Analysis and Prediction of Performance for Evolving Architectures

E.M. Eskenazi, A.V. Fioukov, D.K. Hammer
Department of Mathematics and Computing Science, Technical University of Eindhoven,
Postbox 513, 5600 MB Eindhoven, The Netherlands
{e.m.eskenazi, a.v.fioukov, d.k.hammer}@tue.nl

H. Obbink
Philips Research
Professor Holstlaan 4
5656 AA Eindhoven
The Netherlands
Henk.obbink@philips.com

B. Pronk
Philips Semiconductors
Professor Holstlaan 4
5656 AA Eindhoven
The Netherlands
ben.pronk@philips.com

Abstract

This paper describes a method for the "Analysis and Prediction of Performance for Evolving Architectures" (APPEAR). The method aims at performance estimation of adapted parts of software product families during the architecting phase. It combines both structural and statistical techniques in a flexible way, that is, it allows choosing which part of the component is structurally described, modeled and simulated, and which part is statistically evaluated. The method was exemplified by case studies in the Consumer Electronics and the Medical Imaging System domains. The results of the initial validation of the method are encouraging.

1. Introduction

Early estimation of software performance makes it possible to verify the feasibility of products of a product family before their implementation, thus saving money and effort otherwise devoted to developing potentially infeasible products. The possibility to evaluate the software performance (e.g. response time, latency, average CPU utilization, execution time) at an early stage can help, for instance, in evaluating the impact of architectural decisions beforehand and in quickly selecting the most appropriate one.

Software architects need thus a method to estimate the performance of software early, during the architecting phase. This method should be a) fast in comparison to software implementation and subsequent measurements, b) simple so that less time and fewer human resources are required, c) general so that it can be applied to any type of software, and d) accurate in order to provide useful results.

To date, two types of methods were used for performance evaluation: a) purely simulation-based models and b) mathematical models (e.g., queuing networks [18], [19]). Both types turned out to be unsuitable for evaluating the performance of complex embedded systems. The first type of the methods suffers from the combinatorial explosion of details, whereas the second often makes too specific assumptions about the system under consideration. These assumptions do not hold for many systems, and thus models based on these assumptions can be both inaccurate and inadequate.

The APPEAR (Analysis and Prediction of Performance for Evolving Architectures) method [7], [8] combines the best elements of several existing estimation techniques. The method employs both simulation and statistical models. The former describe the evolving performance-relevant parts that are not yet implemented. The latter are used to abstract from details that are not performance-relevant and to model those parts of a system that remain unchanged for a long time during the evolution of components. Abstracting from irrelevant details helps one reduce the modeling complexity. This mix is supported by the fact that fewer and fewer software-intensive systems are currently being developed from scratch.

This paper is structured as follows. Section 2 summarizes related work. Section 3 presents the requirements for the APPEAR method. Section 4 describes the basic constituents and essential steps of the method. Section 5 presents the application of the APPEAR method to performance prediction of a component of a Medical Imaging software system. Section 6 describes the application of APPEAR to the Teletext decoder of a modern TV set. Finally, Section 7 summarizes the paper and sketches the future work.

2. Related work

Significant research effort has been taken in the performance-engineering domain. The main investigations were aimed at the development of methods for the early performance estimation of software-intensive systems, and at defining the theoretical basis for software performance engineering [18].

Classical approaches [18], [19] to performance estimation use queuing network models, derived from the structural description of the architecture and performance-critical use cases. Other approaches concern specific architecture description styles [2]. In [21], Wu et al. use pre-calibrated performance models of software components to predict the performance of component assemblies. These models abstract from performance irrelevant details and use a component-based modeling language (CBML) for the specification. These models are also based on layered queuing networks. In [1], UML design diagrams are translated into queuing networks. Liu et al. [16] abstract from irrelevant details of complex applications and build performance models from a design description. These models input the results of simple benchmarks and yield the performance estimates.

The aforementioned modeling techniques treat the behavior of a component in a restrictive manner, as they describe it in terms of queuing networks that are not always an adequate behavior description formalism. Moreover, the availability of the entire code of software and figures about its resource consumption and overhead are often unrealistic. These techniques are hardly applicable to modern component-based software, as they neglect input parameters and consider scenarios instead of components.

A well-known practice for early performance analysis is the construction of a simulation model that captures the performance-critical parts of the software. The results from such a model, executed using different parameters, are either estimates for performance attributes or intermediate data that can be used for building other mathematical models. For instance, an interesting approach is proposed in [12]. The executable prototype (a simulation model) generates traces that are expressed in a specific syntax (angio-traces). These traces are used for building performance prediction models, based on layered queuing networks.

In [3], Avritzer et al. describe the early estimation of the performance impact of a small change in a rule-based system. A simulation model is constructed to estimate the CPU utilization, based on rule firings (inter-arrival times). The results of this simulation are compared with the measurements taken from the system before the change to estimate the performance degradation. This degradation is expressed in terms of unprocessed alarms (tasks) in a system due to long task queues and limited CPU capacity.

