Palladio-based Performance Blame Analysis

Frank Brüseke, Gregor Engels
s-lab – Software Quality Lab
University of Paderborn
Paderborn, Germany

fbrueseke@s-lab.upb.de,
engels@s-lab.upb.de

Steffen Becker
Heinz-Nixdorf Institute
University of Paderborn
Paderborn, Germany

steffen.becker@upb.de

ABSTRACT

Performance is an important quality attribute for business information systems. When a tester has spotted a performance error, the error is passed to the software developers to fix it. However, in component-based software development the tester has to do blame analysis first, i. e. the tester has to decide, which party is responsible to fix the error. If the error is a design or deployment issue, it can be assigned to the software architect or the system deployer. If the error is specific to a component, it needs to be assigned to the corresponding component developer. An accurate blame analysis is important, because wrong assignments of errors will cause a loss of time and money.

Our approach aims at doing blame analysis for performance errors by comparing performance metrics obtained in performance testing and performance prediction. We use performance prediction values as expected values for individual components. For performance prediction we use the Palladio approach. By this means, our approach evaluates each component's performance in a certain test case. If the component performs poorly, its component developer needs to fix the component or the architect replaces the component with a faster one. If no component performs poorly, we can deduce that there is a design or deployment issue and the architecture needs to be changed. In this paper, we present an exemplary blame analysis based on a web shop system. The example shows the feasibility of our approach.

Categories and Subject Descriptors

C.4 PERFORMANCE OF SYSTEMS, D.2.5 Testing and Debugging

General Terms

Measurement, Performance, Design

Keywords

CBSE, blame analysis, test, performance prediction, Palladio

1. Introduction

Performance is an important quality attribute for business information systems. Performance requirements are usually specified at the system boundary of such systems. These requirements can be validated using performance tests. The tests are performed after the implementation of all parts of the software participating in the test. However, to optimize the performance at design time, one can use model-based performance prediction. These approaches analyze a model of the system's software architecture to deduce what the performance of the system will be.

Suppose the tester has spotted a performance error. So, the system is not fulfilling one of its performance requirements. In traditional approaches, the error is then given back to the software developers who fix the error. If necessary, they escalate the error to the party that is responsible for the artifact in which it is occurring first, e. g. the software architect. In traditional approaches, it is relatively easy to find the responsible parties to fix the error in the blamed development artifacts. Usually, there is only one developer group that is responsible for each kind of artifact (i. e. software architecture, source code, etc.).

In component-based software development, various component developers provide components. Therefore, it is important that the tester assigns errors accurately to the responsible component developers. Moreover, the components may be acquired from the market. So, it may incur additional costs to mistakenly assign an error to a particular component developer. The tester must decide whether an error is a design or deployment issue, or a component issue. An example for a design issue might be that the component composition employs an unsuited design pattern. An example for a deployment issue is that the components are allocated to hardware in a non-optimal way. If an error is not a design or deployment issue, the tester has to decide, which component developer needs to fix a certain component they contributed to the system. The problem described is called "blame analysis" [4].

Some design issues can already be identified at design time using model-based performance prediction. Thus, system architects can optimize their design models and avoid implementing such flawed designs. However, sometimes design issues remain undetected despite performance predictions being used. These design issues as well as the component-related errors can only be sorted out by using tests.

When it comes to individual components, one might assume that one can analyze a performance error early by the use of component tests. However, this is not feasible because component tests only evaluate the implementation of the component and ignore its environments. This is also applies to component tests that are specifically made for component-based system, such as built-in tests [3]. Built-in tests are component tests shipped alongside or inside a component. In spite of being ignored in component tests, the environment of a component is crucial to its performance [13]. The environment of a component consists of:

- other components,
- usage of the system,
- hardware.

With the approach we present in this paper, we want to contribute to the blame analysis for performance issues. Our approach is able to decide for each component, if it performs well in a certain test

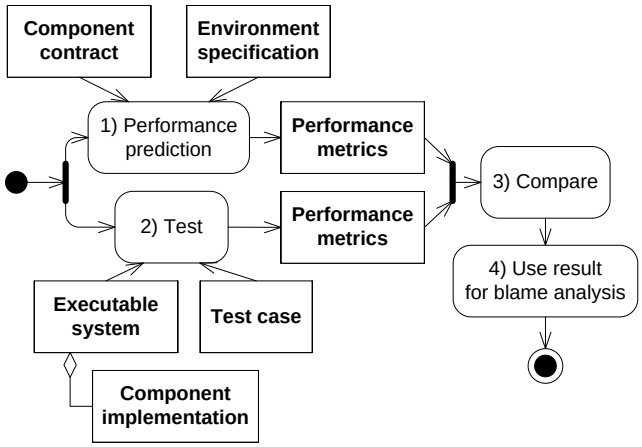

Fig. 1: Overview of our approach

case. By this means, we want to identify who needs to fix component-related errors. If no component performs poorly in a test, we deduce a design or deployment issue in the software architecture.

