Decision Support via Automated Metric Comparison for the Palladio-based Performance Blame Analysis

Frank Brüseke
s-lab – Software Quality Lab
University of Paderborn
Paderborn, Germany
fbrueseke@s-lab.upb.de

Gregor Engels
s-lab – Software Quality Lab
University of Paderborn
Paderborn, Germany
engels@s-lab.upb.de

Steffen Becker
Heinz Nixdorf Institute
University of Paderborn
Paderborn, Germany
steffen.becker@upb.de

ABSTRACT

When developing component-based systems, we incorporate third-party black-box components. For each component, performance contracts have been specified by their developers. If errors occur when testing the system built from these components, it is very important to find out whether components violate their performance contracts or whether the composition itself is faulty. This task is called performance blame analysis. In our previous work we presented a performance blame analysis approach that blames components based on a comparison of response time values from the failed test case to expected values derived from the performance contract. In that approach, the system architect needs to manually assess if the test data series shows faster or slower response times than the data derived from the contract. This is laborious as the system architect has to do this for each component operation. In this paper we present an automated comparison of each pair of data series as decision support. In contrast to our work, other approaches do not achieve fully automated decision support, because they do not incorporate sophisticated contracts. We exemplify our performance blame analysis including the automated decision support using the "Common Component Modeling Example" (CoCoME) benchmark.

Categories and Subject Descriptors

D.2.5 Testing and Debugging, D.2.8 Metrics, G.3 Probability and Statistics

Keywords

Performance blame analysis, CBSE, data series comparison, performance prediction, performance test

1. INTRODUCTION

In component-based software engineering, software architects develop systems by composing third-party black-box components. For each component, functional and non-functional component contracts have been specified by their developers. After composing all components, software architects test the composition in test cases before shipping the system to its end-users. We focus here on a subset of these test cases, which test the fulfillment of the system's performance. If the architect discovers errors while executing such a performance test case, she has to investigate in a so-called performance blame analysis activity whether components violate their performance contracts or whether the composition itself is faulty.

In order to blame components, architects face the problem to identify components violating their performance contract. To tackle this problem, we have proposed a performance blame analysis process in previous work [2] (c.f. Figure 1). It is based on the collection of measured component performance metrics from the failed performance test case and expected performance metrics derived from the component's performance contract (Step 1). Performance contracts are formalized using the Palladio Component Model (PCM) [1]. If the measured performance metrics from the performance test case violate their performance contract, architects must blame the respective component (Step 3). However, in order to compare the measurements to the specification, the architect currently has to compare both performance metrics of each component. The performance metrics are either represented as complex raw data sets or statistical characterizations. This comparison is tedious and error prone. In this paper, we add a novel automated decision support step (Step 2) to our process to speed up the performance blame analysis.

Existing semi-automated decision support approaches for component-based performance blame analysis are limited. If they use components without performance contracts, decision support is restricted to data aggregation only leaving the tedious comparison task to the system architect. In case they do support component performance contracts, these contracts are not parameterized by the component's context (e.g. [15, 17]). The context of a component is defined as the component's usage, its connected external services, and its allocation on execution environments. As the context impacts a component's performance, approaches for non-

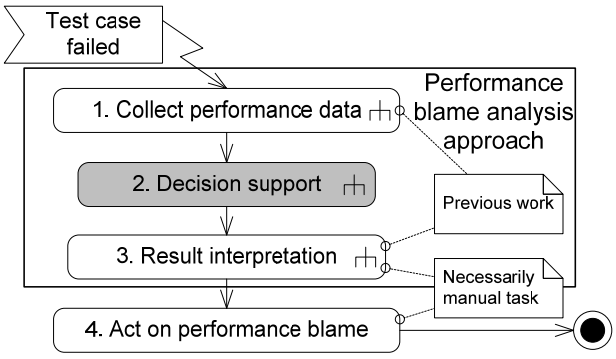

Figure 1: UML 2 activity diagram showing the steps in our previous performance blame analysis approach [2] plus the novel decision support step (inside the box)

ICPE'13, March 21–24, 2013, Prague, Czech Republic.

parameterized component performance contracts have to replace missing context information by manual or heuristically generated specifications. Our approach is the first to explicitly consider context influences. By exploiting the parameterized component performance contracts, we have achieved a sophisticated automated comparison of performance contracts and performance measurements from failed test cases.

In this paper, we enhance our existing blame analysis process by a novel automated decision support step to drastically reduce the manual analysis effort. In order to automate the decision support step, we have derived a set of statistical indicators that can be used in an automated process to identify components with faulty performance. Our tool computes these indicators for the data collected during the execution of the performance test case and the performance metrics derived from a component's performance contract. Computed indicators are mapped to severity levels which direct the system architect to components violating their contractually specified performance. We include these indicators into a novel visualization based on flame graphs [6]. Our extended flame graphs use a coloring scheme based on the indicated severity to highlight contract violations.

We have validated the novel decision support step on two variants of the Common Component Modeling Example (CoCoME) [9]. This system has been designed as a benchmark for component-based analysis methods. It implements the supply chain management of a supermarket chain. Our evaluation shows that the automated decision support step helps the software architect to easily blame the appropriate components. The blamed components' response time during the test case execution violates the expected response time specification. We claim that our visualization is a concise and easy-to-grasp tool for software architects.

The contribution of this paper is an extension to our performance blame analysis process, which semi-automatically identifies components to blame. It uses a set of derived statistical indicators to feed a novel visualization for blame analysis support. We provide a case study based on CoCoME giving evidence for the effectiveness of our approach.

The remainder of the paper is structured as follows. Section 2 introduces a running example that is used throughout the paper. Section 3 revisits our previous work, the Palladio-based blame analysis for component-based systems. Section 4 derives the automated decision support and introduces indicators and visualizations. Moreover, it also discusses necessary changes to the result interpretation. Section 5 explains the automated decision support and result interpretation in detail. It exemplifies these new elements in detail using the running example. Section 6 describes the limitations of our approach. Section 7 discusses related work from the areas of performance visualization and performance blame analysis. Lastly, Section 8 concludes the paper and gives an outlook on our future work.

2. RUNNING EXAMPLE

Throughout this paper we use a running example. The running example is used to exemplify certain points in the reasoning of this paper. An in-depth evaluation of this example follows in Section 5.

The scenario of the example is the most sophisticated use case of the Common Component Modeling Example (CoCoME) [9]. The CoCoME was specified as a benchmark system to compare different approaches for component-based software architecture analysis, such as performance predictions. It specifies a trading system for a supermarket chain that was inspired by a real example. We refer to the trading system as the CoCoME-system. The CoCoME-system includes the whole infrastructure from the cash desks, where items are sold to customers, to the store servers registering those sales and to an enterprise server aggregating all the information. This system structure of the CoCoME is shown in Figure 2.

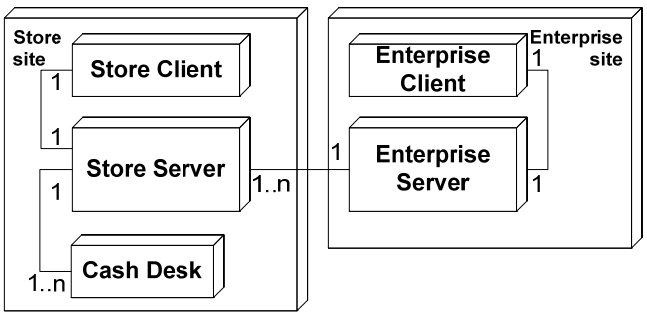

Figure 2: System structure of the CoCoME trading system

The CoCoME architecture is an archetype of a whole class of systems. While the cash desks resemble the embedded system part of the overall system the store and enterprise servers make up the information system part. In this class of systems the data is produced by distributed use of embedded system devices. All data are then stored and processed in the overlaying information system.