Stochastic Petri nets are also widely used for the evaluation of software performance. An approach to the generation of Petri nets from UML collaboration and statechart diagrams is proposed in [14]. These Petri nets are then used to estimate different performance characteristics. Gilmore et al. propose in [10] to use colored stochastic Petri nets (PEPA nets) for the performance modeling of Web-services. Lopez-Grao et al., in [17], and Canevet et al., in [6], translate UML activity diagrams to Petri nets to be used for performance analysis.

In [13], Jain advocates the use of measurements, simulation, and analytical modeling to analyze the performance of computer systems. The results provided by any of these techniques should not be trusted until confirmed by at least one of the other techniques.

An approach presented in [11] is similar to the one presented in this paper. This approach also considers the use of linear regression for performance prediction.

Another inspiration source for our method was the approach described in [5]. Bontempi et al. suggest using linear and non-linear regression (e.g., lazy learning [4]) to predict the performance of embedded software. The models are calibrated using performance-relevant parameters of both software and hardware and the values of performance measurements.

However, both approaches [5] and [11] do not fully support the performance estimation of component-based software at the architecting phase, as the entire program code must be available. We extended these approaches such that the performance models are built on the basis of architectural and design specifications, without requiring the entire code to be available.

3. Requirements

The aim of the "Analysis and Prediction of Performance for Evolving Architectures" (APPEAR) method is to support architects in analyzing the performance of future versions of components during the early phases of product development. By future versions of components we mean adapted versions of existing software or new components that are "sufficiently similar" (see section 4.4) to the existing ones to allow the use of statistical prediction techniques.

We interviewed a number of software architects about the requirements to a valid performance prediction method. The most essential requirements are the following:

1. Allow performance prediction of the adapted components to enable
    - Early estimation of the impacts of architectural decisions on the performance,
    - Finding the appropriate architectural solutions for performance-critical components, and
    - Comparison of different architectural solutions with respect to the performance.
2. Provide insight into the performance-relevant behavior of the components by means of
    - Identification of performance critical parameters,
    - Construction of behavioral models (simulation models) of the components, and
    - Localization of performance bottlenecks.
3. Ensure a reasonable level of accuracy for performance prediction. The required accuracy level is product dependent. Our survey revealed that architects consider an accuracy of 50% to 80% as a definite improvement with respect to the currently used methods.
4. Obtain performance predictions fast in comparison to the time needed for the implementation of a new component and subsequent measurements.

4. Description of the APPEAR method

This section sketches the APPEAR method and enumerates a few assumptions that enable its application.

4.1. Signature type and signature instance

The signature of a component is a set of parameters that provide sufficient information for performance estimation. In this paper, the performance is considered in terms of response and execution times. In principle, the APPEAR method can also be applied for other performance metrics that relate to resource consumption (e.g., average memory demand).

We treat the performance metric P as a function over the signature:

$$P : S \rightarrow C. \quad (1.1)$$

In this formula, $S = \{S_1, S_2, ..., S_N\}$ is a signature type, a vector with real elements S_i, and C is a performance metric such as response time. An example of the signature type of a hypothetical software component is as follows:

S = {Number of memory allocation calls, Number of disk calls, Number of network calls}

The signature type typically correponds with parameters (input parameters, service calls, etc.) that have a serious influence on the performance. It is important to distinguish between the signature type (see above) and a signature instance that contains actual values for a concrete use-case, e.g. s = {132, 57, 21}.

4.2. Essence of the method

This section overviews the basic principles of the APPEAR method.

The APPEAR method suggests the following view of the software stack. The software comprises two parts (see Figure 1): (1) components and (2) a Virtual Service Platform (VSP). The first consist of evolving components that are specific for different products of a product family, whereas the second encompasses stable components that do not significantly evolve during the software lifecycle of a product.

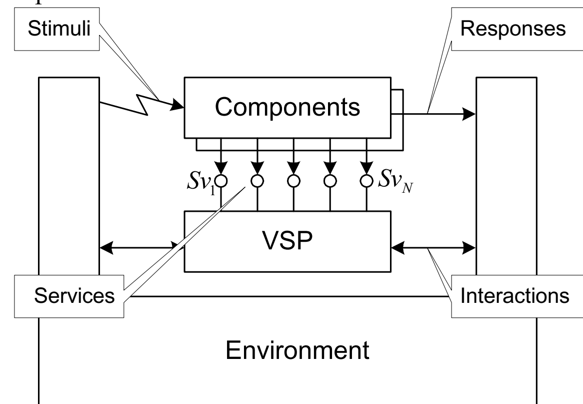

Figure 1. APPEAR view of the software stack

Each use case of interest is represented by a pair stimulus-response. As a result of an input stimulus, a component can invoke several VSP services to perform the necessary functionality. After completing these calls, the component produces a response to the environment. The timing dependency between the stimulus and response can be characterized by some performance measure.

