Appeared in A. Winter, R. Ferenc, and J. Knodel (eds.), Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009), pages 47–57. IEEE. March 2009.

Automatic Failure Diagnosis Support in Distributed Large-Scale Software Systems based on Timing Behavior Anomaly Correlation*

Nina Marwede
BTC Business Technology Consulting AG
Escherweg 5, 26121 Oldenburg, Germany
nina.marwede@btc-ag.com

Matthias Rohr
BTC Business Technology Consulting AG
Escherweg 5, 26121 Oldenburg, Germany
matthias.rohr@btc-ag.com

André van Hoorn
Graduate School TrustSoft
University of Oldenburg, Oldenburg, Germany
van.hoorn@informatik.uni-oldenburg.de

Wilhelm Hasselbring
Software Engineering Group
University of Kiel, Kiel, Germany
wha@informatik.uni-kiel.de

Abstract

Manual failure diagnosis in large-scale software systems is time-consuming and error-prone. Automatic failure diagnosis support mechanisms can potentially narrow down, or even localize faults within a very short time which both helps to preserve system availability. A large class of automatic failure diagnosis approaches consists of two steps: 1) computation of component anomaly scores; 2) global correlation of the anomaly scores for fault localization.

In this paper, we present an architecture-centric approach for the second step. In our approach, component anomaly scores are correlated based on architectural dependency graphs of the software system and a rule set to address error propagation. Moreover, the results are graphically visualized in order to support fault localization and to enhance maintainability. The visualization combines architectural diagrams automatically derived from monitoring data with failure diagnosis results. In a case study, the approach is applied to a distributed sample Web application which is subject to fault injection.

1 Introduction

For many companies, software systems such as online store applications or information systems are business-critical. Due to their complexity, such large-scale software systems are practically never free of software faults, and especially software faults are a major cause for system failures [9]. Manual failure diagnosis can be very time consuming and error-prone, since debugging is basically a search in space across a program state to find infected variables, and a search in time over millions of program states [4]. This motivates the development of automated processes for failure detection, fault localization, and fault removal [2].

Software behavior, such as timing behavior or control flow, and its statistical analysis have been demonstrated as valuable for failure diagnosis [1, 8, 16]. Many such approaches use the concept of anomaly detection: Current system behavior is compared to a profile learned from historical timing behavior in order to find fault indicators. Detecting anomalies in a software system is only the first step of failure diagnosis. The second step is to localize a failure's root-cause based on the component anomaly scores in the context of the system's architecture. Ideally, the component with the highest amount of anomalies or strongest anomalies is the root-cause. However, a component that shows anomalies might only be anomalous because it depends on another component that is the real root-cause. Especially anomalies such as exceptional long response times tend to propagate through the software architecture. Therefore, correlating anomaly scores [14] is required to compensate such propagation effects to provide accurate and clear end-to-end fault localization. Anomaly correlation approaches do not necessarily suggest only one architectural element as root-cause, but usually at least provide a significant reduction of the search space of possible causes by declaring a large part of the system as not being the fault.

This paper introduces the anomaly correlator *RanCorr*. It analyzes the results from timing behavior anomaly detection in the context of an architectural model consisting of calling dependencies between software components. The architectural model is automatically derived from monitor-

* This work is supported by the German Research Foundation (DFG), grant GRK 1076/1.

Appeared in A. Winter, R. Ferenc, and J. Knodel (eds.), Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009), pages 47–57. IEEE. March 2009.

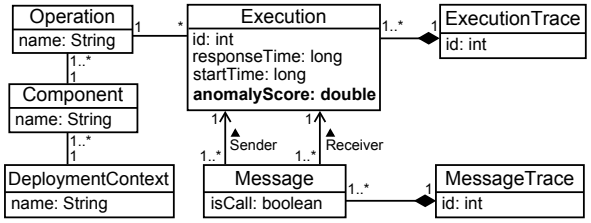


Figure 1. Input data structures for RanCorr.

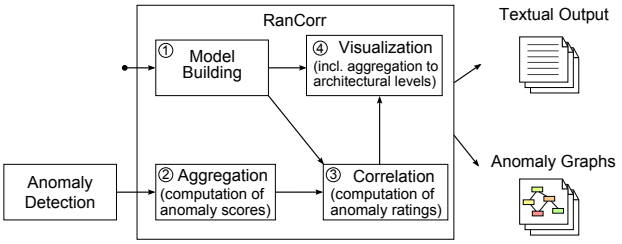


Figure 2. Conceptual architecture, analysis activities and output of RanCorr.

ing data. RanCorr uses a set of rules that reflects general assumptions of propagation effects within the software architecture. Additionally, the approach contributes hierarchical visualization of the anomaly detection results to support administrators in failure diagnosis. An important aspect of the visualization is the clearness, i.e., a measure for the contrast of the visual impression, and an indicator for the quality of the correlation results.

In a case study, the approach is demonstrated by injecting faults into a distributed multi-user Java Web application. The combined results of the timing behavior anomaly detector and the anomaly correlator support the claim that timing behavior is a valuable indicator for failure diagnosis. Furthermore, the results of the case study show that the requirements in terms of monitoring overhead and maintainability are satisfied.

The remainder of this paper is structured as follows. Section 2 provides the basic concepts of software timing behavior anomaly detection. An overview and the details of our anomaly correlation approach are presented in Sections 3 and 4. Sections 5 and 6 provide the case study and a discussion of results. Related work follows in Section 7 before the conclusions are drawn in Section 8.

2 Software (Timing) Behavior and Anomaly Detection

It is assumed that the software system under supervision is composed of components hosted on *deployment contexts*. The components provide *operations*, e.g., implemented as Web services or plain Java methods. Primary artifacts of runtime behavior are *executions* of the operations. A finite sequence of executions resulting from a request is denoted a *trace*. We limit the scope to synchronous communication between executions as defined in the UML [10]: The caller of an operation is blocked and has to wait until the callee returns a result before it continues its own execution. A trace is a complete representation of the control flow originating from a request.

