AVA: Automated Interpretation of Dynamically Detected Anomalies

Anton Babenko
University of Milano Bicocca
viale Sarca, 336
20126 - Milano, Italy
a.babenko@campus.unimib.it

Leonardo Mariani
University of Milano Bicocca
viale Sarca, 336
20126 - Milano, Italy
mariani@disco.unimib.it

Fabrizio Pastore
University of Milano Bicocca
viale Sarca, 336
20126 - Milano, Italy
pastore@disco.unimib.it

ABSTRACT

Dynamic analysis techniques have been extensively adopted to discover causes of observed failures. In particular, anomaly detection techniques can infer behavioral models from observed legal executions and compare failing executions with the inferred models to automatically identify the likely anomalous events that caused observed failures.

Unfortunately the output of these techniques is limited to a set of independent suspicious anomalous events that does not capture the structure and the rationale of the differences between the correct and the failing executions. Thus, testers spend a relevant amount of time and effort to investigate executions and interpret these differences, reducing effectiveness of anomaly detection techniques.

In this paper, we present Automata Violations Analyzer (AVA), a technique to automatically produce candidate interpretations of detected failures from anomalies identified by anomaly detection techniques. Interpretations capture the rationale of the differences between legal and failing executions with user understandable patterns that simplify identification of failure causes. The empirical validation with synthetic cases and third-party systems shows that AVA produces useful interpretations.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Tracing, Monitors*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*

General Terms

Verification

Keywords

Anomaly detection, Dynamic Analysis

ISSTA'09, July 19–23, 2009, Chicago, Illinois, USA.

1. INTRODUCTION

Anomaly detection techniques can support testers by automatically providing information about the unexpected events that have been observed in failing executions [13, 19, 11, 14, 27, 20, 26]. Focusing failure investigation on the information provided by these techniques often facilitates the understanding of the possible failure causes and simplify the localization of faults.

Anomaly detection techniques usually work in two steps. In the first step, they generate a model of the legal behavior of a target system by tracing successful executions at testing-time, and then synthesizing models from the collected traces. In the second step, they compare the set of events detected during failing executions with the generated models to identify the anomalous events that will be presented to testers.

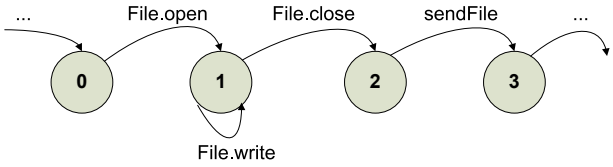

Figure 1: A sample FSA used for anomaly detection.

Even if data about anomalous events can be useful, in many cases investigating and interpreting such data can be extremely time consuming. Consider the case of a technique that derives the Finite State Automaton (FSA) shown in Figure 1 to detect anomalies. The FSA specifies that the target system usually generates the following sequence of events during correct executions: a file is opened, data are written in the file, the file is closed and finally sent through the network. In case a fault causes the file to not being closed, the anomaly detection technique would compare the sequence of events `File.open`, `File.write`, ..., `File.write`, `sendFile`, observed in the failing execution, with the model. The result would be that the event `sendFile` is not accepted by the model, and `sendFile` would be presented to testers as anomalous event. Testers may erroneously think that something is wrong in the functionality for sending files, and they would start their investigation from components and libraries for managing file transfer. Once this investigation failed, testers would probably consider other possibilities and identify the right cause of the failure.

This simple investigation may require several minutes or even hours to be completed. In case many anomalous events or complex re-arrangement of events are detected in failing

executions, e.g., interleaved events, postponed events, etc., the effort required to interpret and investigate anomalous events can be extremely high, with a dramatic loss of cost-effectiveness of the technology.

In this paper, we present a technique, called Automata Violation Analyzer (AVA), that automatically analyzes anomalous events identified by dynamic analysis techniques that infer FSA. The technique produces multiple interpretations of the detected anomalies, prioritizes them according to likelihood to explain differences between correct and failing executions, and presents the final result to testers. We focus on FSA because they are commonly used to represent component behaviors and interaction protocols, can be inferred by many algorithms [3, 1, 14], and are used by several anomaly detection techniques [1, 13, 20, 26, 14].

We integrated AVA with Log File Analysis (LFA), a technique developed for automated analysis of log files [13]. We experienced the integrated solution with both synthetic cases and third-party systems. Synthetic cases are used to simulate the different kinds of anomalies that can be experienced in software systems. Third-party systems are used to validate our solution with large applications. Empirical results show that AVA correctly presents a suitable interpretation of failure causes within the fifth interpretation of the ranking.

The paper is organized as follow. Section 2 summarizes the main steps of the AVA technique. Section 3 describes LFA. Section 4 presents strategies to identify basic interpretations from detected anomalies. Section 5 presents strategies to identify composite interpretations from detected anomalies. Section 6 describes the prioritization criterion used to order interpretations. Section 7 presents empirical results obtained with synthetic cases and third-party systems. Section 8 discusses related work. Finally, Section 9 concludes the work.

2. AVA

AVA can automatically provide an effective interpretation to a set of anomalies that has been detected by an anomaly detection technique that uses inferred FSAs to represent the expected behavior of software systems. Thus, the complete solution is the result of the integration of two main techniques: an anomaly detection technique and AVA.

The anomaly detection technique is responsible for monitoring a target system, deriving a model of the expected behavior, and identifying anomalous events in failing executions. AVA is responsible for interpreting data, identifying candidate explanations to observed problems and prioritizing the results.

In the next section, we shortly describe LFA, the technique for anomaly detection that we integrated in our solution [13]. Here we summarize the three main phases of AVA: identify basic interpretations, identify composite interpretations, and prioritize interpretations.

Identify basic interpretations consists of analyzing both anomalies and inferred models used to identify such anomalies to generate a more relevant, even if simple, interpretation about the unexpected events. For instance, the anomalous `sendFile` event identified with the model in Figure 1 generated by a failing execution that did not close the file before sending it through the network can be more relevantly explained as a missing `File.close` event.

Identify composite interpretations consists of analyzing and combining the available set of simple interpretations to identify richer interpretations of failing executions. For instance, if both a missing event and the addition in the future of the same missing event are identified as simple interpretations, a postponed event represents a more precise interpretations of the failing execution.

