Quantitative prediction of quality attributes for component-based software architectures

Quantitative Prediction of Quality Attributes for Component-Based Software Architectures

Evgeny Eskenazi
Alexander Fyukov

The work described in this thesis has been carried out in close cooperation with Philips Research Laboratories in Eindhoven and Philips Medical Systems in Best.

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Quantitative prediction of quality attributes for component-based software architectures / by Evgeny Eskenazi and Alexander Fyukov.
- Eindhoven : Technische Universiteit Eindhoven, 2004.
Proefschrift. - ISBN 90-386-0992-2
NUR 992
Subject headings: software design / software quality / discrete simulation / regression analysis
CR Subject Classification: D.2.11, C.4, G.3, I.6.8, I.6.7

This project was sponsored by Technologiestichting STW and conducted within the PROGRESS research project AIMES EWI.4877.
The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithms).
IPA disseration series 2004-19

Keywords: Software architectures / Component-based software engineering / Quality attributes / Memory consumption estimation / Performance prediction / Component composition / Simulation models/ Statistical models

Cover design: Paul Verspaget.
Printing: Universiteitsdrukkerij Technische Universiteit Eindhoven.

Quantitative Prediction of Quality Attributes for Component-Based Software Architectures

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de Rector Magnificus,
prof.dr. R.A. van Santen, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op maandag 6 december 2004 om 15.00 uur

door

Evgeny Eskenazi
geboren te Sint-Petersburg, Rusland
en
Alexander Fyukov
geboren te Sint-Petersburg, Rusland

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.Dipl.-Ing. D.K. Hammer
en
prof.dr.ir. J.F. Groote

Acknowledgements

The history of this thesis began more than ten years ago when Yuri Karpov, professor from the Technical University of Saint-Petersburg, and Dieter Hammer, professor from the Technical University of Eindhoven, met each other, and discussed the idea of exchanging students for research projects. We are sincerely grateful to prof. Yuri Karpov, who was one of our favorite tutors for three years at TU\SPb, for introducing us to prof. D.K. Hammer and for recommending us as suitable candidates for the research project at TU\e. We also want to thank Kirill Bolshakov, our group mate and friend from the TU\SPb, as he, working in the group of prof. Karpov, pinpointed us as the best candidates.

Our greatest thanks are expressed to prof. D.K. Hammer who was the key person for making this thesis a reality. He organized the research project AIMES (STW project EWI.4877) and proposed us the positions of Ph.D. students within this project at Technical University of Eindhoven. He introduced us to many interesting and remarkable people in the research community. He was a perfect supervisor for four years, immensely helping us to address both technical and organizational problems. He also was the thorough reviewer of this thesis, and long discussions that found place in his hospitable house in Luttenberg have definitely improved the quality of the thesis.

We are grateful to our technical supervisors– Henk Obbink and Sjir van Loo– at Philips Research Eindhoven. Despite the huge amount of their own research activities, they regularly participated in discussing the project status. Henk provided us with a lot of valuable research ideas, including the basic idea of the APPEAR method. He also arranged for us a case study at Philips Medical Systems where we could conduct research experiments in a real industrial setting for one year. Sjir helped us to arrange a case study in the Consumer Electronics domains and gave us constructive criticism on the methods developed, which enhanced the overall quality of the research results.

Our deep gratitude is also devoted to prof. Peter de With. Being a perfect external advisor for our project and a member of the reading commission, he contributed significantly into the quality of this thesis. His advices on both research methodology and writing style have always been sound, clear, inspiring, and friendly. We are grateful to Michel Reniers for teaching us the basics of formal languages, and especially MSC. We owe thanks to Michel Chaudron who permanently supported our research, invited us for exciting scientific discussions, shared his experiences, and helped in editing the conference papers. We also want to thank Jan Friso Groote, who was the supervisor of our project from the side of TU\e, for perfect project management and for his interest in our research. We also thank Jan Friso for being a member of reading commission and for relevant remarks on this thesis.

At a workshop in Sweden, we got acquainted to prof. Ivica Crnkovic whose research interests turned out to be aligned with ones of us. The common problems that we studied served a basis for many motivating discussions, and led to his visit to our university. We are very thankful to him for the organized workshops and symposiums on component-based software engineering and for inviting us there. We were exceptionally happy when Ivica agreed to become a member of the reading commission, and we would like to express him our sincere respect and gratitude for precious comments to this thesis.

We appreciated free time that we have spent with our friends during these four years. Our gratitude for this time is addressed to Sergei Shumski, Elena Shumskaya, Dmitri Chkliaev, Mugur Ionita, Stas Shumyacher, Victor Shcherbatyuk, Alexei Sintotski, Alexei Nesterenko, Alexander Popov, Michail Sorokin, Igor Nagorski, Sergei Lukin, Slava Pranovich, Alex Spektor, Egor Bondarev. We apologize to those whom we forgot to put in

this list. We have special thanks for Dennis Abrazhevich who reviewed our thesis in a quite informal way.

Acknowledgements by Evgeny Eskenazi

I would like to thank Chritiene Aarts from Philips Research, who together with Rob van Ommering, was my technical mentor. They provided me not only with technical support, but were also willingly discussing the obtained results and the direction of the future work. Special thanks are also expressed to Pierre van de Laar from Philips Research Eindhoven who not only teamed up with me to implement instruments for measuring the TV software, but also provided valuable feedback on our research. I am also grateful to the development team of TV software from Philips Semiconductors: Gerard van Loon, Maarten Pennings, Dirk Borgman and many others who allowed and helped me in performing case studies at their premises. I owe thanks to my fellow cluster members at Philips Research– Hugh Maaskant, Pjotr Kourzanov, and others– who gave me valuable feedback on my research.

I would sincerely like to give a credit to my landlady, Karen Mauve, who not only tolerated my presence during these four years, but also supported me and helped me in orienting in the Dutch society. She became a good friend of mine. I would like to thank my parents Irina and Mikhail, my brother Alexey, and my girlfriend Valeria. Without their love and dedication that they gave me during these years, I would not be able to write this thesis.

Last but not the least, I would like to express my gratitude to my friend and colleague Alexander Fyukov, with whom we wrote this thesis. He provided constructive criticism to my research ideas and inspired me in generating new ones. He also supported me in difficult times, and shared my joy in good times.

Acknowledgements by Alexander Fyukov

I express my gratitude to Ben Pronk, my mentor at Philips Medical Systems in Best, who always, regardless of project deadlines and time of the day, managed to find a time slot for me for discussing the results of the experiments and the next steps. I am also thankful to Ivo Canjels who became my mentor afterwards and provided me with sufficient technical support and was always eager to discuss research issues with me. I thank other employees of Philips Medical Systems: Shamsuddin Slegers, Arie van de Spoel, Marnix van Kempen, Henk van Dijk, Tom Fransen, Ron Verhoeven, and Erik Mellsen. They deeply supported me in studying the medical imaging software and in having a nice stay within their development group.

I am very grateful to my colleagues at Philips Research Eindhoven who surrounded me with the atmosphere of professionalism and kindness. First of all, I have to express my gratitude to Jaap van der Heijden, leader of the “Software Architectures” group for his attention and support during four unforgettable years. I am grateful to Cristian Huiban and Sorin Cristescu for the roles they played in my introductory research project. I want to express special gratitude to Marc Stroucken, Pierre America, Eelco Rommes, and Wim van der Linden for their feedback on my work. I am thankful to Rob van Ommering who introduced me to the world of Koala and TV software, and who generated a lot of inspiring ideas during our discussions.

I am very grateful to my landlady, Ada Jongsma, for her hospitality, for teaching me many Dutch habits and a Dutch language, and for tolerance to my behavior during these four years.

My best regards are directed to my colleague and co-author of this thesis Evgeny Eskenazi. I could always rely on him when working in a research team of this project. Evgeny was continuously stimulating and challenging me with his original research ideas. Based on his valuable and critical remarks, I performed research and wrote my part of this thesis of much better quality than I expected.

Finally, I would like to faithfully confess that without love and care of my parents, Elena and Vladimir, my sister, Nina, and my girlfriend Valeria, this work would hardly get that far. I admire their attitude and hope to return at least a small piece of the warm feelings that they gave me during the last four years. Because of this, I could hardly notice the distance of 2000 kilometers between us.

Contents

1 Introduction

During the past decade, the functionality of modern electronic systems has increased drastically: TV sets can now be used for internet communication; mobile phones can record and play back movies; medical imaging systems can perform 3D reconstruction of blood vessels in real-time, and etc. For all these systems, it is their internal software that provides the added value. The role of software has become essential, as the implementation of diversity and huge amounts of functionality of the products in dedicated hardware only would require unacceptable effort and enormous investments. In many cases, implementation of the functionality in software allows using cheaper general-purpose hardware. Moreover, in opposite to hardware, the software is easily changeable, upgradeable and customizable. The aforementioned attributes of software lead to significant increase of the profit margin of the product.

Software complexity and size are growing together with its functionality. For example, the size of binary code of software in modern TV sets has changed according to the Figure 1.1 during last years.

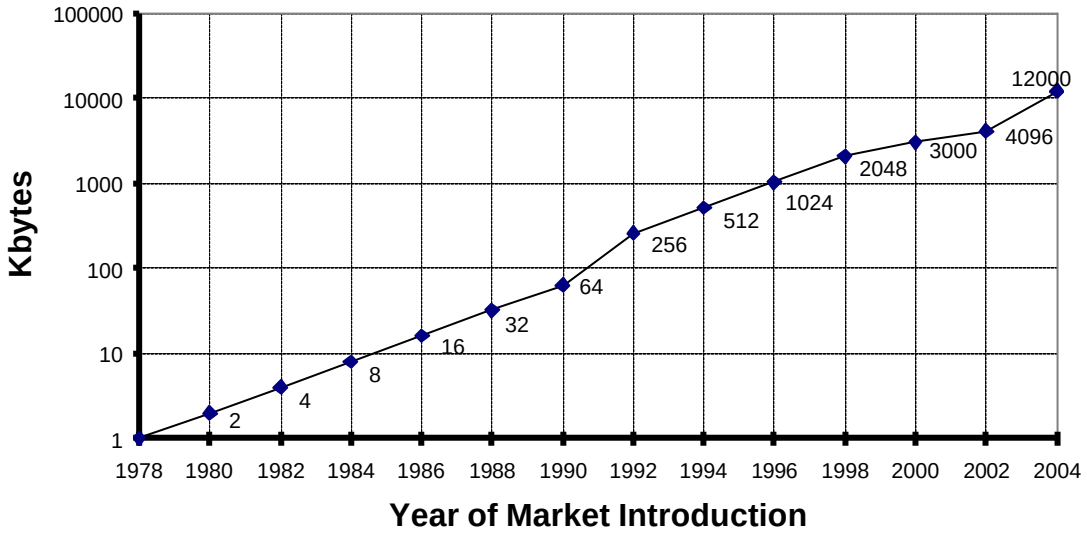


Figure 1.1: Code size evolution of High End TV software

At the same time, companies are willing to remain competitive on the market, and, thus, either have a) to reduce time-to-market and/or production costs or b) have to diversify the products to cover as much as possible market sectors. These steps can help the companies to be successful in the market, to increase their benefit, and to be able to react to the changes in the business world in a more flexible way.

Not only software functionality plays a role, but also other software qualities such as maintainability, reliability, usability as so on, as they define how well the software can address different aspects of its intended context. These aspects relate to the concerns of the respective stakeholders: customers, developers, users, etc.

The properties of software with respect to the aforementioned qualities are usually called *quality attributes*[1]. An extensive taxonomy of quality attributes is given in [BKL95]. However, for the purpose of this thesis it is sufficient to distinguish between *run-time (dynamic)* and *development* related *(static)* quality attributes [Bos00]. Examples of the former are performance and reliability, whereas examples of the latter are modifiability and portability. Quality attributes usually emerge at the system level, rather then being localized in particular software parts, which a system is built from, and they can make a significant impact on the entire product.

A response to the aforementioned challenges presumes dealing with all relevant stakeholders, managing business aspects of the software, and addressing both functionality and quality attributes of software at early development phases. Addressing these factors later, may often result in the development of presumably infeasible or low-quality products and, thus, in the waste of time and money. These considerations resulted in the genesis of a new systematic approach to software development, the *software architecting* (SA).

The software architecting approach claims that the development of a competitive product is not possible without appropriate software architecture, in the same way as construction of a ship or building is impossible without an appropriate blueprint. Many authors rely on this construction engineering metaphor to define the term *software architecture*, conveying the meaning of a high-level design plan [Bos00], [HNS00]. We use the definition of software architecture given by Bass *et al* in [BCK03]:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them.

The software architecture plays an important role in software development process because of the following [BCK03], [Bos00], and [HNS00]:

- *Enabling communication between the stakeholders*. Software architecture represents an abstract description of the system at sufficiently high-level for communicating between the different system's stakeholders.
- *Constraining the quality attributes by making early design decisions.* The design decisions taken at the architecture phase reduce the space of values that the particular quality attributes can be. These decisions have to be carefully revised, and the respective quality attributes assessed, as changing these decisions in the later development phases may have profound consequences on both project schedule and budget.
- *Enabling reuse by consolidating the manageable abstractions of a system.* Software architecture represents a concise design plan of a system that can be grasped by a single person (the architect). This design plan can be applied to other systems that exhibit similar requirements (e.g., product-lines).
- *Outlining system for development*. Software architecture provides the developers of a complex system with a clear view on the roles, responsibilities, dependencies, and interactions of the software components that comprise this architecture. Such a view helps the developers in realizing the component requirements that relate to its environment, in separating the tasks, and in establishing the development plan. This view also eases the cooperation between the development teams.
- *Road-mapping with marketing trends and expectable changes*. The hardware capacity doubles every 18 months according to the Moore's law. The lower speed

[1] Quality attributes are often also called non-functional properties

of software production results in a considerable time gap between the introduction of new hardware and the realization of software functionalities that take advantage of this new hardware. By accounting for the marketing trends (e.g., growing hardware capacity), the software architecture can assist in decreasing the production time, and ensure the introduction of a software product to the market at the optimal moment. By anticipating changes expected in the next versions of a software system, the software architecture can help in reduction the costs of maintaining and evolving of this system.

- *Identifying and modeling the critical parts of a software-intensive system at early development phases*. Being an abstract representation of the system, the software architecture can allow the architect to ascertain which of the parts of this system are the most relevant ones with respect to the requirements. These parts may need preliminary architectural modeling in order to ensure, before investing significant design and implementation effort, that certain requirements can be met.

Software components, being units of composition with "contractually specified interfaces and explicit context dependencies only" [Szy98], are integral parts of software architecture. The software architecture "embodies information about how the components interact with each other" [BCK03], while abstracting from the internal details of components. A separate discipline– component-based software engineering (CBSE) – focuses on component development and assembling the systems from software components. Introduction of a notion of component model [HeC01], [Szy98] made component development more systematic. A component model describes how software components are defined and specified, how they interact, how they are deployed, etc. There exist a number of industrial-strength component model implementations, i.e., the dedicated sets of executable software elements to support the execution of components that conform to the model [HeC01]. The examples of these are COM [Box97], Java Beans [Han01], Koala [OLK00], CORBA [Bol01], and so forth. It is widely acknowledged that using these component models facilitates software reuse, independent development, and separation of concerns.

However, the CBSE approach has not turned out to be a "silver bullet" for producing complex, but still competitive software of sufficient quality. State-of-the-art component-based approaches have a number of shortcomings, i.e. they deal only with the functional aspects of software and do not support the specification and evaluation of quality attributes. Often, quality attributes are addressed only during the last phases of software development. The software is mostly developed according to "fix-it-later" principle [SW02]: the focus is made on software correctness, and quality considerations are postponed until the integration or testing phases. This often results in serious redesign effort spent on tuning the software or hardware in order to meet the quality requirements.

For example, consider response time of a medical imaging software system. It is responsible for acquiring, viewing, archiving, etc of medical images. Initially, two main components responsible for image viewing and acquiring were designed and implemented on dedicated hardware. They had the highest execution priority. A third component, which was responsible for archiving images and assigned a lower execution priority, was developed and tested separately from other components; it exhibited acceptable response time when executed in isolation. After integration of this component with the other two components, its response time degraded drastically, because of resource contention and pre-emption by the high-priority components. To solve this problem, expensive dedicated hardware had to be added to the system, so that the entire composition could work simultaneously. Moreover, significant re-design effort was required to enable parallel execution.

The quality attributes should thus be assessed at the early architecting phase and preferably be considered as an integral part of the CBSE paradigm. This is equally true for operational and development quality attributes. At the early architecting phase, the implementation does not exist yet, but the architects are interested in some qualitative or quantitative estimates of the quality attributes. Here are the three major reasons for early estimation of quality attributes.

1. *Assessment of the quality attributes at the architecting phase is essential to justify design decisions early*. By selecting the appropriate design decisions before starting the product design and implementation phase, it is possible to reduce development costs and, in many cases[2], also production time, as expensive re-development effort is avoided at later stages. It is a well-known fact from the software engineering practice, that it is 20 times more costly, both in terms of time and money, to modify code versus modifying design [SW02].
2. *Assessment of the quality attributes at the architecting phase is required to reuse components from third parties*. By successfully integrating already existing and tested components into a system, the lead-time and costs of products can be significantly reduced, and their diversification can be extended. However, thorough verification is needed to explore the influence of these third-party components on the functionality and quality attributes of the product. The earlier this exploration is performed, the more integration effort is saved.
3. *Assessment of the quality attributes at the architecting phase can make the use of the hardware more efficient*. Currently, due to the lack of means to estimate the required hardware capacity for software-intensive products in advance, the following approach is often used: the hardware is selected in such a way that the software is guaranteed to satisfy its resource constraints anyway. This leads to the underutilization of hardware and, thus, to the waste of resources. Consequently, product costs unnecessary increase and company profit margin decreases. However, preliminary quantitative analysis of software resource demands can assist in more rational choosing of cheaper, but yet capable hardware. This analysis can also assist architects to trade the capacity (and the price of hardware) against future flexibility of the product.

Currently, the techniques for the analysis and estimation of development related quality attributes (e.g., maintainability [BeB00]) of component-based software have already been paid attention to. These techniques are often expert-based and perform only qualitative analysis.

To date, there exist two types of methods for quantitative estimation of operational quality attributes (e.g., timeliness, performance, and reliability): purely simulation-based models and b) mathematical models (e.g., queuing networks [SG98], [SW02]). Both types turn out to be unsuitable for evaluating the quality attributes of complex software-intensive 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. Thus, the quantitative prediction of operational quality attributes of component-based software became the main objective of our research. Particularly, we concentrated on methods for the assessment of performance and static memory demand.

[2] In order to enable selection of the appropriate design decisions, this early assessment should not consume significant effort and should provide architects with accurate and reliable estimates.

1.1 Essence of the thesis

The thesis is structured as follows (see Figure 1.2). The research context, problem definition and research questions to be answered are presented in Chapters 1 and 2. Chapter 3 overviews the current state of the art in the area of software architecting and assessment of quality attributes (QA). The rest of the thesis is subdivided in two parts (unequal is size): one part concerns static quality attributes, whereas the other relates to dynamic quality attributes (see Figure 1.2). These two parts can be read independently of each other. The approach to quantitative estimation of static quality attributes, illustrated by an example from an industrial setting, is described in Chapter 4. The largest part of the thesis is dedicated to estimation of the performance of component-based software. Chapters 5 to 8 address the problems concerning performance estimation for components considered in isolation, including two examples from two industrial domains: Consumer Electronics (CE) and Professional Systems (PS). Chapters 9 to 11 consider different aspects in the research area of performance prediction for component compositions and also provide examples from the two industrial domains. The conclusions and future work are presented in Chapter 12.