The APPEAR method constructs a prediction model, which is fitted to the measurements from the existing component(s) by means of regression techniques [4], [9], [15], and [20]. This model reflects the correlation between the performance metric of interest and signature instances of the existing components. The correlation can be used to extrapolate the performance of adapted components during the architecting phase, as the existing and adapted components use the common VSP and share the same signature type.

To gain insight into the execution architecture and its performance, it is also advisable to construct a high-level simulation model of the component(s) under consideration. Such a model should capture performance-relevant properties of the component(s).

The APPEAR method includes two phases: (1) calibrating the prediction model on the existing components and (2) applying this prediction model to the adapted component to estimate its performance.

4.2.1 Phase 1. First, it is necessary to identify the signature type and construct a statistically valid prediction model. A prediction model is said to be statistically valid if it satisfies a number of statistical tests indicating its quality. The prediction model needs to be calibrated on the signature instances. This can be accomplished according to the following procedure (see Figure 2):

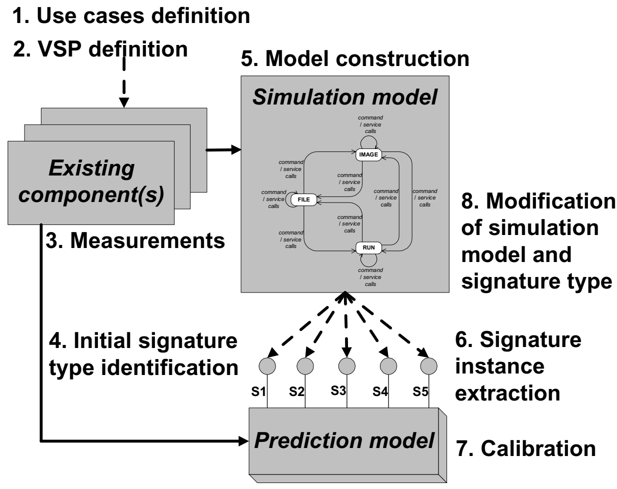

Figure 2. Calibration of the prediction model

Step 1, Use cases definition. Based on the requirements specification, the architect chooses a relevant set of use cases. These use cases are used to extract signature instances, i.e. obtaining the values of the signature parameters.

Step 2, Virtual Service Platform identification. Based on the architectural specification, the software stack is subdivided into two parts: components and VSP.

Step 3, Measurements. For the use cases chosen in step 1, the performance of the existing component is measured, for example, by instrumenting and profiling the code. The collected measurements are treated then as the values of a dependent variable, a variable that needs predicting.

Step 4, Identification of the initial signature type. The initial set of performance relevant parameters is deduced from the analysis of execution profiles, architectural documentation, etc.

Step 5, Construction of the initial simulation model. Based on the available architecture description, a simulation model needs to be built to extract the signature instances, i.e. to determine the values of the performance relevant parameters that can be observed at architecture level.

Step 6, Signature instance extraction. The simulation model calculates one signature instance per use case for the defined set of the use cases. These signature instances are then stored together with the corresponding measurements obtained during step 3.

Step 7, Prediction model calibration. The results of steps 3 and 6 form the calibration data for building a model that predicts the performance, depending on the signature instance. Each sample of this data corresponds to a use case.

Step 8, Tuning of simulation model and signature type. It can be the case that the prediction model is not statistically valid during step 7. This means that either the signature type is chosen wrongly or the simulation model misses performance relevant details. Steps 6-8 must therefore be repeated until the prediction model becomes statistically valid.

4.2.2 Phase 2. After having the prediction model calibrated, the performance can be predicted for adapted components according to the following procedure (see Figure 3):

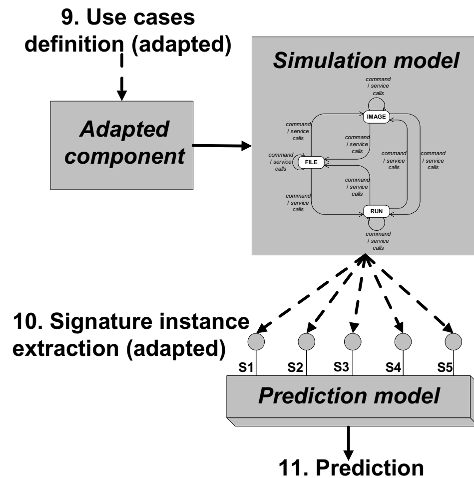

Figure 3. Performance prediction for the adapted component

Step 9, Definition of use cases for an adapted component. For the adapted component, the architect determines a set of use cases that needs performance prediction. According to these use cases, the initial simulation model of the existing component is modified (if needed), with the signature type kept intact.

Step 10, Signature instance extraction for the adapted component. By executing the simulation model, the signature instance is calculated for the use cases defined in step 9. This signature instance can be used for predicting the performance, if the adapted component is sufficiently similar (see section 4.4) to the existing one.

Step 11, Predicting the performance of the adapted component. By applying the prediction model calibrated at phase 1 to the signature instance obtained during step 10, the performance of the adapted component is estimated. The architect must further interpret this estimate with respect to performance requirements.