As we already pointed out, we cannot just employ performance component tests for this task, because of the lacking environment information. We also cannot evaluate the performance of individual components in system tests, as we have no expected values for their performance useable as test oracles. Expected results for individual components are neither given in the requirements nor is it possible to derive exact values from the software architecture. However, model-based component-prediction often works with contracts stating how well a component is meant to perform. These contracts are used to predict how well the component-based system will perform in a particular situation and setup. For example, this can be done with the Palladio component model (PCM) [2]. Performance prediction can also supply performance metrics on component-level.

Our approach is depicted in Fig. 1. It uses the performance prediction results as expected values (cf. step 1 in Fig. 1). In step 3, we compare performance prediction results from step 1 for individual components with corresponding test results from step 2. The results of this comparison are then used to evaluate, if each of the components perform well. Otherwise, we suspect a design or deployment issue. The central activities of our approach are steps 3 and 4, while the first two steps are prerequisites.

In this paper, we show test cases as well as performance models based on a web shop example. The example serves for illustration

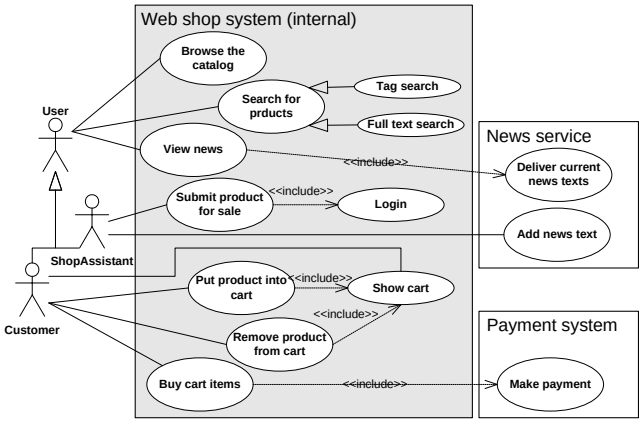

Fig. 2: Use cases of the web shop example system

Table 1: Web shop requirements

Response time requirements:
90% of requests per hour belonging to use case "Browse catalog" shall be answered in less than 200 ms.
80% of the requests per hour belonging to the use case "Search product" shall be answered in less than 500 s.
Throughput:
90000 requests in a two hour period, distributed among the use cases as follows:
45% Browse catalog
35% Search for product
12% View News
…

purposes as well as proof of concept. As a next step, we want to apply our approach to the Common Component Modeling Example (CoCoME) [7]. In the future, we want to use our approach on a real-world open source component-based business information system.

The contribution of this paper is that it introduces an approach enabling the tester to do blame analysis. The tester can decide for each component, if its performance was sufficient in a certain test case. If not, the corresponding component developer needs to fix the error. If all components perform well, the tester can deduce a design or deployment issue. By this means, one can analyze who needs to fix performance errors found in testing. The blame analysis is done by comparing measurements for individual components from performance tests to those from model-based performance prediction.

The remainder of the paper is structured as follows. In Section 2 we introduce an example that is used throughout the paper to explain our approach. Next, we introduce the Palladio approach (cf. Subsection 3.1) and its models as a foundation of our work. Moreover, we also introduce performance test cases (cf. Subsection 3.2) as one of the key artifacts of our approach. In Section 5, we illustrate our approach. First, we introduce test cases based on Palladio models (cf. Subsection 5.1). Second, we explain an example of blame analysis using our approach (cf. Subsection 5.2). Lastly, Section 6 concludes the paper and gives an outlook on our future work.

2. Example introduction

This section introduces an example that we use throughout the paper to explain our approach. As our approach focuses on business information systems, we have chosen a web shop system as an example. The web shop allows customers to either browse through the products organized in categories or to search products via tags or full text search. Customers can also put products into their shopping cart. After browsing the shop customers can proceed to the checkout and pay for the products in their shopping cart. Fig. 2 depicts all the use cases of the exemplary web shop.

Performance-wise the web shop system needs to fulfill certain response time and throughput requirements. These requirements are stated using quantile values. This is common for business information systems. The requirements for our exemplary web shop are listed in Table 1.