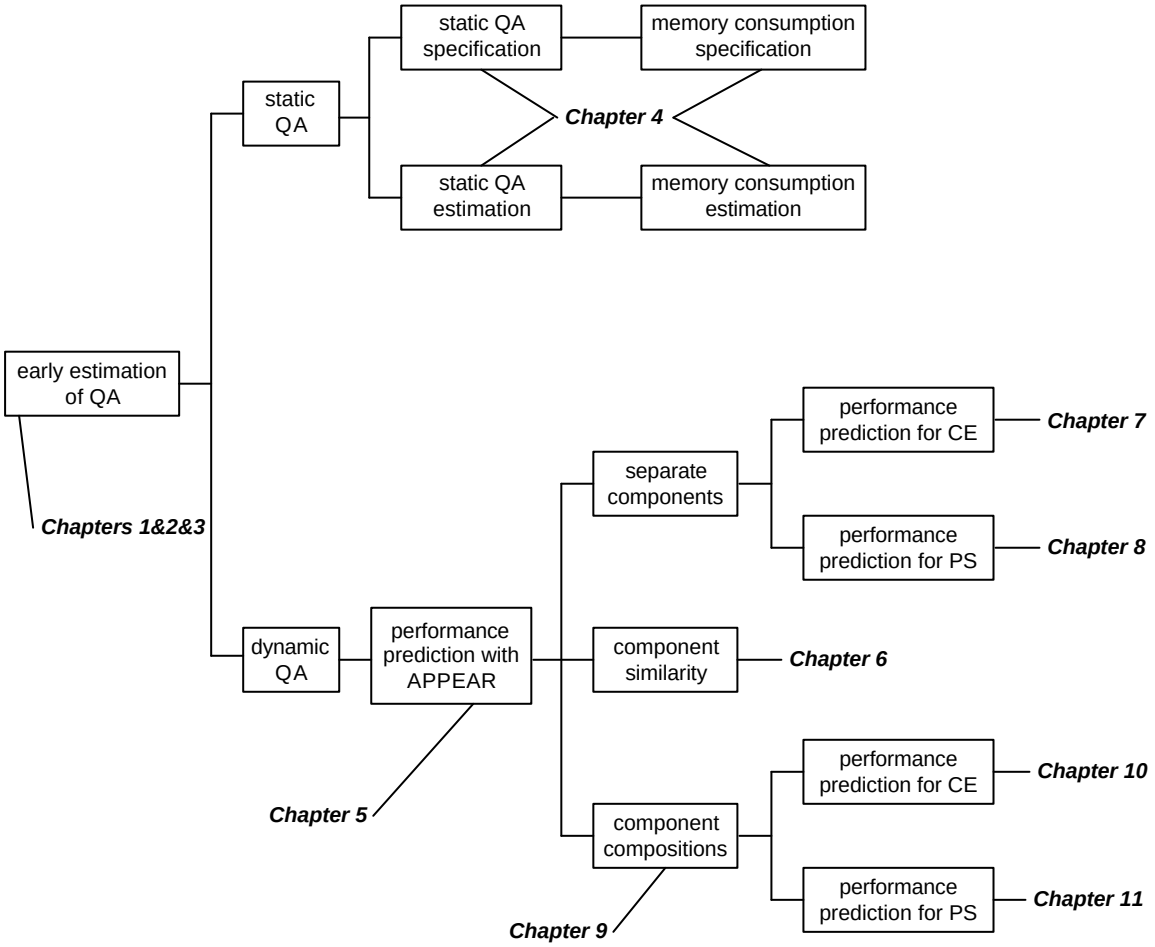


Figure 1.2: Thesis structure

The research work in the project and writing of this thesis were separated as follows. Both authors worked together on research questions, related work, and generalized conclusions. Thus, Chapters 1, 2, 3, and 12 are joint parts. E.M. Eskenazi developed an approach to estimation of additive static QA's and an approach to performance prediction for component compositions. Based on his results, he wrote Chapters 4 and 9. A.V. Fyukov developed an approach to performance estimation of components in isolation and

investigated a problem of component similarity. Based on his results, he wrote Chapters 5 and 6. Furthermore, the work on this thesis is mainly partitioned along the two industrial case studies. E.M. Eskenazi performed two case studies studying the performance of both separate components and component compositions in the CE systems. These case studies are described by E.M. Eskenazi in Chapters 7 and 10, respectively. A.V. Fyukov performed two case studies studying the performance of both separate components and component compositions in the PS systems. These case studies are described by A.V. Fyukov in Chapters 8 and 11, respectively.

The subsections below outline the contents of each chapter.

Chapter 2: Problem description

This chapter describes the main subject of our research: early estimation of quality attributes. First, the chapter explains the importance of various quality attributes for software projects in the Consumer Electronics and the Professional Systems domains, and relates these attributes to the requirements that we collected in the beginning of our research. Second, it enumerates the basic obstacles towards solving this problem in the context of component-based architectures. Third, it introduces the notions of static and dynamic quality attributes. Fourth, this chapter explains why our research concentrated on two particular quality attributes: static memory consumption and performance. Finally, it formulates the major research questions to be addressed in the project and to be answered in this thesis.

Chapter 3: Related work

Both research community and industry have invested substantial effort in developing methods and tools for the early evaluation of quality attributes for component-based software architectures. This chapter overviews the research directions and findings relevant for the main topic of the thesis: the component-based software architecting, evaluation of static properties, and performance estimation of component compositions. For each direction, the achievements and drawbacks of the existing approaches are summarized. The role of our research is discussed with respect to the advantages and disadvantages of the contemporary methods.

Chapter 4: Specification and evaluation of additive static quality attributes

This chapter describes a method for evaluating the additive static quality attributes of component assemblies, based on the properties of its constituents. The chapter identifies relevant factors influencing the static quality attributes of a component composition: diversity and binding. Then, various techniques for specification of static quality attributes of separate components, e.g., reflection interfaces or a dedicated XML-based specification language are exemplified and compared.

The evaluation process can be performed based on specifications of the static quality attributes of components. Depending on components and diversity parameters relevant for an estimation formula, two evaluation approaches– exhaustive and selective– are proposed. These two approaches allow a flexible trade-off between the estimation effort and precision by choosing which components and which diversity parameters are accounted for. Both approaches were validated by an industrial case study: the prediction

of static memory consumption for Koala component compositions. Both approaches exhibited an estimation error that did not exceed 2%.

Chapter 5: Specification and evaluation of dynamic quality attributes: the APPEAR method

This chapter describes a method for the "Analysis and Prediction of Performance for Evolving Architectures" (APPEAR). The method aims at performance estimation of newly developed parts of software-intensive product families during the architecting phase. Early performance estimation makes it possible to verify the feasibility of systems before their implementation, thus saving money and effort otherwise devoted to developing potentially infeasible products.

This chapter explains the drawbacks of the contemporary approaches for performance evaluation of modern complex software architectures. One of the main problems with these approaches is combinatorial explosion due to considering too many irrelevant details.

The APPEAR method avoids this trap by abstracting the irrelevant details by means of statistical modeling. The performance relevant details are, in opposite, considered explicitly and described by a simulation model. This combination serves a basis for the APPEAR method, which helps the architects to construct simple models and, therefore, to obtain performance estimates quickly. The method allows one to flexibly select which parts of the software are simulated, and which parts are described by a statistical model. This flexible choice allows the architects to balance estimation accuracy against estimation effort.

Chapter 6: Similarity of software components

The prediction models provided by the APPEAR method are calibrated on the existing software. As a consequence, the method can only be safely applied to new components that are sufficiently "similar" to the existing ones. Therefore, specific quantitative and qualitative criteria for characterizing similarity are needed. These criteria are based on the heuristics collected during the case studies on the validation of the APPEAR method.

The chapter approaches the similarity of software components in three steps. First, it defines the notion of component similarity. Three questions about the component similarity need to be answered:

How to ensure the accurate performance prediction for adapted components?

How to judge the similarity of the existing and adapted components?

How to incorporate similarity-relevant factors into performance assessment of an adapted component?

Second, the chapter identifies similarity conditions: (1) internal computations of the components, (2) difference in signature type, and (3) distance between signature instances. Third, this chapter proposes a similarity metric. Finally, the chapter describes a number of "escape routes" that can be taken by the architects if these similarity conditions are not satisfied.

Chapter 7: Application of the APPEAR method in the Consumer Electronics domain

This chapter describes the first case study on the APPEAR method validation. The method was applied to the software of modern TV sets. Particularly, the APPEAR method was used (1) to analyze the CPU demand of the existing Teletext acquisition component and (2) to predict the CPU demand of the enhanced version thereof. To achieve the first objective, the simulation model of the existing acquisition component was constructed. To achieve the second objective, a statistical prediction model was fitted to the measurements from the existing acquisition component, and the simulation model was modified to account for new features of the adapted Teletext acquisition component.

The predictions made for the adapted component by the APPEAR method were compared to the measurements collected from the implementation of this component. The average prediction error did not exceed 11%, which demonstrates the good predictive power of the method.

Chapter 8: Application of the APPEAR method in the Professional Systems domain

This chapter describes the second case study for the validation of the APPEAR method. The method was applied to a software component implementing the reviewing of medical images in a medical imaging system. Based on the analysis of the documentation and on performance measurements, the performance significant parameters were identified. It was remarkable that out of more than 100 parameters only 4 parameters were performance-significant and constituted the signature type of the component. Identification of these parameters helped the architects in gaining an architectural insight into the performance bottlenecks: the image displaying procedure, programming of the image processing hardware, and drawing the graphical comments above images. This discovery led to severe code modifications to improve the performance of the reviewing component.

Afterwards, the simulation and prediction models for performance estimation were built and validated. The prediction models were validated by using the observations that were not used for calibration, and this resulted in the maximal relative prediction error of 8% only. Then, one of the models was used to predict the performance of a non-implemented function. Based on the function design, the simulation model was adapted, and 95% prediction intervals were constructed for the response time of this function.

Chapter 9: Performance prediction for component compositions

This chapter proposes a hierarchical approach for predicting the performance of component compositions. This approach allows (1) flexible selection of the abstraction level for behavior modeling, and (2) balance between the estimation effort and estimation accuracy. Additionally, the method employs well-known software engineering notations, e.g. control flow graphs, and does not require much additional skills from software specialists.

This approach considers the following major factors influencing the performance of component compositions: (1) component operations, (2) activities, and (3) composition of activities. The performance model of the entire system is built hierarchically. First, the

contribution of component operations to performance is modeled by means of the APPEAR method. Then, the performance models of activities are specified in terms of control-flow graphs. Finally, the composition of concurrent activities scheduled on non-shareable resources is considered. During each analysis step, various models– analytical, statistical, simulation– can be constructed to specify the contribution of each factor to the performance of the composition. The architects can flexibly choose which model they use at each step. The approach is illustrated by an example of performance prediction for a Car Navigation System.

Chapter 10: Performance prediction for component compositions in the Consumer Electronics domain

This chapter describes the first case study on the validation of the approach to performance prediction for component compositions described in Chapter 9. For this experiment, we considered the software of a TV that performs in steady state, that is, the user just watches the TV but does not try to control it. The goal was to predict the CPU utilization of the major activities that execute in steady state.

In this experiment, we considered only the last two factors: activities and their compositions. The first step, modeling the performance of component operations, was omitted, as there was no sense to model individual component operation implemented within fine-grained Koala components that were not intended for reuse. On the other hand, it was the activities that required an architectural insight into. Therefore, we decided to model, by the APPEAR method, the performance of the entire activities. The final step, building the performance model of the activity composition, was implemented not only by constructing an analytical formula but also by simulation. The reasons for using both techniques were as follows:

- Although the formula provided only slightly larger average prediction errors (6%) than were required (5%), it did not help in gaining the insight into performance relevant aspects.
- The simulation model provided us with insight on performance relevant factors (e.g., scheduling and blocking on the resources) and exhibited average prediction error of less than 2%.

Chapter 11: Performance prediction for component compositions in the Professional Systems domain

This chapter describes the second case study on the validation of the approach to performance prediction for component compositions described in Chapter 9. For this experiment, the medical systems software that consists of several components was selected. We aimed at estimating the response time of the "Archiving" component, executed concurrently with the "Reviewing" component.

During the first step of the approach from chapter 9, the APPEAR models for components were constructed. The APPEAR models for the "Reviewing" component had been constructed in advance, and they are described in Chapter 7. Another component– "Archiving"– is considered in this chapter, and both APPEAR models are built for it.

During the second step, we modeled two activities. In our case, each activity contained only one operation of either "Reviewing" or "Archiving" component. Thus, modeling of branches and loops was not needed, and this step was omitted.

During the third step, we analyzed the contention of these two activities for shared resources. The initial formula was constructed that included all these performance-relevant aspects. Such a form of this formula was based on the observations, analysis of software design, and discussions with architects. Based on the real measurements, a prediction model was created that allowed to confirm the significance of the previously identified performance-relevant factors. The formula was then validated against the real measurements. The predictions obtained via this formula had a relative prediction error less than 2%.

Chapter 12: Conclusions

This chapter summarizes both theoretical and practical results of our research. It explains how and in which chapters the research questions posed in chapter 2 were answered. The main achievements of our research are the methods for evaluation of static and dynamic quality attributes. Both methods help the architects in gaining architectural insight, in producing reliable quantitative estimates, and in trading estimation effort against estimation accuracy. Both methods have been positively validated in real industrial settings. The chapter highlights the advantages of both methods and compares them with the approaches there existed in the architecting field so far.

This chapter also presents lessons learnt during our case studies. A typical instance of such a lesson is the result of the use of statistical models. As such models are the results of curve fitting to measurements, they often do not reflect the real software implementation and dependencies in the system adequately. For example, substitution of certain parameters of such a model may result in a negative value of time.

One of the most important lessons enumerates several guidelines for architects willing to apply the APPEAR method. These guidelines describe the key decisions to be taken during the main steps of method: use case selection, signature type selection, calibration dataset selection, etc.

This chapter also indicates directions for future research.

1.2 What can and cannot be found in this thesis

This section summarizes the main research issues addressed and not addressed in our work.
The following research issues were tackled and thus **can be found** in this thesis:

1. *Estimation of additive static quality attributes*. Our investigations in two industrial domains and interviews with the architects resulted in identification of the most relevant static QAs. All of these static QAs were additive, and, thus, we focused only on QAs of this type.
2. *Performance prediction of average values*. In our approach for the performance prediction, we use the simulation and prediction models for non-implemented components. These models can provide us only with average values of estimates. For obtaining the estimates for the WCET and BCET, the entire implementation is required. Moreover, the worst-case and best-case values were not the primary concerns of our industrial partners.
3. *Performance prediction for adapted versions of existing components, based on the information from their previous versions*. Based on our observations in two industrial domains, we can assume that, nowadays, there always exists initial

software stack, prototype, legacy code, etc. We use the existing components as a basis for constructing performance prediction models for the adapted ones.

4. *Guidelines for and examples of building simulation models containing performance relevant details.* We formulate the major requirements for constructing a proper simulation model (see Section 5.9), and demonstrate the use of the simulation models in Chapters 7, 8, and 10.
5. *Guidelines and examples on the application of linear regression for performance analysis and prediction*. We describe basic rules for constructing the prediction models and criteria for estimating their quality. The examples of the use of linear regression are presented in Chapters 7, 8, 10, and 11.
6. *An approach to performance prediction for component compositions*. Despite the fact, that, due to complexity of this problem, no unified approach exists, we proposed a hierarchical approach that allows one to decompose the problem. After decomposition, various modeling techniques can be applied to each constituent, depending on the goal of the modeling, required accuracy, etc.
7. *Explanation and illustration of the principles that allow trading estimation accuracy against estimation effort.* In the approaches for the estimation of both static and dynamic QAs, we let the architect flexibly vary the level of abstraction and the amount of the details to be accounted for the estimation.

However, a number of research issues were beyond the scope of our research and, thus, cannot be found in this thesis. These issues are listed and commented below.

1. *Estimation of qualitative QA's (safety, maintainability, etc.)*. These QAs were not the goal of our research. Additionally, the software architects that we interviewed did not rank them as the most critical ones[3]. To date, several approaches for evaluating the qualitative QA's exist, e.g. [CKK02].
2. *Estimation of other quantitative QA's (e.g. reliability, availability, etc) besides additive static ones and performance*. As it is impossible to develop a unified method for all quantitative QA, we focused on the two most relevant ones, based on the opinions of the interviewed architects.
3. *Estimation of non-additive static QA's*. We decided to leave these static QA outside the scope of our research, as we did not receive an acknowledgement of their relevance in the industrial domains we operated in.
4. *Performance prediction for components developed from scratch.* We assume that, nowadays, there always exists initial software stack, prototype, legacy code, etc. This assumption allows us to use the statistical models in our approach. For the components developed from the scratch, traditional approaches to performance analysis [Jai91], [SW02], [FM03] can be used instead of the APPEAR method.
5. *Estimation of WCET's*. The estimation of WCET's was not the main concern of our industrial partners, and an estimation error of 20% for the average case was considered sufficient. Additionally, for the estimation of WCET's, the entire component code is required. We aimed at early performance prediction for future versions of software without implementing them, but using the performance models instead.
6. *Performance models of the underlying hardware*. In our approach, we assume that underlying hardware remains unchanged. This allows us to abstract from all the hardware details. However, in the cases when hardware changes, some performance relevant details of the hardware should be modeled explicitly. A short discussion on this problem is presented in Chapter 8. This issue is also indicated as one of the main directions for the future work.

[3] Some of the other important QA's were already addressed by the chosen component technology and organizational measures. The performance and memory consumption were however not properly tackled.

7. *Guidelines for construction of simulation models in general and selection of the best modeling formalism for these models.* The choices of the appropriate abstraction level for modeling and the adequate modeling notation can often be domain- and product-dependent. Thus, we cannot provide a unified set of recommendations for the construction of such a simulation model in general case. Moreover, it is also worthwhile not to force architects to use a particular modeling technique, but to allow them to be flexible.
8. *Construction of prediction model with guaranteed quality and selection of the best regression technique for this model.* The process of construction of the performance model, and the quality of the model are severely dependent on the measured data. Thus, we provide the architects only with basic criteria for estimation of the model quality. Linear regression turned out to be sufficient to satisfy our needs, but analysis of different regression techniques belongs to completely different research domain.
9. *A unified approach to performance prediction for component compositions.* Due to many complex factors (see Chapter 9), influencing the performance of component-based software, there does exist a "silver-bullet" approach suitable for all domains and products. However, we propose a high level hierarchical approach that pinpoints the main steps of performance analysis of component compositions.
10. *Recommendations for proper instrumentation of the code for performance analysis.* This issue lies beyond the scope of our research as well. More information related to instrumentation of the code can be found in [Jai91].

1.3 Publications

This section presents the list of publications that served a basis for the chapters of this thesis.