4.3. Assumptions

The following assumptions must be fulfilled to apply the APPEAR method:

1. The performance of a component depends only on its internals and VSP, but not on other components.
2. The services of the VSP are independent. There are no interactions that significantly influence the performance, e.g. via exclusive access to shared resources.
3. The order of service calls does not matter.
4. The adapted and existing components are similar (see section 4.4). Otherwise, the prediction can fail, because the observation data are not applicable anymore.
5. The product (family) evolves gradually. During the evolution of a product family, a significant portion of the software remains unchanged.
6. A sufficient number of components are available for training the prediction model.
7. The software is instrumented to collect performance measurements.

4.4. Similarity of software components

The accuracy and trustworthiness of predictions obtained using the APPEAR method are questionable. An adapted component may have different behavior than the existing ones; e.g., it may use the VSP in a different manner. As a result, the prediction model may provide incorrect results. Moreover, the architects do not have any measurements to validate these predictions, as the component is not implemented yet. Consequently, they need other means to judge the trustworthiness and accuracy of the estimates.

We define the notion of component similarity as follows:

The existing and adapted components are similar if the performance of the adapted component can be predicted with a known accuracy and confidence using the prediction model fitted on the existing component.

The architects can ascertain the similarity of existing and adapted components by the following formula:

$$Similar = ST \wedge SI \wedge IC \quad (1.2)$$

In formula (1.2), $Similar$ is a Boolean variable that indicates whether the components are similar or not. The IC, ST, and SI Boolean variables denote the three similarity criteria, as explained in Table 1. These similarity criteria are discussed in the subsequent sections.

Table 1. The three similarity criteria

Variable	Aspect	Meaning
ST	Signature types	The signature types are the same for the existing and adapted components.
SI	Signature instances	The signature instances extracted from the simulation model of the adapted component are close to the ones from the existing component.
IC	Internal component calculations	The internal calculations are the same for the existing and adapted components.

4.4.1 Signature types. The existing and adapted components must have exactly the same signature type. The prediction model can only input the signature parameters that were used for its calibration. These signature parameters are determined on the basis of the existing component.

4.4.2 Signature instances. Trustworthy predictions can only be obtained for the signature instances that are close to the ones used to fit this prediction model [15], [20]. It is therefore necessary to check the distance between the signature instances generated for the existing and adapted component.

4.4.3 Internal component calculations. It is not always the case that the most of performance is determined by the VSP. Both existing and adapted components can have timing dependencies or CPU-intensive internal calculations that contribute to the overall performance. These internal calculations and timing dependencies may be component-specific.

5. Performance prediction for Medical Imaging software system

This section describes our experience in building APPEAR models for prediction of the response time of the "Reviewing" component of a Medical Imaging system.

5.1. Identification of the Virtual Service Platform

After selecting the most relevant use cases, the architects assisted us in the identification of a VSP (see step 2 from section 4.2.1). According to the structure of the "Reviewing" component (see Figure 4), the VSP includes the following subcomponents: "Graphics", “Image Board Controller”, and “Database”.

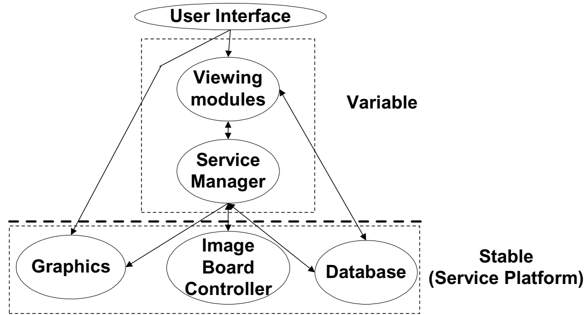

Figure 4. Virtual Service Platform (VSP) of the Medical imaging component.

These subcomponents are a) defined by the underlying hardware and b) common for other components (e.g., image acquisition). The subcomponents above this level are regularly improved, and, thus, belong to the variable part.

5.2. Signature extraction

The examined use cases dealt with *images* or *image sets*. An *image* is a single medical image of a particular patient. An *image set* is a collection of images obtained between the activation and termination of an acquisition process. A *file* contains all *image sets* of a patient.

The most time-consuming calls to the VSP were the main candidates for signature parameters. The execution time of most time consuming calls was directly related to the number of *images* or to the number of *image sets*. Some calls to the “Image Board Controller” and to the “Graphics” subcomponents were also the most time consuming ones. Both types of calls concerned updating the state of the image processing hardware (“*Update*”) and graphics (“*Paint*”) that overlay the medical images.

Finally, the signature type can be represented as a vector consisting of four elements:

Signature = {#Image Sets, #Images, #Update, #Paint}

5.3. Construction of simulation model

This section describes steps 5 and 8 of the APPEAR method (see section 4.2.1). The simulation model inputs external parameters of the software: the number of *images*, the attributes of the images, user commands, etc. When a user command is handled, the simulation model generates a signature instance.

The behavior of the “Reviewing” component is described in terms of state machine (see Figure 5).