The basic software runtime behavior model described above can provide input data for anomaly detectors that compute anomaly scores by comparing an execution's response time with historical ones. Such anomaly detectors can for instance be found in [1, 3, 5, 16].

The combination of anomaly scores and the runtime software behavior provides the input data for anomaly correlation. For our anomaly correlator RanCorr, this model is defined as shown in Figure 1. In addition to a *response time* and a *start time*, each execution has an *anomaly score* which is assumed to be in the interval $[-1, 1] \subset \mathbb{R}$. A score of -1 means that the execution has a normal behavior, and 1 means that this execution is considered to be very anomalous.

3 Overview of our Anomaly Correlation Approach

Our correlation approach draws conclusions from the arrangement of the timing anomalies in the calling dependency graph to identify which component and deployment context (e.g., a virtual machine) contains the fault – provided the failure significantly influenced the timing behavior.

A calling dependency graph consists of nodes which represent software operations, and directed edges that represent call actions between operations. In addition, the graph is hierarchical: A component has a set of operations, and a deployment context hosts a set of components. The calling dependency graph is automatically constructed from monitoring data.

It is a common assumption that anomalies propagate through the calling dependency graph in backwards direction of calling dependencies as suggested e.g. by Gruschke [6]: If a graph node is behaving anomalous, then the node's callers typically show anomalous behavior themselves. Our approach performs a backward propagation analysis by evaluating each node's neighborhood for propagation patterns.

An overview of RanCorr's four anomaly correlation activities and their relations is given in Figure 2.

1. **Model Building**. Based on the monitoring data and

Appeared in A. Winter, R. Ferenc, and J. Knodel (eds.), Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009), pages 47–57. IEEE. March 2009.

the evaluation results of the anomaly detectors, the caller–callee relations as well as their anomaly scores are combined into a model of the application under analysis. This graph-like representation of the application contains both static relations of the elements of the architectural structure, and anomalies in their dynamic behavior.

2. **Aggregation**. The anomaly scores of all executions are aggregated to an *anomaly score* for each architectural element, which is a single number representing an element's degree of being anomalous.

3. **Correlation**. The anomaly scores are correlated within the architectural context to determine an *anomaly rating* for each architectural element. An anomaly rating expresses RanCorr's estimate that an architectural element is the root cause of a fault that caused the anomalies. In the correlation, edges of the calling dependency graph are analyzed for anomaly propagation effects between operations. Ideally, the correlation analysis is able to perform a negation of the propagation effects. The correlation analysis is implemented by applying a rule set that is based on general assumptions about propagation.

4. **Visualization**. The result of the correlation activity is a list of anomaly ratings for the architectural elements of the application under analysis. Initially, this list contains anomaly ratings for all instrumented operations (i.e., the software methods). Using additional aggregation, ordered lists of ratings for all components and deployment contexts are returned. The visualization activity produces a graphical visualization on operation-, component-, or deployment context level for supporting administrators. Component level or deployment context level visualizations are more coarse-grained and more suitable to provide a first overview than the most detailed operation level visualization. An example visualization that covers all three levels is provided in Figure 3. The case study in Section 5 contains further examples.

The correlation activity forms the core of the failure diagnosis, since it localizes faults based on evaluating anomaly ratings in the architectural context. The anomaly ratings are single real-valued numbers in the interval $[-1, 1] \subset \mathbb{R}$, created by aggregating all anomaly scores of one graph node. The ratings reflect to what extent a node is suspected to be the root-cause of failure. The correlation step basically redistributes the anomaly ratings within the graph structure. The anomaly ratings have a maximum value of 1, meaning a significant anomaly, while -1 means perfectly normal behavior. An anomaly rating of 0 means that the classification is ambiguous.

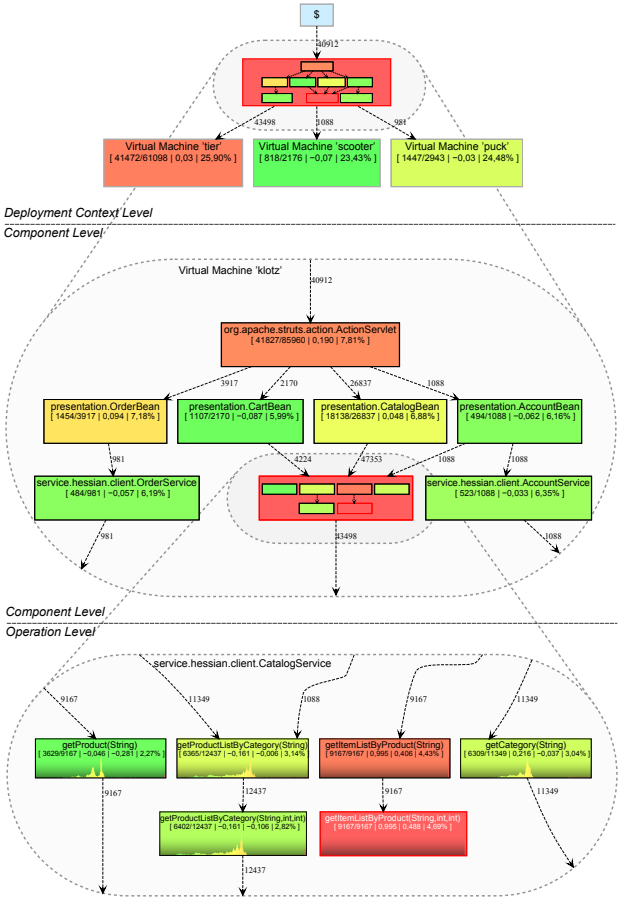


Figure 3. Visualization on three architecture levels.

4 Anomaly Correlation Algorithms

After the strategy and activities of RanCorr were presented above, next the aggregation and correlation activities are detailed. Three different algorithms that both cover aggregation and correlation are introduced in the following Sections 4.1–4.3. Figure 4 shows how these three algorithms are structured according to the architectural levels (operation, component, and deployment context), and what techniques are used. This shows that we apply correlation on the operation level, which is the most detailed architectural level in the system model.