Chapter 4 is based on the following publications:

- *E.M. Eskenazi, A.V. Fioukov, D.K. Hammer, M.R.V. Chaudron, Estimation of Static Memory Consumption for Systems Built from Source Code Components, Workshop on Component-Based Software Engineering at the 9th IEEE Conference and Workshop on Engineering of Computer Based Systems (ECBS), Lund, Sweden, April 2001.*
- *A.V. Fioukov, E.M. Eskenazi, D.K. Hammer and M.R.V. Chaudron, Evaluation of Static Properties for Component-Based Architectures, Component-Based Software Engineering Track of the 28th Euromicro Conference, Dortmund,Germany, September 2002.*

For Chapters 5, 6, 7, and 8 the following papers were used:

- *E.M. Eskenazi, A.V. Fioukov, D.K. Hammer, H.Obbink and B. Pronk, Analysis and Prediction of Performance for Evolving Architectures, Workshop on Software Infrastructures for Component-Based Applications on Consumer Devices in conjunction with EDOC 2002 (6th IEEE Int. Conference on Enterprise Distributed Object Computing), Lausanne, Switzerland, September 2002.*
- *E.M. Eskenazi, A.V. Fioukov and D.K. Hammer, Performance Prediction for Software Architectures, NWO (Dutch Nat. Science Organization) Progress Workshop, Utrecht, Netherlands, October 2002.*
- *E.M. Eskenazi, A.V. Fioukov, D.K. Hammer and H. Obbink, Performance prediction for industrial software with the APPEAR method, PROGRESS Workshop 2003 on Embedded Systems, Nieuwegein, Netherlands, October 2003.*

- *E.M. Eskenazi, A.V. Fioukov, D.K. Hammer, H.Obbink and B. Pronk, Analysis and Prediction of Performance for Evolving Architectures, Component-Based Software Engineering Track of the 30th Euromicro Conference, Rennes, France, September 2004.*

Chapter 9 is the extended version of the paper:

- *E.M. Eskenazi, A.V. Fioukov, and D.K. Hammer, Performance Prediction for Component Compositions, submitted for 7th CBSE (Component-Based Software Engineering) symposium, adjunct with 26th ICSE (International Conference on Software Engineering) conference, Edinburgh, Scotland, May 2004.*

2 Problem description

2.1 Non-functional requirements in the CE and PS domains

Our research in the area of early estimation of quality attributes was driven by the problems identified in real industrial settings. We have focused on the quality attributes that were found important for two product family projects, running at Philips Electronics in two industrial domains: the Consumer Electronics (CE) and the Professional Systems (PS). Within the CE domain, the software of a TV set was considered, whereas within the PS domain the software of a medical imaging platform was analyzed.

For each project, we collected a set of requirements about quality attributes. This allowed us to devise the requirements that were relevant in both investigated domains. As these domains are different, the requirements were considered to be representative enough to ensure developing estimation methods that can be applied in multiple domains.

The requirements were collected by the following means:

- Interviewing the key architects,
- Browsing through architecture and development documentation,
- Studying System Requirements and Functional Requirements Specifications.

The interviews with architects were conducted in the following way. First, we developed a questionnaire (see Appendix A.1) concerned with non-functional requirements for the software products. This questionnaire served a basis for the interviews. We interviewed eight architects in total. The questionnaire was sent to the architects in advance so that they could prepare for the interview. After conducting the interviews, we summarized the results and sent them back to the architects for verification.

The non-functional requirements are briefly described in Table 2.1. The definitions of the quality attributes related to these requirements are presented in Appendix A.2. The leftmost column enumerates the requirements. Then, the specialization of each requirement for the particular domain is given together with its importance. Note that the comparison of importance is meaningful within a single domain only. The importance of each requirement is assigned as follows:

- **'+++'** means very important,
- **'++'** means moderately important,
- **'+'** means not very important,
- '-' means not applicable.

The importance ranking was agreed upon with the interviewed architects. They assigned ranks to each of the requirements based on their judgment and experience. Notice that due to a limited number of architects and only two projects revised, generalizing the collected requirements to the entire domains cannot be justified. The conducted study about the requirements was aimed only at devising the relevant research questions, and from this viewpoint it was sufficient.

Table 2.1: Non-functional requirements in the CE and PS domains

REQUIREMENT	Professional Systems		Consumer Electronics	
	Interpretation	Importance	Interpretation	Importance
Performance	Important points: • Start-up time, • Response-to-user time, • Image generation speed, • Network throughput.	+++	Important points: • Start-up time • Response-to-user time, • Interrupt latencies. • Average CPU utilization	+++
Low memory consumption	Image size	+	Low footprint: small amount of data and code.	+++
Diversity	• The same generic components are to be used within different modalities[4], • Different types of image acquisition hardware within a modality.	+++	Diversification points: • User's language, • A/V hardware, • Transmission standards • Features (e.g., high- and low-end).	+++
Configurability	Components should be tuneable for different environments[5]	+++	Tenability for various environments via: • Early binding mechanisms, • Parameters in non-volatile memory.	+++
Timeliness	Important for some modalities dealing with: • Real-time image acquisition, • Real-time image storing.	+++	Drivers for a hardware chassis require strict deadlines satisfaction: • EPG, • Remote control, • Teletext.	+++
Reliability	Important points: • Long mean-time between failures, • Data integrity, • Correct interaction between the components.	+++	Increasing of software quality in order to reduce the rate of field calls	++
Safety	Patients and medical personnel are not endangered[6]	+++	Reduction of hazard probability by means of proper hardware/software design	++

[4] A modality is a particular field of medical care equipment (e.g., ultra-sound or x-ray equipment)

[5] Besides the components should be flexibly configurable, they should provide forward and backward compatibility

[6] Safe functioning of the system is addressed within each of the modalities in a modality-specific way.

REQUIREMENT	**Professional Systems**		**Consumer Electronics**	
	Interpretation	Importance	Interpretation	Importance
Availability	• Graceful functionality degradation in case of overload, • Resuming tasks after failure (robustness), • Localised failure, • Low failure rates, • Self-optimization regarding resources.	+++	Seamless restart of the system in the case of fatal error	++
Maintainability	The software stack lives for a long time (10-15 years). Documentation and code should be properly organized.	++	The software stack lives for a long time (4-6 years). Documentation and code should be properly organized.	++
Extensibility	Remote component delivery and addition (without dynamic reconfiguration)	++	Not relevant, as products are not upgradeable	-
Legacy code support	Support of code and data models from the existing modalities	++	Reuse of information from the old software stack.	++
Portability	• Usage of different operating systems within different modalities, • Component technology dependencies must be minimized and localized, • Usage of different hardware configurations to execute the system • Different display sizes and color models	++	Porting on a next generation hardware chassis should be planned	++

REQUIREMENT	Professional Systems		Consumer Electronics	
	Interpretation	Importance	Interpretation	Importance
Scalability	• Control over resource budgets, • Performance scalability with respect to image storage, transfers, and display.	++	Control over resource budgets	++
Reusability	Components should be usable within different modalities	+++	Components should be usable in various products	+++

The collected requirements on quality attributes are mostly related to three aspects:

1. The usability aspects of products, i.e. the aspects that define to which extent a product can be used by its users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use [ISO 9244-11].
2. The product-family-oriented architecting. The requirements related to this property were the same for both domains, e.g. diversity, configurability, and reusability.
3. The resource constraints imposed by the reduction of hardware costs.

In the Consumer Electronics domain all three aspects were equally important, whereas in the Professional Systems the third aspect was less important.

This was a significant difference between the two domains. On one hand, Consumer Electronics is a typical high-volume electronics domain. As the profit margin of each consumer device is small, the ability to use the cheapest hardware is crucial and makes one to use devices that are less powerful than contemporary mainstream hardware. This imposes critical constraints on the resource consumption for the embedded software. On the other hand, the Professional Systems domain is a typical low-volume electronics domain, with products being produced in significantly fewer numbers than in high volume electronics domain. As the profit margin of each system is high, the major price factor is not the costs of hardware, but the costs of the software development. Thus, many resource constraints are not relevant (in comparison to Consumer Electronics).

The study on the non-functional requirements delivered two essential outcomes: findings and decisions.

2.2 Findings

Based on the interviews with architects and documentation, we summarized three major obstacles that complicate early estimation of quality attributes so far.

First, only coarse architectural models can be used for the evaluation of quality attributes, as the design and implementation of components usually do not exist at the architecting phase. Four approaches for constructing such models have been identified in the literature [Bos00]: scenarios, simulation, mathematical, and subjective expert-based reasoning.

The *scenario-based techniques* define evaluation scenarios, which describe certain profiles related to the quality attribute of interest. Then, each candidate architecture is 'benchmarked' against the set of relevant scenarios, and the most appropriate one is chosen. The examples of such techniques are described in [BeB00][CKK02].

In the *simulation approach*, the quality attributes are assessed based on the results of simulation of components and system environment.

For estimation of some operational quality attributes (e.g., performance or timeliness), *mathematical models* have been developed. These models can usually be applied faster than simulation and scenario-based techniques, but their applicability scope is much narrower. Examples of such models for estimating the performance and timeliness are given in [SW02] and [KRP93], respectively. Finally, an experienced developer or architect can initially guesstimate the expected values of quality attributes, based on intuition. He or she then has to apply other approaches to support his judgment. In this case, we have *subjective expert-based reasoning*.

Second, the quality attributes often cannot be attributed to specific components; they usually emerge from the cooperation of a number of components. In this case, the quality attributes of the entire composition have to be derived not only from the quality properties of components, but also from the specification of component interactions. This phenomenon makes it difficult to reason about these emergent qualities in a compositional way, i.e. to derive the qualities of a composition from the properties of separate components. An exhaustive taxonomy of quality attributes and their dependencies on properties of separate components can be found in [Lar04].

Third, usually several quality attributes are in conflict. This means that design decisions that improve one of the quality attributes may deteriorate other(s). For example, portability requirements can lead to selection of platform-independent middleware (e.g., Java and JVM) that can significantly decrease the performance of the application. Finding a set of design decisions that maintain all quality attributes at the required level is often difficult (if at all possible). The art of the architect is to be able to find such a set of design decisions. Some guidelines to achievement the best "mixture" of quality attributes are described in [Sva03].

2.3 Decisions

Table 2.1 presents 14 different QA's that had to be accounted for in the CE and PS domains. There exist no "silver-bullet" that can address all these QA's. Nor can such a approach be developed, as these QA's concern often unrelated aspects of the software development cycle. Thus, we decided to perform our research incrementally: we started with finding an approach for predicting a couple of the most relevant QA's. During the interviews, the architects were requested to indicate the most critical of the QA's, for which no industrial-strength solutions were delivered so far. After analyzing the opinions of the architects, performance was chosen as the quality attribute of interest, as it was rated as "very important" in both domains. Static memory consumption was chosen for the CE domain due to strict memory constraints within this domain. These two particular attributes were selected, since the problems with other attributes had been already (partially) addressed in both projects, whereas the techniques for estimation of these attributes were completely immature.

For the project within the PS domain, the important performance metrics were response time, start-up time, and network throughput. For the project within the CE domain, performance was defined in terms of average CPU utilization and average response and execution times of certain activities. The memory consumption was measured in terms of static memory required by component code and data.

2.4 Static and dynamic quality attributes

Let us discuss the difference between *static* and *dynamic* QA's in more detail. Static ones describe the properties of a component composition that do not change at run-time. These static quality attributes (QA) are determined by the structure of software, but not by its behavior. This structure describes how components are bound to each other, that is, it specifies what the component 'uses' relations are. Component developers can determine these attributes separately for each component. The static quality attribute considered in our research is static memory consumption. Software complexity metrics such as McCabe's cyclomatic number [Mcc76] are other examples of static quality attributes.

Dynamic quality attributes are run-time attributes; they may change in run-time. For their estimation, it is important to consider not only the way that components are bound, but also the way that they interact with the environment and each other. The dynamic quality attribute considered in our research is performance.

Dynamic quality attributes can emerge from the collaborative behavior of the components constituting a composition and, in general case, cannot therefore be attributed to separate components only, in contrast to static quality attributes. For instance, static memory demand can be specified per component, as it is determined only by component internals. However, the timing behavior of a component is determined not only by its internals, but can also depend on the behavior of other components and their interactions. In this case, response time of a composition has to be specified as a function that depends on performance relevant properties of the constituent components and the interactions between these components.

2.5 Research questions

This thesis contributes to the area of component-based architecting by answering a number of research questions. These research questions were derived from the two outcomes of our investigation about the requirements that had been posed for two product family projects in two distinctive industrial domains (see Section 2.1). The questions are classified into several groups and presented in the subsequent sections of this chapter.

Static QA's of separate components are in general easier to determine and specify than dynamic ones, since static QA's can usually be represented by the numbers or simple formulas that can be quickly derived by software developers in advance. Therefore, we focused in our research and in this thesis on the compositional estimation of static quality attributes only.

It is important to emphasize that we studied *additive static QA's* only, as we could not find examples of relevant non-additive QA's neither in research literature nor in industrial software. The additive QA's have the following important property: a QA of a component composition is a weighted sum (linear combination) of the QA's of the composed components. For the sake of space, additive static QA's are further referred as static QA's.

The specification and evaluation of dynamic QA's of a component (composition), in opposite, requires significant effort and, thus, we addressed two apart research problems in our thesis: a) specification and evaluation of dynamic QA's for components themselves, and b) specification and evaluation of dynamic QA's for component compositions.

2.5.1 Specification and evaluation of additive static quality attributes

For the specification of an additive static quality attribute, there are usually many factors related to structure of the composition, type of quality attributes, etc. Considering all the factors in the specification can be redundant and/or time-consuming. **The identification of major (relevant) factors that have a significant impact on the additive static quality attribute** of interest is required for the following reasons: a) making the specification less redundant, b) better acceptable by the architects, and c) reducing the specification and evaluation effort.

Estimation of additive static QA's[7] of a component composition at the architecting phase helps to check early if the software meets the planned resource budgets. This check needs to be performed on architectural models and allows the architect to take more appropriate architectural decisions and to avoid development of unfeasible products. **Construction of the appropriate models for the estimation of additive static quality attributes** is also the subject of our research. For example, checking if the memory space of all components exceeds the memory constraints of a system can indicate that more lightweight components have to be selected.

The models for estimation of additive static quality attributes need to be described in a specification language. This language is used to specify the relevant modeling details in a formal way. Thus, **a specification technique for providing an architectural insight into additive static quality attributes** is needed. This technique should include an explicit and comprehensive view on the relevant details. A formal specification language also serves as a basis for more efficient communication between different architects, as they have to estimate static quality attributes of the composition of components from multiple parties. For example, specification of additive static component properties can be based on a widely accepted standard language (e.g. XML, ASN, UML).

The answers to these questions are provided in Chapter 4 (see Figure 1.2). This chapter presents various techniques for the specification and evaluation of static quality attributes, and illustrates them by means of examples.

2.5.2 Specification and evaluation of the performance of components

Components can be used in different contexts. It is thus necessary to have a model for **estimation of the performance-related properties of a certain component** in a new context. Performance models of such adapted components need also to be constructed. These performance models can be used by the architect to model the performance of the entire component composition. Additionally, these models can be used to check if the hardware, the component will be executed on, can satisfy the resource demands of the component with respect to required performance. As a result, either component implementation can be changed or other hardware can be selected.

Another research issue relates to the modifications of the existing components that already work as parts of component-based software. In this case, **performance estimation of the adapted versions of components** shows if the adapted component fits into the old infrastructure and how it influences the overall performance.

[7] Here, only static QA's that relate to resource consumption (e.g., static memory demand) are considered.

Similarly to specification of static properties, the **specification techniques are required for performance estimation mechanism**, to provide architects with performance insight in terms of only relevant behavioral details of a component, and to serve a basis for architect's communication.

The APPEAR method for performance analysis and prediction answers all the research issues above. This method is detailed in Chapter 5 of the thesis (see Figure 1.2). The Chapters 7 and 8 (see Figure 1.2) describe two case studies that demonstrate the application of the method in different industrial domains.

Performance prediction for an adapted component before implementation is beneficial if it is based on the already existing performance model of the existing component. However, this model can be applied only if the adapted component exhibits similar performance-related behavior as the existing one. Thus, the quantitative metrics of component similarity have to be introduced. These metrics should describe the accuracy of the performance prediction for an adapted component. The quantitative information about accuracy of the prediction can help architects in taking more justified decisions about modifications of components and resulting performance impact of these modifications.

Chapter 6 of the thesis (see Figure 1.2) addresses the problem of **the applicability and accuracy of the APPEAR method for adapted components**.

2.5.3 Specification and evaluation of the performance of component compositions

Performance estimation for component compositions is a complex research topic, and, therefore, we considered it apart from other ones.

Similarly to the estimation of static properties, i.e., for the sake of clarity and time, the architects need to concentrate only on relevant details contributing to the performance of the component composition. In this case, **identification of major factors influencing the performance of a component composition** is needed. Besides performance models of the components, the major factors can include various compositional aspects, e.g. scheduling, release patterns, etc.

Similarly to specification of static properties, **the specification techniques are required to describe an input for an estimation mechanism**, to provide architects with a performance insight in terms of only relevant compositional aspects and to serve as a basis for the communication between the architect and stakeholders. These techniques need to account for both (1) components' contribution to the dynamic quality attribute, and (2) contribution due to component interactions with one another. For example, this technique can specify the behavior of the resource scheduler that assigns resources such as a CPU and communication bus to different components.

Performance prediction for component compositions allows architects to choose the most appropriate architectural solutions before implementation and to better determine the hardware resource budgets for the architecture. Additionally, this estimation facilitates reuse of the components. If these components are accompanied with appropriate performance specifications, an architect can determine in advance how they impact the performance of the entire composition. This feature of the early estimation allows increasing the quality of the software, as it is constructed from already developed and tested components, and shortening the time-to-market, as component development is not required. The method for the **performance estimation of a component composition given the performance-related properties of the components** is intended to support the software architects during the product development. To make the method usable and

transferable to the architects, a stable and defined process (sequence of steps, deliverables, etc.) has to be established. This process has to be followed by the architect when applying the method.

The answers to all these questions described above are based on the extensions of the APPEAR method, and presented in Chapter 9 (see Figure 1.2). Chapters 10 and 11 (see Figure 1.2) present two examples (from different industrial domains) that illustrate an approach to the performance prediction for component compositions.

The estimation technique for both static and dynamic quality attributes must provide an architect with accurate and reliable estimates. The level of accuracy is usually defined by the architect. Due to complex dependencies of static and dynamic properties on component structure and behavior respectively, the specification and evaluation of these attributes with required accuracy can cost significant effort. This effort may be unaffordable at the early phases of product development. Thus, it is important that methods for estimation of quality attributes could allow architects **flexibly trade the specification and evaluation effort against the estimation accuracy**.

This issue was considered when developing methods for evaluation of both static and dynamic properties. Thus, the solutions can be found in both Chapters 4 and 5 (see Figure 1.2), describing the estimation methods.

3 Related work

3.1 Software architecting and architecture evaluation

The most elaborated introduction to software architectures is presented in [BCK03]. The authors describe the fundamentals of the software architecting, the role of the architecture in the business cycle of product development, guidelines to creating and analyzing the software architectures, etc. Several chapters of [BCK03] are devoted to the architectural quality attributes (e.g., performance, security, portability, etc.) and the satisfaction of quality requirements by selecting the appropriate architectural styles. The suggested architecting principles are illustrated using various practical case studies that describe architectures from different domains (air traffic control, World Wide Web services, etc.) and for different quality attributes (interoperability, availability, etc.).

A good survey about different subtopics related to software architecture can be found in [Lan02]. Particularly, Land overviews the following topics: architectural views, architectural styles and patterns, architecture description languages, and architecture evaluation methods. A detailed discussion of those subjects is beyond the scope of the thesis. Only certain topics, which are relevant for the presented research, are discussed in more detail in the subsequent subsections.

A need for evaluation of software architectures and role of this evaluation (with the emphasis on the quality attributes) has been explained in [CKK02]. Clements *et al*. express the necessity of early assessment of the architectural quality attributes and advocate the need for systematic and repeatable approach to this evaluation as "a cheap way to avoid disaster". They also suggest three different approaches to architecture evaluation: SAAM (Software Architecture Analysis Method), ATAM (Architecture Tradeoff Analysis Method), and ARID (Active Reviews for Intermediate Designs). The first method analyses how the quality of the architecture will change as a result of certain modifications in the future. The second method concentrates on the interaction (a trade-off) between different qualities of the architecture. Both methods base their evaluations on scenarios developed by system stakeholders and an evaluation team. The important benefits of these methods are organization of the interaction between the stakeholders, architects, and evaluation team as well as additional documentation of the architecture. The ARID method is applied to evaluate the suitability of design approaches and partial architectures at the early architecting phase. This method incorporates the strong qualities of both ADR (Active Design Review) and ATAM methods. This combination allows bringing together the stakeholders and the designers at early phases of software development. The stakeholders generate scenarios for analysis of the architecture suitability, whereas the designers use these scenarios to brainstorm on the designs and their further testing. Passing of the tests generated by stakeholders will ensure the appropriateness of the design.