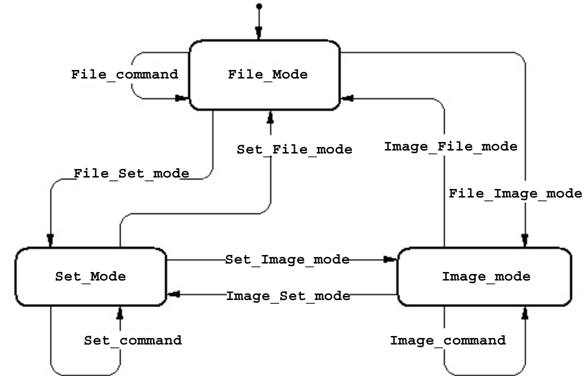

Figure 5. State machine describing the system behavior

The system can function in three modes: “Image”, “Set”, and “File”. In these modes, an *image*, *image set* or *file* are displayed and browsed, respectively. In the “Image” mode, a single *image* is displayed. In the other two modes, an *image set* and a *file* are displayed in an interleaved manner.

Some user commands trigger the switching between modes. Other user commands can be invoked while the system is in a particular state. Both the execution of a command in a certain state and the change of the state result in generating the signature instance.

5.4. The prediction model

This section describes the construction and validation of the prediction model for the “Reviewing” component. The prediction model was constructed by linear regression (using the S-PLUS tool [15]) and calibrated on signature instances generated by the simulation model. The model was validated in the following way: ten arbitrarily chosen points were taken out from the measurements, and the model was calibrated on the remaining data points. First, the quality of the model was assessed by analyzing statistical characteristics of the model such as the R^2-coefficient, maximal absolute error, and a few residual diagnostic plots. Second, response times for the excluded points were predicted and compared with the measurements.

Due to the peculiarities of the “Reviewing” component, the number of image sets and the number of images could not be varied independently, but only according to the following expression:

$$\#ImageSets = 1,\ if\ \ \#Images\ >\ 1$$
$$\#Images = 1,\ if\ \ \#ImageSets\ >\ 1 \quad (1.3)$$

As a result of this dependency, the prediction model had a complex form:

$$Response = \begin{cases} \alpha_0 + \alpha_1 \cdot \#Images + \alpha_2 \cdot \#Update + \\ +\alpha_3 \cdot \#Paint, \quad if\ \#Images > 1 \\ \beta_0 + \beta_1 \cdot \#ImageSets + \beta_2 \cdot \#Update + \\ +\beta_3 \cdot \#Paint, \quad if\ \#ImageSets > 1 \end{cases} \quad (1.4)$$

In formula (1.4), α_i and β_i denote the two sets of linear regression coefficients.

For both cases, the prediction model exhibited high prediction quality: residual distribution was close to the normal distribution, and R^2-coefficients had the values of 0.94 and 0.99, respectively.

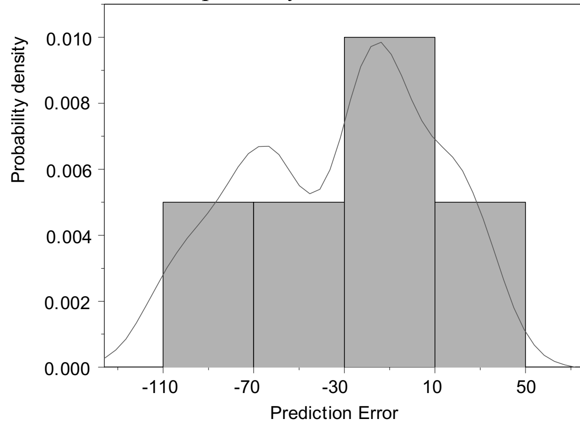

Figure 6. Prediction error distribution

The first case provided the average relative error $E = 0.0151$. The prediction error distribution is depicted in Figure 6.

For the second case, the error distribution was similar, and $E = 0.0865$.

The obtained results have significantly higher quality than for the previous experiment described in [7] due to (1) finding the dependency between the number of images and the number of image sets and (2) refining the prediction model accordingly.

6. Performance prediction for Consumer Electronics software

We applied the APPEAR method to the existing and adapted versions of the Teletext decoder of a modern TV set. The execution time needed to acquire Teletext data by the adapted version of the Teletext decoder was estimated using the prediction model calibrated on the existing version of the Teletext decoder. The predictions were then compared to the measurements from the implementation of the adapted Teletext decoder. More details about this experiment can be found in [8].

6.1. Overview of the Teletext decoder

Each Teletext transmission [22] comprises packets that together can form pages. Some packets are not directly related to a particular page, but rather to a magazine or a broadcast service. Different types of packets are discerned by packet numbers, which lie in a range 0 to 31. Packets with numbers greater than 24 are additionally discerned with designation codes: numbers in the range 0 to 15. Depending on the designation code, the function of a particular packet may change.

Teletext packets are transmitted during Vertical Blanking Intervals (VBI). In each VBI, up to sixteen packets can be transmitted.

All packets received during a single VBI must be processed before the next one. This fact introduces a soft deadline for the duration of a VBI that equals 20 ms.