The so-called *trivial* algorithm uses averaging as aggregation and correlates by using the identity function, which can also be considered omitting correlation. The main purpose of the trivial algorithm is to provide a reference point for a later quantification of other algorithms' benefits. The *simple* and *advanced* algorithms use rules, implemented as conditional mathematical functions, to inverse propagation

Appeared in A. Winter, R. Ferenc, and J. Knodel (eds.), Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009), pages 47–57. IEEE. March 2009.

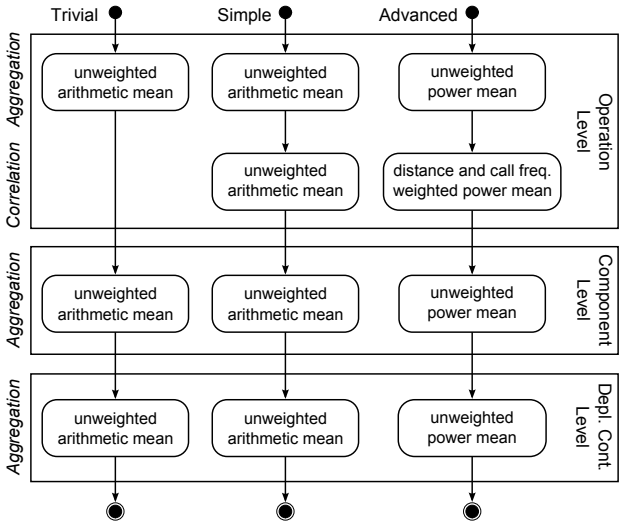


Figure 4. Three algorithm variants derive anomaly ratings from anomaly scores provided by an anomaly detector. Anomaly ratings are computed for operations, components, and deployment contexts.

effects. The advanced algorithm uses enhanced aggregation and correlation methods to consider more assumptions on how propagation effects distribute anomaly scores among the components.

4.1 Trivial Algorithm

The trivial algorithm aggregates anomaly scores $s \in S_i$ to determine the anomaly rating r_i of an operation i (the i-th operation provided by the anomaly detector). The aggregation uses the unweighted arithmetic mean as shown in Equation 1.

$$r_i := \overline{S}_i := \frac{1}{|S_i|} \cdot \sum_{s \in S_i} s \tag{1}$$

As displayed in Figure 4, the unweighted arithmetic mean is also used to aggregate a component's anomaly rating, and a deployment context's rating from the corresponding operation ratings and component ratings respectively.

4.2 Simple Algorithm

Two rules are used to detect configurations in the anomaly structure that are relatively clear to understand and to implement. Two specific conditions are tested, and an increase or decrease flag is set. Additional information is completely ignored, because its effects are unknown and could be misleading. The cause rating is then derived from the anomaly rating, according to the flags.

More precisely, the anomaly rating is increased if the unweighted arithmetic mean of the anomaly ratings of the directly connected callers (upwards in the calling dependency graph) is greater than the anomaly rating of the currently considered operation. This means that this operation is likely to be the cause of failure, because the dependent operations show significant anomalies. The rating is decreased if the maximum of the anomaly ratings of the directly connected callees (downwards in the graph) is greater than the anomaly rating of the current operation. This means that this operation's rating is likely to be a propagation from another operation it depends on. Under all other conditions, as well as in special cases such as singular connections, and the root operation, the value of the anomaly rating is forwarded without change.

Equation 2 defines the correlation function of the simple algorithm that computes r_i as the anomaly rating for operation i, $\overline{S}_i$ as the (unweighted arithmetic) mean anomaly score for operation i, $\overline{S}_i^{in}$ as the mean of all anomaly scores corresponding to operations with calls to i, and max_i^{out} as $max\{\overline{S}_k | k$ is operation called by operation $i\}$.

$$r_i := \begin{cases} \frac{1}{2} \cdot (\overline{S}_i + 1), & \overline{S}_i^{in} > \overline{S}_i \ \wedge \ max_i^{out} \leq \overline{S}_i \\ \frac{1}{2} \cdot (\overline{S}_i - 1), & \overline{S}_i^{in} \leq \overline{S}_i \ \wedge \ max_i^{out} > \overline{S}_i \\ \overline{S}_i, & \text{else} \end{cases} \tag{2}$$

The function for increase and decrease ($\frac{1}{2} \cdot (\overline{S}_i \pm 1)$) is chosen for its simple linear graph, staying in range $[-1, 1]$. Again, the aggregation is done through an unweighted arithmetic mean calculation on all three levels.

4.3 Advanced Algorithm

The advanced algorithm extends the simple algorithm by the following features:

- The anomaly rating for each node is computed by aggregating the anomaly scores of its forward and backward transitive closure. For instance, the anomaly rating for the node D (Figure 5) is computed based on the aggregated anomaly scores (values inside the nodes) of A, B, H–J, F, and G, in contrast to the simple algorithm that considers only A, B, F, and G.
- Distance and edge weights, i.e., the calling frequencies, are used to weight the connections. Together with the consideration of the transitive closure described above, this models the observation that anomalies propagate via the edges over the complete calling graph with descending strength by increasing distance.

Appeared in A. Winter, R. Ferenc, and J. Knodel (eds.), Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009), pages 47–57. IEEE. March 2009.

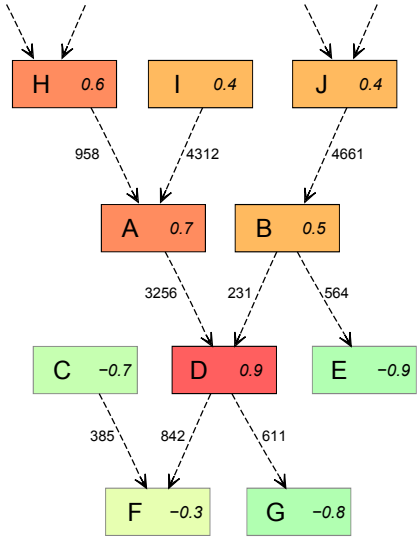


Figure 5. Example: An anomaly in D is propagated upwards in the calling dependency graph. Neighbor and callee elements are not affected. The nodes and edges are annotated with anomaly scores and calling frequencies.