The overall component architecture of the system as envisioned by the software architect is depicted in Fig. 3. The central "Web"-

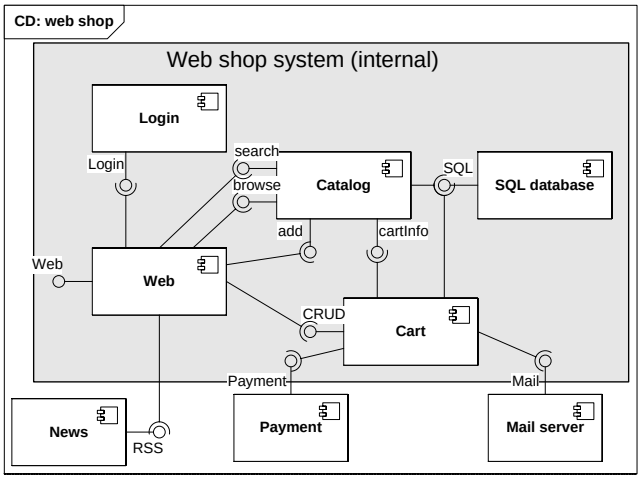

Fig. 3: components of the example web shop system

component generates HTML-pages and coordinates the workflow internally. The other components are doing the actual work. For example, the "Cart"-component manages the shopping cart of each customer. Moreover, the "Catalog"-component holds the catalog of all available products and their distinction into categories. For instance, it participates in the "Browse catalog"-use case.

3. Foundations

The approach presented in this paper makes use of the Palladio approach to analyze performance models. Subsection 3.1 introduces Palladio. It summarizes its models and the roles that are responsible for them.

The performance test cases are important artifacts of the tests preceding our approach. Test cases are important because they influence the test result's interpretation. We will give a short overview of test case structure and characteristics in Subsection 3.2.

3.1 Palladio

In Fig. 1, we give a general overview of our approach. The upper part of the diagram comprises the performance prediction part of the approach. It illustrates that we need component contracts and environment specifications to predict the performance of the system.

We choose the Palladio component model (PCM) [2], [1], [9] for the performance prediction part of our approach. Our approach, as well as Palladio, focuses on component-based business information systems. Palladio is able to evaluate typical performance requirements of business information systems including mean and quantile metrics. Furthermore, Palladio is able to model the component's environment (cf. Section 1). I. e. other components, including external services, can be specified. Palladio can analyze their interaction, including data flow and component configuration. In addition, the PCM can express the usage of the system by the user. The hardware environment and the allocation of the component can also be modeled. Lastly, Palladio offers solid tool support. It not only allows us to model business information systems, but also to analyze and simulate Palladio models to predict their performance.

The PCM consists of different diagram types that assist the different roles participating in the component-base development process. Palladio offers separate diagram types for each of the roles. The different diagram types and the roles responsible for them are:

- component repository (component developer, software architect),
- service effect sepecification (component developer),
- system model, also named composition (system architect),
- resource environment (system deployer),
- allocation (system deployer).

A component repository is supplied by the component developer. It defines the components and their interfaces. These component definitions are also done by the software architect. The software architect uses them as blue prints to acquire components. Each component in the system needs to be in one repository. The component repository supplied by the component developer also includes service effect specifications (SEFFs). They describe the performance-relevant abstractions of each interface method. SEFFs can be seen as performance contracts for the specific method they describe. SEFFs are very similar to UML activity diagrams, but add resource usages as well as probabilistic loops and branches. SEFFs can be parameterized to be environment-independent. Parameters are, for example, used to model the effect of different inputs to the service or different configurations of the component.

The software architect then assembles components from different vendors into a complete system. The software architect specifies this composition in a system diagram. The system deployer specifies on which hardware the resulting system will be deployed (resource environment). The system deployer also models which components will be allocated on which hardware. The domain experts specify the user behavior in usage models. Usage models cover the order and frequency of user calls to certain methods at the system boundary and are similar to SEFFs. Both, usage models and SEFFs, are similar to UML activity diagrams. In usage models, only method calls can be modeled. In SEFFs, one can also model internal computation, resource acquisition etc. Lastly, the QoS analysts are responsible for integrating the models and executing the performance analysis.

3.2 Performance test cases

As the lower part of Fig. 1 shows, our approach interprets measurements obtained in testing. Test cases are a key artifact in our approach, because test cases are important for the interpretation of measurements.

Test cases in general consist of preconditions, input values, expected results, and expected postconditions [14]. The IEEE 829 standard [8] also structures test cases in a very similar way (cf.

Table 2: test cases according to the IEEE 829 standard [8]

2.	Details (once per test case)
2.1.	Test case identifier
2.2.	Objective
2.3.	Inputs
2.4.	Outcome(s)
2.5	Environmental needs
2.6.	Special procedural requirements
2.7.	Intercase dependencies

Table 2). However, they subsume the expected values and post-conditions under the heading "Outcome(s)". The preconditions are included under the heading "Special procedural requirements". This section of the test case also comprises instructions for the test case execution, such as what tools need to be used. Most notably, they add "environmental needs" to test cases. This can be used to specify the test environment needed for this test case, which is very important in performance testing. The performance depends on the hardware used in the test as well as the allocation of the components to the hardware.