There are several presentation levels of Teletext data– 1, 1.5, 2.5, and 3.5– that determine the information transmitted by a broadcaster and enhancements that can be made to a Teletext page. In addition, Teletext can support two types of navigation: (1) First Level One Facilities (FLOF) and (2) Table of Pages (TOP). Both TOP and FLOF implement hypertext-like navigation.

The simplified structure of the Teletext sub-system and its dependencies on the environment are sketched in Figure 7.

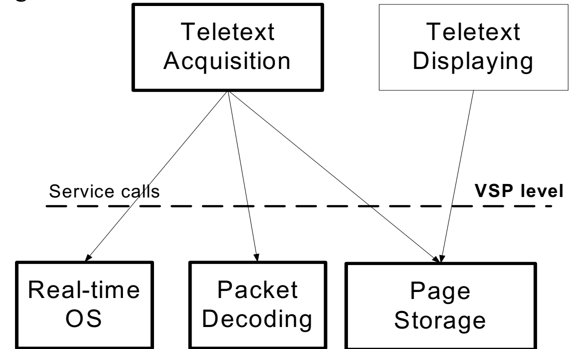

Figure 7. Structure of the Teletext subsystem

The Teletext acquisition component, a part of the Teletext sub-system, builds upon the VSP formed by the following components: the real-time operating system, the Packet Decoding component, and the Page Storage component.

The arrows in Figure 7 depict the ‘uses’ relationship. The dashed line corresponds to the abstraction level of the VSP. The bold rectangles denote the components that are relevant to the performance analysis of the Teletext acquisition component. The normal rectangles denote the

components that do not influence Teletext acquisition (e.g., Teletext Displaying component).

After acquiring all data packets arriving in a single VBI, a high priority task decodes and stores the packets. This task will be referred to as the *Teletext VBI routine* in the rest of this paper. The task is implemented within the Teletext acquisition component as its internal operation. It uses the service calls provided by the Real-time OS, Packet Decoding and Page Storage components. Teletext packets are decoded by the Packet Decoding component and then stored within the Page Storage component in a local page cache. They are moved to a global page store when all the packets of a particular page are received.

6.2. Experiment scheme

Two versions of the Teletext acquisition component were considered. The first one supports Teletext presentation level 1.5 and the FLOF navigation only, whereas the second one supports Teletext presentation level 2.5 and both TOP and FLOF navigation systems. We will hereinafter refer to these components as the Teletext 1.5 and Teletext 2.5 acquisition components, respectively.

The aim of the experiment was to predict the execution time of the *Teletext VBI routine* (see section 6.1) of the Teletext 2.5 acquisition component using the prediction model calibrated on the Teletext 1.5 acquisition component. It was required that the maximal prediction error did not exceed 1 ms, because the *Teletext VBI routine* had a soft deadline of 20 ms.

The experiment was conducted as follows. *First,* we applied the first phase of the APPEAR method to the Teletext 1.5 acquisition component. We then predicted the performance of the Teletext 2.5 acquisition component by constructing the corresponding simulation model and using the already calibrated prediction model. *Finally*, we compared the predictions with the actual measurements from the implementation of the Teletext 2.5 component.

6.3. Definition of use cases

The use case considered was watching a TV channel that carried Teletext information. This means that the TV set performed in a steady state and was collecting the Teletext data without any interference. Thirty use cases were chosen arbitrarily among the real broadcasts transmitted via cable to drive both the implementation and the simulation model.

6.4. Simulation model for the Teletext 1.5 decoder

The simulation model mimics the behavior of the Teletext acquisition component. Most of the functionality of this component is implemented within the *Teletext VBI routine* (see section 6.1). This routine accepts the packets received in a certain VBI and invokes the corresponding *packet processing routine* for each packet.

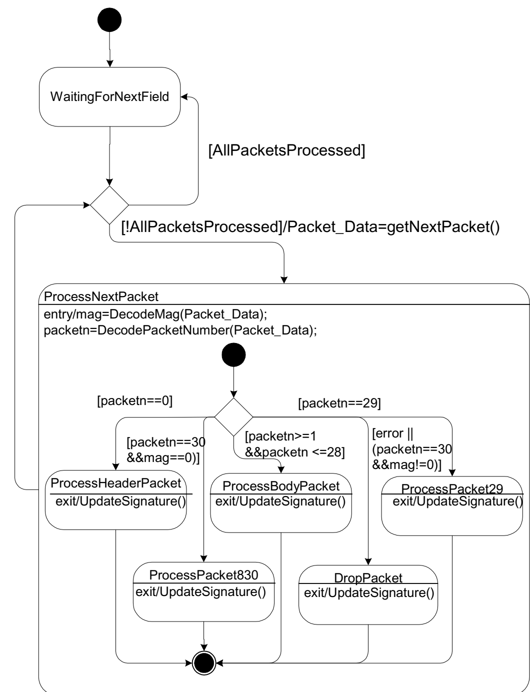

Figure 8. The high-level behavior of the Teletext acquisition routine

Figure 8 presents the UML state chart that describes the behavior of the *Teletext VBI routine.* The *packet processing routine* corresponds to the *ProcessNextPacket* composite state. Both packet and magazine numbers of recently-arrived packets are decoded. Depending on these numbers, further processing is delegated to one of the following states: *ProcessHeaderPacket*, *ProcessBodyPacket*, *DropPacket, ProcessPacket29*, or *ProcessPacket830*. Notice that these states correspond to functionality executed higher in the call hierarchy than the VSP level. The invocations of service calls are not depicted in Figure 8 for the sake of simplicity.