- The power mean is used for aggregation and correlation, instead of the arithmetic mean. This allows one to vary the influence of extreme values within the anomaly scores. A power mean with exponent 1 is equal to the arithmetic mean. Exponents smaller than 1 reduce the influence of outliers, while exponents greater than 1 increase it.

The advanced algorithm is detailed in the remainder of this section.

4.3.1 Weighted power mean

Common definitions for the power mean are only valid for positive values and do not weight values. An extended power mean $\widetilde{S}$ is provided in Equation 3. $\widetilde{S}_{i,p}$ aggregates the anomaly scores $S_i = \{s_1, \ldots, s_n\}$ for an operation i with the corresponding weights $W_i = \{w_1, \ldots, w_n\}$. The parameter p is the exponent of the power mean.

$$\widetilde{S}_{i,p} := \gamma\left(\frac{\sum_{j=1}^{n} w_j \cdot \gamma(s_j, p)}{\sum_{j=1}^{n} w_j}, \frac{1}{p}\right), \qquad \gamma(a,q) := |a|^q \cdot \begin{cases} 1, & a >= 0 \\ -1, & a < 0 \end{cases} \tag{3}$$

4.3.2 Aggregation of anomaly scores

According to Figure 4, the three aggregation steps use an unweighted power mean to combine all anomaly scores of an operation (or higher level element) into a single number. The weights in the power mean (Equation 3) are set to 1 in anomaly score aggregation.

4.3.3 Correlation of anomaly ratings

The correlation function uses the anomaly rating of the highest rated callee (successor in the graph) like in the simple variant, but additionally includes indirectly connected operations. The mean calculation of the caller operations (predecessors in the graph) is now weighted by distance and edge weight. The edge weight can be either absolute, or relative.

A function traverses the dependency graph upwards and downwards in a tree-like depth-first search, storing the length of the shortest path to each directly or indirectly connected other operation, and the number of connections, i.e., the number of executions on that path. It is assumed that the intensity of a propagated anomaly decreases by distance to its origin. Each sample weight $w \in W_i$ is defined as the edge weight e divided by the distance d as shown in Equation 4 where z is a distance intensity constant.

$$w := \frac{e}{d^z} \tag{4}$$

The anomaly rating r_i for an operation i is defined as follows:

$$r_i := \begin{cases} \frac{1}{2} \cdot (\widetilde{S}_{i,0.2} + 1), & \widetilde{S}^{in}_{i,1} > \widetilde{S}_{i,0.2} \wedge max^{out}_i \leq \widetilde{S}_{i,0.2} \\ \frac{1}{2} \cdot (\widetilde{S}_{i,0.2} - 1), & \widetilde{S}^{in}_{i,1} \leq \widetilde{S}_{i,0.2} \wedge max^{out}_i > \widetilde{S}_{i,0.2} \\ \widetilde{S}_{i,0.2}, & \text{else} \end{cases} \tag{5}$$

In Equation 5, $\widetilde{S}^{in}_i$ and max^{out}_i include the directly and indirectly connected operations in the dependency graph as described above. The power mean exponent is set to 0.2 by default (thus weakening the influence of outliers). On the higher aggregation levels, the power mean exponent is set to 2.0 and 3.0 for component and deployment context level, respectively.

5 Evaluation

In the following, the applicability of RanCorr's three algorithms is evaluated. For this, several fault injection scenarios are applied to a distributed system.

5.1 Evaluation Goals and Metrics

The two evaluation goals are to evaluate 1) whether injected faults are accurately localized, and 2) whether RanCorr

Appeared in A. Winter, R. Ferenc, and J. Knodel (eds.), Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009), pages 47–57. IEEE. March 2009.

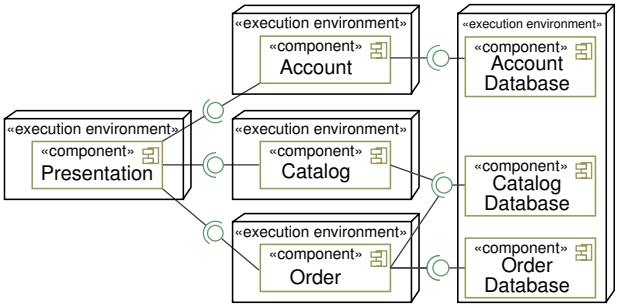


Figure 6. Deployment architecture of the distributed JPetStore.

clearly ranks the components. Furthermore, the different algorithm variants and parameter values are to be evaluated with respect to these goals. Concrete metrics for localization accuracy and clearness are defined as follows.

Accuracy

For the n decreasingly ordered anomaly ratings $(r_1, \ldots, r_n)$ and r_i corresponding to the architectural element which was subject to fault injection, let accuracy be defined as follows (Equation 6).

$$accuracy(\{r_1, \ldots, r_n\}) := \frac{n - rank(r_i)}{n - 1} \qquad (6)$$

where $rank$ denotes an architectural element's position in the corresponding list of decreasingly ordered anomaly ratings. The $accuracy$ is 100% if the faulty element has been assigned the highest rating, and 0% if the lowest anomaly rating is assigned to the faulty element.

Clearness

In addition to accuracy, we define clearness in the following Equation 7.

$$clearness(\{r_1, \ldots, r_n\}) := \frac{r_j}{\sum_{k=1, k \neq j}^{n} \frac{r_k}{rank(r_k)} + 1} \qquad (7)$$

where component j has the highest anomaly rating. A higher clearness indicates a faster decrease in anomaly ratings within the list, thus a higher contrast and understandability in visualization.

5.2 Experiment Setup

The software system for the case study is the iBATIS JPetStore[1] which is a demo Java Web application implementing an online store scenario. In order to get a more realistic scenario, we divided the JPetStore into components, and deployed the system to five machines as illustrated in Figure 6.

The application is monitored by our monitoring framework Kieker [13] which records timestamps for method executions and monitors control flow. In total, 34 Java methods of the application have been instrumented with monitoring probes.

We expose the software to probabilistic workload that is generated by Apache JMeter[2] and our extension Markov4JMeter [15] that generates probabilistic workload based on Markov chains. The workload is set up to constantly simulate 15 concurrent users, which is far less than the overall system capacity of about 80 concurrent users as observed in preparative experiment runs.