Prioritize interpretations consists of ordering basic and composite interpretations according to their likelihood to effectively explain differences between correct and failing executions. Ordering interpretations is useful to start the investigation of failure causes from good interpretations, and avoid the investigation of interpretations with little relevance.

3. LOG FILE ANALYSIS

LFA is a dynamic analysis technique that automatically identifies anomalous events in log files recorded during failing executions [13]. LFA is based on three main phases: monitoring, model generation and failure analysis.

Monitoring consists of collecting traces of correct executions from target systems. A trace is a sequence of events annotated with attribute values. Monitoring takes typically place at testing time, when it is simple to distinguish failing and correct executions.

Model generation is the most complex phase of the technique and consists of generating a FSA that summarizes and generalizes all the collected traces. LFA initially processes trace files to automatically identify event names and attribute values. This operation is executed because LFA does not require the complete knowledge of the log format in advance, but heuristically infers the format. It is sufficient that event names and their attribute values are recorded in a text file separated by a known character. In other words it is only known when an event entry (event name followed by attribute values) begins and ends. Thank to this feature, it is extremely simple to adapt LFA to different log formats.

Since attribute values often capture information that is specific to a given execution, but with little relevance when compared with other executions, such as the case of process identifiers, LFA pre-processes attribute values to extract more general and reusable information than concrete attribute values. In particular, LFA eliminates attribute values and modifies event names to incorporate information about the regularity of the distribution of those values across events. The rewriting of the event names is based on the identification of recurrent definition-use patterns across attribute values. For instance, if an analyzed log file includes a trace like

`start p1210`, `start p1084`, `stop p1210`, `stop p1084`

where `start` and `stop` are event names and `p1210` and `p1084` are attribute values, LFA rewrites this trace as

`start_A`, `start_B`, `stop_A`, `stop_B`

where only event names occur[1]. The rewritten trace has the benefit to include information about the expected attribute values, i.e., processes are started and stopped in the same order, even if it only includes event names without attribute values. Thus, the rewritten traces can be used to infer mod-

[1]this example presents the simplest rewriting strategy available in LFA.

els capable to reveal failures that depend on attribute values. Let us consider the sequence of events

`start p4567`, `start p1234`, `stop p1234`, `stop p4567`

which starts and stops processes with a different order than the previous trace. Let us also suppose that a failure occurs when this example sequence is observed. According to the previous strategy, the sequence would be rewritten as `start_A`, `start_B`, `stop_B`, `stop_A`. If we compare the two rewritten sequences, we discover that the first two events match (`start_A` and `start_B`), while the third and the fourth differ (`stop_B` and `stop_A`). This difference successfully indicates that processes are terminated with an unexpected order. Simply ignoring attribute values and only matching event names would not have revealed such difference.

LFA includes mechanisms to automatically and heuristically identify likely related attributes, and independently rewrite different groups of homogeneous attributes. For instance, attribute values that indicate IP addresses and values that indicate meters would be rewritten and verified independently. Thus avoiding comparisons of unrelated attributes.

LFA can derive models at different levels of granularity. For instance, it is possible to derive a unique model that specifies the behavior of the whole system, or to derive one model for each component that is monitored, to better focus on the events generated by each unit. More details about these mechanisms, further technicalities to handle multiple attribute values associated with events, and to increase robustness with respect to noise and incidental values can be found in [13]. For the purpose of the work presented in this paper, it is sufficient to know that LFA considers attribute values in the analysis even if it infers plain FSA.

The model generation phase is completed by using kBehavior to infer the final FSA that summarizes and generalizes the rewritten traces provided as input [14].

When a failure is investigated, LFA reads the trace recorded during the failing execution and uses this trace to extend the model that represents correct executions. Several techniques simply compare a trace with a model to indicate the first event that is not accepted by the model. This strategy hinders visibility of many anomalies that may be located after the first one. Thus a little noise can heavily reduce effectiveness of the technique.

To overcome this issue, LFA does not check whether the model accepts the trace or not, but extends the model considering the trace as an extra input (kBehavior provides the capability to incrementally extend FSA by processing additional traces). The extended model will generate both the language generated by the original model and the trace that corresponds to the failing execution. All the changes introduced during the extension are interpreted by LFA as unexpected event sequences that occurred in the failing execution.

According to experiments reported in [13], LFA can identify a small set of anomalous events (between 0.01% and 28.57% of the total number of recorded events) with an average precision of 0.67 when suitable configurations are selected by testers. Even if a precision of 0.67 indicates that a relevant percentage of the anomalies presented to testers are in effect related to failures, interpreting these data can still be hard and time consuming.

4. BASIC INTERPRETATIONS

The first operation executed by AVA consists of automatically identifying basic interpretations, i.e., interpretations of the anomalies detected in a failing execution specified according to a catalog of user-understandable basic anomaly patterns. Formally, an interpretation is a triple <*anomaly pattern*, *confidence value*, *extra info*>, where *anomaly pattern* is the name of an anomaly pattern; *confidence value* is a confidence value in the range 0 to 1 that specifies how much the anomaly pattern well describes anomalies observed in the failing execution; and *extra info* specifies additional information that easy the understanding of the interpretation. Basic interpretations are derived by locally analyzing single anomalies (the way we implemented locality is presented in Section 4.1).

An example basic interpretation is <*delete*, 1, "*event* `File.close` *generated from state* 2 *has been skipped in the failing execution (closest expected sequence File.open File.write File.close sendFile; observed sequence File.open File.write sendFile)*">. The first item *delete* specifies that the interpretation refers to the *delete* anomaly pattern. The confidence value 1 indicates perfect confidence on the interpretation, thus the only difference between the expected behavior and the observed behavior consists of deleted events. The last element of the triple specifies that only the event *File.close* has been deleted, and presents both the observed event sequence and the expected correct sequence that is closest to the observed one. Section 4.2 describes how we compute confidence values.

The identification of basic interpretations is based on two steps: (1) define scope and expected behaviors, and (2) align and score basic anomaly patters. Define scope and identify expected behaviors consists of identifying the legal behaviors accepted by the inferred FSA that are close to the anomaly under analysis. Align and score basic patterns consists of executing a customized string alignment algorithm to compare possible behaviors with the failing execution and measure how well each basic anomaly patterns describes the observed anomaly.