As performance testing is about creating load on the system, it needs good tool support. This does not only apply to test execution, but also to the analysis of performance measurements. Moreover, it is common to randomly generate input values to get the high volume of test data. To create test cases, the tester needs to specify transactions (e. g. use cases). Then, performance test cases are built by either testing one transaction in isolation or several transaction executed concurrently. Molyneaux [12] describes several types of such performance test cases. Most commonly, the load test is used. It uses a realistic mix of transactions to validate if the system can cope with a realistic load.

4. Related work

The main part of our approach is about testing and the interpretation of test results. There are several approaches of built-in tests (summarized by Beydeda [3]). Built-in tests are component tests that optionally can be generated by test generators. Built-in tests are shipped along-side or inside the component itself. They can then be executed (or generated and executed) by the software architect. The software architect does not need any deep understanding of the internals of the component to do the test. Moreover, the tests are usually given in source code, so that the software architect can see what is being tested and build confidence in them.

The tests or test generators are implemented by the component developers. Built-in tests are self-validating with preset acceptance criteria. While this is acceptable for functional tests, it is not for performance tests. In performance testing, the needed performance depends a lot on the field of application. Built-in tests employ specific knowledge of the component, because they are supplied by the component developers. In contrast, built-in tests are ignorant of the environment of the component. The knowledge of the environment in which the component is used is, however, very important in our case.

Another approach to component-based testing is assume-guarantee testing introduced by Giannakopoulou et al. [5]. Assume guarantee testing checks if a component shows the guaranteed behavior, given that the modeled assumptions hold. The assumptions explicitly model the environment of the components. In this approach, all components are modeled using labeled transition systems. The approach will generate test cases based on the transition systems and test each component in isolation. Doing this, it can check for properties affecting the global properties of the system. However, this approach is only applicable for testing functional properties.

The approach most closely related to our approach is the one introduced by Groenda [6]. Groenda proposes to assess the quality of behavior specifications using an automated model-based testing approach. This approach is also based on Palladio models. It derives test cases from SEFFs and component repository specifications. In this approach each service description (i. e. each method a component provides) is tested in isolation. To measure the

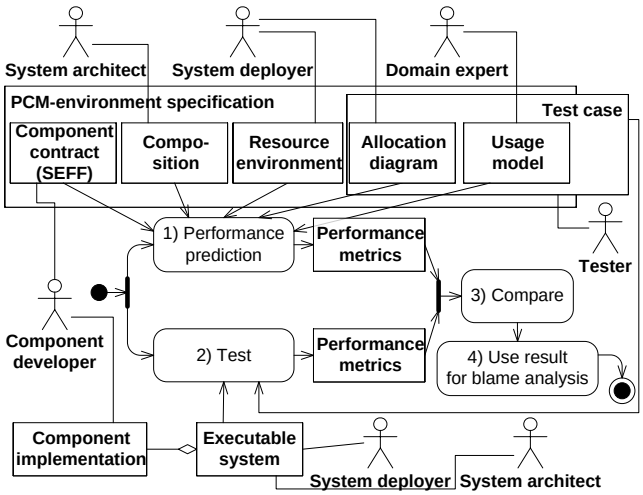

Fig. 4: Refined view of the introduced approach

performance, each section of a SEFF (e. g. internal computation or external call) is instrumented. Groenda uses byte code counting instead of time measuring. This way, they get results that may be ported to different deployment scenarios. The results of these measurements are then compared to those obtained from performance prediction. The deviation of the results indicates the quality of the specification.

Our approach also compares performance metrics from the test with those obtained by performance prediction using Palladio. While Groenda derives test cases from a Palladio model as well, he does so on the basis of SEFFs and component specifications instead of usage models. In our scenario, we rely on regular performance tests. These usually execute several transactions concurrently. So, we need to additionally specify how these more complicated test case inputs shall load the system. Besides, the particular deployment (i. e. the hardware and operating system) and the concurrently running requests can influence each other. So in our case, the performance metrics should be obtained by measurement and not bytecode counting.

5. Approach

As we argued before, we need to analyze the performance of individual components for a successful blame analysis. In blame analysis, one needs to decide if there is a component-related error, or a design or deployment issue. If there is a component-related error, then we need to know which component is affected. Our approach evaluates the performance of individual components. It compares performance metrics for individual components obtained by testing to those obtained by model-based performance prediction.