5.3 Fault Injection

Fault injection is used to provoke failures of different severity and different granularity in the application under analysis. The two major fault scenarios for the case study are programming faults and database slowdown injections. Programming faults are implemented by manual source code mutation, incorrect assignments, or exchanged calls. Database slowdown injections are implemented by adding timing delays to operations that access the database. The delay is set to 10 ms. This slowdown is intended to have significant influence allowing a detection with high probability.

In total, five fault injection scenarios are defined: Three programming faults, and two database slowdowns at different locations. Each of these scenarios is executed three times. Additionally, to get training data that can be assumed to be free of anomalies, three experiment runs are performed without fault-injection. Each experiment run consists of 5 minutes warm-up, 15 minutes monitoring, and 5 minutes pre- and post-processing steps, such as restarting the deployment contexts. This results in about 8 hours pure experiment run time.

5.4 Monitoring Data and Anomaly Scores

The size of monitoring data is 1.7 GB, with 7 million monitored executions in total, i.e., about 370,000 executions for each run.

Regardless of the injection variant and position, the anomaly detector always classifies a considerable fraction of the executions anomalous. The anomaly detector used in this case study (variant of the *plain anomaly detector* [11]) learns two thresholds from the run that was not subject to fault injection. The scoring function of the anomaly detector, which is not a focus of this paper, assigns real-valued anomaly scores in the range $[-1, 1]$, based on the first and

[1] http://ibatis.apache.org/

[2] http://jakarta.apache.org/jmeter/

Appeared in A. Winter, R. Ferenc, and J. Knodel (eds.), Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009), pages 47–57. IEEE. March 2009.

Table 1. Summarized results of the fault injection scenarios (scoring from 1 (- -) to 5 (++)).

Scenario	Injection	Trivial	Simple	Advanced
No. 1	Progr. fault	+	+	+
No. 2	Progr. fault	+	+	++
No. 3	Progr. fault	-	-	+
No. 4	DB slowdown	+	++	++
No. 5	DB slowdown	o	+	++
Averages		3.4	3.8	4.6

third quartile of an operation's response time. These values are mapped to the anomaly scores -0.5 and 0.5 respectively.

5.5 Anomaly Correlation Results

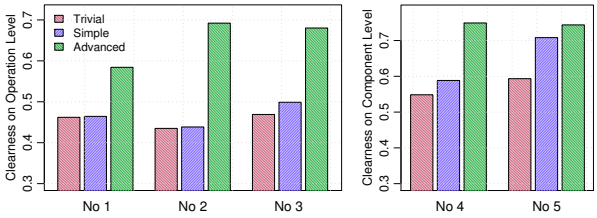
Figure 7 (next page) shows a part of a diagnosis visualization. Due to space restrictions, most parts of the graph are omitted. In this visualization, all three architectural levels (operation, component, deployment context) are activated and small histograms show the distribution of the anomaly scores for each operation. Four components are deployed within the one of the four deployment contexts shown.

Example visualizations for accurate localization and a demonstration of the clearness metric are provided in Figure 9 for fault scenario 5 (DB slowdown). In the visualizations from both the trivial algorithm and the advanced algorithm, all elements corresponding to the fault are deeply red colored and have high anomaly ratings. The visualization produced by the advanced algorithm is more clear in pinpointing to suspected architectural elements. The highest ratings on their respective levels are correctly assigned to the faulty elements (ItemSqlMapDao and 'tier').

All five fault scenarios are accurately localized by the advanced algorithm variant as shown in Table 1. In contrast, the localizations performed by the trivial and simple algorithms are successful in only four of the five cases, symbolized by '++', '+', and 'o' signs. These three differ in mathematical clearness, and in overall visual impression. In scenario 3, although the trivial and simple algorithms reach respectable values for clearness (see Figure 8), in the list of decreasingly ordered operations, the rank of the operation where the fault has been injected is less than one, thus the accuracy does not reach 100% as shown in Table 2.

5.5.1 Accuracy

Table 2 presents the accuracy results. In the majority of the evaluations, the accuracy is 1 (100%), which means that RanCorr correctly assigned the highest anomaly rating to the architectural element that contained the fault.

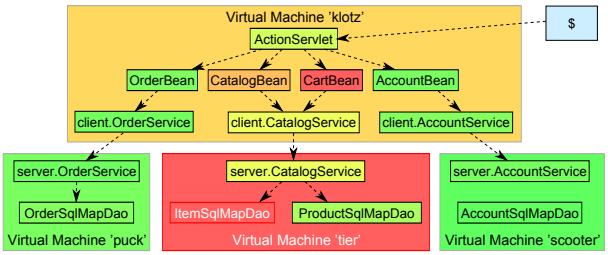


(a) Programming fault scenarios (b) DB slowdown scenarios

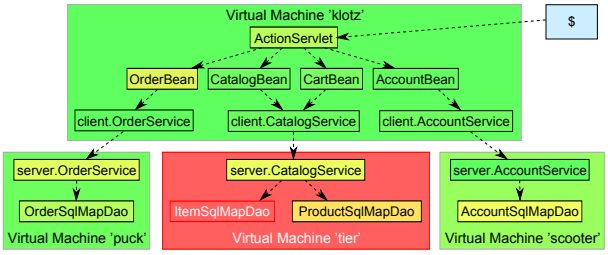
Figure 8. Clearness results.

Table 2. Anomaly correlation accuracy.

Scenario	Injection	Trivial	Simple	Advanced
No. 1	Progr. fault	1	1	1
No. 2	Progr. fault	1	1	1
No. 3	Progr. fault	0.97	0.97	1
No. 4	DB slowdown	1	1	1
No. 5	DB slowdown	1	1	1



(a) Result of trivial algorithm



(b) Result of advanced algorithm

Figure 9. Component-level visualization: Database slowdown scenario.

Only the advanced algorithm produced completely accurate results, while the trivial and simple algorithms fail to produce accurate results for the third programming fault scenario.

