attached explanations, provides the rationale of a failure, which can be quickly validated, and then either confirmed or discarded by the developers.

The paper is organized as follow. Section II overviews the AVA technique. Section III describes the toolset implementing the technique. Section IV presents a running example. Section V summarizes the main empirical results that we have obtained by applying AVA on multiple faults. Section VI discusses related work and Section VII provides final remarks.

II. AVA: Automated Violation Analyzer

AVA is a technique that, given a Finite State Automaton (FSA) that describes the behavior expected for a software program and a trace obtained by monitoring the same program, identifies the anomalous events occurred in the trace, and produces interpretations that explain why the detected events should be considered anomalous [6].

The FSA could be part of the program specification or could be automatically inferred by monitoring test case execution, as it usually happens in anomaly detection techniques that compare passing and failing executions [3], [7].

AVA detects anomalous events using KLFA[1], which is a technique that can compare the events in a trace with an FSA, identify the sub-traces accepted by sub-automata in the FSA, and report the events that are not accepted by any sub-automaton as anomalous events [8]. In contrast with simply checking the trace using the model, the way KLFA behaves has the advantage of detecting multiple suspicious events in a single trace. In this way the noisy events that may occur at the beginning of the trace do not hinder the detection of the other anomalous events occurring successively in the trace.

KLFA implements other functionalities that are relevant in the scope of log-file analysis, such as the ability of inferring

[1]KLFA acronym stands for **K**Behavior **L**og **F**ile **A**nalysis, thus indicating that KLFA analysis is based on the KBehavior incremental inference engine.

FSAs from logs recorded during valid executions, the ability of handling parameters associated with events, and the ability of inferring the syntax of a log file. The interested reader can refer to [8] for more detailed information.

AVA produces an interpretation for each sequence of anomalous events identified with KLFA. Even if AVA is integrated with KLFA, it can be generally applied to create interpretations from anomalous events detected by comparing a trace with a FSA, regardless the use of KLFA. AVA works by comparing the events that occur in the trace in the neighborhood of the anomalous events, with the behavior expected by the automaton in replacement of the anomalous events. The behavior in the trace (which consists of a string) and the behaviors accepted by the FSA (which consist of multiple strings) are compared using a customized version of the global alignment algorithm, which is a string alignment algorithm [9]. The customization of the algorithm serves the purpose of classifying the differences between the actual and expected behaviors according to a set of known patterns.

The patterns supported by AVA are:

- **Delete:** which is the case of an event sequence that is expected to occur according to the FSA, but does not occur in the trace.
- **Insert:** which is the case of an event sequence that is not expected to occur according to the FSA, but occurs in the trace.
- **Replace:** which is the case of an event sequence that is expected to occur according to the FSA, but is replaced with a different event sequence in the trace.
- **Early termination:** which is the case of a trace that terminates earlier than expected.
- **Anticipation:** which is the case of an event sequence that occurs earlier than expected.
- **Postponement:** which is the case of an event sequence that occurs after than expected.
- **Swap:** which is the case of two event sequences that are replaced one with each other in the trace.

When the detected anomalous events are reported together with their interpretations it is quite simple to understand if and why a given event should be considered anomalous. For instance, if an anomalous event is decorated with the *delete* interpretation, it means that another event should have occurred before the anomalous event. An AVA interpretation is always annotated with contextual information, in the case of a *delete* interpretation the contextual information includes the name of the event that should have occurred before. The information reported in an interpretation is usually enough for developers to distinguish a true anomaly from a false alarm. In this example, if the anomalous event would not have been decorated with the *delete* interpretation the developers would have to autonomously understand that the event has been considered anomalous because something occurring before the event is missing.

AVA assigns a likelihood value, in the range $0 \ldots 1$, to each detected interpretation. The number represents how well a sequence of anomalous events fits a given interpretation. For instance, if a sequence of anomalous events consists of four new events that occur in a trace and one expected event missing from the trace, the *insertion of new events* is a possible interpretation for the detected anomaly, but since it does not perfectly fit the case, it would be assigned with a value smaller than 1.