The three methods described above have the following important properties: a) they can be employed for high-level analysis of the entire architecture and do not focus on details, b) they enable qualitative analysis only, and c) they indicate what should be improved in the architecture to meet the quality requirements. The methods are quite general, but this is also a disadvantage, as significant acquaintance effort has to be taken to apply them to a particular architecture.

The focus of our research, in comparison to this work, is on the quantitative evaluation of quality attributes. We deal with concrete class of the architectures: component-based architectures. Instead of considering architecture as a whole, we zoom in and consider details at the level of individual components. Besides the identification of the quality of the

architecture, our method can also identify the reasons why certain quality level is achieved or not (architectural insight). We reduce the scope of quality attributes to static additive quality attributes and to performance, yet keeping our approach extendable to more quality attributes.

3.2 Component-based software engineering

Component-based software engineering (CBSE) has already become widely adopted software development paradigm. Component-based approach helps to deal with the growing complexity of software, enables reuse of already implemented software parts, facilitates the independent development, etc. This approach is practice-oriented, and thus only a limited amount of work on fundamentals and theory of component-based technologies is therefore available in the literature.

In [Lan02], Land explains that the research areas of CBSE [Szy98], [CL02] and of software architectures [Bos00], [HNS00] intersect. The software architecture puts emphasis on the structure of software consisting of components, whereas the focus of CBSE is on components themselves and the technology for their integration.

One of the worldwide-known publications on component-based software is [Szy98]. This book provides an extended rationale for introducing the component-based approach into the software development field. It demonstrates the differences between obsolete object-oriented and component-oriented development paradigms and explains why the former does not completely meet the requirements of the modern software market. Moreover, it provides necessary information for making easy step from OOP to CBSE.

All essential definitions of the basic entities of the component-based approach (e.g., component, interface, etc.) are discussed in [Szy98]. The author stresses the multi-functional nature of a component: a unit of composition, unit of independent deployment, unit of encapsulation, etc. Szyperski also presents a detailed taxonomy on the existing components models (COM, CORBA, JavaBeans), explaining their basic principles, rules, advantages, and disadvantages. Additionally, the book catalogs a number of valuable practical guidelines for the development of components, component frameworks, and component architectures. Finally, Szyperski speculates about the future of components: technical tensions, market evolution, and software developer evolution.

Another extensive publication on component-based theory is [LS00]. The authors present the underlying theory for multiple component-oriented solutions (e.g., introduction of *provides* and *requires* interfaces). Further, they focus mostly on two main issues: a) description syntax for components and their interconnection and b) semantics for component behavior and interaction. This book contains only high-level abstractions of component-based systems and does not concentrate on particular component technology.

A good collection of the foundations in CBSE is presented in [BBB00]. The report highlights the main concepts, such as component, interface, component model, component framework, etc. Besides the foundations, the authors also sketch a vision for necessary future research in the field, namely, derivation of the properties of component compositions from the properties of components.

There are two other interesting collections of articles in component-based software engineering field: [HeC01] and [CL02]. The first collection [HeC01] presents various aspects of component-based software engineering. It starts with definitions of components and different views of component technologies. Several parts of this book are dedicated to component infrastructures and component-based architectures. A component infrastructure is responsible for the operational context of components and for supplementary units that

ensure the proper functioning of the components (with a certain quality level). A component architecture usually integrates a number of components, responsible for certain tasks, using pre-defined rules and constraints. The book also contains a number of overviews on state-of-the-art component technologies (COM+, EJB, CORBA) and descriptions of cases and practices illustrating the use and status of CBSE in different contexts.

The second collection [CL02] is mostly concerned with the emergent quality of the component-based software. This book provides a detailed description of CBSE discipline in terms of challenges, objectives, requirements, concepts, etc. Is also contains an overview of modern component technologies. A large part of this book describes the specification and verification of component semantics and component properties (both static and dynamic). Closely related to this part is a part about the integration of components and prediction of the properties of component compositions.

Three other parts of this book are relevant for our research. The first one explains the principles and process of construction of product line architectures from components, and takes the Koala component model [OLK00] as an example. The second one presents a detailed overview of all significant issues for designing of the real-time systems from components (e.g., scheduling, WCET verification, etc.). The third one describes the main aspects of software development for embedded systems and illustrates the basic principles via a case study. The special flavor of this book is conveyed by using various industrial experiences in the field of component-based architecting. Descriptions of the case studies on industrial automation and on business applications facilitate comprehension of the rationale for CBSE and of the component-based development process.

A good overview of component technologies- JavaBeans, COM+, CCM, OSGi, and .NET- currently applied in industry and the assessment of the concepts and principles thereof are presented in [EF02]. The authors identified a number of common aspects of the existing technologies (e.g., interface, assembly, etc.) and claimed that it would be possible to find the basic blocks of component models and, probably, develop a general component model.

Most of the contemporary component models- COM [Rog97], CORBA [Bol01], JavaBeans [MoH01]- require extra computing and memory resources to implement and to run the components. Therefore, these models cannot be successfully used in embedded systems. An effort of several research partners on the development of a lightweight component model resulted in genesis of the Koala component model [OLK00].

3.2.1 Important CBSE notions

In this thesis, we use the definition of a component given by Szyperski from [Szy98]:

A software component is a unit of composition with contractually specified interfaces and explicit component dependencies only. A software component can be deployed independently and is a subject to composition by third party.

Binding defines the connections between components, and, also, which components are included into a composition. For example, the component may need to be excluded from a composition if its interfaces are not required by other components[1].

Depending on a particular component model, binding can be performed at a) compile time (in Koala), b) link time, c) initialization time, and d) run-time, at any moment. The

[1] The Koala component model [OLK00] performs a so-called *reachability analysis* that determines if a component needs building, based on monitoring of component's *provides* interfaces.

first two types are often referred to as *early binding*, whereas the last ones are examples of *late binding*.

The binding is also discerned on its location: (1) *endogenous* and (2) *exogenous*. The former is implemented by the code of components, whereas the latter is implemented outside of them. The exogenous binding makes it easier for third parties to assemble components, whereas the endogenous one makes it more cumbersome. For example, the COM component model is based on endogenous binding, whereas the Koala component model employs exogenous binding.

A reusable component must be customizable for the use in a particular context. For instance, consider a component that must execute its operations in a thread-safe manner. Depending on the context in which the component is used, locks may need to be introduced in order to ensure thread-safeness. However, this kind of information may only be known at binding time, and the component consequently has to be configured at that time. This brings us to a notion of *diversity* of a component that describes possible configurations of a component. The component has a special interface that allows to tune it the environment. For example, the Koala component model introduces a special form of *requires* interfaces, called *diversity interfaces*. A diversity interface specifies a set of parameters that describe what customizations of a particular component are possible. For example, a memory management component may be configured via such a diversity interface to reserve a memory pool of certain size.

We illustrate the important CBSE notions by describing the Koala component model [OLK00] in more detail in the section below.

3.2.2 An example of component model: Koala

Koala is a proprietary component model primarily designed for resource constrained embedded systems and applied in Philips Consumer Electronics products such as TVs. This model was developed to tackle the problems of growing complexity and diversity in resource constrained software-intensive systems often met in the Consumer Electronics domain and to shorten development time. The Koala model supports fast development of software for product families by enabling the reuse of already existing components and by using the explicit architecture. To describe component architectures, a dedicated language, the Koala ADL (Architecture Description Language), was introduced. This language is employed to specify and to parameterize components and component binding.

Three other important Koala features– (1) the use of source code (C language) components, (2) static binding and (3) exclusion of unbound components from the composition– make the resource demands of a component composition low and predictable. This allows one to use cheap hardware in systems produced in high volumes, with profit margin increasing.

However, Koala component model does not incorporate specification of quality attributes. Consequently, it is does not allow yet the architects to reason about the quality attributes of the component compositions.

On the other hand, because of its features, the Koala component model is a good candidate for making extensions to it to enable the prediction of quantitative QA's for component compositions. The subsequent subsections discuss those features in more detail and explain why these features facilitate the prediction of QA's. More information about Koala can be found in [OLK00].

A) Components and Interfaces

Koala components are units of design, development, and – more importantly – reuse. A component communicates with its environment through *interfaces*. A Koala interface is a small set of semantically related functions. A component provides functionality through interfaces, and to do so may require functionality from its environment through interfaces. In the Koala component model, components access all external functionality through *requires* interfaces.

An interface is described syntactically in an Interface Definition Language (IDL). For instance, this is the ITuner interface:

```
interface ITuner
{
   void SetFrequency(int f);
   int GetFrequency(void);
}.
```

ITuner is an example of a specific interface type, which will be provided and/or required by only a few different components. Koala interfaces may contain not only declarations of functions, but also declarations of attributes. An example of such an interface is as follows:

```
interface IRtkConfiguration
{
  int MaximalTaks;
  int MaximalMessages;
  …
}.
```

A Koala component is an encapsulated set of software, with a clearly defined relation to its environment. A component

- *provides* interfaces to its environment. These interfaces are called *provides intrafces*;
- *requires* interfaces from its environment; These interfaces are called *requires intrafces*.

Each component is described in a Component Description Language (CDL)

Two types of components are distinguished: *basic* and *compound*. Basic components are built in C and cannot contain other components; compound components are built by instantiating other components and connecting them together, thereby forming a *containment* or *decomposition hierarchy*. Components of both types may contain a number of *modules*, which implement a part of the component functionality in the C language.

Figure 3.1 shows an example of a component graphically. The large box is a component, whereas the small ones with triangles inside are interfaces. The direction of the triangle denotes the direction of function call: the triangle tip pointing inside the component indicates a *provides interface* and pointing outside – indicates a *requires interface*. The internals of the *CComp* component are discussed in Section B) in more detail.

Explicit component boundaries allow assigning the component certain QA-related properties. These properties describe the contribution of a component to the QA of the entire component composition. For brevity, we call these properties QA's of components. For instance, consider static memory consumption. In this case, such the QA of a component describes its static memory demand.

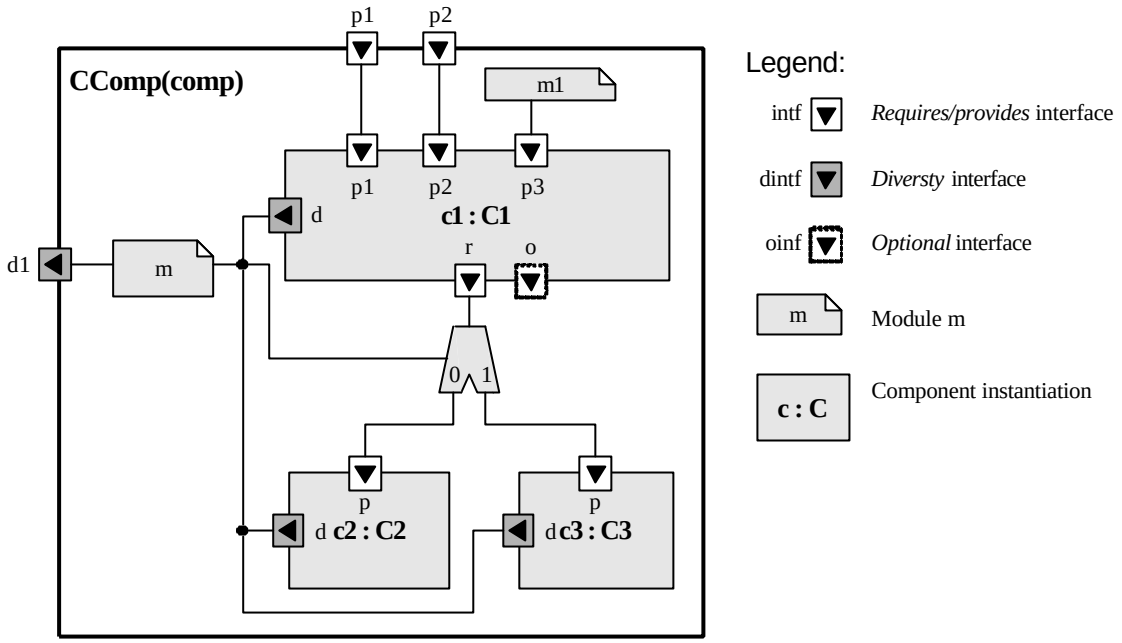


Figure 3.1: A compound Koala component

The possibility to describe interfaces that contain attributes can be used to specify the QA's of components. Such an interface is called a *reflection interface*

The containment hierarchy facilitates hiding the details of an inner component, which may be irrelevant at certain abstraction level. The QA's of component can be described in terms of the QA's of its immediate subcomponents only.

B) Configurations, Binding and Diversity

Figure 3.1 shows how a compound component can be constructed from basic components:

- Interfaces of the compound component can be implemented by subcomponents;
- Interfaces of subcomponents can be connected *directly* (not shown), through *modules* (m) or through *switches* (s);

Diversity is handled by making explicit variants of components, or by using *if-statements* within a component. These mechanisms are called *structural* and, respectively, *internal* diversity. Variants are created when roughly 80% of the code is different; otherwise, the diversity is handled internally. In other cases, the component boundaries have not been chosen properly.

Both types of *diversity* of a component are controlled through *diversity interfaces*. These diversity interfaces are interfaces containing attributes and describe sets of parameters (cf. properties in Visual Basic), which are called *diversity parameters*. These attributes are given a specific value to tune the component to its environment. In Figure 3.1, these interfaces are the *d* interfaces of the compound and inner components.

The diversity interfaces are a special type of *requires* interfaces such that a user of the component *must* fill them in. This compelling allows to attract the attention of the user to component configuration process. This configuration process is important, as if performed properly, it allows for certain optimizations with respect to the resource demand of a component [OLK00].

Structural diversity is implemented by introducing switches in the binding. Such switches are usually controlled by diversity interfaces of the compound component, thus making internal and structural diversity two faces of the same coin.

In Figure 3.1, both the switch and the internal diversity of the subcomponents are handled by a module m, containing expressions that formulate these parameters in terms of the diversity parameters of the compound component. This allows using different sets of parameters at different levels in the decomposition hierarchy.

This explicit management of diversity parameters enables accounting for their influence on the QA's of individual components and component compositions. Particularly, the values of diversity parameters may determine the resource demand of a component, which must be known to be able to predict the resource-consumption-related QA's of a composition. Without the explicit knowledge of the values of these parameters, describing the component resource demand would be impossible, as this demand significantly depends on these parameters. In other words, not taking into account diversity will result in highly non-deterministic predictions.

C) Optional Interfaces

Sometimes it is possible to obtain "enhanced" functionality or non-functional behavioral aspects of a component or subsystem, if it is provided with some additional features (e.g. having an ability to use hardware-accelerated functions if the appropriate hardware is available). The notion of optional required interface has been introduces in Koala to deal with such kind of issues explicitly.

Each *optional interface* is associated with an "*iPresent()*" function. This function may be evaluated `false` (the interface is *not* implemented) or `true` (the interface *is* implemented). Note that non-optional interfaces must always be implemented.

The explicit handling of optional functionality enables taking this functionality into the account, when predicting the QA's of individual components and component compositions. Similarly to the case of explicit management of diversity parameters, this explicit will allow to reduce the non-determinism in QA predictions.

3.2.3 Early assessment of QA's of a component composition

One of the main challenges of CBSE is the development of component compositions with predictable quality attributes [Lar04]. This challenge presumes that QA's of the composition can be derived from the properties of the components.

The requirements to the techniques for development of predictable compositions are sketched in [Sch01]. In [Pel99], the author identifies a number of problems that have to be solved when composing COM components to provide predictability of composed behavior.

An interesting approach to static verification of the correctness of component compositions is described in [GZ01]. Approaches to prediction of the properties of the assemblies based on the properties of constituents were proposed in [LWN02], [Mas02] and [MHW02].

Hissam *et al.* in [HSM03] introduce prediction-enabled component technology (PECT) as a means of packaging predictable assembly as a deployable product. A PECT is the integration of a component technology with one or more analysis technologies. The latter support prediction of assembly properties and also identify required component properties and their certifiable descriptions. The authors discuss the means of validating the predictive powers of a PECT, which provides measurably bounded trust in design-time predictions. In [Lar04], Larsson analyzes a number of existing component models (e.g. [OKL00]) in the light of PECT.

The PACC (Predictable Assemblies from Certifiable Components) initiative was started by SEI (Software Engineering Institute) to address the behavioral and quality aspects of component compositions. The focus of this initiative is on prediction of the assembly behavior from the properties of components. One of their significant research results in this direction is the compositional language CL presented in [ISW02]. This composition language defines a formal syntax and semantics for component interaction and composition. This language describes a composition in terms of components and their interconnections, and the compositional semantics is described in CSP process algebra. The component behavior is specified in terms of reactions; assemblies are constructed from components, but they cannot contain other assemblies. The application of CL is illustrated with examples that demonstrate component property specification and composition property calculation.

In [JMC03], de Jonge *et al.* describe a specification and estimation technique for predicting the use of dynamic memory by an assembly of components. The component interfaces are enriched with the specification of the control flow graph represented by an MSC [Ren99] and the amount of memory allocated and released by each operation. A formal semantics for calculating the resource demand is suggested and demonstrated by predicting the dynamic memory demand for a simple application.

3.2 Estimation of static quality attributes

Development of the techniques for compositional reasoning on static properties is one of the promising research directions in component-based software engineering. The early estimation of some compositional static properties (e.g. the upper bound of resource consumption) is often crucial for ensuring the feasibility of architectural decisions.

Some research effort was taken towards the prediction of memory consumption. An approach, proposed in [USL00], is based on the abstract interpretation theory. A high-level language program is automatically transformed into a function that calculates the worst-case usage of stack and heap space. This worst-case usage is a static quality attribute. The authors of [ZG94] propose and evaluate the methods for modeling the memory allocation behavior of the software. By means of this modeling, the estimation of memory consumption can be performed with reasonable accuracy.

The aforementioned approaches are based on rigorous mathematical theories that pose idealistic assumptions. They also do not deal with complex software input and diversity parameters that can have significant influence on the static property of interest. Some of them do not deal with component compositions, but they can be applied when the entire program code is available.

The added value of our research is the early quantitative estimation of the static composition properties that are vital for embedded resource-constrained applications. We consider component-based systems, and the estimation is based on the specification of the properties of each component. An important feature of our approach is that it accounts for input and diversity parameters of the components when calculating the estimates of static properties. The estimation can also be performed even if some components are not implemented yet, but their static properties are budgeted. Finally, our approach provides an architect with an opportunity to balance estimation effort versus estimation accuracy.

3.3 Performance estimation techniques

There are a number of dynamic quality attributes that are of particular interest of software architects. We decided to concentrate on software performance, as this dynamic

attribute was a concern of our industrial partners (see Section 2.3). Thus, we surveyed research results that were produced in the field of software performance engineering and evaluation.

In our research, we aim at the development of extension to the existing component-based technologies that enables the quantitative estimation of the quality attributes of component-based software. Taking the best features of the conventional component models as a foundation, we focus on the generic enhancement that would be suitable for all of them. The most important requirements for the component models extension so far are presented in [HC01]. This enhancement includes specification of non-functional properties of the components and techniques to assess these properties for component compositions. In addition to this enhancement, our goal is to come up with a set of guidelines (limitations) for the architects that use current component technologies. When following these guidelines, the architects will analyze and assess the architectural quality attributes more efficient.