5.5.2 Clearness

The two failure diagnosis visualizations shown in Figure 9 are an example for differences in clearness of results from the different algorithms. Figure 9 depicts the results for

Appeared in A. Winter, R. Ferenc, and J. Knodel (eds.), Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009), pages 47–57. IEEE. March 2009.

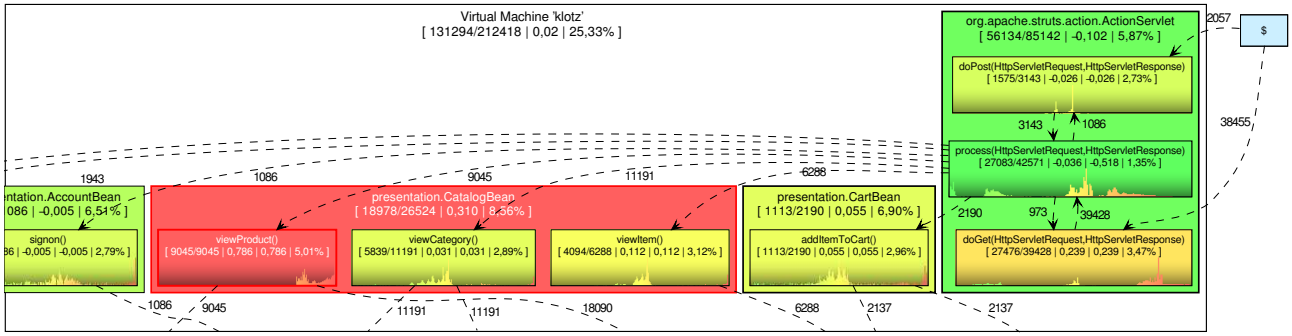
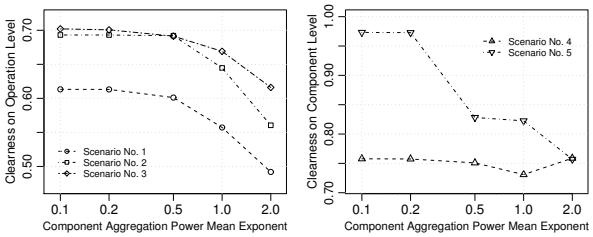
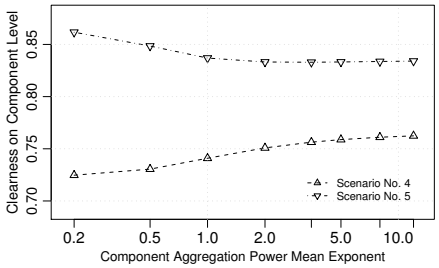


Figure 7. Partial example of the graphical result of RanCorr. The percent values, augmented with colors, represent the probability for the elements to be the cause of failure.



(a) Programming fault scenarios (b) DB slowdown scenarios

Figure 10. Clearness for varying operation aggregation power mean exponents.



(a) DB slowdown scenarios

Figure 11. Clearness for varying component aggregation power mean exponents.

a "database connection slowdown" scenario, for the trivial and the optimized algorithm, reduced to component and deployment context level. The correct elements are highlighted in both cases, and in deep red color, while the other elements are mostly green with a high contrast for the optimized algorithm. On the other hand, there is much yellow and orange color for the trivial algorithm, meaning uncertainty regarding the cause of failure. A strong propagation effect is visible up to the presentation layer (Virtual Machine 'klotz'), still present in the final diagnosis visualization by the trivial algorithm (Figure 9(a)).

5.6 Influence of Correlation Algorithm Parameters

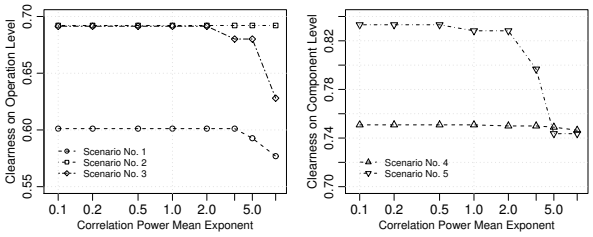
The influence of the algorithm parameters on the correlation quality is summarized in the following paragraphs:

Operation Aggregation Power Mean Exponent The power mean exponent affects the influence of extreme values on the mean calculation. With increasing exponent, the influence of outliers is also increased. Figure 10 shows that for the aggregation of anomaly scores on operation level, a small exponent gives better results for all fault scenarios.

Component Aggregation Power Mean Exponent Because of our iterative approach as depicted in Figure 2, the higher-level processing does not affect the lower-level's results. Thus, for scenarios 1–3, where the clearness on operation level is the fundamental benchmark, these values remain constant for higher-level aggregation. However, an influence on the overall (visual) results is reflected by the clearness on those levels, being a subordinate benchmark, that shows a distinct advantage of larger aggregation exponents. Figure 11 reveals ambiguous results for the scenarios 4 and 5. While scenario 4 benefits from a higher exponent, and the aggregation on deployment context level (not shown here) supports this for both scenarios, the curve for 5 does not, though the range of values is small.

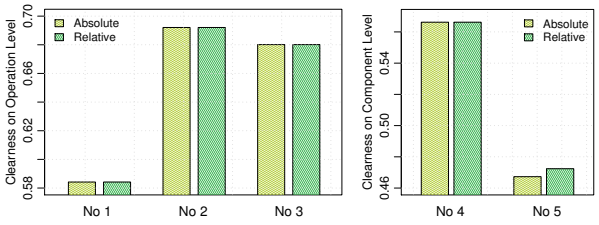
Correlation Power Mean Exponent Similar to the exponents for aggregation, the power mean exponents for correlation are analyzed. Figure 12 shows that in most cases, a smaller value gives a better result.

Appeared in A. Winter, R. Ferenc, and J. Knodel (eds.), Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009), pages 47–57. IEEE. March 2009.