In Fig. 4, we have refined the general overview (cf. Fig. 1) to include the Palladio artifacts (cf. Subsection 3.1). Furthermore, it also associates the responsible roles to each of the artifacts according to the Palladio process [10]. In particular, the "environment specification" breaks down into the following parts:

- composition (system diagram),
- resource environment,
- allocation diagram,
- usage models.

The component contracts are modeled using SEFFs in Palladio. They are supplied by the component developers. We assume that the component developers guarantee the performance that the

analyses or simulations of their SEFFs yield in the context of a complete Palladio model. In addition, the software architect uses the Palladio model including the contract SEFFs to optimize the software design. So, this Palladio model is either specifying the system or is at least an abstraction of the specification used to optimize design model. Either way, the used Palladio corresponds to the specification.

The central "compare" activity of our approach (cf. activity 3 in Fig. 4) relies on the performance metrics provided by performance prediction and test (cf. activities 1 and 2 in Fig. 4). Our approach is about finding out who needs to fix an error that occurred after completion of the *testing* phase. Performance prediction is only a means to interpret measurements obtained during the *test*. This means that we react on failed performance tests[1] by interpreting test results using performance prediction results. Thus, there needs to be performance prediction results for all failed performance test cases.

But not any performance prediction result can be used to interpret the results of a certain test case. The performance prediction results need to stem from a performance prediction scenario that is comparable to the test case in question. In Subsection 5.1, we explain how test cases can be written comprising Palladio models. Fig. 4 depicts that that these test cases can not only be used for testing but also for performance prediction. For these test cases one can always perform a performance prediction using the Palladio models in the test case. Thus, we can guarantee that there always are either comparable performance prediction results available or that they can be produced easily.

In Subsection 5.2, we show an example for the actual blame analysis. I. e. we want to find out who needs to fix a particular error in the exemplary web shop system. We present performance metrics obtained in test[2] and performance prediction respectively. Both of them rely on a test case specified using Palladio models (cf. Subsection 5.1). Thus, we can draw valid conclusions.

5.1 Comparable performance metrics from test and performance prediction

In this section, we will show test cases that are specified using Palladio models. We need comparable performance prediction results for any failed test case. Defining test cases with the help of Palladio models enables us to get the required prediction results. For these test cases we are using diagrams of the Palladio model to replace certain parts of the test case. These diagrams need to be based on other parts of a complete Palladio model. Ideally, they are based on the model used to optimize the design models. We need to make some assumptions on the underlying Palladio model.

We assume that the specification of hardware resources and hardware usages is consistent throughout all project artifacts. This includes the resource environment specifying the resource and the SEFFs specifying the resource usages. For example, a consistent understanding of CPU usage exists, whether a CPU usage of 10 CPU units is interpreted in the same way by all component developers, e. g. all software architects or QoS-analysts. Our approach inherits this assumption from Palladio as this is needed for proper performance prediction as well.

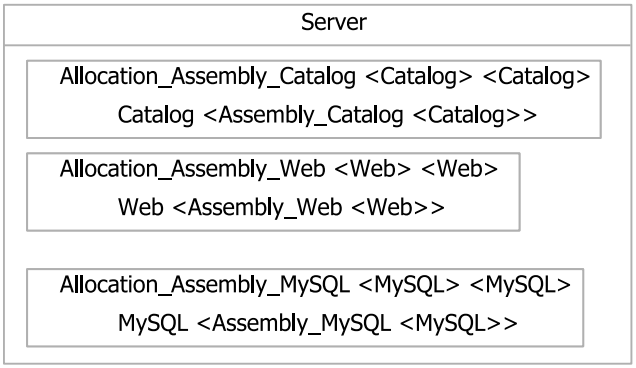

Fig. 5: Example PCM allocation diagram

Moreover, we must add the assumption that resource specification and resource usage values are consistent to the real test environments. This means, that the relation of a resource usage to the total amount of resources available needs to be realistic. For example, suppose that a method uses 10 CPU units and the CPU is specified with a total capacity of 1000 units. For example, this corresponds to reality, if the method uses 10 ms CPU time in our test environment. The relation of used resources to available resources is in both cases 10/1000.

We propose to write test cases using Palladio diagrams. As shown in Table 2 test cases mainly consist of preconditions, inputs, outcomes (i. e. expected outputs and postconditions), and the test environment. The inputs can be modeled with usage models and the test environment can be modeled using allocation diagrams. We expect that any hardware resource that might be needed in the allocation diagram is already defined in the resource environment. Moreover, we expect that the software architecture is fixed at the time of test design, such that all components and interfaces are available in a component repository. In the following, we will investigate how usage models and allocation diagrams can be used in test cases.