This paper focuses on the store server and enterprise server of the CoCoME-system. These make up the information system part of the trading system. In this setting, we investigate the scenario that solely consists of the use case "inter-store exchange" for shipping items among the stores of an enterprise. This use case is the most sophisticated use case of the CoCoME. It is triggered by each sale. A store server checks if the items' stock falls below a minimal threshold. If so, it triggers the enterprise server to cause near-by stores to send some of the missing items to the store in need. It then optimizes the transports such that no store runs out of the transported items and the overall distribution distance is minimal.

Performance blame analysis, as outlined in our previous work [2], deals with the response times of component operations. Figure 3 outlines the calls of the component operation in our running example. Figure 3 shows the participating component objects (meaning an instance of a deployed component; cf. Cheesman and Daniels [3]) being deployed on several nodes. The enterprise server and several store servers participate in the scenario. The solid arrows along the sockets and interfaces represent component operation calls and the dashed arrows stand for operation returns. These messages are numbered in the order of their execution. Since most return messages are given directly after the corresponding operation call these return messages are not numbered.

The scenario in question starts with a store server (node on top in Figure 3) registering a sale (message 1). The origin of message 1 is a test driver component (cf. Figure 3). The registered sale lowers the store's stock, such that it crosses the minimal threshold for the item stock. The store then finds all missing products (message 2) and communicates to the enterprise server (message 3) to initiate a transport that replenishes the store's stock for these products. The `ProductDispatcher` component first determines the relevant enterprise information (message 4). The enterprise information also contains references that enable the `ProductDispatcher` to contact all stores. The `ProductDispatcher` queries the stock information for each missing product in all the other stores (messages 5). Messages crossing the border to the multi-node "Other store servers" stand

for several messages. One message is sent to each store node in consideration. The enterprise server then creates an optimized transport plan and lowers the stock for the transported products in those stores that need to send products (messages 6). The enterprise server then returns the amount of transported products back to the initial store server (message 7). Finally, the store then saves the products as incoming.

3. PREVIOUS WORK

The approach presented in this paper improves our previously introduced performance blame analysis approach [2]. Our previous work is based on comparing performance metrics from testing with performance metrics derived from performance contracts. In our previous work and also in this paper, we exclusively deal with the performance metric response time. The response times are derived from the performance contracts that the component developers supply in the form of the Palladio Component Model (PCM) [1]. This partial PCM model can then be incorporated into a system model by the system architect. The first step of our approach (cf. Figure 1) is then the collection of response time measurements from testing and from the PCM system model for a failing test case. To derive expected values from the PCM system model, the system architect uses performance prediction, i.e. the PCM simulation. Next, these two data series are used to produce a histogram. The system architect shall use the histogram to interpret (cf. step 3 in Figure 1) if a particular component operation violates its contract and therefore needs to be blamed. The system architect decides this by judging if the test data series has overall higher response time values than the data series of expected values.

Our blame analysis approach relies on several assumptions. First, the test case and the performance analysis scenario must be equivalent. Our approach does not guarantee this equivalence, but we have suggested including PCM models in the test case specification [2]. Then, the system architect can draw test cases and PCM models for performance prediction from the same source, which implies equivalence. Second, the system architects need PCM contracts that adequately reflect the component developer's intentions for the performance of each component. The system architect must then correctly assemble a system model containing those PCM contracts. Moreover, the hardware usages included in the PCM contracts must be modeled in a standardized way. Combined with the hardware definitions used in the project they must result into response time values that can be easily compared to the measurements acquired in testing.

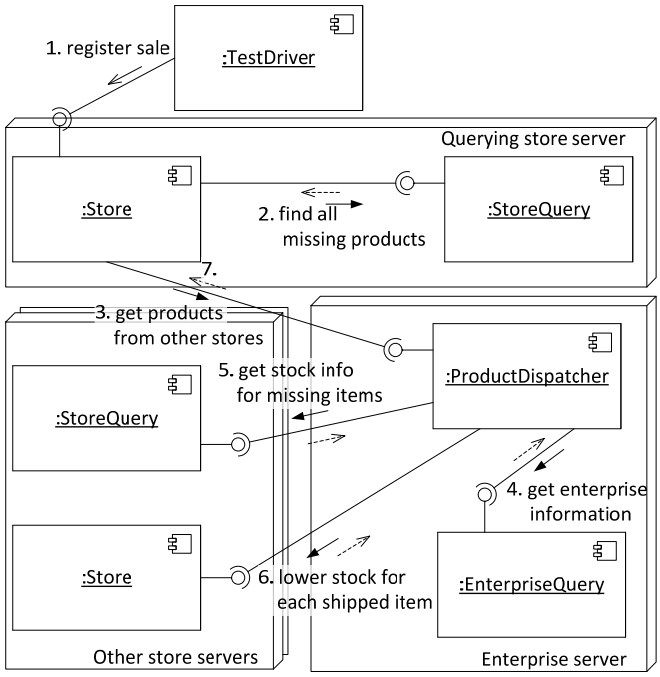

Figure 3: Scenario with all participating components and their exchanged messages

The following subsections elaborate more on the data collection and result interpretation of our previous work. Figure 4 gives a more detailed overview of the performance blame analysis approach. The data collection parts for testing and performance prediction as well as the result interpretation are introduced in turn. Subsection 3.1 introduces the test part of data collection while Subsection 3.2 deals with the performance prediction part. The result interpretation is covered in Subsection 3.3. The result interpretation is more explicitly explained, because this paper introduces an automated decision support speeding up result interpretation.

3.1 Data Collection – Test

As stated before, blame analysis always starts with a failing test case[1]. The system architect collects the performance measurements for the scenario that are included in the test case results. She checks if the data taken while executing the test case are sufficient. For our blame analysis approach the system architect

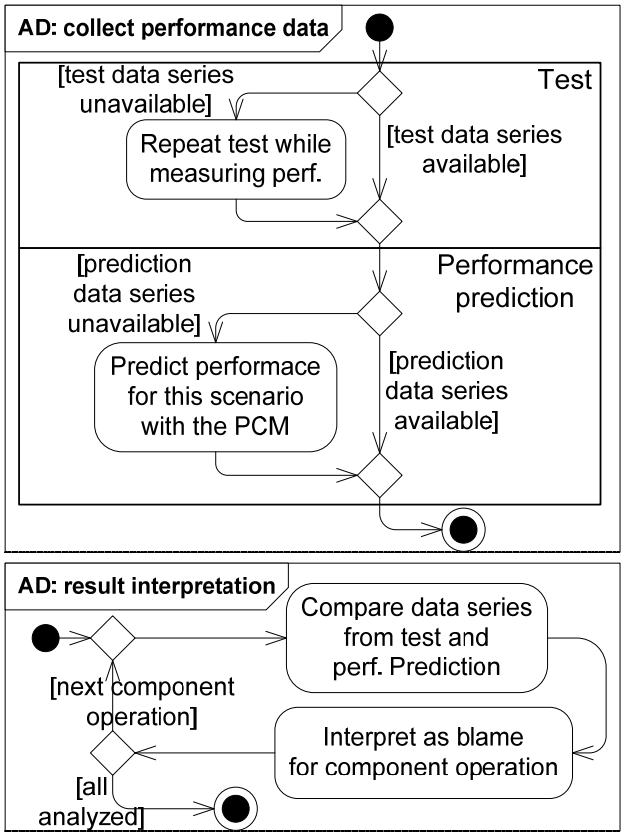

Figure 4: Previous performance blame analysis process [2]

[1] Please note that we assume that failing test cases can be identified with the help of detailed requirements, which specify the expected performance at the system boundary.

must collect the response times for all component operations that are involved in the test case. In our running example she must measure the response time of each operation call indicated in Figure 3. For message 4 "get enterprise information" she has to measure the response time of the call to the operation `queryEnterpriseByID()` in the component `EnterpriseQuery`. If all the needed response time measurements were collected, the system architect proceeds, otherwise she must repeat the test case and take the necessary measurements while doing so.