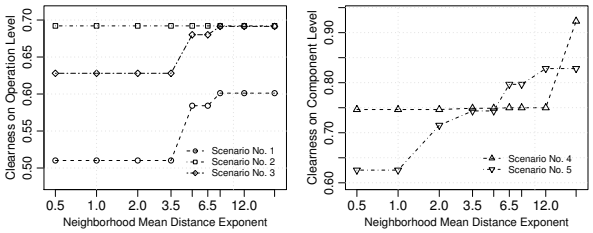
(a) Programming fault scenarios (b) DB slowdown scenarios

Figure 12. Clearness for varying correlation power mean exponents.



(a) Programming fault scenarios (b) DB slowdown scenarios

Figure 13. Clearness ratings for the two alternative edge weighting methods.



(a) Programming fault scenarios (b) DB slowdown scenarios

Figure 14. Clearness for varying neighborhood mean distance exponents.

Edge Weight Method Two different methods to calculate the edge weights can be used in the correlation activity. The "absolute" method uses the number of connections between the elements, while the "relative" method uses their percentages. Figure 13 reveals no difference in the results for scenarios 1–4, and only a very small advantage of the "relative" method for scenario 5. However, the clearness on higher levels is more distinctly increased in scenario 5, and our other experiments with more algorithm variants (not part of this paper) continuously confirmed this advantage.

Neighborhood Mean Distance Exponent As part of the edge weight calculation (see Equation 4), the neighborhood mean distance exponent is used to emphasize ($z < 1$) or weaken ($z > 1$) the influence of indirectly connected architecture elements – according to their distance – during the correlation on operation level. Figure 14 shows a trend that in four of the five scenarios, larger exponents provide better results, i.e., the influence of more distant elements is considerably weakened.

5.7 Summary

The advanced algorithm performs best in the case study. In average, it provides accurate and more clear results than the other two algorithms. The simple algorithm produces better results than the trivial variant, but the difference is small in most of the cases. The simple algorithm produces graphs containing less yellow and orange shading than the trivial algorithm, which can be considered to be more certainty or precision in localization, and supports the understandability of the visualizations.

6 Discussion

The quality of RanCorr's results depends on the completeness of the dependency graph. If the monitoring instrumentation is too coarse-grained, i.e., only few monitoring points exist, then the evaluation will be imprecise, in that dependencies are missing, and therefore connections between the affected components cannot be recognized. If the instrumentation is too fine-grained, i.e., many methods are instrumented, the amount of data will increase without being an advantage to the result.

Computational Requirements The implementation prototype reasonably scales: The complete analysis of the monitoring data of one experiment of about 41,000 traces containing 262,000 executions takes round about 60 seconds on a 1.5 GHz desktop PC with 1 GB main memory. A large part of this time is required for loading and parsing the monitoring data, reconstructing the control flow and computing anomaly scores, while the anomaly correlation takes about 7 seconds.

Monitoring Overhead As mentioned in the evaluation section, the Kieker framework [13] is used for monitoring calling dependencies and response times. Kieker is intended to be used for continuous monitoring during regular operation. Thus, the monitoring overhead must be considerably lower than that of profiling tools. Kieker's monitoring code is woven into the application code. The cost of measurement is constant for each activated monitoring point. In benchmarks, we observed an overhead that was in the order of microseconds for each activated instrumentation point. In order to decouple the application control

Appeared in A. Winter, R. Ferenc, and J. Knodel (eds.), Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009), pages 47–57. IEEE. March 2009.

flow from the measurement storage, the measurement data is asynchronously written to permanent storage.

Of course, the overall overhead depends on the number of activated monitoring points and its activation frequency. A key challenge in instrumentation is always the selection of feasible monitoring points, i.e., which operations to instrument. In the case of failure diagnosis, the instrumentation is guided by the desired granularity of fault localization. In the case of component-level fault localization, the instrumentation should be based on the granularity of the corresponding component interfaces.

In our ongoing field study, the monitoring overhead was reported to be very small if only the major platform services (e.g., ≤ 50) are instrumented.

Maintainability Our diagnosis approach requires an instrumentation for monitoring response times and control flow of the system to be diagnosed. Maintainability is reduced if the monitoring code is manually integrated and mixed with the source code of the business logic, because this reduces source code readability. A less intrusive alternative is Aspect-Oriented Programming (AOP), since monitoring is a so-called cross-cutting concern, i.e., it is typically used at many places within a software. An AOP implementation can reduce the required integration steps to only minimal changes (in our case 1–2 lines for each monitoring point) in the application source code, software build scripts, and application server configuration in order to enable monitoring.

Anomaly Detector Requirements The quality of RanCorr's results depends on the quality of the results provided by the anomaly detector. Anomaly detection is usually subject to false positives (false alarms). It is expected that a certain amount of false positives can be tolerated by the anomaly correlator, as long as the false positives are approximately equally distributed among all architectural elements. Obviously, only faults can be localized that cause anomalies that can be detected by the anomaly detector in use.

Reconstruction of Architecture Models from Monitoring Data The calling dependency graphs (CDGs) used in our approach are completely reconstructed based on monitoring data. An alternative is given by static analysis tools that reconstruct such graphs based on source code analysis. CDGs reconstructed from monitoring data may be incomplete as not all possible interactions among architectural elements are represented by edges in the graph, while static source code analysis identifies all possibilities for call actions. In the context of failure diagnosis, reconstruction from monitoring data is more suitable: It is reasonable to assume that if no interaction between two architectural elements takes place during the time period before a failure, then there is also no error- and anomaly propagation between those elements. A more detailed discussion of this issue is provided by Gupta et al. [7].

7 Related Work

Agarwal et al. [1] evaluate response times of internal components, similar to our approach. Anomalies result from comparing the average response times of all operation calls that can be connected to recent end-to-end SLA (service level agreement) violations based on historical average response times. In other words, the anomaly scores quantify the recent shift in the average response times (of a subset of the operations). In contrast to our approach, the existence of end-to-end service level agreements is assumed. Their correlation uses a clustering algorithm combined with a ranking logic to sort the suspicious components.