The anomalies, together with the detected interpretations, are ranked according to likelihood values, and inspected in this order by developers.

III. Tool Support

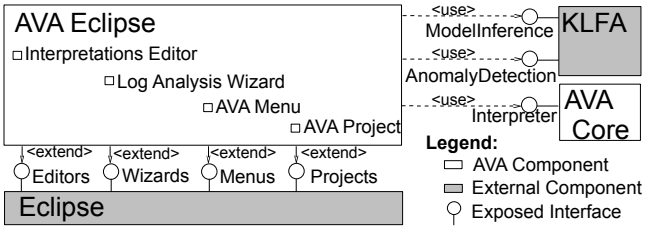

Figure 1. Architecture of the AVA tool.

Figure 1 shows the architecture of the AVA tool, which consists of two main components: the AVA-core library and the AVA-Eclipse Eclipse plug-in.

The *AVA-core* library implements the AVA technique and is meant to be used by researchers who want to integrate AVA in other tools. In particular, *AVA-core* implements an API that can be invoked from third-party programs to generate interpretations from anomalies. The API can be used to programmatically execute AVA and retrieve the results of the analysis, namely the interpretations.

The *AVA-Eclipse* plug-in is meant to be used by software engineers and system administrators to run daily debugging activities. *AVA-Eclipse* implements a GUI, which can be used to specify the log files that must be analyzed; multiple editors, which can be used to elaborate the AVA output; and multiple wizards, which assist developers with the configuration of the tool.

Our current implementation of the AVA technique is natively integrated with KLFA. KLFA is transparently executed on log files corresponding to legal executions to infer a FSA model that captures and generalizes the behavior in the log. KLFA is also used to compare the inferred FSA with a log file corresponding to a failed execution to automatically identify anomalous sequences of events. Thus, the typical input of our tool consists of a log file with multiple legal execution and a log file with a failed execution, and the output consists of a set of anomalous events with the corresponding interpretations.

Our tool can be easily used to analyze many different logging formats. When the format of the log file is unknown

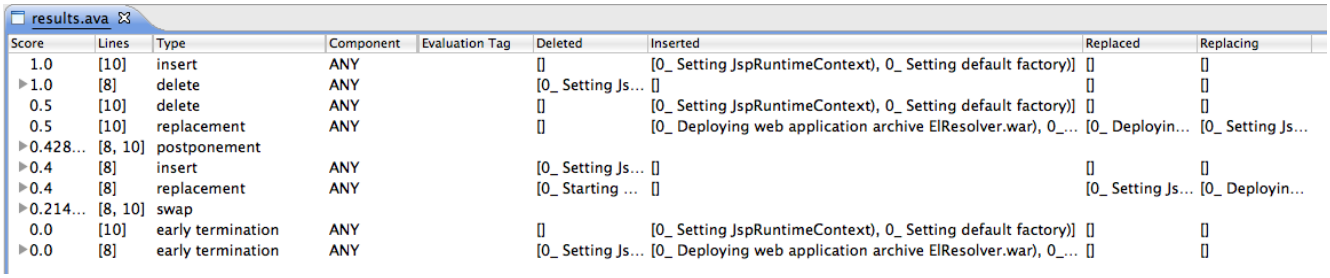

Figure 2. The AVA Anomaly Interpretations Editor.

or a specification is not available, users can rely upon the KLFA capability of automatically recognizing the structure of a log file. On the contrary, if the format is known and can be expressed using regular expressions, AVA can exploit such a specification to extract the relevant information from the entries in the log files.

AVA-Eclipse processes the anomalies detected with KLFA using the *AVA-core* API. The resulting interpretations are saved in the Eclipse workspace to be later inspected by software developers. *AVA-Eclipse* provides an editor that visualizes the interpretations produced by AVA.