3.2 Data Collection – Performance Prediction

In our approach, the system architect utilizes the Palladio Component Model (PCM) [1] for performance prediction. We have chosen the PCM for this because of its role model. The role model has an explicit role for the component developer. It specifies which diagrams and model contents have to be delivered by the component developer. Moreover, it states how these parameterized PCM models have to be completed by the system architect, such that it reflects a specific system with specific usage scenarios. So, the component developers are able to specify their components' performance contracts, such that the specification is usable in different systems and for different use cases. This makes PCM performance specifications suited for exchanging between component developers and system architect.

In a full PCM model all parameters have been subsequently filled in by the various roles creating the model [1]. This PCM model explicitly covers the components' context. I.e. it models the system composition including external services, the allocation and the system usage. The usage model specifies the workload produced by virtual users. It includes frequency and order of calls as well as optional parameters like think time and input characterizations (e. g. value or number of elements).

The system architect must supply predicted response times for the failed test case by analyzing such a full PCM model. The system architect checks if they already have matching data series from performance analysis during the design time (cf. Figure 4). If they do not have predicted the performance for this environment setting and scenario yet, they must do so now. The system architect needs the predicted response times for any component operation participating in this scenario. In our running example the system architect must supply a response time data series for each of the operation calls in Figure 3, just like in testing.

For proper comparison, the performance data series from testing on one hand and from performance prediction on the other hand need to stem from the same scenario. In particular, the input workload must be the same performance-wise, and also the deployment must be the same.

3.3 Result Interpretation

Finally, the system architect can proceed to result interpretation (cf. Figure 4). In our previous work, the interpretation of the measured performance metrics from test and performance prediction is manual. To decide if a component operation needs to be blamed, the system architect has to compare the two response time data series. Then, the system architect has to decide whether the component operation exhibited higher or lower response time values than predicted. In our previous work [2], we have suggested that the system architect creates histograms with bins of the same size (e.g. 125 ms each). Each histogram visualizes the relative frequencies of the response time data series from testing and performance prediction side by side.

Figure 5 depicts an example histogram showing the response time distributions for the component operation `queryLowStockItems` of the component `StoreQuery`. This operation call corresponds to message 2 in Figure 3. The histogram in Figure 5 shows the relative frequency of the response time values from testing and the expected response time values from performance prediction. The X-axis exhibits the response time (in ms) and the Y-axis depicts the relative frequency how often this response time occurred. Figure 5 shows the histogram bins not as bars. This is also correct, because all the bins are of the same width, i.e. 125 ms wide. The curves for testing (blue) and performance prediction (red) each have only one peak. So, in this case it is clear that the test response time values are overall lower than predicted. However, the decision is not always trivial. In our experience the decision is hard to make, when the curves intersect more often. Also, if the blue curve had a different shape and had far more frequent occurrences of the response time around 1000 ms, the decision would be quite hard to make. Recognizing that an operation has overall slightly lower or higher response time values only by looking at a histogram such as this is quite hard.

The system architect has to construct and interpret one response time histogram, as shown in Figure 5, for each component operation. In our running example, the system architect needs to analyze 6 different component operation calls (cf. Figure 3). Thus, the comparison step is quite time consuming and tedious. This paper shows how an automated decision support step can look like.

4. DERIVING AN AUTOMATED DECISION SUPPORT SUB-PROCESS

This section discusses what indicators and visualizations are suitable for the automated decision support introduced in this paper. Subsection 4.1 discusses statistical indicators that can decide whether the response time data series from the test has higher values than the response time data series from performance prediction. Subsection 4.2 presents visualizations for these statistical indicators. Subsection 4.3 introduces a decision support subprocess that combines the indicators and visualizations. It also deals with how the obtained results of the decision support need to be interpreted by the system architect to identify the blamed component operations.

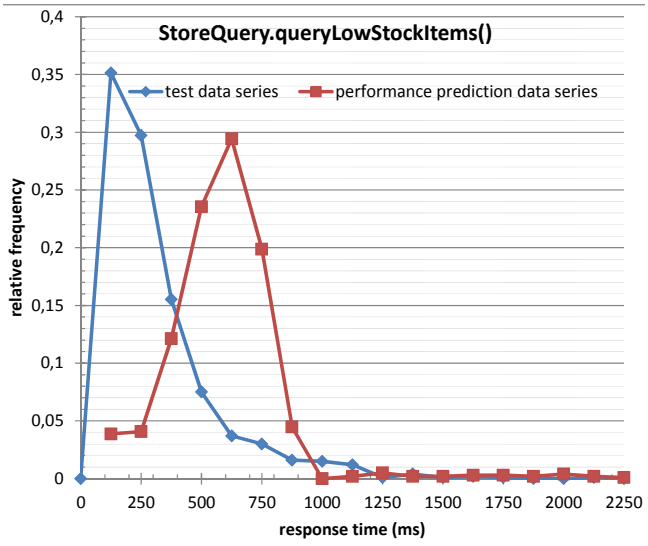

Figure 5: response time chart for manual comparison

4.1 Decision Criteria for an Automated Decision

This subsection develops automatable decision criteria in terms of indicators which help to decide whether the values of one response time data series exceeds the values of another response time data series. The following questions are relevant with respect to the decision criteria:

- Which indicators are suited to automatically decide whether a component operation is to be blamed or not?
- Which of these indicators are most robust (i.e. give the correct result in most/all cases)?

The questions also imply requirements on a suitable indicator or decision criterion: 1.) it must work with response time data series; 2.) it must be able to decide whether one data series has lower or higher values than another one; 3.) the decision must be correct in as many cases as possible. The first two requirements can be checked by consulting the definition of the indicator. This subsection discusses only indicators that match these two requirements. The third and most important requirement can practically only be decided using example data series. We have used data series from our experiments with CoCoME. Our test set included 20 pairs of data series. For each pair of data series, we manually assessed whether the test data series exhibits higher, lower, or about equal values than the performance prediction data series. To decide whether an indicator is suited, we tested whether it decides the same way that we have decided manually. The more often the indicator decides as we would have done, the better it is suited for our performance blame analysis approach. We used this benchmark for all the indicators (point estimators and statistical tests) described in this subsection. We use this example-based benchmark to derive a heuristic that works well for CoCoME and likely also applications similar to CoCoME.

The most obvious comparison indicators for data series are point estimators. Point estimators summarize a data series with a single numerical value. Two data series can then be compared by contrasting the point estimator value for each data series. The most prominent point estimators are minimum, maximum, mean, and the quartiles (i.e. 25%-quartile, the median and the 75%-quartile) [4, 10]. We have tried all these indicators to compare the data series in our test sample. The minimum and maximum are the two point estimators whose decision differed from the manual decisions most often. The minimum had seven and the maximum three false indications. The mean differed less from the manual decisions with only two wrong decisions in close cases. In one example where the mean fails, the test data series has a long tail, i.e. it has a lot of outliers near the maximum. The outliers caused the mean to fail, because in that case the mean and median differed significantly. The 25%-quartile, the median and the 75%-quartile performed best amongst all point estimators. They only failed in one case.

While some point estimators already qualify as comparison indicators, we looked for better indicators in statistical testing. The Kolmogorov-Smirnov-Test (KS-Test) [16] is a test that quantifies whether one data series generally has high or low values. There are two versions of the KS-Test that test if one data series has higher or lower values than another one, respectively. To the best of our knowledge the KS-Test is the only test that can test on lower and higher values. Other statistical test can only test equality (e.g. the Chi^2-test [4, 16]).