4.1 Define scope and expected behaviors

AVA analyzes each anomaly by comparing the expected event sequences and the observed event sequence. To reduce the size of the problem and concentrate the analysis on the target anomaly, the comparison is restricted to the expected and observed events that are close to the place where the anomaly has been detected. The exact scope of the analysis depends on the kind of analyzed anomaly. Since LFA represents anomalies as extensions of FSAs, the kind of anomalies to be analyzed can be classified according to the type of extension introduced by LFA. In particular, LFA can extend a model in three possible ways: adding a branch, adding a tail, and adding a final state.

Branch extension

A branch extension consists of extending a FSA with the addition of a new branch, as shown in Figure 2. A new branch has an *extension start point* and an *extension end point*. The extension start point is the state of the FSA augmented with a new outgoing transition. The extension end point is the state of the FSA augmented with a new incoming transition.

We can have two kinds of branch extensions: branches

pointing to the future and branches pointing to the past. We have a branch pointing to the future when the extension end point is reachable from the extension start point. This extension indicates that a new behavior has been observed instead of an expected behavior. We have a branch pointing to the past when the extension end point is not reachable from the extension start point in the original FSA. This extension indicates that an expected behavior has been unexpectedly observed multiple times.

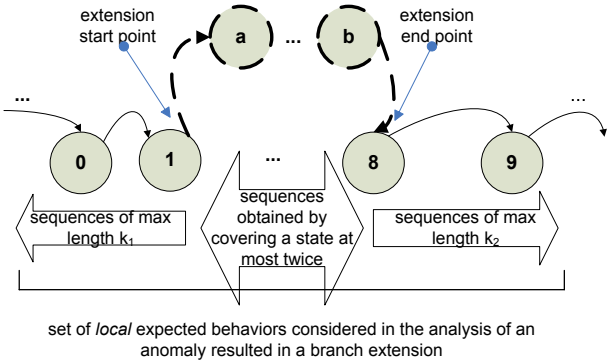

Figure 2: A branch extension pointing to the future

We define the scope of the local analysis depending on the kind of branch that has been added. In the case of a *branch pointing to the future*, we consider behaviors immediately preceding the anomaly, i.e., immediately preceding the extension start point, alternative to the anomaly, i.e., between the extension start and end points, and immediately following the anomaly, i.e., immediately following the extension end point. In particular, we consider the event sequences that can be obtained by concatenating the sequences of maximum length k_1 that can be generated before the extension start point, the sequences of events that can be generated between the extension start point and the extension end point by traversing each state at most twice, and the sequences of events of maximum length k_2 that can be generated after the extension end point. Parameters k_1 and k_2 are integer values defined by testers to tune the scope of the local analysis. We extend the boundaries of the local analysis outside extension start and end points because some interpretations can depend on events located before or after the extension points, e.g., postponed events and added events. Figure 2 visually shows the set of events that are considered in the local analysis of a branch pointing to the future.

The subsequence of the events observed in the failing execution to be compared with the expected behaviors is selected with an analogous strategy, i.e., we consider the k_1 events before the first event in the branch pointing to the future, the events in the branch pointing to the future, and the k_2 events after the last event in the branch pointing to the future. Figure 3 shows the expected and observed sequences that are considered for the analysis with an example FSA and $k_1 = k_2 = 1$.

In the case of a *branch pointing to the past*, we consider behaviors immediately preceding the anomaly, i.e., immediately preceding the extension start point, and immediately following the anomaly, i.e., immediately following the extension start point. In particular, we consider the event sequences that can be obtained by concatenating the sequences of maximum length k_1 that can be generated before the extension start point and the sequences of events of maximum length $k_2 + l$ that can be generated after the extension start point, where l is the length of the branch pointing to the past. We take into account the length l of the branch generated by the anomaly to consider expected sequences with the same length than the subsequence considered for the failing execution. Figure 4 visually shows the set of events that are considered in the local analysis of a branch pointing to the past.

We adopt an analogous strategy to select the subsequence of the events observed in the failing execution to be compared with the expected behaviors. In this case we consider the k_1 events before the first event in the branch pointing to the past, the l events in the branch and the k_2 events after the last event in the branch. Figure 5 shows the expected and observed sequences that are considered for the analysis with an example FSA and $k_1 = k_2 = 2$.

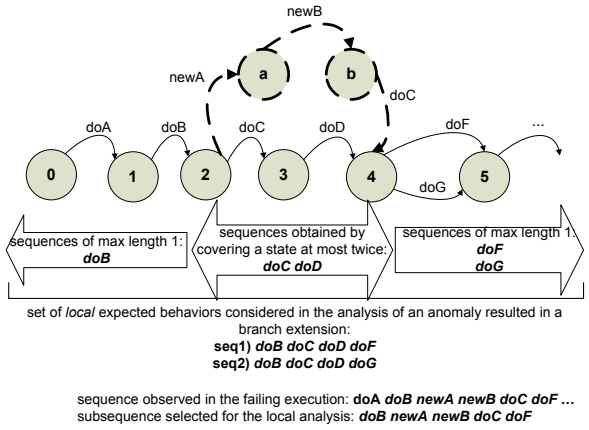

Figure 3: An example set of behaviors considered in the analysis of a branch extension pointing to the future

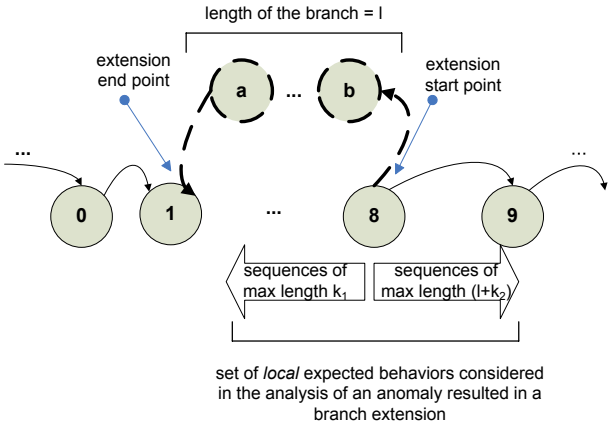

Figure 4: A branch extension pointing to the past

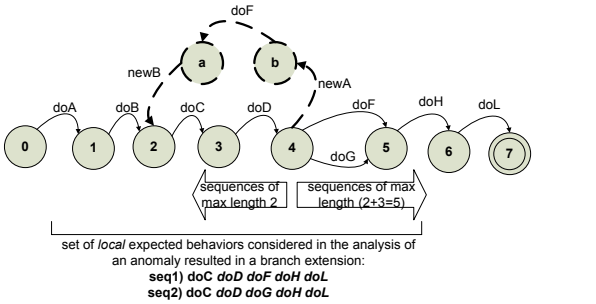

Figure 5: An example set of behaviors considered in the analysis of a branch extension pointing to the past