Figure 2 shows the *Interpretations Editor*. For each interpretation, the *Interpretations Editor* shows: the score associated to the given interpretation, the range of lines in the log file that originated the anomaly corresponding to the interpretation, the name of the interpretation, the component affected by the anomaly (the name of the component is reported only if this information is included in the log file, otherwise `ANY` is shown in the column), a tag with the evaluation of the usefulness of the interpretation (assigned by the end-user), the list of inserted events (i.e., the unexpected events that occurred in the trace), deleted events (i.e., the expected events that did not occur in the trace), replaced events (i.e., the expected events that were replaced with others in the trace), replacing events (i.e., the events that occurred in the trace instead of some others). Depending on the interpretation only a subset of the columns inserted, deleted, replaced and replacing might be filled with some value.

For a same anomaly, AVA can generate multiple interpretations. This could happen because there could be multiple ways of explaining the reason of an anomalous event. These ways could consist of a same interpretation referring to different subsets of events. To keep the output manageable, when a same type of interpretation occurs multiple times for a same sequence of anomalous events, the Interpretations Editor shows only the one with the highest score, assuming that the others represent a less effective way of explaining the same case. Different interpretations are ordered according to their score.

For example, Figure 2 shows that for the anomaly occurring at line 10 (see column `lines`) all the four basic interpretations could apply (*insert*, *delete*, *replacement*, and *swap*), but the one with the highest score is *insert*.

Both *AVA-core* and *AVA-Eclipse* are available for download at the following URL: http://www.lta.disco.unimib.it/tools/ava/.

IV. Debugging with AVA

In this section we describe the typical flow of activities executed by developers when using AVA to debug programs. The basic flow, which is shown in Figure 3, consists of three sequential activities: collect logs, setup the analysis, inspect results. We present these activities referring to the debugging of a real fault reported in Tomcat version 6.04.

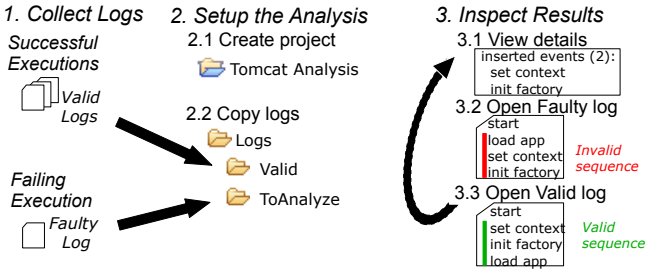

Figure 3. The AVA debugging methodology.

The fault in Tomcat version 6.04 prevents the ELResolver web application to be automatically started during the booting of the system. In particular, the web application correctly starts just after the deployment, but does not start anymore when Tomcat is succesively rebooted [10].

To debug a fault with AVA, software developers need to *collect logs* recorded during multiple passing executions and a failed execution. In this case, we collected correct logs from two executions: starting Tomcat when no web application has been deployed and starting Tomcat with few applications deployed. We collected the log with the failure by restarting Tomcat after having deployed the ELResolver web application.

Developers *setup the analysis* by using the AVA log analysis wizard to create a new log analysis project in the Eclipse workspace. The wizard creates a new project containing a configuration file, *analysis.alfa*, and the folders

logs/correct and *logs/toAnalyze*. Developers then simply copy the logs belonging to passing and failed executions in the appropriate folders (in the running example it is enough to copy the two correct logs and the log with the failure in the appropriate folders), and then start the AVA analysis through a contextual menu. AVA runs the analysis in the background and automatically opens the editor with the interpretations when the analysis finishes.

Figure 2 shows the editor with the results generated for the Tomcat case. The first two interpretations explain the failure cause. The first interpretation, an *insert*, indicates that the events `Setting JspRuntimeContext` and `Setting default factory` were not expected in line 10. The second interpretation, a *delete*, shows that in line 8 the same events (and few others) are missing, which clearly indicate that the default factory has been set too late.

Figure 4 shows an excerpt of the faulty log. The failure occurs because the default factory object is set in line 11, while the ELResolver application requests the factory before, as shown by log line 9. Tomcat should have set the factory in line 8 to run properly.

AVA users *inspect the results* downward from the top of the list. Each item of the list can be expanded using a detailed view, which shows the list of deleted, inserted, replaced and replacing events, and also the result of the alignment between the observed and expected events, with the mismatching events highlighted. Figure 5 shows the detailed view for the first result of the list. The highlighted lines show that the alignment of the expected and observed sequences lead to the identification of two unexpected events.