We use the "less"- and the "greater"-variant of the KS-Test (as implemented in R [14]). We investigate the hypotheses whether the test data series exhibits *lower* response time measurements than the predicted data series and whether test data series has *higher* measurements. As a consequence, we gather more information than when using a single test. In a clear case we can reject one of the two hypotheses with confidence. When the two data series are very close to one another, both tests will show the same result indicating that the KS-Test cannot tell them apart. This additional information enabled the KS-Test to outperform the point estimators. The KS-Test decided according to our manual decisions for the test data series. However, in two cases it made no decision, i.e. both KS-Tests indicated that their hypothesis could be rejected. Manually, we were able to decide one of the two cases, but could not do so for the other. As a result, the KS-Test was undecided in one case where a decision was possible. Thus, it did a little better than the quartiles, which made one actually wrong decision.

Altogether, our experiments suggest that the three point-estimators 25%-quartile, median, and 75%-quartile are reliable comparison indicators. The KS-Test seems to the best comparison indicator. It does not only decide, but it also indicates cases in which a decision is hard and which therefore need human intervention.

4.2 Visualizations for the Decision Criteria

Subsection 4.1 identified the 25%-quartiles, median, 75%-quartile, and the KS-Test as good indicators for performance blame analysis. Now, we look for a helpful visualization of these indicators. Most importantly, the visualization needs to be easily interpretable in terms of blaming. If possible, the visualizations shall not only present the indicator in question, but also other relevant data. Other relevant data confers to data that quantifies performance or that reflects influences on performance. For response time data, an important supplement is the call tree, the deployment and architecture, and the input data.

A box-and-whiskers-plot [10] can visualize the quartiles as well as the minimum and maximum. Showing two box-and-whiskers-plots (one for each response time data series) in one chart helps comparing whether the values of one of the data series exceed the values of the other one. This comparison can be done by comparing the position of the boxes of the data series. Each box stands for 50% of the measurements. So, if one box is drawn above the other, the data series likely has the higher values. As the quartiles only serve as a secondary indicator, we did not try to couple the quartiles with other relevant data.

An execution of the two variants of the KS-Test gives one of three possible results: 1.) the test data series shows lower response times; 2.) the predicted data series shows lower response times; or 3.) the data series are so close the test cannot decide clearly. These three result types stand for three different severity levels. Having three different severity levels suggests using a "traffic light indicator". Green indicates that the test data series has the lower values, yellow indicates that the KS-Test is unable to decide, and red indicates that the predicted data series has the lower response times. Such a traffic light can visualize the KS-Test outcome for each component operation that contributed in the test case. This traffic light coloring cannot only be used on its own, but it can also be incorporated into other visualizations that cover all the component operations participating in the test case.

Profilers are well matured performance analysis tools. Profilers often visualize caller/callee relationships of operations along with operation response time (cf. Section 7). This data combination is the main data that profilers present for analyzing the response times. We take this as evidence that this data combination can be seen as "other relevant data".

Brendan Gregg [6] presented his "Flame Graphs" which are a variation of the usual profiler charts. This particular variant shows

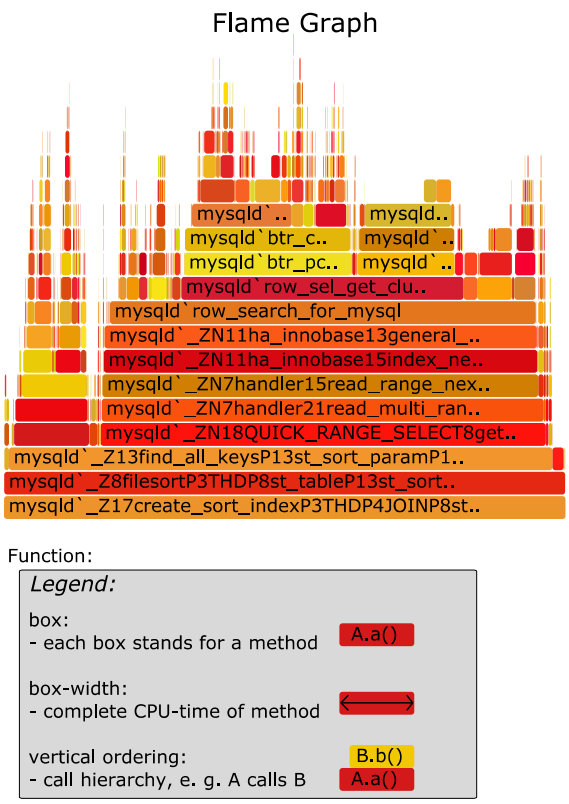

Figure 6: sample "Flame Graph" [6] with additional legend

the mentioned information in a compact form. Figure 6 shows an excerpt from a sample Flame Graph [6] presenting data measured in a MySQL run. A legend has been added in Figure 6 to give additional explanations. The chart shows the CPU-time of functions in MySQL. Each function is represented by a box. The width of the box shows the complete CPU time of the function during the test. The vertical ordering in the chart stands for the call hierarchy. A calls B if the box representing B is drawn on top of the box representing A. The width of a box that is not covered by a box on top of it represents the CPU time consumed by computations inside this function. In the original Flame Graphs the box colors have no meaning. Also the positioning of the boxes from left to right conveys no meaning.

The compact representation of the Flame Graphs requires to omit some information. The names of the functions (or operations) and the exact number for the CPU time cannot be shown in each box (cf. Figure 6). The Flame Graphs circumvent this limitation using interactivity. Flame Graphs are interactive SVG charts. The missing information for a particular box is shown next to the "Function" text at the bottom of Figure 6, when moving the mouse cursor over that box.

We see the call hierarchy in combination with a response time metric as performance relevant information. The Flame Graphs show these metrics. They have the advantage of being compact, when compared with other usual profiler visualizations (cf. Section 7). Moreover, we have found that the KS-Test result can be sensibly visualized using traffic light colors. Until now, the colors in the Flame Graphs have not conveyed meaning. As a consequence, we can adapt Flame Graphs with the KS-Test traffic light coloring as box colors. Moreover, the box width can represent the median response time of the component operation in the test case. It showed that the median is indicative for our performance blame analysis (cf. Subsection 4.1). This way, we get a compact diagram that includes the KS-Test results as well as the median response time and call hierarchy. We call this adapted version of the graph "Blame Graph".

In this subsection, we have introduced the Blame Graph that couples the KS-Test result in the form of traffic light colors with the average response time for each component operation and the position in the call tree. The KS-Test results are our prime decision criterion and are also coupled with other relevant decision criteria. Next, we visualize the quartiles using box-and-whiskers-plots. We have not coupled this secondary indicator with other relevant data.

4.3 Incorporating Decision Support

This subsection presents our decision support sub-process that incorporates the indicators and visualizations that the subsections 4.1 and 4.2 have introduced. Moreover, this subsection discusses how the result interpretation is affected by introducing automated decision support into the performance blame analysis.

The starting point of the decision support sub-process is that the system architect has response time data series from testing and performance prediction available for the test case in question. Figure 7 shows the additional decision support process. It is split in two parts. On the left hand side, Figure 7 shows the action that the system architect needs to perform and on the right hand side Figure 7 shows steps that are automatically computed. We have implemented a Python script that uses the R language [14] for statistical computations.