Yilmaz et al. [16] present a fault localization approach that compares method execution times to learned timing behavior profiles. The so-called Time Will Tell approach creates for each method a Gaussian Mixture Model (GMM), i.e., a multi-dimensional probability density model, where in this case the dimensions are given by execution times of sub-called methods. Execution time observations are evaluated in the context of the normal timing behavior of its subcalls. The Time Will Tell approach does not explicitly correlate anomaly scores and performs a ranking of aggregated anomaly scores similar to the trivial algorithm presented in this paper.

Kiciman and Fox [8] describe how component interactions and control-flow path shapes can be observed and analyzed to detect and localize anomalies. During a training phase, a reference model is learned from monitoring data. For anomaly detection, the current system behavior is compared to the reference model. Kiciman and Fox [8]'s approach focuses on component interaction anomalies (control-flow), while our focus is on timing behavior anomalies. In contrast to our approach, it is required that some requests are known to be failed. As in our approach, the correlation step uses interaction paths to invert propagation effects.

8 Conclusions and Future work

Due to their complexity, large-software systems are practically never free of faults, and manual failure diagnosis is time-consuming and error-prone. Thus, the automation of this task is desirable. Automatic failure diagnosis can be divided into two steps: the computation of component anomaly scores, and the global correlation of anomaly scores for fault localization.

Appeared in A. Winter, R. Ferenc, and J. Knodel (eds.), Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009), pages 47–57. IEEE. March 2009.

In this paper, we presented the anomaly correlation approach RanCorr for supporting failure diagnosis. Anomaly scores for operation executions are correlated based on architectural calling dependency graphs and a rule set to address error propagation. RanCorr computes anomaly ratings on operation-, component-, and deployment context level, and visualizes the diagnosis results in architectural diagrams. All steps of the approach are completely automatic: monitoring, initialization of the anomaly detector, derivation of architecture dependency graphs from monitoring data, and failure diagnosis.

In a case study, RanCorr was applied to a distributed Java Web application, subject to fault injection of different types and severity, and exposed to probabilistic workload. The correlation performed by RanCorr is based on anomaly scores provided by the timing behavior anomaly detector [11, 12]. The results of the three algorithms were analyzed with respect to accuracy and clearness. Moreover, we presented results analyzing the influence of the algorithm parameters on the correlation quality.

The results of the evaluation show that the presented algorithms are able to perform significantly better than a plain aggregation of anomaly scores. The case study provides additional empirical support for the claim that timing behavior anomalies can be a valuable indicator in failure diagnosis. Therefore, timing behavior based failure diagnosis can be considered an efficient complement to other failure diagnosis methods that focus on the observation of other system characteristics.

Future work consists of evaluation in the field. Two main long-term evaluation goals are 1) to quantify failure diagnosis accuracy and clearness for real faults in real systems and 2) evaluating benefits from the visualization approach used, i.e., presenting anomaly ratings on three architecture levels of different granularity, in the context of maintenance with human administrators and developers.

References

[1] M. K. Agarwal, K. Appleby, M. Gupta, G. Kar, A. Neogi, and A. Sailer. Problem determination using dependency graphs and run-time behavior models. In A. Sahai and F. Wu, editors, *Proceedings of the 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2004)*, volume 3278 of *LNCS*, pages 171–182. Springer, Nov. 2004.

[2] A. Avižienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[3] R. M. Bailey and R. C. Soucy. Performance and availability measurement of the IBM information network. *IBM Systems Journal*, 22(4):404–416, 1983.

[4] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 342–351. ACM Press, May 2005.

[5] A. Diaconescu and J. Murphy. Automating the performance management of component-based enterprise systems through the use of redundancy. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*, pages 44–53. ACM, 2005.

[6] B. Gruschke. A new approach for event correlation based on dependency graphs. In *Proceedings of the 5th Workshop of the OpenView University Association*, Apr. 1998.

[7] M. Gupta, A. Neogi, M. K. Agarwal, and G. Kar. Discovering dynamic dependencies in enterprise environments for problem determination. In *Proceedings of the 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'03)*, volume 2867 of *LNCS*, pages 221–233. Springer, Oct. 2003.

[8] E. Kiciman and A. Fox. Detecting application-level failures in component-based internet services. *IEEE Transactions on Neural Networks: Special Issue on Adaptive Learning Systems in Communication Networks*, 16(5):1027–1041, Sept. 2005.

[9] P. Küng and H. Krause. Why do software applications fail and what can software engineers do about it? A case study. In *Proceedings of the IRMA Conference: Managing Worldwide Operations and Communications with Information Technology*, pages 319–322. IGI, 2007.

[10] Object Management Group (OMG). Unified Modeling Language: Superstructure Version 2.1.1, Feb. 2007.

[11] M. Rohr. *Workload-sensitive Timing Behavior Anomaly Detection for Automatic Software Fault Localization*. PhD thesis, Department of Computing Science, University of Oldenburg, Oldenburg, Germany, 2009. (work in progress).

[12] M. Rohr, S. Giesecke, and W. Hasselbring. Timing Behavior Anomaly Detection in Enterprise Information Systems. In J. Cardoso, J. Cordeiro, and J. Filipe, editors, *Proceedings of the Ninth International Conference on Enterprise Information Systems (ICEIS'07)*, pages 494–497. INSTICC Press, June 2007.

[13] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stoever, S. Giesecke, and W. Hasselbring. Kieker: Continuous monitoring and on demand visualization of Java software behavior. In *Proceedings of the IASTED International Conference on Software Engineering 2008 (SE 2008)*, pages 80–85. ACTA Press, Feb. 2008.

[14] M. Steinder and A. S. Sethi. The present and future of event correlation: A need for end-to-end service fault localization. In *Proceedings of IIIS SCI World Multi-Conference on Systemics, Cybernetics and Informatics 2001, Vol. XII*, pages 124–129, 2001.

[15] A. van Hoorn, M. Rohr, and W. Hasselbring. Generating probabilistic and intensity-varying workload for Web-based software systems. In *Proceedings of the SPEC International Performance Evaluation Workshop 2008 (SIPEW '08)*, volume 5119 of *LNCS*, pages 124–143. Springer, June 2008.

[16] C. Yilmaz, A. Paradkar, and C. Williams. Time Will Tell: Fault localization using time spectra. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pages 81–90. ACM, May 2008.