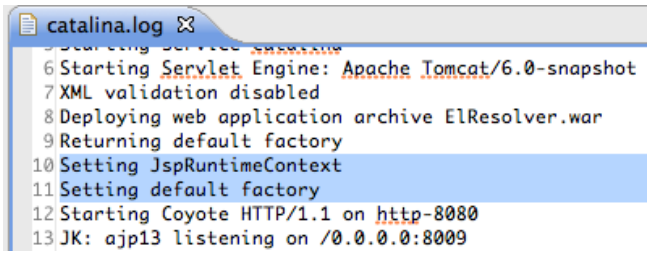

Figure 4. The faulty log with the first anomaly of Figure 2 highlighted.

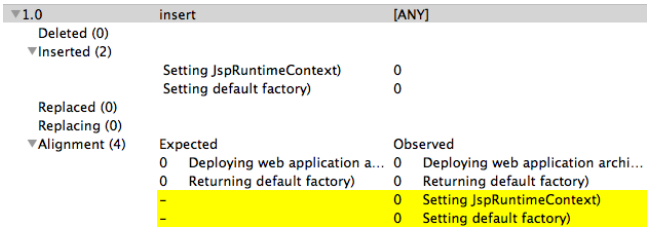

Figure 5. The detailed view for the first interpretation of Figure 2.

To properly understand the cause of a failure it is often necessary to inspect the place in the log where the anomaly has been detected, to better understand the failure context. To this end, the interpretations editor can be used to directly access the logs with the failed and the passing runs. If the log with the failed execution is opened, the editor highlights the lines that produced the anomaly that originated the selected interpretation (lines 10 and 11 in the case of the first interpretation in Figure 2, see Figure 4). If the log with passing executions is opened, the editor highlights an example of expected behavior detected in one of the passing runs which is close to the behavior that produced the anomaly in the failed execution. Figure 6 shows a log with passing executions and the expected behavior that is close to the second interpretation in Figure 2. In practice, the behavior in the passing log shows that the default factory must be set before deploying the *ElResolver*.

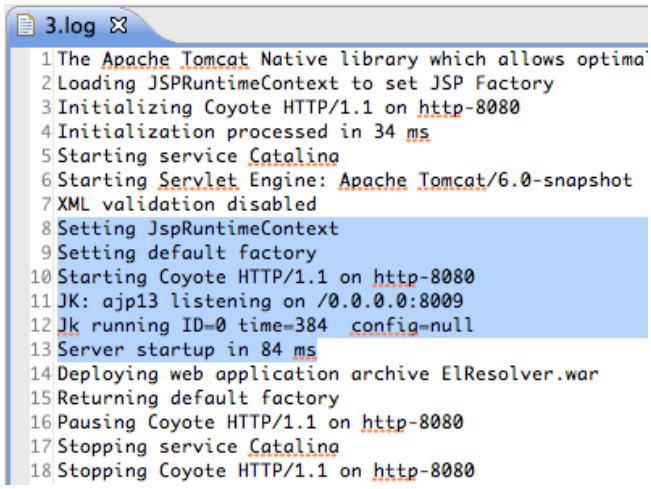

Figure 6. The passing log with the expected behavior highlighted.

To understand the cause of a failure developers may need to inspect multiple anomalies. To help end users in this process the editor supports tagging of anomalies (see Figure 7). Four tags are supported: *failure*, which indicates an interpretation that is a consequence of the observed failure; *failure cause*, which indicates an interpretation that points at the failure cause; *wrong*, which indicates a false alarm; and *maybe related*, which indicates an interpretation with an unclear role.

V. Empirical Results

So far, the AVA tool has been used to analyze faults experienced in enterprise applications [6]. Table I shows 5 case studies regarding Glassfish [11], which is a JavaEE Application Server (about 2 millions lines of code), and Tomcat (about 300.000 lines of code). The case studies consist of known faults and typical configuration issues affecting Glassfish and Tomcat. The logs processed by AVA in such an empirical experience are the logs natively produced by these systems.