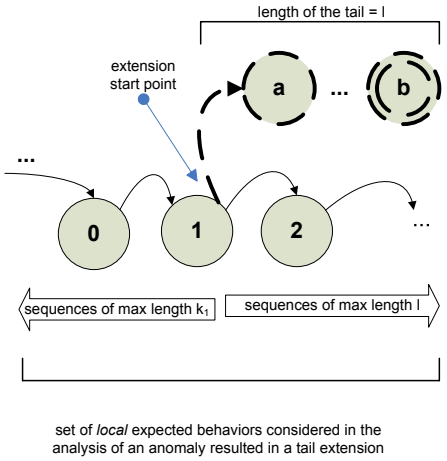

Figure 6: Tail extension

Tail extension

A tail extension consists of extending a FSA with the addition of a new tail, as shown in Figure 6. A new tail has an *extension start point* and a final state at the end of the tail. The extension start point is the state of the FSA which is augmented with a new outgoing transition.

We define the scope of the analysis as the set of behaviors immediately preceding the anomaly, i.e., immediately preceding the extension start point, and immediately following the anomaly, i.e., immediately following the extension start point. In particular, we consider the event sequences that can be obtained by concatenating the sequences of maximum length k_1 that can be generated before the extension start point with the sequences of events of maximum length l that can be generated after the extension start point, where l is the length of the tail (this choice guarantees the analysis of expected and failing sequences of the same length). Figure 6 visually shows the set of events that are considered in the local analysis of a tail.

The subsequence of the observed behavior to be compared with expected behaviors is selected with an analogous strategy, i.e., we consider the k_1 events before the first event in the tail and the l events in the tail. Figure 7 shows the expected and observed sequences that are considered for the analysis with an example FSA and $k_1 = 2$.

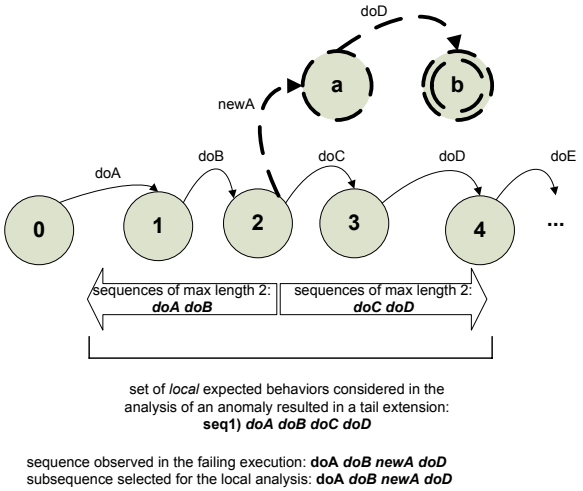

Figure 7: An example set of behaviors considered in the analysis of a tail extension

Final state extension

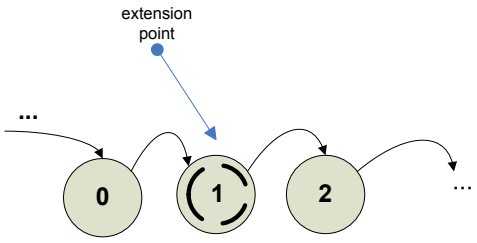

Figure 8: Final state extension

A final state extension consists of extending a FSA by replacing a regular state with a final state, as shown in Figure 8. A final state extension has an *extension point* which is the replaced state.

This kind of extension is a special case because there are no unexpected events: the execution simply terminated at an anticipated time. In this case, local analysis is not needed because the anticipated interruption of the execution is identified with perfect confidence.

4.2 Align and score basic patterns

In the previous step, we selected the expected behaviors to be considered for local analysis. In this step, we compare each expected behavior with the sequence observed in the failing execution to interpret differences. These differences will be presented to testers as likely failure causes.

To evaluate the relevance of a given interpretation, we compute its confidence values as the distance between each pair <*expected behavior*, *anomalous behavior*>. A distance

is a real value in the range 0..1 that indicates how well a given interpretation fits the case under analysis.

Traces collected during failing executions often include noise, i.e., events that are not accepted by the model and are not directly related to the investigated failure. Since LFA extends a FSA with a new trace by matching subsequences with sub-automata, and adding the extra states and transitions necessary to accept the whole trace, presence of noisy events can modify the sequences in the added branches or even cause the addition of new branches. To be applicable to large and complex cases, the strategy to recognize anomaly patterns must also work when extensions include noise.

To this end, we use a string alignment algorithm that compares the local expected behavior with the local anomalous behavior and finds a proper fitting of events, despite presence of noisy data. We run our string alignment algorithm with different configurations corresponding to the kind of basic interpretation that is investigated. Each possible alignment of the two analyzed event sequences is associated with a score that indicates how much the two strings are close according to the selected configuration. For example, an observed and an expected sequence that only differs for deleted events will be extremely close according to the delete interpretation.

The string alignment algorithm that we use is a modified version of the Needleman-Wunsch global alignment algorithm [17]. Our modified version can be configured to evaluate the differences between two aligned strings according to different weights. Depending on the alignment, we can have four results when comparing two aligned symbols: match, mismatch, deleteGap, and addGap. Figure 9 shows these cases.

We have a match when there is the same symbol at the same position. We have a mismatch when there are two different symbols at the same position. We have a deleteGap when there is a symbol in the expected sequence and a gap in the observed sequence. We have an addGap when there is a gap in the expected sequence and a symbol in the observed sequence. Gaps are automatically introduced by the alignment algorithm to find the best correspondence between the two strings under analysis. We modified the alignment algorithm to support a different evaluation for the deleteGap and addGap. The original version of the algorithm cannot distinguish between these gaps. We modified the algorithm because gaps have not the same semantics in our domain, e.g., a deleteGap is a strongly favorable indication of deleted events, while an addGap is against this same interpretation.