The simulation model inputs the descriptions of events that correspond to packet arrivals in a particular VBI. It calculates a signature instance for this VBI, based on the packets received.

Because the long-term history proved to significantly influence the performance of the Teletext acquisition, it also had to be modeled. The Page Storage component

maintains this long-term history by tracking all packets and pages received after the channel switch.

Notice that the Page Storage component belongs to the VSP. Although the pure APPEAR method, described in section 4, models explicitly only components that do not belong to the VSP, this component also had to be modeled explicitly to obtain a statistically valid prediction model.

6.5. Signature type

The identified signature type accounts for different types of packets, their encoding, the way they are stored, and etc. The signature type consists of thirteen signature parameters in total. Notice that the value of one signature parameter had to be extracted from the simulation model of the Page Storage component (a part of the VSP), whereas the rest of the signature parameters were extracted from the simulation model of the Teletext acquisition component.

6.6. Calibration of prediction the model

The linear regression tool S-Plus [15] was used to calibrate the prediction model. The prediction model has the following structure:

$$y = \beta_0 + \sum_{i=1}^{13} \beta_i \cdot s_i . \qquad (1.5)$$

In formula (1.5), y is the predicted execution time; β_i are linear regression coefficients; S_i are signature parameters (see section 6.5).

After calibrating the model (1.5), the following results were obtained. The multiple R^2-coefficient is 0.974. This means that the model explains well the variation of the execution time of the Teletext field routine. Moreover, all regression coefficients proved to be significant, with a significance level of 0.05.

6.7. Simulation model for the Teletext 2.5 decoder

The simulation model of the Teletext 2.5 acquisition component resembles the one of the Teletext 1.5 component (see Figure 8). The differences between the two components amount to the following:

1. Handling of packets 27 of a page with a non-decimal page number (e.g. Magazine Inventory Page).
2. Handling of a packet 28 with a designation code greater then one, or packet 27 with a designation code greater then three.

The modification of the Teletext 1.5 simulation model to obtain a simulation model for Teletext 2.5 took only one man-day.

6.8. Results

The implementation and simulation model of the Teletext 2.5 decoder were driven by the same broadcasts. The predictions were compared to the measurements from the implementation.

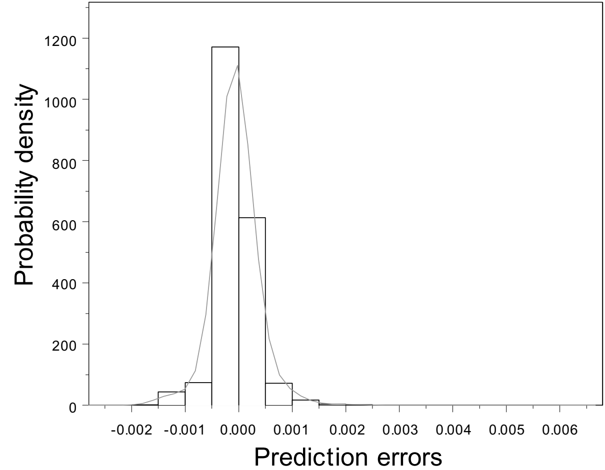

Figure 9. Prediction errors for entire broadcasts

The probability density and histogram of the prediction errors are shown in Figure 9, which covers all acquired VBI's. In this plot, the y-axis is the probability density, whereas the x-axis is the prediction error measured in seconds.

Figure 9 demonstrates that the bulk (more than 98%) of prediction errors lies within a ±1 ms range for the entire broadcasts. This fact means that the required accuracy (see section 6.2) is achieved for 98% of VBI's. Please notice that this range widens, if one considers packets specific for Teletext 2.5 and TOP navigation only.

Predictions made for the entire broadcasts (see Figure 9) suffer an average relative error of only 11%. For VBI's that are specific for Teletext presentation level 2.5 and the TOP navigation only, this error increased to 16%.

7. Conclusions

The APPEAR method, presented in this paper, allows the early estimation of the performance of an adapted component, based on the measurements from and simulation model of the existing component(s).

We exemplified the method by two industrial case studies. *First*, we applied it to a medical imaging software. The prediction quality was checked by cross-validation, i.e., a part of measurements was used to calibrate the prediction model, whereas the other was used to make predictions for. *Second*, we applied the method to two versions of the existing Teletext software.

One version was used to construct and calibrate the prediction model, whereas the other was used to predict the performance. In both cases, the predictions were compared with the measurements to ascertain the quality of the prediction.