First, the system architect needs to prepare the computation by mapping component operations to PCM entities. Then the system architect gives conversion factors for the test and the predicted data series. These factors will be used to convert the response time data series to a common time unit. After that, computation begins. The computation takes the configuration mentioned before as well as the various response time data series as input and produces two different outcomes. First, it produces the Blame Graph which summarizes all the component operations

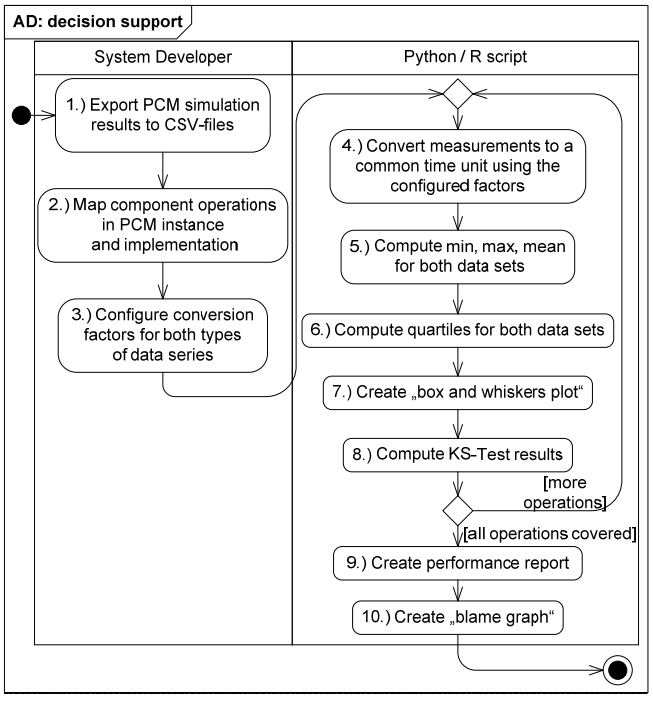

Figure 7: the additional decision support sub-process

involved in the test case in one chart. This chart shows the median response time for each component operation in context of the call hierarchy. We chose the median, as it turned out to be indicative for performance blame analysis (cf. Subsection 4.1).

Moreover, the computation produces a performance report that consists of two sheets for each of the component operations participating in the test case. One sheet shows a box-and-whiskers-plot for the two response time series associated with it, i.e. the data series from test and performance prediction. The next sheet exhibits all the point estimators for the two data series as well as the KS-Test results. The point estimators covered in the performance report are the quartiles, the minimum, the maximum, and the mean value.

The system architect must then look at the blame graph and the performance report to find blamed component operations. Figure 8 shows the updated result interpretation sub-process. In contrast to our previous work (cf. lower part of Figure 4) it relies on the Blame Graph and the performance report. The Blame Graph shows all the component operations in one sheet. All blamed operations are drawn in red. In addition, the system architect can also prioritize the blamed operations by call hierarchy (vertical box order) and median response time (box width). So, the blame graph supplies the system architect with three different relevant decision criteria. The system architect can then look into the performance report for more detailed figures about each component operation. If the system architect is still in doubt whether or not to blame the component operation in question, the system architect should of course make her own analyses.

5. EXEMPLIFIED AUTOMATED DECISION SUPPORT

This section exemplifies the automated decision support as depicted in Figure 7. It covers the configuration by the system architect as well as the automated analysis. This analysis results in the Blame Graph and the performance report. Next, this section discusses how to interpret these two artifacts in terms of blaming (cf. Figure 8). Also, it illustrates how each step of the decision support and the result interpretation has been performed in our running example (cf. Section 2) and its results. Please note that you can find all example data and analysis results along with our analysis scripts on this paper's companion website[2].

Subsection 5.1 first introduces the implementations and the test bed that have been used in the case studies with both CoCoME implementations. This includes the instrumentation used to produce the response time measurements. The Subsections 5.2 to 5.5 introduce the automated decision support and the result interpretation showing the results produced with the CoCoME reference implementation. Subsection 5.2 introduces the mapping from the component operations in the implementation to their counterparts in the model (cf. steps 1 and 2 in Figure 7). It also covers the subsequent conversion of the values of the data rows into a common time unit (cf. steps 3 and 4 in Figure 7). Subsection 5.3 presents the performance report and the statistical computations related to it. That subsection covers the steps 5, 6, 7, 8 and 10 in Figure 7. Subsection 5.4 comprises the creation of the Blame Graph and the related statistical computation. It covers the steps 8, 9 and 10 in Figure 7. Subsection 5.5 discusses the result interpretation by the system architect (cf. Figure 8). Finally, Subsection 5.6 covers the lessons learned from the case study with the CoCoME 2 implementation.

[2] http://homepages.uni-paderborn.de/bruesie/icpe2013/

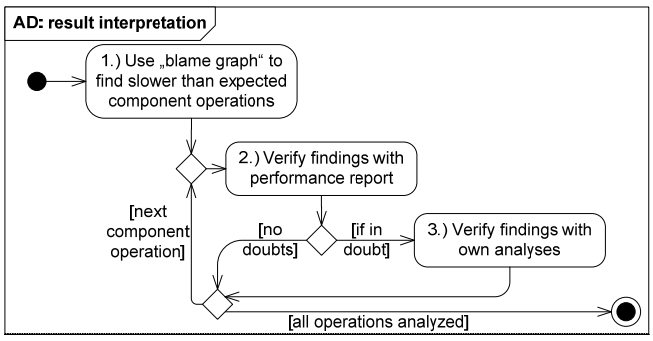

Figure 8: the updated result interpretation sub-process

5.1 Tested Implementation and Test Bed

We have chosen CoCoME as our running example, because it is a good representative of an information system that is fed by data generated by embedded system devices. It consists of several subsystems. The information system that we focus on consists of nine different components. These components are deployed on four different types of nodes, i.e. store server, store client, enterprise client, and enterprise server. Each store servers hosts five component objects and the enterprise server hosts six component objects. The clients are not participating in the test case used in the running example.

The CoCoME was originally released with a reference implementation[3] written in Java. We have fixed some bugs in the reference implementation, such that it can execute the scenario of our running example (cf. Section 2). This fixed CoCoME implementation comprises 6320 lines of Java code. The following refers to this implementation as CoCoME RI. Subsections 5.2 to 5.5 deal with the CoCoME RI case study in detail. Additionally, the original reference implementation has been fixed, improved and released as CoCoME 2[4]. The Karlsruhe Institute of Technology (KIT) created this version for the same reasons which made us develop a fixed version of the reference implementation. The KIT still maintains this version for further usage. This improved implementation comprises 9788 lines of Java code. Subsection 5.6 briefly summarizes the lesson learned in the CoCoME 2 case study.

To test the use case "exchange products among stores", introduced in Section 2, we have adapted the test bed to our needs. We have introduced a load test driver and response time measuring using bytecode instrumentation in both implementations. The instrumented bytecode measures the time between method entrance and return on the server side. This metric also includes possible queuing delays. This is consistent with the performance prediction of the PCM. The measurement bytecode writes this response time to a file and it also includes other important information, namely, the method name, the current time, the current stack trace, and the thread identifier. BTrace[5] supplies the actual bytecode instrumentation, data query, and file writing facilities. We created a short program using BTrace to enable response time measuring. While this custom measurement solution could be more efficient and also could trace cross-process control flow, it suits our purpose. Our measurement solution adds an overhead to

[3] see http://cocome.org/

[4] see http://sdqweb.ipd.kit.edu/wiki/CoCoME2 and http://sourceforge.net/projects/cocome/

[5] see http://kenai.com/projects/btrace

Table 1: mapping between implementation and PCM

Java stack trace	CSV-file from PCM
(9) overall	bookSale0.csv
(8) otherStoreInterchange	orderProductsAvailableAtOtherStores0.csv
(7) markStock	markProductsUnavailableInStock1.csv
(6) solveOptimization	solveOptimization0.csv

the response time of the tested methods. The method directly called by the test driver shows about 34.6 ms (i.e. 6%) slower response times when instrumented. This overhead is tolerable.

The test setup was very simplistic. The test setup was used for either implementation, i.e., CoCoME RI and CoCoME 2. A single dedicated PC (Pentium D 3 GHz single core CPU with two threads, 2 GB RAM, Windows 7 with 64 bit) was used to execute the CoCoME system and also the load test driver. The executed CoCoME system consisted of three store servers and an enterprise server as well as the necessary infrastructure, i.e. the RMI registry, the Active MQ JMS messaging server, and the Apache Derby database included in each CoCoME implementation. The load test driver simulated eight parallel users. Each user queries the same store server 125 times (1000 requests total). This setup should produce a certain amount of contention in the system. However, we did not quantify contention (e.g. by measuring queue length).