The allocation diagram describes which components are allocated to which hardware resources. The allocation model shall reflect the test environment in our test cases. Fig. 5 contains a sample allocation diagram. It shows one block for the only server in our example set up and all the components in our exemplary web shop inside. In our example, all components are allocated to one server. If there were more servers, there would be one block for each of them. The connections between the servers are not part of the allocation model. They are specified in the resource environment.

The usage model states how the system is going to be used. Usage models specify, which interface methods a user calls in succession. This specification may include branches and loops. I. e. a user can call a method not at all or several times in a certain instantiation of the usage model. Furthermore, one can specify how many users are loading the system.

In Fig. 6 we see an example of a usage model. It depicts the "Browse catalog" use case of our web shop example. Usage models are similar to UML activity diagrams. According to the exemplary usage model, the user calls the "startPage", "catalogPage", "categoryView", and "productView" methods of the Web component at the system boundary in succession. The "productView"-method is optional to call, as it is called zero times with 20% probability. But the method may also be called up to eight times.

[1] We assume that the tester can always evaluate, if a test failed or passed by consulting the requirements.

[2] The test results are hypothetical, as we have not implemented the example web shop.

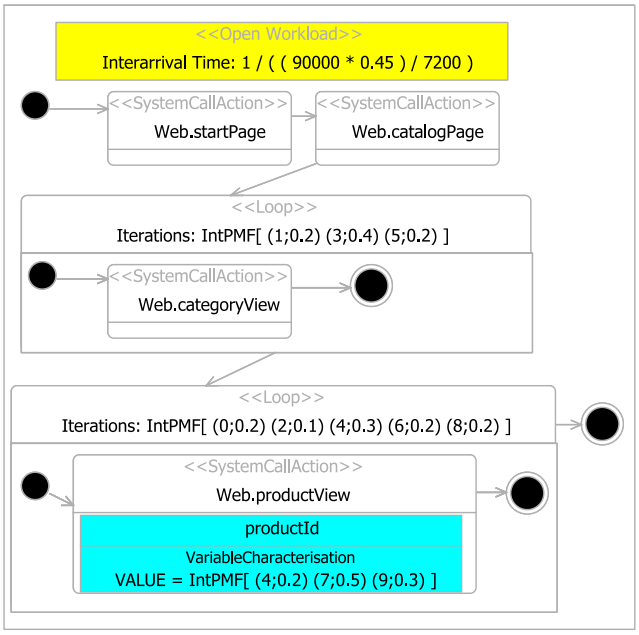

Fig. 6: PCM Usage model of the "Browse catalog" use case

Eight calls per user happen 20% of the time. Moreover, we also modeled the possible input values for the "productId" parameter of the "productView" method. It can assume the values 4, 7, and 9 with 20%, 50%, and 30% probability respectively. Instead of the actual values of "productId", we could also model its byte size or the number of elements in it. Lastly, Fig. 6 shows how often (in seconds) a user enters the system according to the "open workload" block at the top.

The example usage model in Fig. 6 shows that usage models are able to specify the inputs of performance test cases. It allows us to model the sequence and frequency of method calls. Moreover, we can model the actual input values where necessary and can also use probability functions to state how to generate input values. However, usage models lack expected results and expected post-conditions as well as the preconditions of the test case. So, these parts of the test case are not given in terms of Palladio models.

In Table 3, we show a sample test case for the "Browse catalog" use case. The test case includes the Palladio usage model and allocation model shown in Fig. 6 and Fig. 5. We can also analyze the performance of this use case with Palladio. The test case supplies the usage model and the allocation model. In addition, we need a component repository, a system model (i. e. component composition), and a resource environment. These models form a complete Palladio model together and enable us to analyze the performance of the model according to the test case.

5.2 Performance metric comparison and blame analysis

Our approach helps analyzing performance errors. After performing a test, the tester needs to check first, if the test passed according to the requirements. If the test failed our approach can be used for analysis. Our approach can be used to decide whether each of the components in the system is performing well. In this case, the poorly performing components are considered erroneous and their developers should fix them. If all components perform well, the problem lies elsewhere. So, we suspect a design or deployment issue. In the following, we will give an example for this analysis.