The results of the APPEAR method validation appeared to be positive. The prediction accuracy was acceptable with respect to the requirements of the architects (e.g., see section 6.2).

We consider the following points for further investigation:

1. *Performance estimation for component compositions.* It must be possible to estimate the performance of a component composition, given the performance models of the components.
2. *Evolution of the platform.* Both components and platforms can evolve. During this evolution, it is important to maintain the predictability of performance.

8. Acknowledgment

The work presented in this paper was conducted within the AIMES project (EWI.4877) and funded by STW.

We are grateful to Sjir van Loo from Philips Research Laboratories for constructive discussions about the APPEAR method. We would also like to thank Rob van Ommering, Chritiene Aarts, Wim van der Linden, Marc Stroucken, and Pierre van de Laar from Philips Research Laboratories for their technical support.

9. References

[1] A. Alsaadi "A Performance Analysis approach based on the UML class diagram", In proceedings of the 4th International Workshop on Software and Performance (WOSP), USA, 2004.

[2] F. Aquilani, S. Balsamo and P. Inverardi, "An Approach to Performance Evaluation of Software Architectures", Research Report, CS-2000-3, Dipartimento di Informatica Universita Ca' Foscari di Venezia, Italy, March 2000.

[3] A.Avritzer, J.P. Ros, E.J. Weyuker "Estimating the CPU utilization of a rule-based system", In proceedings of the 4th International Workshop on Software and Performance WOSP, California, USA, 2004.

[4] G. Bontempi, "Local Learning Techniques for Modeling, Prediction and Control", PhD thesis, IRIDIA- Universite' Libre de Bruxelles, Belgium,1999.

[5] G. Bontempi, W. Kruijtzer, "A Data Analysis Method for Software Performance Prediction", In the proceeding of DATE 2002 on Design, automation and test in Europe, France, 2002.

[6] C. Canevet, S.Gilmore, J. Hillston, L. Kloul, P. Stevens "Analysing UML 2.0 activity diagrams in the software performance engineering process", In proceedings of the 4th International Workshop on Software and Performance WOSP, California, USA, 2004.

[7] E.M. Eskenazi, A.V.Fioukov, D.K.Hammer, H.Obbink, B. Pronk, Analysis and Prediction of Performance for Evolving Architectures, In Proceedings of Workshop on Software Infrastructures for Component-Based Applications on Consumer Devices, Lausanne, Switzerland, September 2002.

[8] E.M. Eskenazi, A.V.Fioukov, D.K.Hammer, Performance Prediction for Industrial Software with the APPEAR method, In proceedings of STW PROGRESS workshop, Netherlands, 2003.

[9] J.H. Friedman, "Multivariate Adaptive Regression Splines", Tech. Report 102, Department of Statistics, Stanford University, USA, August 1990.

[10] S. Gilmore, J. Hillston, L. Kloul, M. Ribaudo "Software performance modeling using PEPA nets", In proceedings of the 4th International Workshop on Software and Performance (WOSP), USA, 2004.

[11] P. Giusto , G. Martin , E. Harcourt, Reliable estimation of execution time of embedded software, Proceedings of the DATE 2001 on Design, automation and test in Europe, Germany, 2001.

[12] C.E. Hrischuk, C.M. Woodside and J.A. Rolia, "Trace Based Load Characterization for Generating Software Performance Models", IEEE Trans. on Software Engineering, Vol. 25, Nr. 1, pp 122-135, Jan. 1999.

[13] R. Jain, The art of computer systems performance analysis, Techniques for Experimental Design, Measurement, Simulation and Modeling, John Wiley & Sons, 1991.

[14] P. King and R. Pooley, "Derivation of Petri Net Performance Models from UML Specifications of Communications Software", Proc. 11th Int. Conf. on Tools and Techniques for Computer Performance Evaluation (TOOLS), Schaumburg, USA, 2000.

[15] A. Krause, M. Olson, "The basics of S-Plus", 3rd Edition, Springer Verlag, 2002.

[16] Y. Liu, A. Fekete, I. Gorton "Predicting the performance of middleware-based applications at the design level", In proceedings of the 4th International Workshop on Software and Performance (WOSP), USA, 2004.

[17] J. P. Lopez-Grao, J. Merseguer, J. Campos "From UML Activity Diagrams to stochastic Petri nets: application to software performance engineering", In proceedings of the 4th International Workshop on Software and Performance (WOSP), USA, 2004.

[18] C. Smith and L. Williams, "Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software", Addison-Wesley, 2001.

[19] B. Spitznagel and D. Garlan, "Architecture-based performance analysis", in Proceedings of 10th International Conference on Software Engineering and Knowledge Engineering, Knowledge Systems Institute, 1998.

[20] N. A. Weiss, "Introductionary Statistics", Addison-Wesley, 1995.

[21] X. Wu, M. Woodside "Performance Modeling from Software Components", In proceedings of the 4th International Workshop on Software and Performance (WOSP), USA, 2004.

[22] ETS 300 706: "Enhanced Teletext Specification".