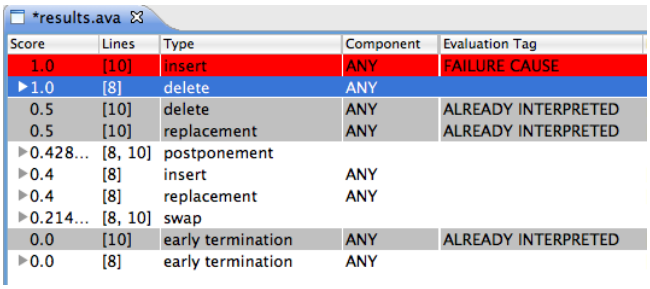

Figure 7. Anomalies editor with the anomalies tagged by the developers.

ID	Case study	Failure Cause
G1	GlassFish (v. 2-GA)	The Java Petstore cannot be correctly deployed because of a configuration error [12]
G2	GlassFish (v. 2-GA)	The Java Petstore cannot be correctly deployed because of a configuration error [13]
G3	GlassFish (v. 3-b01)	The server hangs because of a fault related to classloading [14]
T1	Tomcat (v. 6.0.4)	A web application cannot be started because of a fault in the classloader [10]
T2	Tomcat (v. 6.0.14)	Tomcat is not starting because the default port is already in use [15]

Table I
CASE STUDIES

AVA demonstrated to be useful in all the considered cases. Table II shows summary data about the results that we obtained. Column *ID* indicates the case study reported in Table I. Column *Most useful interpretation* shows the position in the ranking returned by AVA of the interpretation that better describes the problem under analysis. Assuming that the tester is able to recognize the relevance of this interpretation this number is usually also the overall number of interpretations that must be evaluated by the tester before discovering the failure cause. Column *First true positive* indicates the position of the first interpretation related with the fault, such as the logging of an exception caused by the fault. Column *False positives* shows the number of interpretations not related with the fault returned by AVA.

ID	Most useful interpretation	First true positive	False positives
G1	5	1	2
G2	16	1	4
G3	5	5	4
T1	1	1	0
T2	1	1	0

Table II
RESULTS WITH REAL FAULTS

Results show that AVA frequently returns true positives on the highest positions of the list and that the number of interpretations that must be inspected before discovering a fault is usually small (five at most in four out of five case studies). On the other hand the number of false positives returned by AVA in this set of experiments has been limited: four in the worst case. These results suggest that AVA could be a valid debugging support, returning a focused list of candidate failure causes (the anomalous events) decorated with interpretations that ease their analysis.

VI. Related Work

In this section we compare AVA with debugging tools and techniques based on statistical approaches, model based approaches, and self-repairing approaches.

A. Statistical Approaches

Several approaches use statistical indexes to identify the instructions that are better correlated with failed executions. Two notable examples are Tarantula [1] and Crosstab [16], which use statement coverage, and Sober [2], which uses conditions coverage. The output of these techniques is a list of statements ranked according to the likelihood of being faulty. These approaches are useful, but can be applied only when the execution space is nicely covered (e.g., the exception handling code not covered by passing executions might mask faulty lines) and they identify suspicious lines of codes without providing any additional information about why these lines should be considered as suspicious; and this is a known limitation [17]. On the contrary AVA provides not only an indication of the suspicious events that could be responsible of a failure, but also a description (i.e., the interpretation) that facilitates understanding why an event has been selected as suspicious.

B. Model Based Approaches

Model based approaches derive models that generalize the characteristics of data recorded in passing executions, and use these models to identify anomalous events occurred in failing executions. Daikon [18] and DIDUCE [5] are examples of engines that can derive likely program invariants from data recorded during passing executions. These invariants can be checked in failed executions to identify anomalous data values.

Other approaches focus instead on the identification of anomalous event sequences. Both ADABU [19] and GK-Tail [20] for example derive FSAs that capture legal method invocation sequences, and use these FSAs to identify anomalous event sequences.