3.3.1 General approaches to performance analysis and estimation

When analyzing performance of a software system, the most adequate performance approach should be selected based on the structure of the system, required estimation accuracy, performance metric of interest, required effort, etc. This section briefly describes a number of the existing performance approaches. The descriptions of the approaches include the essence of the approach, the scope of the applicability, basic assumptions, limitations, and required effort. The descriptions are concluded with a comparative table for all approaches.

The following approaches are considered (see Figure 3.2): analytical modeling, statistical modeling, and simulation. Methods for worst-case execution time (WCET) estimation are considered apart.

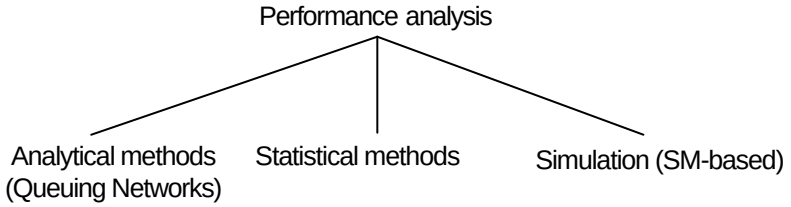


Figure 3.2: Approaches for performance modeling

In [Jai91], Jain advocates the use of all measurements, simulation, and analytical modeling for analyzing 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.

A) Analytical performance modeling

Nowadays, the majority of performance analysis and modeling techniques are based on the theory of queuing networks (QN) [Kle75]. The techniques of this type are usually called analytical[2] [Jai91]. According to the QN theory, a software system consists of *servers* that have to process *jobs*. The jobs have to wait in the *queues*, while the other jobs

[2] This widely adopted term is, in general case, erroneous, as QN models are sometimes solved by means of simulation.

are being served. In modern software systems, CPU, disk, memory and other resources can be treated as servers (see Figure 3.3).

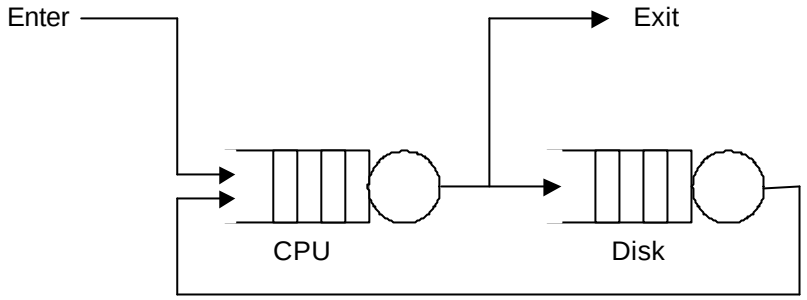


Figure 3.3: A simple queuing network

Further, Table 3.1 describes the parameters of a queuing network.

Table 3.1: Parameters of a QN

PARAMETER	MEANING
l	Arrival rate of the jobs. Most common arrival distribution is the Poisson distribution that means that inter-arrival times are independent and exponentially distributed.
S	Mean service time at server. Most commonly used for these times distribution is exponential distribution.
V	Average number of visits to the server.
Service discipline	For each server, the order in which jobs are served is specified. Most commonly "First Come, First Served" (FCFS) discipline is used.

Based on the aforementioned characteristics, one can estimate various performance metrics of the system: average waiting time for a job, average service time for a job, throughput of a server, etc. These estimations are usually obtained by modeling the behavior of QN in terms of stochastic processes. These processes are functions of time and they represent the state of QN at each particular moment. The state of a system can define, for example, the number of jobs in the system (see Figure 3.4).

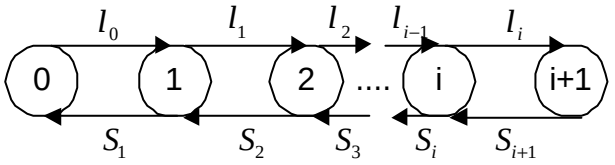


Figure 3.4: States of a QN

The most commonly used processes for modeling the QN are Markov processes[3] [Jai91]. A remarkable property of these processes is that the future state of the process is independent of the past, and depends only on the present state (memoryless distribution of state times). Such a feature significantly eases the analysis of the system. For example, for a simple system with one server, Poisson arrival process, and exponential service time distributions (M/M/1 system), the following performance measures can be estimated:

1. Mean response time:

$$E(R) = \frac{\left(1/S\right)}{1-r} \tag{3.1}$$

[3] Discrete-state Markov processes are called Markov chains.

2. Mean waiting time:

$$E(W) = r \frac{1/S}{1-r} \tag{3.2}$$

In Formulas (3.1) and (3.2), $r = \frac{l}{S}$ - traffic intensity in the system.

The advantages of applying QN theory are the following:

1. The estimates of performance metric of interest can be obtained by solving algebraic equations, and, thus estimation process does not consume much time.
2. As QN's are considered a key approach to performance prediction for many years, enormous amount of experiences and literature on QN's is available.

The following disadvantages can, however, convince one not to use QN's for performance analysis:

1. QN's model the system behavior with a limited language (jobs, servers, and queues). As a result, some performance-relevant behavioral aspects (e.g., interactions between jobs) cannot be described in these terms, and performance insight becomes limited.
2. Certain assumptions must hold for the system in order to enable the modeling and solution of queuing networks. These assumptions are listed below.

The following assumptions (for more assumptions, see also [Jai91]) should hold for application of the Markov processes for analysis of the QN system:

1. Exponentially distributed service times;
2. Individual arrivals of jobs (no bulk arrivals);
3. The jobs are independent, i.e. they do not join, fork, etc;
4. One job does not occupy more than one resource simultaneously.

A classical approach to performance analysis is presented in [Smi90]; it is extended in [SW02]. This work explains the fundamentals of performance analysis and gives the rationale for performance engineering. The authors emphasize the need for tackling the performance problems at the early stages of product development, since correction of performance errors at the later stages is an order of magnitude more expensive than correcting them at early stages. All performance-relevant issues and metrics are classified, and a systematic approach to construction performance models is proposed. Each software performance model is represented with a number of scenarios that are the most important for meeting the performance requirements. The scenarios are described by means of execution graphs. These graphs, system model and resource overhead specifications allow constructing a system execution model. This system execution model builds on the principles of queuing networks and can sometimes be solved analytically. For complex systems, with many scenarios, distributed to different CPU, simulation has to be used to obtain an approximate solution. The supportive SPE-ED tool [SW97] is developed to interactively assist the performance engineer in applying this approach.

There are a number of approaches that aim at early performance estimation, based on architectural models. Classical approaches [SG98] to performance prediction use queuing network models, derived from the structural description of the architecture and performance-critical use cases. Other approaches include specific architecture description styles [ABI00].

An interesting approach is proposed in [HWR99]. An executable prototype (a simulation model) generates traces expressed in a specific syntax (angio-traces). These

traces are used for building performance prediction models, based on layered queuing networks.

Preliminary analysis of the behavior of the investigated software systems in CE and PS domains revealed that aforementioned assumptions (e.g., distribution of service times) do not hold for these systems. Thus, QN-based techniques were not applied for performance analysis in our research.

B) Statistical techniques

Statistical approaches to performance prediction are data-driven approaches that are based on the principles of the statistical regression analysis. *Statistical regression analysis* is a statistical technique which models the relation between a set of *input* variables, and one or more *output* variables, which are considered dependent on the inputs, on the basis of a finite set of observations (measurements). The resulting model is a prediction model that, being fed with values of the input variables, calculates an estimate of the value of the output variable. The goal is to obtain a reliable generalization that means that the prediction model, calibrated on the basis of a finite set of observed measures, is able to calculate an accurate prediction of the dependent variable for a previously unseen value of the independent vector. In other terms, this technique aims to discover and to assess, on the basis of observations only, potential correlations between sets of variables and use these correlations to extrapolate to new scenarios. Usually, the statistical prediction models are constructed as depicted in Figure 3.5.

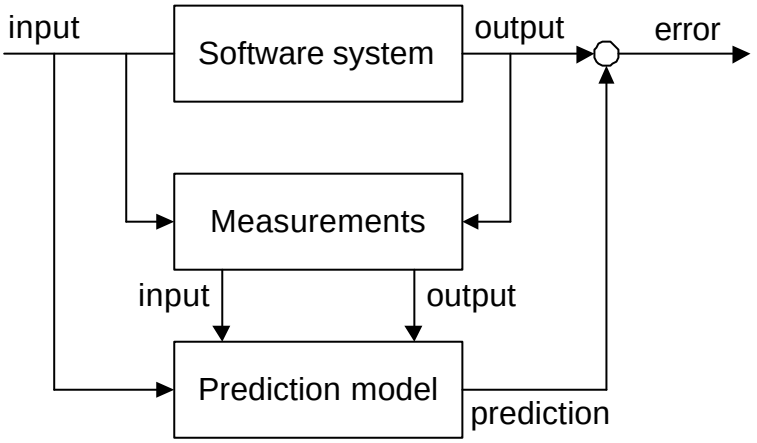


Figure 3.5: Calibration of the statistical prediction model

The calibration process consists of the following basic steps:

1. *Definition of performance-relevant scenarios*. This step aims to select a representative set of execution scenarios to be used for collecting measurements.
2. *Definition of the input parameters*. During this step, performance analysts, together with software designers, have to select a set of parameters to be used as input variables for the prediction model. This step can be repetitive if such parameters are difficult to identify or if a prediction model with pre-selected set of input parameters has insufficient quality. We assume that output variable is pre-defined and it is one of the widely used performance metrics (response time, throughput, etc.)
3. *Collection of measurements.* This step consists in execution of the performance-relevant scenarios and in collection of the values of input and output variables.
4. *Construction of the prediction model by means of regression techniques.* Based on the measurements, regression techniques construct an algebraic expression that approximates the dependency between input and output variables.

5. *Validation of the prediction model*. This step includes checking of the quality of the model and the quality of the predictions for alternate scenarios. The quality of the model is defined by a number of statistical criteria (R-squared coefficient, residual distribution, etc.). The quality of the prediction is usually determined by statistical comparison of the predictions obtained for the scenarios that were used for calibration and for some alternate ones.

Note that this entire process is iterative, and can be repeated until the prediction model has a sufficient quality.

There exist two types of regression models: linear regression models and non-linear regression models. The former approximates the dependencies in a system by linear expressions:

$$y = b_0 + \sum_{i=1}^{N} b_i \cdot x_i \,. \tag{3.3}$$

In this formula, y is a dependent variable (output), x_i are independent variables (input parameters), N is the number of input parameters, and b_i are the regression coefficients. In the latter case, the dependencies are specified by non-linear expressions, e.g.

$$y = b_0 + \sum_{i=1}^{N} x_i^{b_i} \,. \tag{3.4}$$

Linear models are the most commonly used ones, and, thus, we also decided to use them in our research project. A good overview of various existing regression techniques is provided in [Jai91].

The main advantages of using the statistical approaches for performance prediction are the following:

1. Possibility to abstract from irrelevant details by means of the prediction model fitted on the limited set of parameters.
2. Fast performance estimation by prediction model (e.g., in comparison to simulation).
3. Solid mathematical basis for performing various tests on the collected measurements. These tests help in exploring complex dependencies in the system.

The main drawbacks of the statistical models are the following:

1. They do not provide any insight about performance-relevant behavior of a system, since they consider the system as a "black box".
2. They provide reliable predictions only if the assumptions of the underlying techniques are valid for the prediction model (see e.g. assumptions below).
3. They provide reliable predictions only for the measured range of the input variables. For the variables, located outside this range the approximation may turn out incorrect (see Figure 3.6).

There are a number of assumptions needed to ensure the quality of a prediction model constructed by means of linear regression (for more details, see [WEI95], [MoH01], and [MR03]):

1. Residual (prediction error) is normally distributed.
2. Residual standard deviation is constant (homoscedasticity).
3. There are no outliers.
4. There are no influential observations.
5. There is no correlation between input parameters of the prediction model.

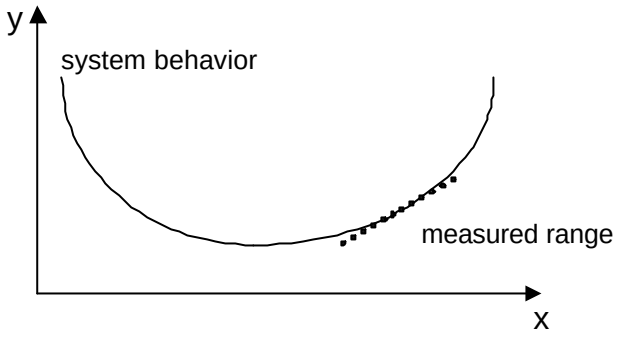


Figure 3.6. Approximation is valid for the measured range only

Our research was in particular inspired by an approach proposed by Bontempi *et al.* in [BK02]. The authors use regression techniques for the performance prediction of various embedded software systems (among others, MPEG2 decoder). The approach proposes to tackle two problems in performance estimation: significant amount of time consumed while executing a code on cycle-true simulator, and selection of the performance-relevant parameters among hundreds of the input ones.

The code of a software system in question is compiled by dedicated tools into the code based on the reduced set of instructions (see Figure 3.7).

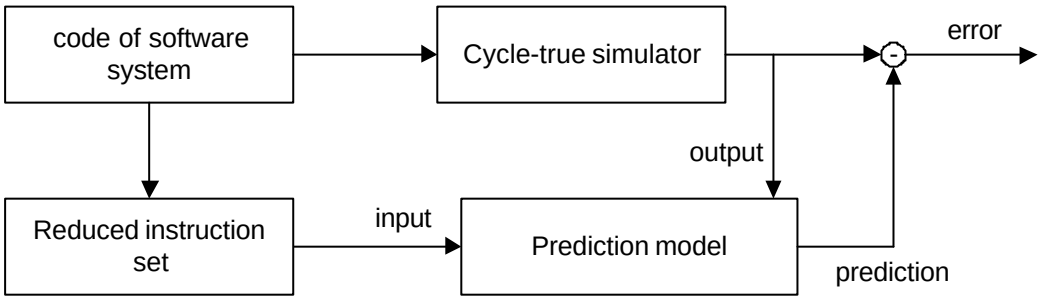


Figure 3.7: A data-driven approach for performance prediction

Afterwards, the numbers of instructions of different types are used as input variables for the prediction model. Execution of the code on cycle-true simulator yields performance measures, i.e. values of output variable for the prediction model. The prediction model is calibrated based on the aforementioned values of inputs and outputs. The performance predictions obtained by means of this model were compared to the measurements on cycle true simulator.

The experiments with the software of different types showed that prediction model had a low prediction error, below 20%. Another important result was the comparison of the estimation effort. Cycle-true simulation took many hours to return a performance measure, while a performance prediction model could provide the result in less than one second.

This approach exhibited a number of positive aspects:

1. It allows one to obtain performance predictions much faster than by means of pure cycle-true simulation.
2. The abstraction by means of reducing of the instruction set and construction of the statistical model does not deteriorate the prediction quality.

However, this approach also has a number of drawbacks:

1. The entire code is required for the performance prediction. A program that consists of the instructions from reduced set cannot be created from architecture and design description.
2. This approach is not oriented at component-based systems as it considers software system “as a whole”.

The work presented in [GMH01] also describes an approach to software performance prediction that employs statistical techniques. It considers the use of linear regression techniques only, whereas the approach described in [BK02] additionally considers the use of adaptive local regression techniques (lazy learning [Bon99]). Both approaches address performance prediction when the software has already been implemented.

We extended the approach from [BK02] such that it does not require the entire code be available to construct high-level performance models. These models are rather built on the basis of architectural and design specifications. Additionally, this extension is component-oriented, that is, the performance estimate of the component composition can be obtained by using the performance models of the constituents.

C) Simulation

A well-known practice for early performance analysis is the construction of a simulation model that captures the performance-critical parts of the software. Simulation is the realization of a model for a system in computer executable form [FM03]. Simulation of the system with the purpose of performance analysis allows one to model the run-time behavior of a system, to gain insight into performance-relevant parts of the system, and to identify performance bottlenecks.

Simulation models are usually constructed when the complexity of the system under consideration does not allow using simple analytical approaches for performance analysis and reliable prediction. Simulation model do not only allow an analyst to decrease the level of complexity, but also allow the analyst to flexibly adjust the level of modeling details according to the goal of the analysis, available timing budget for the simulation, and required accuracy of the results.

Simulation models usually describe the system behavior in a dedicated (high-level) programming language. Often, this language is a part of a dedicated simulation environment that provides user with the tools to specify system behavior with various formalisms (e.g., state charts), to compile the simulation program, to execute it, and to inspect the observable behavior of the model. A number of the existing simulation environments are presented in Appendix S. In our research, we applied the COVERS simulation engine that has the following properties:

1. It allows user to specify the behavior of the active objects with state charts and communication between these objects by means of message passing.
2. It translates these models into C++ code that is compiled and executed.
3. It supports model observation and modification during run-time.
4. It provides an extended class library supporting statistics gathering and visualization.
5. It allows building real-time models to be used with real environment.

A screenshot demonstrating some feature of the COVERS simulation environment is depicted in Figure 3.8.

Some other simulation languages and techniques are described in [FM03].

The use of simulation models for performance analysis has the following advantages:

1. Simulation models can provide much more performance insight than statistical ones can.
2. Simulation models can be used for modeling complex dependencies that cannot be covered by analytical or statistical ones (e.g., dependencies on the history, behavior on the long run, etc.).
3. Simulation models can be made as accurate as needed (e.g., to achieve higher estimation accuracy).

4. Simulation models provide an opportunity to change performance relevant parameters, and architectural solutions and to quickly observe how these changes influence the performance (without implementing them in software product).
5. Some simulation environments support automatic code generation from the models.

However, the use of simulation models also has a number of drawbacks:

1. Selection of both level of abstraction and simulation language are the potential points of failure, since there exist no general guidelines for proper selection, and an error in selection can lead to an inadequate model of system behavior.
2. Construction of a simulation model is an iterative process that can require significant effort.
3. Selection and generation of correct input data for the validation of the simulation model can require a) additional effort for choosing the representative input parameters and sufficient amount of this data for model validation, and b) additional knowledge in statistical distributions.
4. Validation of the simulation can consume much time. However, the models that are not sufficiently and properly validated can provide useless and misleading results.

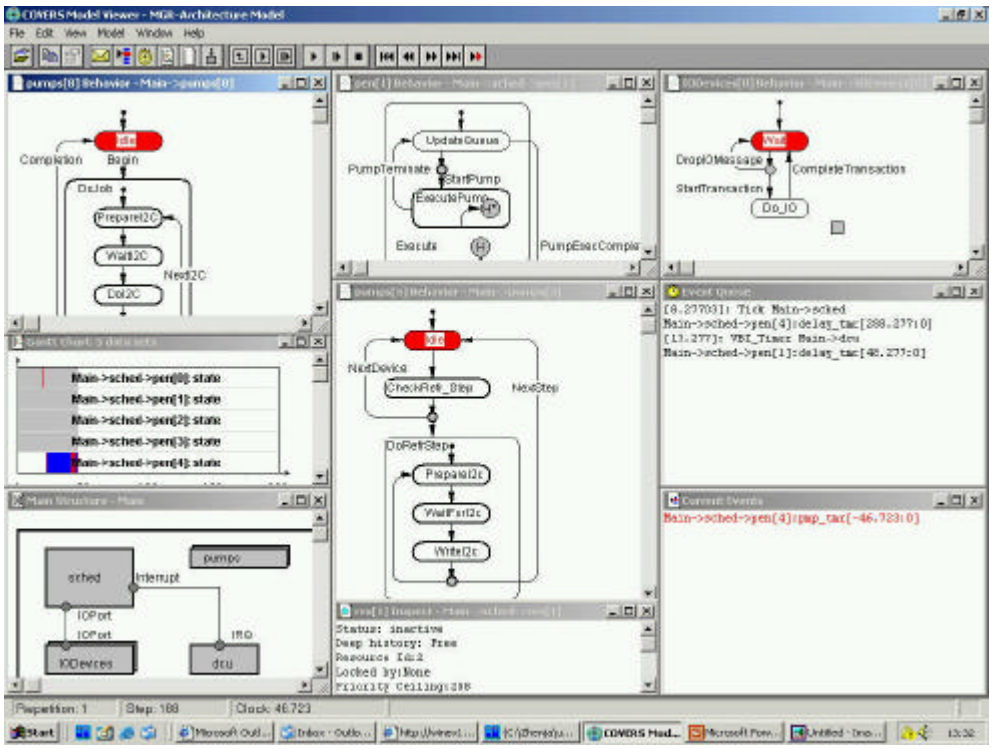


Figure 3.8: COVERS simulation environment

Table 3.2 presents a brief comparison of the performance modeling techniques.