5.2 Map Operations and Unit Conversion

First the system architect needs to prepare some configuration data for the Python/R script, which computes the performance report and the Blame Graph. The system architect needs to ensure that there exists a mapping from the component operations in the implementation to their representation in the performance model. The mapping needs to consider each operation and its complete call hierarchy at operation level. The contents of the mapping need to reflect this. The testing side of the mapping consists of all the Java stack traces from the test case that indicate the call of a component operation. On the performance prediction side, the system architect enters the file name of a set of data for this operation invocation. In the PCM, calls to operations are modeled as part of a component's control flow. So, the PCM can distinguish different invocations of the same operation. The PCM's performance prediction yields a separate set of values for each different invocation of an operation.

Because the mapping takes the call hierarchy of a component into account, the mapping needs to be individually constructed for any system. The system architect may use information from the component developers that states what method in the implementation correspond to which model element, but she needs to add information from the component context.

Both the response time measurements from the CoCoME test and also from the PCM need to be represented in a comma-separated values (CSV) file. One line in the CSV file from the test must consist of the response time and the full stack trace. Having both data series in CSV format, we can easily process the data with our analysis.

For better readability, we have compressed the different Java stack traces by numbering them and added a short text for descriptiveness. In the following, we will only exhibit this short form. Table 1 displays an example of the mapping from the compressed Java stack trace to the CSV file with predicted response time values.

As a next step, the system architect needs to convert the response time values from both sources (test and performance prediction) to a common time unit. The system architect gives two conversion factors to the Python/R script, one for the test data series and one for the performance prediction data series. In our running example, the measurements from testing and performance prediction are converted from nanoseconds and seconds to the common unit milliseconds respectively.

The manual steps (cf. steps 1 to 3 in Figure 7) were finished in about one hour for each implementation of CoCoME in the example. These manual steps have not been automated due to the prototype status of our tooling, but are partly automatable. The measurement export and the computation of conversion factors (if the time units are known) can be automated. Creating the mapping can be semi-automated.

5.3 Performance Report

While the Blame Graph is the primary analysis tool, the performance report is the secondary analysis tool that the automated decision support delivers. The performance report gives a detailed overview of each pair of response time data series for all the component operations. It contains the point estimators, the KS-Test results, and a box-and-whiskers-plot for each operation. Specifically, it comprises all computed point estimators: maximum, minimum, quartiles, and mean. In addition, it visualizes the quartiles in a box-and-whiskers-plot that shows the test and performance analysis data series side by side.

After finishing the configuration (cf. Subsections 5.2), the Python/R script computes the point estimators for each component operation. Table 2 lists the point estimators for the method "(8) otherStoreInterchange". The system architect can compare the point estimator values to get an impression, which data series has higher values. In Table 2, all the point estimators in the test data series have higher values than in the performance prediction data series. The Python/R script also draws a box-and-whiskers plot that shows the test and performance analysis response time data series side by side. Figure 9 depicts this box plot. For better readability the box plot does not include the highest 1% of the measurements of each data series, because the test data series has a few

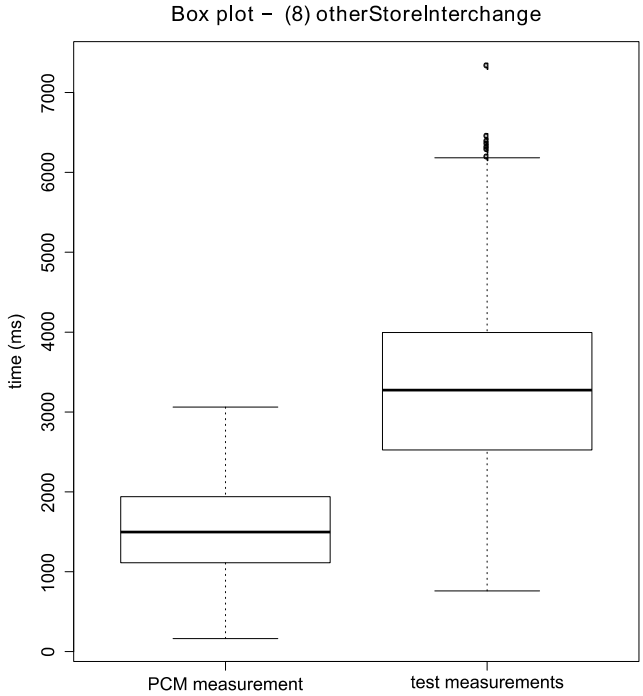

Figure 9:example box-and-whiskers plot

Table 2: point estimators for the operation "(8) otherStoreInterchange"

	PCM – resp. time	Test – resp. time
minimum	163.1 ms	760.7 ms
25%-quartile	1120.8 ms	2528.6 ms
median	1506.2 ms	3279.6 ms
75%-quartile	1952.4 ms	4012.1 ms
maximum	3186.0 ms	24595.4 ms
mean	1500.3 ms	3248.7 ms

outliers that are way higher than most of the measurements. Figure 9 shows that the box for the test data series (which represents 50% of the measurements) lies above the box of the performance prediction data series. This indicates that the values in the test data series are higher than those of the performance prediction data series.

The result of the KS-Test is also shown in the performance report. The report comprises the probability value (p-value [16]) that the "less" and the "greater" variant of the KS-Test result in. Section 5.4 deals with the KS-Test in more detail.

5.4 Blame Graph

The performance report is the secondary indicator for analyzing performance blame, while the Blame Graph is the primary indicator. The Blame Graph depicts operations that are slower than expected as red boxes. The box colors stand for the result of the KS-Test (cf. Subsection 4.3). The KS-Test implementation in R takes both data series as input and results in the p-value [16]. The Python/R script executes the KS-Test and accepts the hypothesis when the p-value is at most 5%. This 5% threshold is a starting value which we will evaluate in future experiments.

The Blame Graph visualizes the results of the KS-tests. It also visualizes the call relationships in the tested implementation and the median response times of each component operation in the test. For the example test case, our script produces the result presented in Figure 10. We added a legend to Figure 10 for better readability. The method names in the Blame Graph are given in terms of the Java implementation of CoCoME. The boxes in Figure 10 are not labeled with fully qualified class names due to the restricted width of the boxes (even the short names may be cut off in small boxes). The fully qualified class name is shown next to the "Function"-text in the original SVG Blame Graph, when moving the mouse over the box representing the operation. In the given Blame Graph the system architect can identify that three methods are blamed, because they are drawn as red boxes. Details on how to interpret the Blame Graph are discussed in Subsection 5.5.

5.5 Result Interpretation

The system architect first looks at the Blame Graph. The system architect identifies the red boxes in the Blame Graph representing the operations that are slower than expected. In our example Blame Graph (cf. Figure 10), there are three blame operations:

- StoreImpl.bookSale(..)
  (also known as "(8) otherStoreInterchange")
- ProductDispatcher.orderProductsAvailableAtOtherStores(..)
- StoreImpl.markProductsUnavailableInStock(..)

Then, the system architect can prioritize the order in which the blamed operations are analyzed by either call hierarchy (vertical box order) or by median response time (box width). For example, the system architect could decide to prioritize the operation "StoreImpl.markProductsUnavailableInStock", as this operation does not call other operations.

The system architect can check the performance report to investigate if the KS-Test has blamed the operations correctly. Using the performance report, the system architect can also prioritize the order to investigate the blamed operations differently. For instance, the system architect could decide to prioritize the component operations according to the difference between the test values and the expected values. If the system architect is still not satisfied, she needs to perform her own analyses.

Compared to the manual process in our previous work, the system architect only needs to configure the Python/R script and can then already interpret the generated results. This saves the system architect the trouble to create the chart by herself. Moreover, the results of the automated decision support can be interpreted easier. The indicators produced by the automated decision support are tailored to judge whether the test data series exhibits the higher values than the performance prediction data series. In our previous work, the system architect needed to judge whether the test data series has the higher values by looking at a specific histogram.