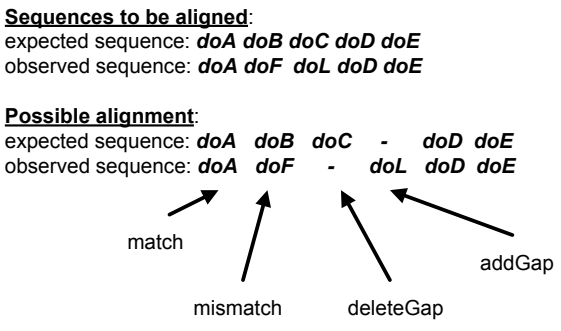

Figure 9: Example alignment.

The final score assigned to each possible alignment is computed by summing the weights associated with each symbol in the aligned sequence and then normalizing in the range 0..1 by dividing for the number of places (a place can be a symbol or a gap) in the aligned sequences. This score indicates the confidence value of the interpretation.

We look for basic interpretations by defining configurations, i.e., sets of weights assigned to the possible result of symbol comparison, that positively evaluate matchings and differences that are consistent with the investigated interpretation, and negatively evaluate differences that are not consistent with the investigated interpretation. We defined four basic interpretations: delete, insert, replace and final state. In the following, we present the configuration of each basic interpretation, and we show with an example how each configuration is evaluated.

It is worth to mention that interpretations are decorated with extra information that specifies the expected sequence that is closest to the anomalous sequence and the start state of the anomalous behavior.

Delete

A delete interpretation is associated with the following weights:

$match = +1$
$mismatch = -1$
$deleteGap = +1$
$addGap = -1$

Since we are analyzing an anomaly, it is not possible that all events match, thus the maximum evaluation is obtained when we have matches and some deleted events. Eventual noise represented by the presence of *addGap* and *mismatch* decreases the value of the interpretation. If we consider the `sendFile` example presented in the Introduction, the best alignment according to the configuration of a delete interpretation is shown in Figure 10. The example shows the case of a pure deleted event and the normalized value of the interpretation is in effect $\frac{4}{4} = 1$.

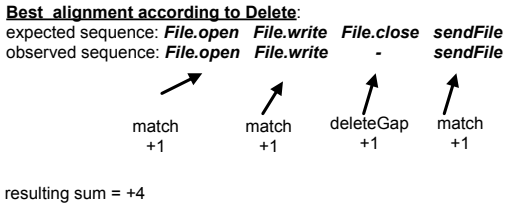

Figure 10: Example of a delete interpretation.

Insert

An insert interpretation is associated with the following weights:

$match = +1$
$mismatch = -1$
$deleteGap = -1$
$addGap = +1$

Since we are analyzing an anomaly, it is not possible that all events match, thus the maximum evaluation is obtained when we have matches and some added events. Eventual noise represented by the presence of *deleteGap* and *mismatch* decreases the value of the interpretation. Figure 11 shows an example interpretation of an insert. The example shows the case of a execution with mostly, but not purely, added events. In fact, the normalized value of the interpretation is $\frac{5}{7} = 0.71$.

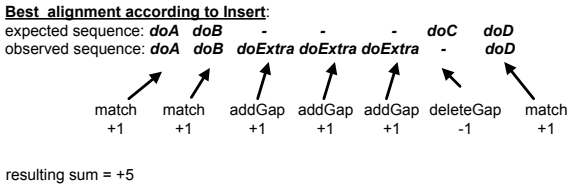

Figure 11: Example of an insert interpretation.

Replace

A replace interpretation is associated with the following weights:

$match = +1$
$mismatch = +1$
$deleteGap = -1$
$addGap = -1$

Since we are analyzing an anomaly, it is not possible that all events match, thus the maximum evaluation is obtained when we have matches and some mismatching events. Eventual noise represented by the presence of *deleteGap* and *addGap* decreases the value of the interpretation. Figure 12 shows an example interpretation of a replace. The example shows the case of an execution with mostly, but not purely, replaced events. In fact, the normalized value of the interpretation is $\frac{4}{6} = 0.67$.

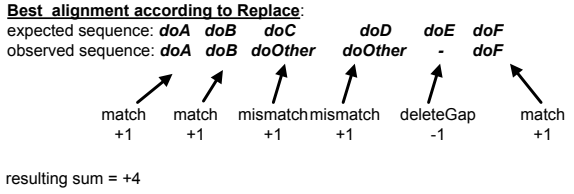

Figure 12: Example of a replace interpretation.

Final state

A final state interpretation does not require fine grained analysis to be discovered. In fact, every time a final state extension is detected, a final state interpretation with perfect confidence is generated. This interpretation indicates that the application terminated earlier than expected.

Since with high probability a failing execution terminates differently than a regular execution, AVA frequently generates this interpretation. However, it exists at most one occurrence of this interpretation per analyzed failure. Thus, this interpretation does not change readability of the output, while providing the useful information about the point where the application terminated its execution.

5. COMPOSITE INTERPRETATIONS

Basic interpretations can be useful in many cases, however there exist further interesting interpretations that can be discovered by combining basic interpretations. We define a composite interpretation as an interpretation that is function of basic or composite interpretations. In the context of this work, we defined three composite interpretations: anticipation, posticipation and swap. In the following, we describe composite interpretations and show a few examples.

Anticipation

An anticipation is the case of an event that occurs earlier than expected. We analyze anomalies to discover if an anticipation occurred in all the cases where we detect two basic interpretations I, D, where I is an insert or replacement interpretation, D is a delete or replacement interpretation, and I occurs before D in the trace recorded during the failing execution.

To discover if an anticipation occurred, we define $I.newEvents$ as the sequence of all the events in the observed sequence that are classified as added or mismatched, and $D.removedEvents$ as the sequence that includes all the events in the expected sequence that are classified as deleted or mismatched. To discover if there are common symbols with same order between these two sequences (thus suggesting an anticipation), we align $I.newEvents$ with $D.removedEvents$ by using the following configuration:

$match = +1$
$mismatch = 0$
$deleteGap = 0$
$addGap = 0$

The sequence *alig* of the symbols that match between $I.newEvents$ and $D.removedEvents$ after string alignment represents the events that have been anticipated. The sequence *notAlig*, which consists of the events that do not match after string alignment, includes events that are only added in the first interpretation, only deleted in the second interpretation or appear with the wrong order. If the number of aligned events is 0, we cannot have an anticipation.