Table 3.2: Comparison of various performance modelling techniques

Approach\ Criterion	Insight	Accuracy	Effort	Assumptions	Simplicity
Analytical methods (QN)	High	Medium	Low	Strict: exponential distribution of arrival and service times	Medium, requires learning time
Statistical models	Moderate	High	Medium, but extensive measurements required	Typical for statistical models (e.g., residual distribution)	Medium, requires learning time
Simulation	High	High	High, for construction and validation	No	High, well-known engineering notations

3.3.2 WCET estimation techniques

WCET estimation techniques were interesting and relevant for our research, as they also deal with the CPU demand of software, which is important for assessment performance. In addition, these techniques consider the control flow within the software, which is also concern in our hierarchical approach, described in Chapter 9, for predicting the performance of component compositions. Although, we hardly employed the WCET estimation techniques in our approaches to performance prediction and in our experiments because of the following reasons:

1. These techniques require the entire code (execution graph) available, while our goal was to build the performance models from design models.
2. During analysis of the modern complex software, these techniques run into the problem of combinatorial explosion of details.
3. We did not deal with hard real-time systems, and, thus, we were rather interested in reasonably accurate average performance estimates then in tight WCET estimates.

The analysis of worst-case execution time (WCET) is important for component-based real-time systems. On one hand, this property is static, as it is usually estimated statically. On the other hand, WCET is required to estimate the schedulability of software, which is a dynamic property. The estimation of WCET for component-based software is complicated, as execution paths often pass through multiple components. Analysis of all paths can lead to combinatorial explosion of details. Despite the fact that WCET estimation was not the primary focus of our research, we investigated the state-of-the-art approaches in this field.

The worst-case execution time analysis can be subdivided into three parts: (1) high level analysis or extraction of the control flow information, (2) low level analysis which takes into account influence of hardware architecture, and (3) actual calculation of WCET.

The main task of extraction of the control flow information is to reduce the pessimism of WCET estimation due to infeasible paths. The extraction can be performed manually or (semi-) automatically. Manual extraction means that functions (component methods) are annotated with the relevant control flow information like the maximum number of iterations [BP01], [KP00], [KP01] and [PK89].

With the automatic extraction one analyses the structure of a program and the use of variables and function parameters to reduce a number of infeasible execution paths. There exist a few approaches for the automatic extraction. The two major ones are based on the theory of abstract interpretation [Gus00], [GE98] and symbolic execution [Alt96], [LS99]. The former suggests a special interpretation of the subset of a high-level programming language. The semantics of operations on variables builds not on scalar values, but on the

sets[4] of values. By doing so it is possible not only to extract the worst-case control flow information, but also to obtain the superset of possible values for every variable of the analyzed function.

A symbolic-execution-based technique typically constructs a dedicated virtual machine that executes a program (function) with the values of variables represented in the symbolic form. During the symbolic execution of the program, the value of each variable is represented as an expression over other variables or constants. By confronting the symbolic values of the variables to the controlling expressions of branch operators and loops, it is possible to eliminate some infeasible paths.

The low-level analysis helps to incorporate the influence of hardware aspects into the execution time of atomic blocks. Typical hardware features that need to be addressed during this analysis are the following: memory, bus, caches, pipelines, branch predictors, etc.

The actual calculation of WCET may be performed with three classical approaches described in the literature: (1) path-based techniques [EES01], (2) tree-based techniques [Alt96], [CP01] and (3) implicit path enumeration techniques (IPET) [KP00], [KP01], [EE00] and [PK89].

The path-based techniques search all possible paths for the longest one, whereas the tree-based techniques build on the principle of bottom-up traversal of a program syntax tree. IPET-based techniques represent the information about control flow and execution times of atomic blocks with a set of algebraic and/or logical constraints. WCET estimates are calculated by maximizing a certain objective function given the set of constraints.

Unfortunately, many approaches proposed in the literature implicitly integrate high-level analysis, low-level analysis, and actual calculation of WCET into an indivisible framework. This makes it difficult to reuse these approaches in a new context, e.g. when the hardware platform has a different CPU. The remarkable exceptions are presented in [BP01], [EE00], [EES01], and [EES01a].

The authors of [GL03] advocate the use of programs with single execution paths only. To that end, it is necessary to identify all (sub-) paths that depend on the input variables and convert those paths to ones that do not depend on input. The identification is based on the abstract interpretation. The abstract values are ID (input dependent) and NID (not input dependent). The authors give an example by providing an operational abstract semantics to interpret a program written in the While language. This kind of analysis is very useful for compositional performance reasoning, as input dependency is also an issue there.

3.3.3 Estimation of performance of component compositions

The problem of performance prediction for component compositions has become a challenging research topic for different groups, and currently it is being tackled in many different ways. There exist a number of practical and academic approaches to compositional performance prediction.

The importance and inevitability of this problem is clearly formulated in [Sit01]. The author puts the main emphasis on the difficulties associated with this problem: proper component specification at component level, complexity of component-based systems, estimation accuracy, etc. The paper presents a concise and clear overview of the current work in the field and main aspects to be considered for solving the problem.

[4] The considered set can be a joint set of scalars and intervals.

The article [WOS03] motivates the need of a sufficiently expressive specification language to formally describe performance-related aspects of components. This language should cover more aspects than a normal functionality specification language does. Major problems occur when a user-define types need to be considered. This is demonstrated with the Map data type example.

The paper [WMW03] describes a framework that relates components with their performance sub-models and component assemblies with the corresponding system level performance models. The authors suggest to use Layered Queuing Networks to model performance at both component and composition level. The approach targets performance estimation only at early stages, that is, no source code is available yet. A special tool, called "component assembler", is introduced to assemble the components. A similar automated tool is to be used to compose the performance models. The approach is demonstrated using a simple industrial case study, in which the authors investigate the best possible distribution of an e-business three-tier application among different CPU configurations. In general, this work seems to be in the very initial stage and needs be elaborated.

Hamlet *et al.* investigate in [HAT03] the problem of transformation of input parameters when a “provides” interface operation invokes a “requires” interface operation. The input parameters of both caller and callee are subdivided into a number of sub-domains, and a transformation matrix is built to map the domain of the caller onto the domain of the callee. By constructing this transformation matrix and sampling the performance measurements in all sub-domains of both components, it is possible to make performance predictions. The authors checked the approach using a simple composition comprising two components. The reported results are encouraging, but there are still a number of unclear points in the approach:

1. How the approach will behave for greater number of parameters?
2. How the approach will behave for many components and complex control flows?

The article [BM03] describes the initial steps to create an automated tool that support CB-SPE paradigm. This paradigm is based on adapting the combination of the SPE tool [SW97] and UML RT to the component-based context. The article states the number of requirements and describes some steps that the target tool should support.

The authors of [MHW03] raise an interesting problem related to standardized representation of measurements and predictions of component and assemblies, respectively. To that end, the statistical descriptive and inferential techniques are suggested to be used. Notions of confidence intervals, normative/informative tolerance intervals are found to be useful. As an example of useful inferential statistics, the authors suggest to use the magnitude of relative error. Also the coefficient of determination that indicates the degree of linear correlation is recommended for normalization.

In [DMM03], Dumitrascu *et al.* consider a general method and supporting tool for predicting the performance of component assemblies. The target component models are wide known industrial ones: COM (+), JavaBeans, .NET, and CORBA. Every concept is demonstrated with an example from .NET. The authors noticed that most of CBSE performance researches ignore the component infrastructure. So, the authors decided to explore the dependency of .NET components performance on different parameters of the infrastructure. The prediction of assembly properties is supposed to be performed using traditional models such as Queuing Networks, Petri Nets, Markov chains, etc.

The authors of [RZJ02] are concerned with the compositional timing analysis of heterogeneous platforms that have different scheduling domains. They distinguish four types of schedulers: RMA, EDF, TDMA, and round-robin. For those schedulers, a number of analysis techniques already exist. In order to combine those techniques, the authors

introduce (1) input and output event model interfaces (EMIF) and (2) event adaptation functions (EAF). The former specify which type of analysis can and/or has to be applied, whereas the latter allows one to adapt an output event model to a certain input event model. The approach is demonstrated with a couple examples (both feed-forward and feed-back) that need predicting the best- or worst-case response times for a composed system. The feedback systems, however, are not necessarily analyzable without appropriate EAFs.

In [BM03] and [BMW04], Bondarev *et al.* describe a scenario-based simulation approach to the prediction of the fraction of missed deadlines in a soft real-time component based software system. The suggested approach reconstructs tasks from component descriptions and applies simulation to assess the response times of these tasks. A simulation model of the scheduler is executed for particular scenarios to obtain the estimates of the response times. The scenarios are suggested to use for reducing the complexity of modeling, associated with combinatorial explosion of details related, e.g., to input parameters. The use of simulation to account for scheduling artifacts such as blocking and preemption is similar to our hierarchical approach described in Chapter 9.

Most of the existing approaches to compositional performance reasoning are analytical. They are built upon the mathematical foundations, and often based upon too strict assumptions about the systems under question. The modern complex software systems with hundreds of components do not satisfy these assumptions and, thus, the analytical approaches are predisposed to combinatorial explosion of details. Moreover, these approaches can hardly be accepted by software architects, as they require solid scientific background and long learning curve. Employing of statistical regression techniques reflects the software behavior only in terms of curve-fitted formulas and hides relevant architectural insight.

Some approaches require the entire code of software system, and, as a result, they are not applicable at the early architecting phase. Simulation-based approaches (e.g., cycle-true simulation) usually imply the simulation of the entire software stack, with all details. Construction of such a simulation model is quite time consuming. Moreover, complex simulation models are as error-prone as the original software. Finally, not many contemporary approaches sufficiently use the knowledge about existing versions of the software. The existing software can be measured to collect the performance relevant information. This information can be then used for constructing performance prediction models for adapted versions of software.

4 Specification and evaluation of additive static quality attributes

4.1 Introduction

Static quality attributes of a component composition are one of essential qualities that need to be assessed as a part of early feasibility checks. These attributes are those quality attributes that do not change at run-time (in opposite to dynamic quality attributes). In this chapter, we treat a subclass of static quality attributes: additive static quality attributes. This additivity property allows the estimation of the quality attribute of a component composition just by summing up the quality attributes of the constituent components. The static memory demand is a typical example of additive static quality attribute. The early assessment of this attribute is needed for checking if the implementation of an embedded system will meet memory constraints.

We propose an approach to the assessment of the additive static quality attributes of component compositions. These quality attributes are expressed in terms of the quality attributes of the constituent components and certain compositional rules. The approach is illustrated by applying it to a Koala component composition to predict the static memory consumption of this composition.

This chapter is structured as follows. Sections 4.2, 4.3, 4.4 introduce the basis of our view on the static quality attributes of a composition, describe factors influencing these quality attributes, and present a general expression for the estimation of these attributes, respectively. Section 4.5 compares various approaches to the specification of the static quality attributes. Section 4.6 explains two approaches– *exhaustive* and *selective*– to the evaluation of these attributes. Section 4.7 elaborates on the construction of estimation formulas and their accuracy. It describes two techniques for constructing the formulas: (1) *empirical* and (2) *statistical*. Section 4.8 extends Section 4.5 by presenting a general way for the specification of static quality attributes via the XML language. Section 4.9 exemplifies the suggested mechanisms for the specification and evaluation by a case study in the domain of TV software. Finally, Section 4.10 summarizes the chapter.

4.2 Method basis

This section introduces a framework and basic foundations for assessing static quality attributes of component compositions.

4.2.1 Components and component instances

The term *component* usually refers to a component class or *component type*, which can be instantiated to a number of *component instances*. In the CBSE literature [Lar04], the term component can be used to refer either to component type or component instance, depending on the context. In Sections 4.2 and 4.3, we will use the term component, only when is clear from the context whether the component instance or component type is meant.

We assume the following underlying component model. Each component type describes which components can be instantiated by its instances. This instantiation results

in the creation of a hierarchy of subordinate component instances. We call this hierarchy a *containment hierarchy*. An example of such hierarchy is given in Figure 4.1.

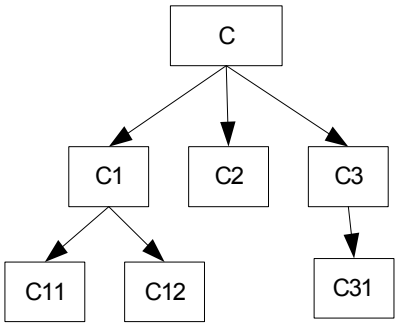


Figure 4.1: An example of containment hierarchy

The rectangles denote the component instances, whereas the arrows show which component instances can be potentially instantiated by which ones. Whether a particular component is instantiated depends on the context, e.g., certain configuration parameters. The outer component instances are also responsible for configuring the inner ones. For instance, the *C1* component instance creates the *C* component instance and configures it.

Component hierarchy has always a tree-like structure, that is, a component instance can be a subordinate only to a single component instance. However, multiple instances of the same type are permitted. For a particular component instance C_i, we denote the corresponding component as $type(C_i)$. For instance the, the following formula can hold for the containment hierarchy from Figure 4.1:

$$type(C11) = type(C31). \tag{4.1}$$

4.2.2 Framework description

As explained in Section 4.2.1, we assume that each component composition forms a containment hierarchy, as many existing component models allow encompassing of component instances by other ones. Within this hierarchy, three types of components are distinguished: (1) basic components whose instances do not contain instances of other components, (2) compound components whose instances may contain instances of other components (see Figure 4.2), and (3) the composition itself that is the top-level component.

There are two inputs for the framework: specifications of the quality attributes of the components and compositional rules both for assembling components and for calculating the static quality attributes of a composition. The framework implements a mechanism that composes component quality attributes according to the rules and estimates the values of these quality attributes for the entire composition. Depending on the type of a static quality attribute, various rules can be used. For additive static attributes, this rule is addition of the quality attributes of all component instances in the composition under consideration. This addition rule is trivial, but obtaining the estimates of quality attributes of the constituent component instances as well as determining what these instances are is not an easy task. Non-additive static quality attributes need compositional rules different from addition.

Using this framework, it is possible to predict the values of the quality attributes for different component compositions By varying component quality attributes one can estimate the quality attributes of the composition without actual building (i.e. without

compiling, linking, etc.). Note that values of static quality attributes of the components can also be budgeted, e.g. when the component is partly implemented.

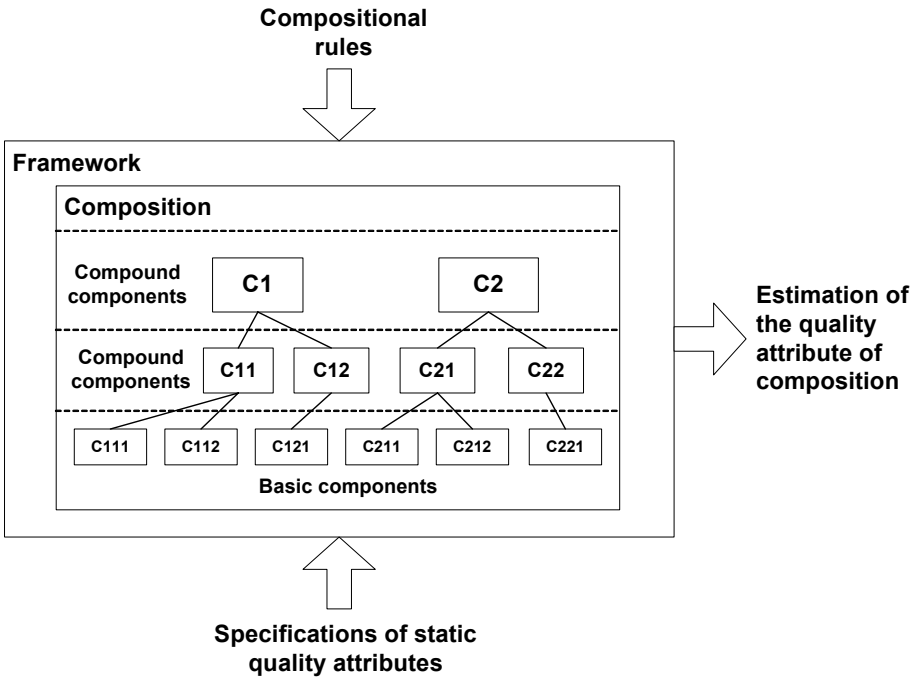


Figure 4.2: Framework for quality attribute estimation

The most important requirements to our method are the following:

- *Each component should include a specification of its static quality attributes.* This specification is essential for the estimation process.
- *The estimation mechanism should be compositional***.** This means that the static quality attributes of the component composition are expressed in terms of the static quality attributes of the constituents, requiring no additional information.
- *The estimation accuracy should be possible to trade against the estimation effort***.** The accuracy of the estimation depends on the amount of the effort invested.

4.3 Sources of variability in component-based software

This section introduces important elements of our model. The model specifies dependencies between static quality attributes of a composition and its constituent components using an analytic expression. These analytical expressions have to account for two basic factors that influence a composition quality attribute: (1) *component diversity* and (2) *component binding*. We discuss these two factors in the subsequent subsections.

4.3.1 Component diversity

The diversity of a component relates to its ability to be tuned for a particular context [OLK00], and is usually defined in terms of *diversity parameters*. Note that depending on the particular component model, the values for the diversity parameters may be selected at different times: component description interpretation, compilation, linking, and run-time.

Our hypothetical component model supports component containment hierarchy, that is, it allows a component to instantiate other components. In this case, the diversity parameters of the inner component instances are usually calculated as a function of the

diversity parameters of the outer component instance that is one level higher in the hierarchy (see Figure 4.3).

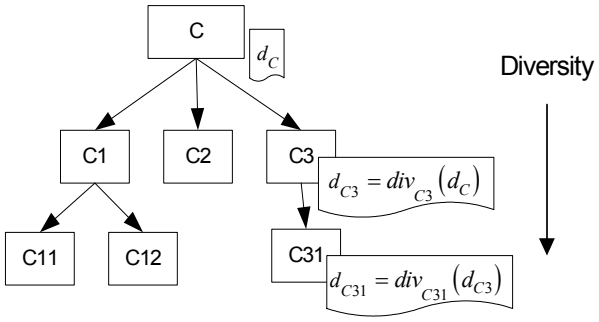


Figure 4.3: Diversity parameters passing

A higher-level component C_i applies the function $div_{C_{ij}}(d_{C_i})$ to its diversity parameters to obtain the diversity parameters $d_{C_{ij}}$ of a lower level component C_{ij}. For example, Figure 4.3 demonstrates that the values of the diversity parameters of component *C3* are obtained from the values of the diversity parameters of the entire composition by applying formula $d_{C3} = div_{C3}(d_C)$. Sequentially applying these functions, it is possible to construct a function that expresses the diversity parameters of any component in terms of the diversity parameters of the entire composition. For example, in Figure 4.3 the diversity parameters of component *C3* are expressed using the following superposition:

$$d_{C31} = div_{C31} \circ div_{C3}(d_C). \tag{4.2}$$

Note that for component instances that are instantiated from the same component type the functions $div_{C_{ij}}(d_{C_i})$ are the same. The actual values of the arguments of this functions may however differ, as these component instances may be created by different parent components.