5.6 Case Study: CoCoME 2

While the Subsections 5.2 to 5.5 deal with our running example in the CoCoME RI, this subsection summarizes what we learned in the case study with the CoCoME 2. In this second case study, we could largely confirm what we learned in the CoCoME RI case study. The system architect could employ the automated decision support to get the Blame Graph and the performance report. The Python/R-script quickly produced this result. The

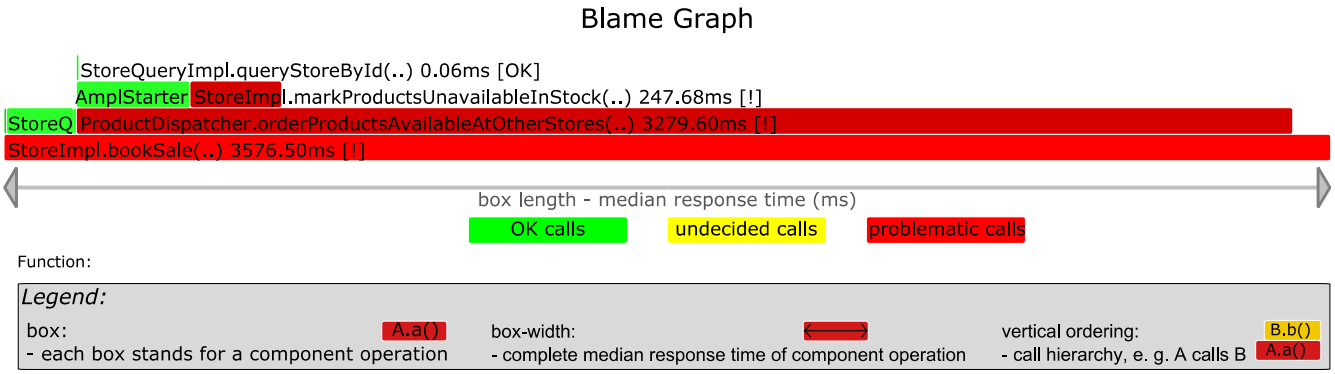

Figure 10: blame graph for the example test case

effort for the analysis was considerably lower than the effort for implementing the test driver and measurement solution.

The Blame Graph for CoCoME 2 gives a quick overview which methods exceed the response time values derived from the PCM contract. But this Blame Graph also shows undecided operation calls that the KS-Test cannot clearly decide. The performance report confirms the results of the Blame Graph, but also shows that one of the undecided results can actually be decided with the performance report (cf. Subsection 4.1). This shows that it is important to also include the performance report as a decision support result. Moreover, the Blame Graph also shows that our current monitoring solution is unable to track requests crossing process boundaries. Operation calls in other processes are drawn just as operations called directly from the test driver.

6. LIMITATIONS

Our approach is currently limited to analyzing the response times of component operations. It can only detect errors in specific component operations. If there are no deviations found for any component operations, our approach assumes that the error is in the composition. However, our approach does not help to detect defects in the architecture, the deployment, etc.

In addition, this approach is based on the differences in the performance contract compared to the actual implementation. If the contracts were correct, comparison between the testing values and expected values from performance prediction will always result in equal values. This is why this approach assumes that the contracts reflect the component developer's intentions of the component's performance.

The approach depends on the quality of the performance contracts that are delivered by the component developers. The component developers could deliver overly simple contracts. Such contracts are only viable in very specific cases and will evaluate to unrealistic expected values in the majority of cases. Moreover, the component developer can betray us with wrong contracts. They could either deliver "safe contracts" or "flattering contracts". "Safe contracts" always result in higher response time values than the values measured in test. This would render this approach useless, but would also lower the chance system architects acquire that component from the market. "Flattering contracts" indicate that the component is performing better than it actually is. Such components are more likely to be acquired from the market, but our approach will also blame such components in case of performance issues. If our approach blames all components, the result is not useful as well. Even when the component developers deliver contracts as intended, the contracts still may not be completely accurate. It is very hard for the component developers to foresee every possible context a component may be used in and to design an according and accurate contract.

The test measurements considered in our analysis are gained with an instrumentation that implies an overhead (cf. Subsection 5.1). This overhead may cause false positives, because it adds to the values in the test data series.

Only measuring response times leaves out certain aspects of the middleware that is used to execute the components. For example the measurements do not differentiate queuing delays and the response time also includes the response time of called components. That means that the blame may travel down the blame graph along the call chain towards the callers. In this case our approach may blame more components than necessary.

The visualization introduced in this paper can also be improved. The Blame Graph does not show process or host boundaries. This is partly due to the monitored data that it visualizes. Our current monitoring solution cannot track requests that cross process boundaries. Thus, the Blame Graph shows internally called methods from another host or process on the same level as calls that are called directly from the test driver. Moreover, the Blame Graph shows the median response time as box-width. It does not cover the amount of times an operation was called and may therefore be deceiving. The number of calls is omitted due to the prototype status of our tooling, but can be easily added to the Blame Graph.

7. RELATED WORK

This paper introduces an automated decision support for performance blame analysis. This decision support delivers the Blame Graph that take the context of the component operations into account. So, Subsection 7.1 discusses other performance blame analysis approaches in terms of their decision support. It evaluates for each approach what their decision support looks like and to what degree it is automated. Moreover, it presents what results the decision support for each approach yields. The set of results shall be easily interpretable and it shall take the context of the results into account.

One contribution of this paper is to present performance metrics visually embedded in the component operation context. Subsection 7.2 presents several visualization techniques from other areas, because visual result presentation is seldom used in performance blame analysis. It covers the areas of performance profiling and software cartography.

7.1 Performance Blame Analysis

One approach that does come from the field of performance blame analysis in component-based systems is presented by Srinivas and Srinivasan [17]. The approach analyzes the call tree of component-based software to perform blame analysis. The call tree is annotated with cumulated percentage costs. The costs in the call tree can be any performance metric such as response times or CPU usages, which are expressed using percentages. The analysis then converts the cumulated costs to absolute costs. The cost conversion takes only a user-defined set of components into account. Calls to other components are defined as in-method computation, and are hence not differentiated in the analysis. Then the analysis computes if there are calls in the altered call tree that exceeds a chosen cost threshold. The result of this approach is a list of component methods with their absolute percentage costs. The threshold can be used to scale the number of results given.

Compared to our approach the approach by Srinivas and Srinivasan [17] has a slightly lower degree of automation. On one hand, the user has to supply several complex configuration settings. The user has to specify a threshold and the set of classes and methods, which are considered in the analysis, as substitute for the contract and the component context, respectively. The configuration settings have massive influence on the quality of results. On the other hand, the decision support as such is automated. The decision support is the computation of the final cost percentages that are higher than the user-defined threshold. The decision support results in an ordered list of methods and cost percentages. In the list the system architect can easily identify the most costly methods. This kind of analysis relies on the observation that often times high cost of a method means that it needs to be improved. However, if the methods that need to be improved are not at the top of the list, they are hard to find. The list can be very extensive if the threshold is set too low. In such a case an additional visualization would be helpful. The cost-metric is not so much dependent on the call tree as pure response time measurements. The algorithm internalizes calls into calling methods, as specified by

the user. Then it computes absolute cost values. These absolute values can be interpreted independent of the call tree.

Another approach that deals with performance blame analysis is exhibited by Rutar and Hollingsworth [15]. The approach computes the blame for variables and data structures on the basis of data flow. Every operation on a variable like a simple assignment or an addition of another variable increases the blame of this variable. This is also the case if the variable is used in loops or branches. The blame is also aggregated in data structures. These blame data are gathered by static analysis and by runtime monitoring. The result of this approach is a list of variables and their blame factor.