If we have at least 1 aligned event, we have a behavior that can be interpret as an anticipation, even if it may be associated with a low confidence value. We compute the confidence value of the anticipation by positively counting the events that are anticipated and the ones that match, and negatively counting the differences that do not contribute to the anticipation. In particular, if the interpretation I has $I.m$ matches, $I.n$ mismatches, $I.a$ addGaps and $I.d$ deleteGaps, and the interpretation D has $D.m$ matches, $D.n$ mismatches, $D.a$ addGaps and $D.d$ deleteGaps, and the sequence *notAlig* is further split into *notAlig.g* gaps and *notAlig.n* mismatches, the confidence value of the interpretation is computed as:

$$\frac{2*\sharp alig - 2*\sharp notAlig.n - \sharp notAlig.g + \sharp I.m + \sharp D.m - \sharp I.d - \sharp D.a}{length(I)+length(D)}$$

where $\sharp$ indicates the number of events in the sequence that follows.

Note that $I.m$, $I.a$, $D.m$ and $D.d$ do not appear explicitly in the formula for computing confidence value because events in these sequences have been used to generate the sequences $alig$ and $notAlig$. Moreover, events in $alig$ and $notAlig.n$ are multiplied by a factor of 2 because two symbols (1 symbol per analyzed string) contribute to generate the case.

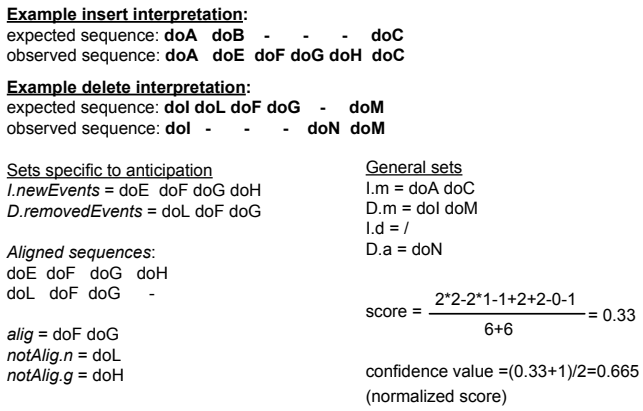

Figure 13: Example of anticipation interpretation.

Figure 13 shows an example application of the anticipation interpretation.

Posticipation

The case of the posticipation is symmetric to the anticipation.

A posticipation consists of events that occur later than expected. We analyze anomalies to discover if a posticipation occurred in all the cases where we detect two basic interpretations D, I, where D is a delete or replacement interpretation, I is an insert or replacement interpretation, and D occurs before I in the trace recorded during the failing execution.

To discover if a posticipation occurred, we define $D.removedEvents$ as the sequence that includes all the events in the expected sequence that are classified as deleted or mismatched, and $I.newEvents$ as the sequence of all the events in the observed sequence that are classified as added or mismatched. To discover if there are common symbols with proper ordering between these two sequences (thus suggesting a posticipation), we align $D.removedEvents$ and $I.newEvents$ with the same configuration used for the anticipation.

The sequence $alig$ of the symbols that match after string alignment represents the events that have been posticipated. The sequence $notAlig$ indicates events that are only deleted in the first interpretation, only added in the second interpretation or appear with the wrong order. If the number of aligned events is 0, we cannot have a posticipation.

If we have at least 1 aligned event, we have a behavior that can be interpret as a posticipation, even if it may be associated with a low confidence value. We compute the confidence value of the posticipation by positively counting the events that are posticipated and the ones that match, and negatively counting the differences that do not contribute to the posticipation, resulting in the same formula used for the anticipation.

Swap

A swap is the case of a sequence of events that are anticipated by replacing others that are posticipated. This interpretation can be easily discovered by combining the anticipation with the posticipation. In particular, if both such interpretations have been discovered for an observed trace, we also generate a swap interpretation with a confidence value given by the average value of the confidence values for the anticipation and the posticipation.

6. PRIORITIZE INTERPRETATIONS

Discovery of basic and composite interpretations ends with a list of possible interpretations that have a confidence value greater than 0. Since each anomaly can be compared with several possible expected behaviors, depending from the structure of the inferred FSA, and each pair $<$ *anomalous behavior*, *expected behavior* $>$ can have multiple explanations, the list of candidate interpretations can be large.

For instance, if an inferred FSA specifies that the system under analysis can continue its execution in three possible ways just after the point where an anomaly has been detected, the anomalous behavior is compared with 3 possible expected behaviors. Considering that until now we have defined 4 possible types of basic interpretations, and 3 types of composites interpretations, the technique may produce tens of interpretations with confidence value greater than 0.

To avoid having testers inspecting data with little interest, we rework the output produced by AVA in two steps: data aggregation and prioritization of the results.

Data aggregation consists of building a unique representation of equivalent interpretations. The same interpretations for a same anomaly are represented as a single interpretation associated with the best confidence value. For instance, if a same anomaly has 3 possible explanations as an *insert* interpretation with confidence value 0.9, 0.7 and 0.2, we display *insert* only once with confidence value 0.9. If necessary, the tester can expand this information and see the other equivalent interpretations that have been hidden. This reduction is applied to both basic and composite interpretations.

Finally, the overall set of resulting interpretations are globally ordered according to their confidence values. Since testers use AVA to look for explanations of observed failures, we present first explanations with high confidence values, thus clearly explaining differences between the expected behavior and the observed behavior, then the ones with low confidence value, which can be harder to read.

7. EMPIRICAL VALIDATION

Interpretations are more descriptive than plain lists of suspicious events by construction. The empirical work aims at demonstrating that interpretations can describe real failure causes and that AVA can effectively discover them. In particular, we show that (1) AVA discovers interpretations that describe the differences between observed and expected behaviors, when these differences match our patterns, and (2) failure causes that can be described as AVA interpretations occur in real systems. To this end, we designed two empirical investigations.

The first investigation focuses on a set of synthetic cases that have been automatically generated by our toolset. These cases cover the possible structure of the differences between expected and observed behaviors. We show that AVA effec-

Anomaly Type	Cases	ranking of the perfect interpretation			Related basic patterns found before
		1	<= 2	<= 5	
delete	36	34	36	36	-
finalState	36	36	36	36	-
insert	36	36	36	36	-
replacement	36	36	36	36	-
anticipation	180	0	158	180	87
posticipation	180	0	111	180	73
swap	180	0	157	180	123

Table 1: Results with ad-hoc cases

tively analyzes these cases.