4.3.2 Component binding

Binding defines the connections between components, and, also, which components are included into a composition. For example, in Koala [OLK00] a component may need to be excluded from a composition if its interfaces are not required by other components[1]. The contribution of a component to a static quality attribute may therefore depend on binding.

To indicate if this contribution needs accounting for, we introduce a function $in(c)$ that takes the identifier of the component instance as argument and valuates to zero or one, depending on whether the component instance needs including or not, respectively (see Figure 4.4).

Note that we allow low level components to be included even if their parent components are not (e.g., components *C1* and *C31* in Figure 4.4). We introduce this feature to support the Koala component model [OLK00] (see also Section 3.2.2). A parent component may just wrap an inner one, while not consuming any additional resources such

[1] For instance, the Koala component model [OLK00] performs a so-called *reachability analysis* that determines if a component needs to be included in a build, based on monitoring of component's *provides* interfaces.

as memory. This feature makes it difficult to judge about the presence of a component instance based on the knowledge about presence of the parent component instance.

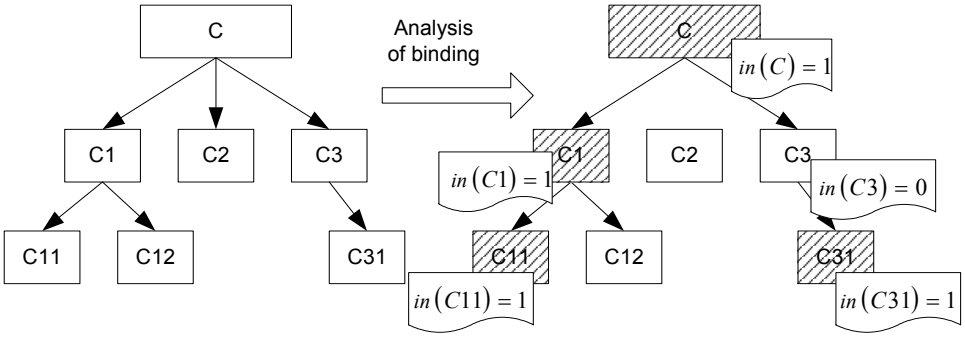


Figure 4.4: Binding analysis

4.4 General estimation formula

The quality attributes of compound components can be expressed in terms of the quality attributes of the inner ones:

$$Q(C,D) = Q_C(D) \cdot in(C) + \sum_{i \in sub(C)} Q\left(i, div_{C,i}(D)\right). \tag{4.3}$$

This formula uses the following notations:

- $Q(C,D)$ is a function that expresses the value of the quality attribute Q of the component instance C for a given set D of diversity parameters,
- $Q_C(D)$ denotes the contribution of component instance C to quality attribute Q given a set of diversity parameters D, without accounting for its sub-components,
- $sub(C)$ is the set of the sub-component instances of component instance C,
- $div_{C,i}(D)$ is a function that equals the values of the diversity parameters of sub-component instance i of component instance C, given the diversity parameters D of component instance C,
- $in(C)$ is a function that returns one or zero depending on whether the contribution of component instance C itself to the quality attribute needs accounting for.

Formula (4.3) is evaluated recursively, starting from lowest level components up to the entire composition. The values of function $in(C)$ should be provided by the tools supporting the component model, whereas the values of $Q_c(D)$ may be approximated on the basis of quality attribute measurements on the existing component or budgeted. Notice that the Formula (4.3) is valid for all additive static quality attributes, assuming that the underlying component model supports *containment hierarchy*, as described in Section 4.2.1, *diversity propagation*, as stated in Section 4.3.1, and *component binding*, as explained in Section 4.3.2.

4.5 Specification of additive static quality attributes

Each component is annotated with one or more formulas, each specifying one static quality attribute. This formula estimates the value of the static quality attribute based on the values of diversity parameters, constant numbers, etc.

We consider three approaches for specifying static quality attributes of a component. The *first* approach is the introduction of an auxiliary interface that specifies the static quality attribute of a component. We call such an auxiliary interface an *IResource* interface. For instance, such an auxiliary interface can provide information about the memory space occupied by the code and static data of a component. The attributes[2] of this interface correspond to particular type of memory. For each type of quality attribute, a formula that estimates the occupied space is provided. Section 4.9.5 demonstrates the application of this first approach.

The *second* approach is the annotation of a component with a separate description of its static quality attributes. This means that the original component descriptions (e.g., code) are kept intact and a separate, auxiliary file (e.g., in XML) is added. This file consists of the same sort of information as in the case of the *IResource* interface. Section 4.8 demonstrates the application of this second approach.

The *third* approach is the modification of a component description language and existing tool chain to allow describing static quality attributes of a component in the component definition language. This can be considered as the best approach, since we obtain a new component definition language, which allows one to specify static component quality attributes in an explicit and comprehensible way. Unfortunately, this approach requires the extreme amount of effort and may introduce incompatibilities with the existing component descriptions.

Table 4.1 summarizes the comparison of the three approaches.

Table 4.1: Possible options for static quality attribute specification

Option	Advantages	Disadvantages
Addition of an auxiliary reflection interface to a component (IResource)	• Possibility to use the existing tools (of some component models) for performing automatic calculations of static quality attribute of a composition • Relevant descriptions of component static quality attributes are known from its description • No binary code is added to a component after compilation	• The resource demands of the component are not vividly specified, since the existing constructions of component description language are used • Component description file is affected • Approach is component-model-specific
Addition of a special file to a component definition file (e.g. XML-based file).	• Does not require changes in the syntax of component description language, so that the existing tool chain can be kept as is • The syntax of the additional file can be easily changed	• Necessity to implement additional tools for XML-files generation and interpretation • The functional and non-functional information is kept separately, and, thus, the problem of description consistency arises
Modification of the component definition language and corresponding tool chain	• Expressiveness of the new descriptions. It is easy to distinguish between non-functional aspects and functional aspects, as they are expressed with a different syntax • The most flexible way	• Much effort is necessary to extend the existing tool chain • Approach is component-model-specific

During our research, we considered the first and the second approach only. For the first approach, we completed a successful validation experiment (see Section 4.9). The second approach was elaborated but not checked by practical cases. The third approach was not thoroughly investigated due to enormous effort required and limited project timeframe.

[2] An *attribute* is an interface member in a form of variable.

4.6 Evaluation of additive static quality attributes

This section explains two approaches to static quality attribute estimation. It describes which components have to be taken into account for the estimation and the order these components should be considered.

4.6.1 Exhaustive and selective evaluation

Measurement and annotation of all components with the specifications of their static quality attributes and subsequent calculation of the resulting quality attributes can require significant effort for complex systems with hundreds of components and thousands of diversity parameters. This approach can be efficient only for comparatively low-scale compositions (up to 20 components with few diversity parameters). Thus, considering that the annotation and measurement effort should be reasonable, it would be wise to handle not all the components and not all the parameters, but only the relevant ones. This presumes more thorough analysis of contribution of each component into the static quality attribute of a composition and annotation of only those components that have tangible influence on the static quality attribute of interest.

The same holds for diversity parameters. Considering all diversity parameters of each component may result in the waste of time and human resources for two reasons:

- Only some of the diversity parameters may significantly influence the static quality attribute of a component. Accounting for insignificant parameters requires extra effort that does not improve prediction accuracy.
- A component can be always used in the context narrower than its diversity parameters allow. Analyzing the values of diversity parameters that never occur in practice is also a waste of resources. Components are likely to have narrower diversity, if they are always used as sub-components in the same component.

Considering all above, two approaches can be suggested: (1) *exhaustive* approach and (2) *selective* approach.

In the *exhaustive* approach, all the diversity parameters and all components are taken into account (see Figure 4.5: and Figure 4.6:).

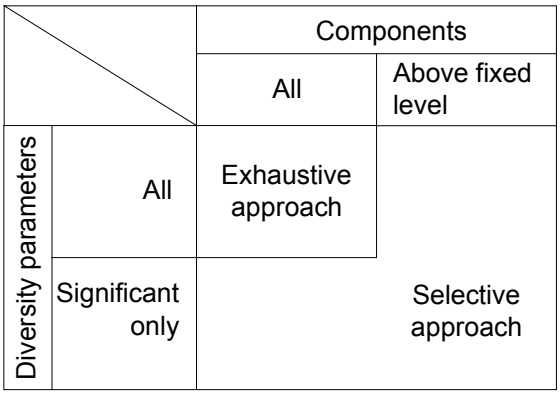


Figure 4.5: Treating components and diversity parameters in the exhaustive and selective approaches

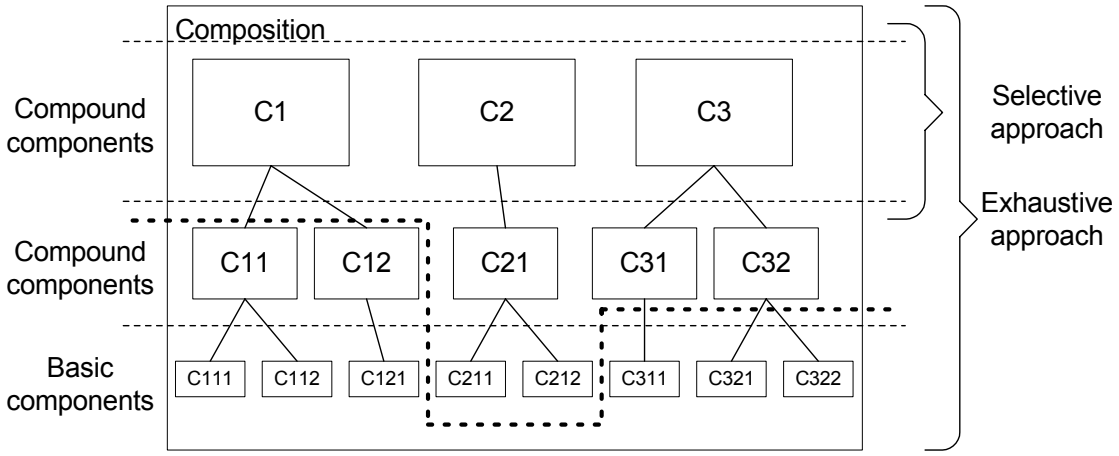


Figure 4.6: Treating component hierarchy in the exhaustive and selective approaches

The containment hierarchy is traversed in a bottom-up way, starting from the basic components up to ones at the defined level of the hierarchy. A formula is constructed for each component, until a formula for the entire composition is determined.

The *selective* approach deals only with the components and diversity parameters that are sufficient for achieving a desired level of precision with a fixed limited effort. The approach starts at some fixed level of the composition hierarchy (in Figure 4.6, e.g. components *C1*, *C2*, and *C3*). If some components of lower level are subjects for reuse, the selected level may be lowered. The selected components are analyzed such as they are basic components. Afterwards, the approach works in exactly the same way as the exhaustive one. Note that the level of hierarchy where the analysis is performed may not always be purely horizontal, but also multi-graded. The reason is that particular inner subcomponents can be units of reuse (e.g., the annotation can be required for components *C1*, *C211*, *C212*, *C31*, and *C32*, see Figure 4.6).

The selected diversity parameters influence the accuracy of estimation formulas for the quality attributes of components. The highest accuracy can be reached if all diversity parameters are accounted for. However, particular diversity parameters may not significantly influence the quality attribute of the component. These can be often omitted at the cost of slight accuracy degradation, thus saving effort for constructing the formula.

The formula for a top-level component builds on the formulas of its constituents. The formulas can be built either in (1) an *empiric stepwise* way (see Section 4.7.1) or by (2) using *factorial analysis and regression techniques* (see Section 4.7.2). In both cases, it is necessary to be able to measure or calculate the actual values of static quality attributes of a component for different values of its diversity parameters. The empiric stepwise construction of a formula is performed by sequentially changing the diversity parameters, measuring the actual values of the static quality attribute, and modifying the formula such that it fits also the new measurements.

Note that both *exhaustive* and *selective* approaches support budgeting, i.e. the expected quality attributes of non-existing components can be incorporated. Moreover, these budgets can also be combined with estimation formulas constructed for the existing components.

4.6.2 Selection of a particular approach

There are a number of important factors to be accounted for when choosing a particular approach. The general guideline is that the chosen approach should be a compromise between the amount of the effort and the accuracy of the results.

A) Estimation accuracy

The exhaustive approach presumes that all components in the hierarchy are annotated with the formulas specifying their static quality attributes and all their diversity parameters are considered. If all formulas are sufficiently accurate, the exhaustive approach ensures (in the general case) that the required level of accuracy for the entire composition is reached.

The selective approach, however, may not ensure required level of accuracy as some of relevant diversity parameters may be missed. Hence, this approach may require the analysis of more diversity parameters if the achieved level of accuracy is not sufficient.

B) Time budget

In the general case, it is not clear which of the approaches– selective or exhaustive– requires less time, given that the prediction accuracy is fixed. The time needed to apply either approach is primarily determined by the amount of measurements needed to construct estimation formulas. This amount, in turn, depends on the number of diversity parameters that have to be varied to collect the sufficient number of measurements. As the diversity parameters of lower-level components may depend on the diversity parameters of higher-level components (see Section 4.3.1) in a non-trivial way, the total number of diversity parameters that need varying may be both larger and smaller for the exhaustive approach than for the selective approach. The same relation persists for the times needed to apply these approaches.

However, when the components of higher levels of the hierarchy tend to use the subcomponents only with a limited subset of possible values of diversity parameters, the selective approach is likely to be advantageous in terms of time needed for the measurement.

C) Reuse of components

Component reuse may be an important factor for choosing between the exhaustive and selective approaches. If some components of the analyzed composition are considered for reuse in other products, their annotation might be worthwhile, even if this is not necessary for the current estimation.

4.7 Construction of estimation formulas

Two approaches to the construction of estimation formulas are possible:

- An empiric approach, which estimates the coefficients of a formula by manually tweaking them;
- An approach, which estimates the coefficients of a formula by using statistical regression techniques;
- An analytical calculation.

The first approach is conceptually simpler than the second one, but it may require more effort spent in guessing the proper values of formula coefficients. On the other hand, the second approach requires the knowledge of statistical techniques, such as (multiple) linear regression. These two approaches are detailed in the subsequent subsections. The third approach can be applied only for component models that use strict composition rules and are based on restricted programming language. The estimation formulas could be

constructed on the basis of results of static analysis of component compositions (including their implementation). As it is unlikely to meet such a component model in an industrial setting, where we target our method at, we do not elaborate this third approach any further.

4.7.1 Empirical formula construction

This section describes an algorithm for estimating static quality attributes of a composition. The estimation process consists of a number of actions, which are performed according to the flowchart shown in Figure 4.7.

The following actions need to be taken:

1. Define the top-level component (entire composition).
2. Define the necessary estimation accuracy.
3. Choose one evaluation approach (exhaustive or selective) and define an appropriate level of component hierarchy (if selective).
4. Identify the entities influencing the estimation accuracy and effort. The choice of the entities is based on the information provided by component producers and judgment of the architect, or by applying statistical techniques such as factor analysis (see Section 4.7.2). The relevant entities are to be found amongst the following:
   - Relevant components. The contribution of these components into the static quality attribute cannot be neglected due to their size, accuracy requirements, etc;
   - Relevant diversity parameters. The influence of these parameters on the estimated quality attribute cannot be neglected with respect to the defined accuracy;
   - Relevant values of diversity parameters.
5. Manipulate the relevant diversity parameters of component of interest[3].

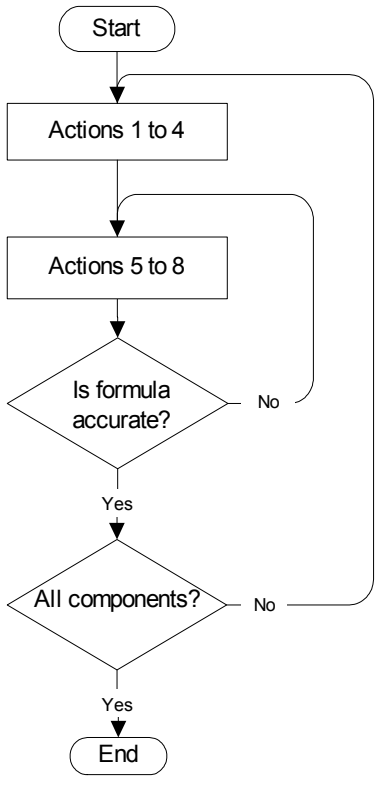


Figure 4.7: Algorithm for constructing a formula for a single component

[3] Initially, the chosen component is the top-level one.

6. Build/modify the estimation formula and extend the description of the component with the specification of static quality attributes (e.g., by means of a reflection interface).
7. Obtain the actual values of the quality attributes by measuring the component.
8. Verify the formula for the component: check if the accuracy of the formula is in accordance with the accuracy specified by the architects. If the formula is sufficiently accurate then stop. Otherwise, refine the formula in a step-wise manner by repeating steps 6 to 8.
9. If necessary, repeat Actions 5 to 9 for all relevant components.

After completing the actions above, the architect can apply the constructed formulas to obtain estimates of the quality attribute of the composition. The accuracy of the obtained estimates is discussed below.

A) Estimation accuracy

As explained in Section 4.1, we consider only additive static quality attributes of a component. Consequently, an estimate at the composition level is the sum of estimates at the component level. The corresponding estimation errors add up in the same way.

Let us denote the accurate value of a variable to be estimated at the compositional level with Q_c, and the corresponding values at the component level with Q_i; the estimations are denoted as Q_c^* and Q_i^*, respectively, where $Q_c^* = \sum_i Q_i^*$ and $Q_c = \sum_i Q_i$. Consider also that the estimations at the component level are provided with certain relative error δ_i^*:

$$Q_i - \delta_i^* \cdot Q_i \le Q_i^* \le Q_i + \delta_i^* \cdot Q_i, \tag{4.4}$$

where Q_i^*, Q_i, and δ_i^* are greater then zero. The absolute error of the estimation Q_i^* is the following:

$$\Delta_i = Q_i^* - Q_i. \tag{4.5}$$

Combining Formulas (4.4) and (4.5), it is easy to bound the value of Δ_i:

$$|\Delta_i| \le Q_i \cdot \delta_i^*. \tag{4.6}$$

The relative error δ_c of the estimation for the component composition can be calculated by the following formula:

$$\delta_c = \frac{\left|\sum_i \Delta_i\right|}{\sum_i Q_i} \le \frac{\sum_i |\Delta_i|}{\sum_i Q_i}. \tag{4.7}$$

Then applying Formula (4.6) to Formula (4.7), δ can further be bounded:

$$\delta_c \le \frac{\sum_i |\Delta_i|}{\sum_i Q_i} \le \frac{\sum_i \delta_i^* \cdot Q_i}{\sum_i Q_i} = \sum_i \delta_i^* \cdot \frac{Q_i}{\sum_j Q_j} \tag{4.8}$$

Formula (4.8) bounds the worst-case relative error for the estimation at the composition level, given the estimations and their relative errors at the component level. Note that the relative error of each component-level-estimation is weighed by the accurate value at the component level divided by the accurate value at the compositional level. This makes it

difficult to predict the accuracy at the composition level, knowing only the accuracy of the estimation for a single component (without knowing the values at the compositional level).

However, it is possible to find the upper bound of the relative error for the composition by considering that $\forall i \left(\delta_i^* \le \delta_{\max}^* \right)$ in Formula (4.8). In this case δ_c satisfies the following equation:

$$\delta_c \le \sum_i \delta_i^* \cdot \frac{Q_i}{\sum_j Q_j} \le \frac{\delta_{\max}^* \cdot \sum_i Q_i}{\sum_j Q_j} = \delta_{\max}^* . \qquad (4.9)$$

Formula (4.9) relates the accuracy at the component level and at the entire composition level.