Table 3: example test case with Palladio models

2	*Details (once per test case)*
2.1	*Test case identifier:* 1_Server_UC_Browse_001
2.2	*Objective* Validate if the use case "Browse catalog" fulfills its response constraints in isolation using a realistic load for this use case.
2.3	*Inputs* Cf. Fig. 6
2.4	*Outcome(s)* 90% of the requests shall be answered in less than 200 ms (cf. Table 1)
2.5	*Environmental needs* Cf. Fig. 5
2.6	*Special procedural requirements* No preconditions Execute the test case for one hour.
2.7	*Intercase dependencies* None

Presuming that the test case for the "Browse catalog" use case is executed (cf. Table 3) and the tester finds that the test fails and that the requirements violated. Consequently, the blame analysis begins. I. e. we want to analyze who needs to fix the error the tester found. As Fig. 7 depicts, the components "Web", "Catalog", and "SQL database" participate in the use case. The use case covers three requests. The tester communicates that the test case failed during the "productView" requests. The tester can give us this information because the test case (cf. Table 3) is about checking request response times. If the test case would have constrained a full execution of the use case, finding out which request is taking longer as it should, would have been part of the blame analysis. However, this could be easily accomplished by looking at the requirements, as requests are called at the system boundary.

Fig. 8 shows the additional response time measurements for the request "productView" and the inner methods it calls. The diagram depicts a histogram of their response time. On the x-axis, the

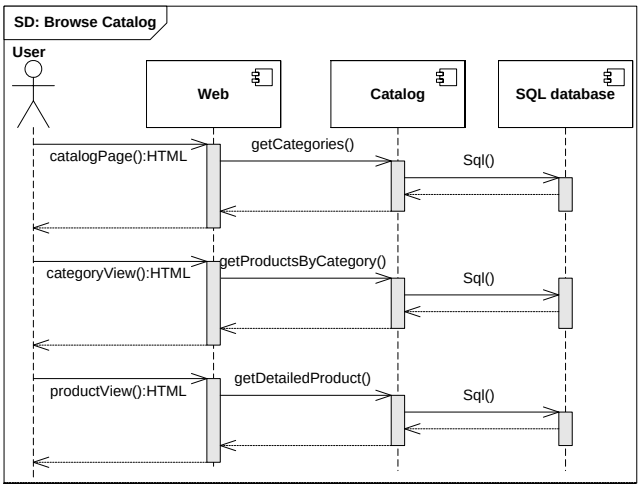

Fig. 7: Detailed view of use case "Browse Catalog"

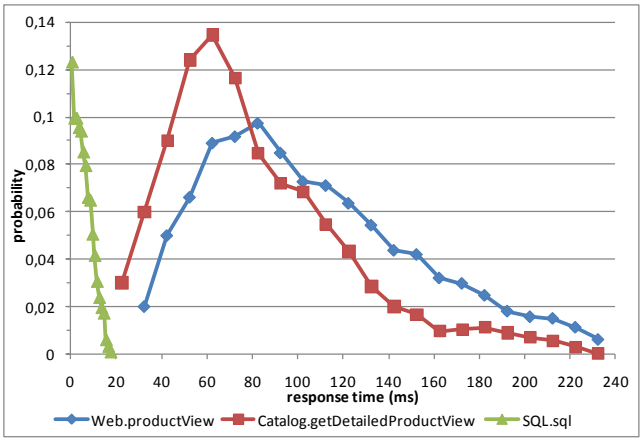

Fig. 8: Combined test results – use case "Browse catalog"

diagram shows response time measurements aggregated in clusters. The response time of each method includes its own execution time and the time the method waited for results from internally called methods. On the y-axis, the diagram depicts the probability with which a measurement falls into the arctual cluster. The diagram depicts that the "sql" method is executing fairly fast, while the other two methods take quite long to execute.

Now, we compare the test results (cf. Fig. 8) to the performance prediction results depicted in Fig. 9. The performance prediction results stem from a performance simulation based on the Palladio models in the test case defined in Table 3. Thus, we can draw valid conclusions by comparing the performance metrics in Fig. 8 and Fig. 9.

Fig. 9 depicts the same performance metrics as Fig. 8. I. e., the x-axis shows clusters of method response time while the y-axis shows probability that a measurement is part of the respective cluster. In Fig. 9, the distribution of the measurement for each of the methods differ are very similar. The inner methods have a higher probability of lower response times, and the outer methods have a higher probability of higher response time. However, there is no such huge gap as in Fig. 8.

In Fig. 8, the response time for the "sql" method is almost identical as in Fig. 9. However, the methods "productView" and "getDetailedProduct" show a much higher response time than in Fig. 9. So, we suspect that both of them are erroneous. Now we aim at narrowing the field of slow methods down to one. The response time for the method "productView" subsumes the response time for the method "getDetailedProduct". This hints at problems in the method "getDetailedProduct" of the "Catalog" component. Moreover, the response time difference between the "getDetailedProduct" curve and the "productView" curve is a bit bigger in Fig. 8, but still similar to the difference in Fig. 9. We can deduce that the most serious problem occurs in the method "getDetailedProduct" of the component "Catalog". The vendor of this component shall fix this error.