The second investigation focuses on a set of third-party systems that are affected by known issues. We show that AVA effectively produces correct interpretations that can support and easy failure analysis activities.

7.1 Synthetic Cases

To evaluate the capability of AVA to correctly interpret differences between legal and failing executions, we designed a set of synthetic cases that stress the set of possible interpretations defined in this paper. In particular, we obtained the set of cases to be investigated by automatically generating expected and observed sequences that differ according to the following dimensions: the type of the anomaly, the starting point of the anomaly, the length of the anomaly, and, only for composite anomalies, the number of events that separate the two basic anomalies that compose the composite anomalies.

The type of the anomaly can be any of the basic and composite interpretations with the exception of the final state interpretation that can be trivially identified by AVA. The starting point of the anomaly indicates the first event that occurs in the anomalous behavior and it can be at any of the following positions with respect to events in the observed sequence: 0 (intended as the first event of the sequence), 1, 2, 3, end of the sequence minus 3, and end of the sequence minus 2. The length of the anomaly, intended as the set of symbols in the anomalous behavior, can be 1, 2, 3, 7 and 12. The distance between two basic interpretations that are part of a composite interpretations can be 1, 2, 3, 10, or 15 events.

If we combine the parameters above, we obtain 36 cases for each basic pattern and 180 cases for each composite pattern. We analyzed each case with AVA with $k_1 = k_2 = 4$. Table 1 summarizes results.

We can notice that all basic interpretations are successfully recognized and presented to testers at the top of the ranking, with few exceptions regarding the delete interpretations. In particular, for 2 out of the 36 cases related to the delete interpretation, AVA classifies replacement as a better description of the differences than delete. This happens when the deleted events occur at the end of the sequence.

We can also notice that it seldom occurs that composite interpretations are reported at the first position of the ranking. This is inherently related to complexity of these interpretations. However, most of the composite interpretations are reported at the second position, and in all cases within the fifth. Moreover, in a relevant number of cases, composite interpretations are overcome by the basic interpretations that generate the composite interpretations (see last column in Table 1). In such cases, it is extremely simple for testers to recognize that the composite interpretation is the most interesting description of the failure cause.

In summary, synthetic cases show that AVA can effectively recognize the interpretations defined in this paper when they occur.

7.2 Third-party Systems

ID	Case study	Failure Cause
G1	GlassFish (v. 2-GA)	The Java Petstore cannot be correctly deployed because of a configuration error [9]
G2	GlassFish (v. 2-GA)	The Java Petstore cannot be correctly deployed because of a configuration error [10]
G3	GlassFish (v. 3-b01)	The server hangs because of a fault related to classloading [8]
T1	Tomcat (v. 6.0.4)	A web application cannot be started because of a fault in the classloader [22]
T2	Tomcat (v. 6.0.14)	Tomcat is not starting because the default port is already in use [23]

Table 2: Case studies

To show that the interpretations defined in this paper can describe failure causes of real systems and AVA can discover them, we selected 5 case studies based on 2 large third-party systems: Glassfish [7], which is a J2EE Application Server (about 2 millions lines of code), and Tomcat [2], which is a JSP/Servlet server (about 300.000 lines of code). The case studies focus on known faults and typical configuration issues affecting Glassfish and Tomcat (see Table 2). The objective of the empirical validation is to show that AVA generates correct and useful interpretations of the observed failures and presents to testers a limited number of false positives that do not hinder effectiveness of the technique.

The method followed in the empirical study is based on three main steps. We first execute the functionality under test with about 50 test cases that we identified according to the category partition method [18]. During testing we collect traces (we use the log file natively recorded by these application servers) and we use LFA to derive a model that summarizes and generalizes the observed behavior of each component. We then reproduce the failing execution and we use LFA to discover behavioral anomalies. These anomalies are finally interpreted by AVA. To limit the length of the analyzed behaviors, we used a configuration with $k_1 = k_2 =$

Case Study	LFA anomalous events	LFA false anomalies	AVA pos failure interpretation	AVA prev relevant interpretations	AVA false interpretations	Inferred FSA fsa	Inferred FSA states	Inferred FSA trans	Inferred FSA t×s
G1	7	2	5	2	2	7	7	8	1.14
G2	36	9	16	11	4	16	56	442	7.75
G3	54	11	5	0	4	41	8	27	3.38
T1	17	2	1	0	0	6	32	76	2.38
T2	13	1	1	0	0	9	6	8	1.33

Table 3: Results with third-party systems

2. Results are summarized in Table 3.

Column *anomalous events* indicates the total number of anomalous events detected by LFA. Column *false anomalies* indicates the number of anomalies not related with the failure that testers need to investigate when they adopt LFA. Column *pos failure interpretation* indicates the position of the interpretation that pinpoints the behavior that caused the failure in the ranking produced by AVA. Note that AVA identified the interpretation of the failure cause for all the 5 case studies. Column *prev related interpretations* indicates the number of relevant interpretations that occur with better ranking than the interpretation of the failure cause. A relevant interpretation is defined as an interpretation that describes an anomalous behavior that is a consequence of the failure cause. A typical example is the extra events that are usually generated by applications to trace failures. These interpretations are useful to understand the failing execution, but not necessary to understand the failure cause. Column *false interpretations* indicates the number of false interpretations, i.e., interpretations that describe false positives, that are presented to testers before the one describing the failure cause. Finally, columns *fsa*, *states*, *transitions*, and $t \times s$ indicate the number of FSAs generated by LFA, and the average number of states, transitions and transitions per state in the inferred FSAs, respectively. These data show that our technique is able to work with multiple models of relevant size and complexity.

Note that AVA provided a description of the failure cause at one of the first 5 positions of the ranking for 4 case studies out of 5 (in 2 cases is at the top of the ranking). Moreover, in all the cases testers have to analyze at most 4 false interpretations. These results confirm effectiveness of AVA when analyzing real failures.

In addition to better describe failure causes than anomalies, interpretations also reduce the number of false alarms investigated by testers when using LFA. In fact, the number of false anomalies investigated by testers is always greater or equal than the number of false AVA interpretations.

8. RELATED WORK

There exist several techniques that support testers when investigating failure causes. Many debugging techniques provide automated and semi-automated ways to inspect the behaviors of a target system and find faults. However, most of the existing debugging techniques work only when the source code is available and the state of the application can be accessed, and eventually altered [30, 21, 4, 15, 12, 29].