The approach by Rutar and Hollingsworth works on a different abstraction level than our approach. They are doing a white-box analysis covering the variables and are therefore working on very detailed level. This implies that the source code of all components needs to be available, since static analysis works on the source code. The static analysis is not fully automated and needs to be assisted in complicated cases. Moreover, the result list does neither cover blamed component operations nor components. Therefore, the system architect has to manually map the variables to components and component operation in order to find the blamed components or component operations. This means that the approach has no decision support.

The last approach tests the conformance of a component implementation to its performance specification. Groenda's approach [7, 8] is also based on the PCM. On one hand, Groenda measures the performance of the implementation during test cases derived from PCM instances. On the other hand, he derives performance measurements from PCM performance prediction. These two data series are then compared in terms of equality. The comparison is done on component operation level.

In contrast to our approach Groenda [7, 8] investigates whether the data series for a component operation in a specific test case from testing matches an equivalent data series from performance prediction. He statistically tests the hypotheses whether the two data series deviate or match. He interprets the test result as "yes" or "no" with the help of a user-defined error threshold. While this approach exemplifies the automated use of statistical testing as part of the decision support, it does not elaborate further on this topic. In particular, Groeda does not mention how the single results of the repeated tests are aggregated and how the result is presented.

7.2 Performance Visualization

This subsection deals with the areas of performance profiling and software cartography. It describes which visualizations are used in these areas and how they compare to the Blame Graph. The Blame Graph is the prime visualization introduced in this paper.

The popular open source profiler JFluid [5] (also known as Netbeans Profiler) exhibits a tabular representation based upon the Calling Context Tree. This presentation consists of a table of methods that incorporates a tree view. Each row stands for a called method. The methods it calls are drawn with more indentation in the table rows underneath and the called method's name is connected by a line to its caller's row. Each row is annotated with the information how many times the method was invoked as well as its net and total response time.

The commercial profiler JProfiler [11] exhibits a similar view than JFluid. JProfiler does not have the tree included in a table, but it exhibits a call tree view drawn with vertices and arrows between them. Each vertex carries the same information as the table rows in the JFluid view. It is worth noting that JProfiler features a similar view for memory usage.

The JFluid and JProfiler visualizations are both similar to the blame graph. They display a similar set of metrics. However, the Blame Graph additionally includes the KS-Test result that is visualized using a color code. The Blame Graph also is more compact. Especially when compared to the JProfiler view. Because of its compactness the Blame Graph does not include each method name and operations with very short response time are barely visible.

The "j2eeprof" approach [12] presented by Kłaczewski and Wytrębowicz introduces another view of the call tree. In this view every method called is represented by a box. A box for the method B() is drawn on top of the box for method A() if A calls B. The width of each box then resembles the response time of each respective method. This view is essentially identical to the Blame Graph. The Blame Graph defines the box width as median response time of the test data series and adds the box color as visualization of the KS-Test result.

Software cartography describes how the techniques of cartography, e.g. used for city maps, can be carried over to software. Krogmann et al. [13] have shown that this approach is well suited for the presentation of performance metrics. They have focused on the presentation of resource usages in the context of a component-based system. They have created several views that use the data from several scenarios as overlays. The overlays can be switched on and off at will to allow rapid comparison.

While Krogmann et al. show that the concepts of software cartography are very advanced and allow for good result representation, we cannot directly use their results as they focus on resource usage. However, concepts such as overlays are also desirable for our result representation. For example, the Blame Graph can use overlays in terms of different color codes that stand for different metrics.

8. CONCLUSIONS AND FUTURE WORK

In this paper we have extended our previously introduced performance blame analysis process [2] with an automated decision support step. The decision support step statistically and automatically decides whether the test data series has the overall higher response time values in comparison with the expected values derived from the component's performance contract. The decision support evaluates this by the KS-Test and by comparing point estimators. The automated decision support results in the Blame Graph and the performance report. The Blame Graph concisely presents the blame (from the KS-Test results) for each component operations along with the call hierarchy and the median response time. The Blame Graph is the main tool for the system architect to determine blame. She uses the performance report to back her decision up with detailed results for the KS-Test and the point estimators.

The automated decision support helps the system architect to interpret performance measurements in terms of performance blame analysis. Instead of looking directly at the raw data series or the complex statistical characterizations, the system architect can work with the Blame Graph and the performance report. Both artifacts make it easy to judge whether the test data series has higher or lower response time values compared to the expected values. The Blame Graph exhibits this directly using a color code that stands for the KS-Test results and the performance report visualizes point estimators for both data series side by side.

In the future we will continue to work on the approach presented in this paper. As a next step, we want to evaluate the reasoning of our automated blame analysis with an even more realis-

tic example system. We want to assess if the Blame Graph and the performance report are sufficient decision criteria or if we have to add more criteria (like the difference between the values of the two data series) to form a decision tree. Moreover, we want to set up an empirical case study testing if the proposed visualization is user-friendly and adequate in all circumstances. We might need to add different views to our visualization using the overlay concept from software cartography [13]. In addition, we are developing a test case notation that incorporates PCM instances. Such enriched test cases constructively ensure that the test case scenario is always compatible to the PCM simulation scenario.

9. REFERENCES

[1] Becker, S. et al. 2009. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*. 82, 1 (Jan. 2009), 3–22.

[2] Brüseke, F. et al. 2011. Palladio-based performance blame analysis. *Proceedings of the 16th International Workshop on Component-oriented programming - WCOP '11* (Boulder, Colorado (USA), 2011), 25–32.

[3] Cheesman, J. and Daniels, J. 2000. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley Longman, Amsterdam.

[4] Devore, J.L. and Berk, K.N. 2012. *Modern Mathematical Statistics with Applications*. Springer.

[5] Dmitriev, M. 2004. Selective profiling of Java applications using dynamic bytecode instrumentation. *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on - ISPASS*. (2004), 141–150.

[6] Gregg, B. 2011. Flame Graphs: *http:///dtrace.org/blogs/brendan/2011/12/16/flame-graphs/*. Accessed: 2012-10-04.

[7] Groenda, H. 2011. An Accuracy Information Annotation Model for Validated Service Behavior Specifications. *Models in Software Engineering*. J. Dingel and A. Solberg, eds. Springer Berlin / Heidelberg. 369–383.

[8] Groenda, H. 2010. Usage profile and platform independent automated validation of service behavior specifications. *Proceedings of the 2nd International Workshop on the Quality of Service-Oriented Software Systems* (New York, NY, USA, 2010), 6:1–6:6.

[9] Herold, S. et al. 2008. CoCoME - The Common Component Modeling Example. *The Common Component Modeling Example*. A. Rausch et al., eds. Springer Berlin / Heidelberg. 16–53.

[10] Jain, R. 1991. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons.

[11] Java Profiler - JProfiler: *http://www.ej-technologies.com/products/jprofiler/overview.html*. Accessed: 2012-03-23.

[12] Kłaczewski, P. and Wytrębowicz, J. 2007. j2eeprof — a tool for testing multitier applications. *Software Engineering Techniques: Design for Quality*. Springer Boston. 199–210.

[13] Krogmann, K. et al. 2009. Improved Feedback for Architectural Performance Prediction Using Software Cartography Visualizations. *Architectures for Adaptive Software Systems*. Springer Berlin / Heidelberg. 52–69.

[14] R Development Core Team 2011. *R: A Language and Environment for Statistical Computing*. http://www.R-project.org.

[15] Rutar, N. and Hollingsworth, J.K. 2009. Assigning Blame: Mapping Performance to High Level Parallel Programming Abstractions. *Euro-Par 2009 Parallel Processing*. H. Sips et al., eds. Springer Berlin Heidelberg. 21–32.

[16] Shao, J. 2003. *Mathematical Statistics*. Springer.

[17] Srinivas, K. and Srinivasan, H. 2005. Summarizing application performance from a components perspective. *ACM SIGSOFT Software Engineering Notes* (New York, NY, USA, 2005), 136–145.