A few techniques address debugging combining multiple types of models. For instance, BCT [3] and RADAR [7] use both models on data values and operation sequences. Probabilistic Program Dependency Graph combines bayesian probabilities with structural information to capture anomalies in the flow of the data [21].

Model based approaches identify the data and the events that differ from the common behavior of the system under analysis and the software developers are asked to understand the failure causes from these misbehaviors. However, these

techniques, contrarily to AVA, do not assist software developers with additional information about why each anomalous event should be considered as a misbehavior.

C. Self-Repairing approaches

Self-Repairing approaches automatically suggest fixes that can repair programs. For example, PACHIKA can automatically add the method calls necessary to satisfy a precondition that has been violated by a given method [22]. Other approaches iteratively alter code fragments using genetic algorithms until a fix is synthesized [23].

Self-repairing approaches are quite close to AVA. In fact a synthesized fix could be considered as an explanation of the failure, indeed a fix includes both the identification of the problem and its solution. However, self-repairing approaches and AVA work at different levels. The former has been demonstrated to be useful to address small unit faults. The latter has been applied to system logs and used to address faults derived from the interactions of multiple components of a system.

VII. Conclusion

In this paper we presented the tool that implements AVA, a technique that can automatically identify the suspicious events produced by a software execution and automatically generate the descriptions that can explain why these events should be considered suspicious.

AVA is available for download at http://www.lta.disco.unimib.it/tools/ava/. Our AVA implementation includes both an API, which can be invoked from third party tools, and an Eclipse plug-in, which can be used by developers to analyze failures traced in log files.

References

[1] J. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 2002, pp. 467–477.

[2] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, ""SOBER: statistical model-based bug localization"," in *Proceedings of the European Software Engineering Conference and Foundations on Software Engineering (ESEC/FSE)*. ACM, 2005, pp. 286–295.

[3] L. Mariani, F. Pastore, and M. Pezzè, "Dynamic analysis for diagnosing integration faults," *IEEE Transactions on Software Engineering (TSE)*, vol. 37, no. 4, pp. 486–508, 2011.

[4] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of the European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*. ACM, 2007, pp. 35–44.

[5] S. Hangal and M. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 2002, pp. 291–301.

[6] A. Babenko, L. Mariani, and F. Pastore, "AVA: Automated interpretation of dynamically detected anomalies," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2009, pp. 237–248.

[7] F. Pastore, L. Mariani, A. Goffi, M. Oriol, and M. Wahler, "Dynamic analysis of upgrades in c/c++ software," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2012, pp. 91–100.

[8] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2008, pp. 117 – 126.

[9] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[10] Tomcat bug database, tomcat fault 40820. https://issues.apache.org/bugzilla/showbug.cgi?id=40820, visited in 2012.

[11] Glassfish Application Server, http://glassfish.java.net/, visited in 2012.

[12] Glassfish user forum, glassfish configuration issue. http://forums.java.net/jive/thread.jspa?messageID=252898, visited in 2012.

[13] Glassfish user, glassfish configuration issue. http://forum.java.sun.com/thread.jspa?threadID=5249570, visited in 2012.

[14] Glassfish bug database, glassfish issue 4255. https://glassfish.dev.java.net/issues/showbug.cgi?id=4255, visited in 2012.

[15] Tomcat bug database, tomcat configuration issue. http://www.blogjava.net/haix/archive/2008/01/16/175592.html, visited in 2012.

[16] W. E. Wong, V. Debroy, and D. Xu, "Towards better fault localization: A crosstab-based statistical approach," *IEEE Transactions On Systems, Man, and Cybernetics*, 2010.

[17] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2011, pp. 199–209.

[18] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[19] V. Dallmeier, C. Linding, A. Wasylkowski, and A. Zeller, "Mining object behaviour with adabu," in *Proceedings of the International Workshop on Dynamic software Analysis (WODA)*. ACM, 2006, pp. 17–24.

[20] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2008, pp. 501– 510.

[21] G. Baah, A. Podgurski, and M. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 528 –545, July-August 2010.

[22] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2009, pp. 550–554.

[23] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the Annual conference on Genetic and evolutionary computation (GECCO)*. ACM, 2009, pp. 947–954.