6. Conclusions and future work

In this paper, we have presented an approach to do blame analysis. It evaluates the performance of individual components. The approach achieves this by comparing component-individual performance metrics that are acquired in performance testing to those that are acquired in performance prediction. Our approach predicts the performance of the system using a Palladio model. We use the metrics from performance prediction as expected values. If

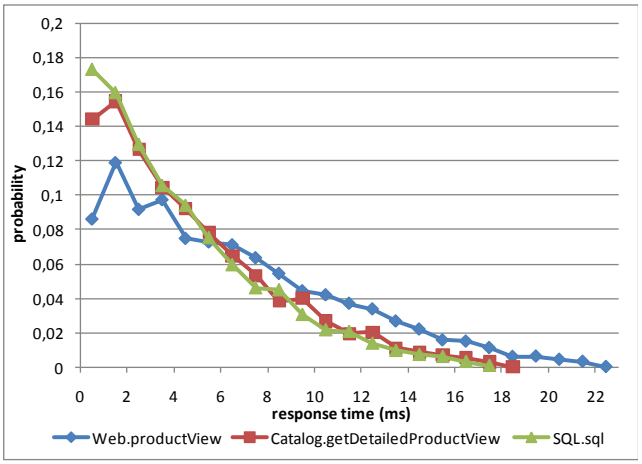

Fig. 9: Combined prediction result – use case "Browse catalog"

a component performs poorly, its component developer needs to fix the error. If no comsponent performs poorly, we can deduce a design or deployment issue.

As our approach relies on comparing performance metrics from testing with those from performance prediction, we need these values to be comparable. Thus, we have analyzed test cases and Palladio models and shown that they partly include the same information. We have proposed to include certain Palladio models in test cases. In particular, the Palladio allocation and usage model shall be used to specify the input values and the test environment in the test cases. These two models built on an existing complete Palladio model, e. g. the one used to optimize the design models. We have shown an example for such a test case (cf. Table 3) based on the use case "Browse catalog" of our example web shop.

Based on this test case we have shown that our approach can be used for blame analysis. We have presented performance metrics from performance prediction and performance test and have compared them. By this means, we have found out that the gravest problem occurs in the "getDetailedProduct" method. Therefore, the component developer of the "Catalog" component would have to fix our example error.

In the future, we plan to use the introduced approach on the Common Component Modeling Example (CoCoME) [7]. CoCoME offers a complete implementation including a test bed. Moreover, there is a Palladio model for CoCoME [11]. This will allow us to validate our approach using a fully implemented and more complex example. Particularly, we aim at creating blame analysis guidelines for several types of errors. This guideline shall include the analysis of utilization and concurrency metrics, as well as response time metrics.

After that, we plan to evaluate our approach using a real-world open-source business information system that had known performance problems. We will apply our approach to the faulty version of the software. This way we can analyze, if we can find out which components are responsible for the performance problems. We can confirm our findings by looking at the fix that the developer used to solve the problem.

7. REFERENCES

[1] Becker, S. 2008. *Coupled model transformations for QoS enabled component-based software design*. Universität Oldenburg.

[2] 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.

[3] Beydeda, S. 2005. Research in testing COTS components - built-in testing approaches. *Computer Systems and Applications, 2005. The 3rd ACS/IEEE International Conference on* (2005), 101.

[4] Gao, J. et al. 2003. *Testing and quality assurance for component-based software*. Artech House.

[5] Giannakopoulou, D. et al. 2008. Assume-guarantee testing for software components. *Software, IET*. 2, 6 (2008), 547-562.

[6] 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.

[7] 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.

[8] IEEE Computer Society 2008. IEEE Standard for Software and System Test Documentation. *IEEE Std 829-2008*. (Jul. 2008), 1-118.

[9] Koziolek, H. 2008. *Parameter Dependencies for Reusable Performance Specifications of Software Components*. University of Oldenburg.

[10] Koziolek, H. and Happe, J. 2006. A QoS Driven Development Process Model for Component-Based Software Systems. *Component-Based Software Engineering*. I. Gorton et al., eds. Springer Berlin / Heidelberg. 336-343.

[11] Krogmann, K. and Reussner, R. 2008. Palladio – Prediction of Performance Properties. *The Common Component Modeling Example*. A. Rausch et al., eds. Springer Berlin / Heidelberg. 297-326.

[12] Molyneaux, I. 2009. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media.

[13] Reussner, R.H. et al. 2004. Parametric Performance Contracts for Software Components and their Compositionality. *Proceedings of the 9th International Workshop on Component-Oriented Programming (WCOP 04)* (2004).

[14] Spillner, A. et al. 2011. *Software Testing Foundations: A Study Guide for the Certified Tester Exam*. Rocky Nook.