Anomaly detection techniques can be used to address the systems that are not manageable by classic debugging techniques, i.e., systems with access limited to binaries or log files, and provide complementary information to classic debugging techniques [11, 19, 27, 24]. In fact, anomaly analysis techniques can indicate likely unexpected events that have been observed at run-time and that should deserve further investigation by testers, while classic debugging techniques focus more on the identification of the code blocks that likely include faults.

AVA augments anomaly detection techniques by providing an automated way to describe the anomalies that have been detected. In particular, AVA provides explanations to differences between failing and correct executions in a human understandable way rather than simply providing a plain set of accepted and reject events.

Some techniques address investigation of failure causes by using various data mining solutions, e.g., cluster analysis [5, 6, 16] and support vector machines [28]. Similarly to AVA, these solutions do not require any knowledge about the target system and purely work with dynamic data. These solutions can effectively pinpoint the anomalous events observed in failing executions, but do not provide any description of the differences between legal and failing executions. Thus anomaly interpretation is left to testers. On the contrary, AVA focuses on the generation of useful descriptions that ease failure investigations, by using a catalog of possible differences.

Finally, the technique by Weimer [25] shares with AVA the idea to compare actual behaviors and expected behaviors, specified with FSA, to detect problems. However, the two techniques differ on the purpose and kind of analysis. AVA compares traces and FSA to provide relevant interpretations of anomalous behaviors, while the technique presented in [25] compares the behavior extracted by analyzing the program source code with the behavior specified by the FSA to suggest simple fixes to statically detected faults.

9. CONCLUSIONS

Anomaly detection techniques have been largely applied to analyze software failures. These techniques can provide useful information in terms of anomalous event sequences that have been observed in failing executions and that are not normally observed in legal executions. However, this information is usually limited to the identification of a set of suspicious event sequences, and does not capture the rationale of the differences between legal and failing executions.

In this paper we presented AVA, a technique to automatically analyze the anomalous events discovered by anomaly detection techniques that use inferred FSAs to produce interpretations. The interpretations generated by our solution capture the differences between legal and failing executions with descriptions well understandable by testers.

Empirical validation shows that AVA effectively identified interpretations both for a large set of synthetic cases and

for several third-party systems. Furthermore, AVA reduced diagnosis effort by reducing the number of false reports inspected by testers to understand failure causes.

Future work concerns the definition of additional interpretations and the preparation of empirical studies to measure the time saved by developers when diagnosing problems by using AVA.

Acknowledgment. This work has been supported by the European Community under the Information Society Technologies (IST) programme of the 6th FP for RTD - project SHADOWS contract IST-035157.

10. REFERENCES

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *proceedings of the 29th Symposium on Principles of Programming Languages*, pages 4–16. ACM Press, 2002.

[2] Apache Software Foundation. Tomcat JSP/Servlet server. http://tomcat.apache.org/, visited in 2009.

[3] A. Biermann and J. Feldman. On the synthesis of finite state machines from samples of their behavior. *IEEE Transactions on Computer*, 21:592–597, June 1972.

[4] L. Briand, Y. Labiche, and X. Liu. Using machine learning to support debugging with tarantula. In *proceedings of the International Symposium on Software Reliability Engineering*, 2007.

[5] M. Y. Chen, E. Kiciman, E. Fratkinand, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *proceedings of the International Conference on Dependable Systems and Networks*. IEEE Computer Society, 2002.

[6] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *proceedings of the 23rd International Conference on Software Engineering*, 2001.

[7] Glassfish application server. Glassfish. https://glassfish.dev.java.net/, visited in 2009.

[8] Glassfish bug database. Glassfish issue 4255. https://glassfish.dev.java.net/issues/showbug.cgi?id=4255, visited in 2009.

[9] Glassfish user forum. Glassfish configuration issue. http://forums.java.net/jive/thread.jspa?messageID=252898, visited in 2009.

[10] Glassfish user forum. Glassfish configuration issue. http://forum.java.sun.com/thread.jspa?threadID=5249570, visited in 2009.

[11] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *proceedings of the 24th International Conference on Software Engineering*, 2002.

[12] J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *proceedings of the Internation Conference on Software Engineering*, 2002.

[13] L. Mariani and F. Pastore. Automated identification of failure causes in system logs. In *proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 2008.

[14] L. Mariani and M. Pezzè. Dynamic detection of cots components incompatibility. *IEEE Software*, 24(5):76–85, 2007.

[15] W. Masri, A. Podgurski, and D. Leon. An empirical study of test case filtering techniques based on exercising information flows. *IEEE Transactions on Software Engineering*, 33(7):454–477, July 2007.

[16] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller. Problem diagnosis in large-scale computing environments. In *proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[17] 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*, 48(3):443–453, March 1970.

[18] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commununications of the ACM*, 31(6):676–686, 1988.

[19] O. Raz, P. Koopman, and M. Shaw. Semantic anomaly detection in online data sources. In *proceedings of the 24th International Conference on Software Engineering*, 2002.

[20] S. P. Reiss and M. Renieris. Encoding program executions. In *proceedings of the 23rd International Conference on Software Engineering*, pages 221–230. IEEE Computer Society, 2001.

[21] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *proceedings of the Internation Conference on Automated Software Engineering*, 2003.

[22] Tomcat bug database. Tomcat fault 40820. https://issues.apache.org/bugzilla/showbug.cgi?id=40820, visited in 2009.

[23] Tomcat bug database. Tomcat configuration issue. http://www.blogjava.net/haix/archive/2008/01/16/175592.html, visited in 2009.

[24] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *proceedings of the European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2007.

[25] W. Weimer. Patches as better bug reports. In *proceedings of the 5th International Conference on Generative Programming and Component Engineering*. ACM, 2006.

[26] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2002.

[27] C. Yilmaz, A. Paradkar, and C. Williams. Time will tell: Fault localization using time spectra. In *proceedings of the International Conference on Software Engineering*, 2008.

[28] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. In *proceedings of the 2006 EuroSys conference*. ACM, 2006.

[29] A. Zeller. Isolating cause-effect chains from computer programs. *SIGSOFT Software Engineering Notes*, 27(6):1–10, 2002.

[30] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufman, 2005.