4.7.2 Use of statistical techniques to construct estimation formulas

Empiric construction of a formula that estimates static quality attributes of a component may be very time consuming. It is necessary to identify the (diversity) parameters that have a major effect on the static quality attributes of interest as well as to express the effects of these parameters in formulas. If the preliminary knowledge about these effects is absent, it may be cumbersome to guess these effects by using the trial-and-error strategy.

We suggest performing a two-stage approach to constructing an estimation formula (see Figure 4.8:).

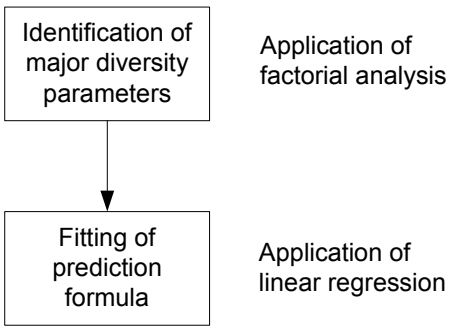


Figure 4.8: An approach to construction of the prediction formula

First, it is necessary to identify the diversity parameters that have a major effect on the static quality attributes of interest. To this end, a number of dedicated measurements have to be performed for discerning the effects of each diversity parameter on the static quality attributes. These measurements are organized such that the information about the effects can be obtained in an efficient way. This measurement scheme is referred to as the full or fractional factorial designs in the statistical literature [JAI91].

Afterwards, more elaborated measurements have to be performed to detail the major effects and to build the estimation formula. Multiple linear regression can be used here [JAI91][WEI95][MON01].

The subsequent sections detail each stage of the approach. As these subsections rely on the notions typically met in the statistics literature, we will use the notations that can be encountered in a typical book on statistics. Particularly, the variable y will denote a dependent variable, which is the same as a quality attribute Q from Section 4.4. Diversity

parameters D corresponds with independent variables, denoted as x in the subsequent sections. Please notice that we did not validate this two-stage approach in practice, as in our case studies (see Section 4.9) the empirical approach performed well.

A) Identification of significant diversity parameters

We consider two types of diversity parameters: (1) *real* and (2) *categorical*. The former allow expressing (possibly continuous) quantities, whereas the latter allow discerning modes, types, etc. For example, consider a component implementing a real-time kernel. This component has the following real diversity parameters: the maximum number "*MaxTasks*" of tasks that can be created and the maximum number "*MaxSemaphores*" of semaphores. It has also one categorical diversity parameter "*DebuggingMode*": the debugging mode. The value of "*DebuggingMode*" equals "*On*", if the debugging is turned on. Otherwise, it equals "*Off*".

We advocate the use of the factorial analysis [JAI91] to identify the significant diversity parameters. All diversity parameters are treated as *factors*, i.e., variables that may have only a few values. These values are called *factor levels*. A factor F is said to be at level f if its value equals f.

For a categorical diversity parameter, a particular value of this diversity parameter equals some level of the corresponding factor. For instance, for the "*DebuggingMode*" diversity parameter, the corresponding factor has two levels: "*On*" and "*Off*".

For each real diversity parameter, an auxiliary factor has to be introduced. This factor has only a few levels, each level being equal to a particular value of the diversity parameter. Typically, only two levels are introduced: one level for a fixed small value of the diversity parameter and another one for a fixed large value of this parameter. Reducing a real diversity parameter to a two-level factor substantially decreases the number of measurements needed to check if this parameter has an impact on the static quality attribute of interest. For example, consider again the real-time kernel component, which will be used in a context where not more than 30 tasks are created. For the "*MaxTasks*" diversity parameter, it is reasonable to introduce the "*MaxTasksFactor*" factor with two levels: "*FewTasks*" and "*ManyTasks*". The "*FewTasks*" level corresponds to the value of "*MaxTasks*" that equals two, whereas the "*ManyTasks*" corresponds to the value of 25. Please notice that this choice is arbitrary; the only rationale for choosing these particular values is the hope that they will allow to discover the influence of "*MaxTasks*" on the static quality attribute of interest.

The variation of a static quality attribute may be accounted to *effects* and *unexplained part*. An effect describes the contribution of a certain level of a factor or the contribution of the interaction of two or more factors. In the latter case, these factors are said to *interact*, that is, the effect of one factor may depend on the level(s) of other factor(s). The unexplained part represents the error term.

It is possible to conduct a series of measurements that allow determining the influence of factors and their interactions on the static quality attribute of interest. This influence can be analyzed by applying factor analysis [JAI91], [MCD85], and [MON01]. Depending on time and human resources available for measurements, different schemes of experiments for collecting measurements can be used: 2^k-factorial designs, 2^{k-p}-fractional factorial design, two-factor full factorial design, general full factorial designs with k factors, and etc [JAI91], [MCD85], [MON01].

Let us demonstrate the use of a general factorial design with 2 factors: "*MaxTasksFactor*" and "*DebuggingMode*". For the sake of simplicity, the third factor,

"*MaxSemaphores",* is considered to be at a fixed level. Table 4.2 summarized the measurements of the sizes of static data for the real-time kernel component. These measurements are collected for different values of the diversity parameters and, consequently, for different levels of the chosen factors.

Table 4.2: Measurements of sizes (in bytes) of static data collected for the real-time kernel component for different values of its diversity parameters *MaxTasks* and *DebuggingMode*

		MaxTasksFactor	
		FewTasks (α_1)	*ManyTasks* (α_2)
DebuggingMode	*On* (β_1)	10	62
	Off (β_2)	7	53

Let us denote the effects of "*MaxTasksFactor*" as α_i and the effects of "*DebuggingMode*" as β_j. The interactions between the levels of those two factors are denoted as γ_{ij}, and the measurements of the sizes are y_{ij}. The relation between these variables can be described by the following formula:

$$y_{ij} = \mu + \alpha_i + \beta_j + \gamma_{ij}. \tag{4.10}$$

In Formula (4.10), μ denotes the average (over the measurements from Table 4.2) size of static data. Obviously, Formula (4.10) contains more unknown variables than measurements. Therefore, additional equations over these variables have to be added to obtain the unique solution. These additional equations enforce orthogonality between the main effects of "*MaxTasksFactor*" and "*DebuggingMode*" and their interactions:

$$\begin{aligned} &\sum_{i=1}^{2}\alpha_i = 0; \quad \sum_{j=1}^{2}\beta_j = 0; \\ &\sum_{i=1}^{2}\gamma_{ij} = 0 \ \forall j; \quad \sum_{j=1}^{2}\gamma_{ij} = 0 \ \forall i. \end{aligned} \tag{4.11}$$

Solving Formulas (4.10) and (4.11) over all unknown variables and substituting the measurement from Table 4.2 provides the following:

$$\begin{aligned} &\mu = \frac{1}{4}\sum_{i=1}^{2}\sum_{j=1}^{2} y_{ij}; \\ &\alpha_i = \frac{1}{2}\sum_{j=1}^{2} y_{ij} - \mu; \quad \beta_j = \frac{1}{2}\sum_{i=1}^{2} y_{ij} - \mu; \\ &\gamma_{ij} = y_{ij} - \alpha_i - \beta_j - \mu. \end{aligned} \tag{4.12}$$

The values of all effects and interactions are obtained by substituting the measurements from Table 4.2 to Formula (4.12):

$$\begin{aligned} &\mu = 33; \\ &\alpha_1 = -24,5; \quad \alpha_2 = 24,5; \quad \beta_1 = 3; \quad \beta_2 = -3; \\ &\gamma_{11} = -1,5; \quad \gamma_{12} = 1,5; \quad \gamma_{21} = 1,5; \quad \gamma_{22} = -1,5. \end{aligned} \tag{4.13}$$

In addition to the values of those effects and interactions, it is also necessary to estimate how much they impact the size of static data. This impact is estimated by observing the amounts of variation of the size of static data due to each effect and interaction.

The entire variation is represented by the total sums of squares (*SST*):

$$SST = \sum_{i=1}^{2}\sum_{j=1}^{2}\left(y_{ij} - \mu\right)^2 \tag{4.14}$$

The *SST* accounts for all deviations of the measurements from a particular mean value. Moreover, it favors positive and negative deviations equally (because of squaring the $\left(y_{ij} - \mu\right)$ term). Finally, it can be partitioned to sum of squares that relate to different effects and interactions.

This partitioning is performed as follows. First, we square both sides of Formula (4.10) and sum over all *i* and *j*. All cross-product terms cancel out because of orthogonality conditions (4.11), and we obtain the following formula:

$$\sum_{i=1}^{2}\sum_{j=1}^{2} y_{ij}^2 = 4\mu^2 + 2\sum_{i=1}^{2}\alpha_i^2 + 2\sum_{j=1}^{2}\beta_j^2 + \sum_{i=1}^{2}\sum_{j=1}^{2}\gamma_{ij}^2. \tag{4.15}$$

Formula (4.15) can be rewritten as follows:

$$\begin{aligned} \sum_{i=1}^{2}\sum_{j=1}^{2} y_{ij}^2 - 4\mu^2 = \sum_{i=1}^{2}\sum_{j=1}^{2}\left(y_{ij} - \mu\right)^2 &= 2\sum_{i=1}^{2}\alpha_i^2 + 2\sum_{j=1}^{2}\beta_j^2 + \sum_{i=1}^{2}\sum_{j=1}^{2}\gamma_{ij}^2, \\ SST = \quad & SS\alpha \;+\; SS\beta \;+\; SS\gamma. \end{aligned} \tag{4.16}$$

Formula (4.16) partitions the entire variation of the size of static data into sum of squares $SS\alpha$ and $SS\beta$ related to the main effects α_i and β_j, respectively, and sum of squares $SS\gamma$ related to the interactions γ_{ij} between the two factors.

It is now easy to assess the fraction of variation explained by each factor and interactions. This can be done by observing the following fractions: $SS\alpha/SST$, $SS\beta/SST$, and $SS\gamma/SST$. For the measurements from Table 4.2, we obtain the following:

$$\begin{aligned} &SST = 2446;\ SS\alpha = 2401;\quad SS\beta = 36;\quad SS\gamma = 9; \\ &\frac{SS\alpha}{SST} \approx 98,2\%;\ \frac{SS\beta}{SST} \approx 1,5\%;\ \frac{SS\gamma}{SST} \approx 0,3\%. \end{aligned} \tag{4.17}$$

Formula (4.17) shows that about 98.2% of the variation is explained by the factor "*MaxTasksFactor*" (with effects α_i). The second factor "*DebuggingMode*" and its interaction with "*MaxTasksFactor*" explains only 1.8% of the variation. This means the variation of the size of static data of the real-time kernel component is primarily due to the variation of the diversity parameter "*MaxTasks*". This diversity parameter is turned out to be the only relevant parameter according to the analysis described above.

Other schemes of experiments for collecting and analyzing measurements (2^k-factorial designs, 2^{k-p}-fractional factorial design, two-factor full factorial design, general full factorial designs with *k* factors, and etc) are implemented similarly to the example that is described above. For more details the reader is referred to the literature [JAI91], [MCD85], [MON01].

B) Construction of an estimation formula for a component by linear regression

Once the main factors (and diversity parameters) are identified, the estimation formula can be constructed by means of regression. We advocate the use of the multiple linear regression [JAI91], [WEI95], [MON01] for the following reasons:

- We consider additive static quality attributes, which complies with the additivity of multiple linear regression models
- Multiple linear regression is a well known statistical technique, and many tools exist for automating the construction and analysis of linear regression models
- Multiple linear regression provides not only prediction formulas, but also powerful means for checking the adequacy and accuracy of these models (e.g., in terms of prediction and confidence intervals, different kinds of significance tests, etc.).

Let us demonstrate the use of linear regression for constructing estimation formulas both in abstract form and by means of the concrete example introduced in Section 4.7.2. A multiple linear regression model describes the following relation between the dependent variable[4] y and independent variables x_i:

$$y = \beta_0 + \sum_{j=1}^{k} \beta_j \cdot x_j + \varepsilon . \qquad (4.18)$$

In this formula, y is the dependent variable, a variable that need to be predicted. It corresponds with the static quality attribute of interest. β_j denote *regression coefficients*, and x_j are independent variables that correspond with diversity parameters. The ε random variable describes the prediction error. It is assumed that ε has normal distribution with mean zero and constant standard deviation. Consider the example of the real-time scheduler component. The dependent variable y is the size of static data. The mapping between the independent variables x_j and the diversity parameters is presented in Table 4.3.

Table 4.3: Mapping between the diversity parameters of the real-time kernel component and independent variables of the linear prediction model

Diversity parameter		Independent variable	
Name	Values	Name	Values
x_1	0..25	*MaxTasks*	0..25
x_2	0..10	*MaxSemaphores*	0..25
x_3	1	*DebuggingMode*	*On*
	0		*Off*

In the general case, this mapping will require more than one independent variable per a categorical diversity parameter that may have more than two values (see Section 4.7.2).

Suppose that along with each observed value y_i of the dependent variable y, the corresponding values x_{ij} of independent variables x_j are also noted. The model from formula (4.18) can be then rewritten in the following form:

[4] In the regression literature, the dependent variable is often called *response variable*, and independent variables are named *regressors* or *predictors*.

$$y_i = \hat{\beta}_0 + \sum_{j=1}^{k} \hat{\beta}_j \cdot x_{ij} + \varepsilon_i . \qquad (4.19)$$

In Formula (4.19), $\hat{\beta}_j$ denote the estimates of true *regression coefficients.* The random variables ε_i represent the prediction error for the i-the observation, or residual. They are assumed to be uncorrelated and to have mean zero and the same constant variance.

The regression coefficient estimates $\hat{\beta}_j$ are obtained such that they minimize the sum of squares of errors $\sum_i \varepsilon_i^2$ (so called least-squared error minimization). The computation of regression coefficients involves matrix operations and is usually performed using a statistical software tool.

After fitting the prediction model, it is important to check that the assumptions of linear regression are not violated, and that the obtained regression coefficients are all significant. The former is usually checked by analyzing residuals, whereas the latter involves either hypothesis testing or construction of the confidence intervals for regression coefficients. For more details, the reader is referred to the regression literature, e.g. to [JAI91], [WEI95], [MON01], or [MR03].

An estimation formula for the static quality attribute of interest can be constructed by inputting the (expressions over) diversity parameters of a particular component to the fitted linear prediction model.

Multiple linear regression provides means to estimate the accuracy of predictions made using linear regression models. This means is prediction intervals [JAI91], [WEI95], [MON01]. A prediction interval allows estimating the limits for the value of a prediction for particular values of independent variables. Notice that the use of prediction intervals is only valid if the assumptions of linear regression are not seriously violated. We therefore recommend using prediction intervals for specifying the accuracy of individual estimation formulas obtained by linear regression.

C) Using linear regression for real and categorical diversity parameters

A static quality attribute of a component corresponds with a dependent variable, and real diversity parameters map on independent variables. It is however no so straightforward for categorical diversity parameters.

The literature [JAI91] suggests using *dummy* variables to model categorical diversity parameters. For instance, consider a diversity categorical parameter d with possible values from a set $\langle A, B, C, D \rangle$. To model this diversity parameter, three dummy variables are introduced:

$$x_1 = \begin{cases} 1, & if\ d = B \\ 0, & otherwise \end{cases}$$
$$x_2 = \begin{cases} 1, & if\ d = C \\ 0, & otherwise \end{cases} \qquad (4.20)$$
$$x_3 = \begin{cases} 1, & if\ d = D \\ 0, & otherwise \end{cases}$$

Formula (4.20) describes the encoding of diversity parameter *d* (see Table 4.4).

Table 4.4: An example of the encoding of categorical diversity parameters using *dummy* variables

Value of d	x_1	x_2	x_3
A	0	0	0
B	1	0	0
C	0	1	0
D	0	0	1

In the general case, a categorical diversity parameter with k possible values requires $k-1$ dummy variables be introduced into the regression model. These dummy variables are then considered as independent variables.

However, the use of dummy variables may significantly increase the number of independent variables in the regression model, especially when interactions with other categorical variables have to be considered. It may be even the case that the available observations become insufficient for the reliable fitting of the prediction model. In this case, it is recommended to build a separate regression model for each value of a categorical diversity parameter and then to use the weighted sum of these prediction models to estimate the total value. Though, this solution complicates making inferences for the obtained composite prediction model. Particularly, confidence and prediction intervals become difficult to calculate.

D) Construction of an estimation formula for a component composition

For additive quality attributes, it is enough to add up the values of a static quality attribute for all components to obtain the value of the composite static quality attribute. The same is equally true for estimates of the static quality attribute. Let us consider an entire composition "*Cmp*". By applying Formula (4.3) from Section 4.4 to all components (in a recursive manner), the following can be obtained:

$$Q_{Cmp}(Cmp, D_{Cmp}) = \sum_{i \in reachable(Cmp)} Q_i\left(D_i\right) \tag{4.21}$$

In Formula (4.21), $Q_{Cmp}(Cmp, D_{Cmp})$ denotes the estimate of the static quality attribute of the composition "*Cmp*" with the diversity parameters D_{cmp}, and $Q_i\left(D_i\right)$ is the estimate of the static quality attribute of the component *i* with the diversity parameters D_i. The set *reachable(Cmp)* describes all components included in a build for the composition "*Cmp*" with diversity parameters D_{cmp}.

However, assessing the accuracy of the composite quality attribute estimate is not that straightforward for the reason explained below.

As stated in Section 4.7.2, for a single component, the accuracy of the estimate of a static quality attribute can be indicated by means of a prediction interval. For the entire composition, prediction intervals are also a good means for representing the accuracy of the estimate. It is therefore necessary to calculate a prediction interval for the sum of the estimates of the static quality attribute, calculated by Formula (4.21). (In statistical terms, this task amounts to estimating a prediction interval for the sum of estimates, each calculated by a linear regression model.) Though, adding up the lower and upper limits of prediction intervals of individual estimates does not result in the correct prediction interval for the entire sum. The interval obtained this way will be much wider then the true prediction interval, as the prediction errors partly cancel out for individual components (as these errors are independent and have the normal distribution).

The literature survey showed that the calculation of a prediction interval for the sum of estimates calculated by linear prediction model is not at all simple. Moreover, standard books about statistics such as [WEI95], [MON01], and [MR03] do not cover this subject. We suggest a solution to this problem. This solution involves complex mathematical derivations, which are described in Appendix C in detail.

4.8 Generic specification of static quality attributes

This section describes our proposal for the specification of components and their static quality attributes in the XML language. We did not validate this approach in practice due to limited timeframe.

4.8.1 Rationale for selection of XML

XML stands for Extensible Markup Language [XML1]. XML is a plain text, Unicode-based meta-language for defining markup languages. Markup languages provide mechanisms for describing the document structure by means of markup tags (<> and </>). XML is employed as a technology for structuring, manipulating, transforming, and querying data. During its initial development, in the middle of 1990th, XML was considered as a description language for data formats for the Internet. Currently, the role of XML is more general: it is useful not only for formatting Web documents, but also for describing structured data of any type. The examples of structured data are spreadsheets, configuration files, network protocols, etc. Essential features of XML such as flexibility, extensibility and portability have destined its wide adoption as the 'lingua franca' for information interchange. For the information about XML syntax and other technical details, the reader is referred to [You01], [PKK98], and [XML1].

We considered XML a good candidate language for describing component and composition quality attributes independently of any particular component model. To give a flavor of such description we worked out a simple example (see Sections 4.8.2 and 4.8.3). However, due to lack of time we did not validate this technique by a more elaborated case study.

4.8.2 Specification of components

This section elaborates on the specification of all the elements required in component specification, including static quality attribute specifications. Let us first present a diagram of the essential elements of component specification (see Figure 4.9).

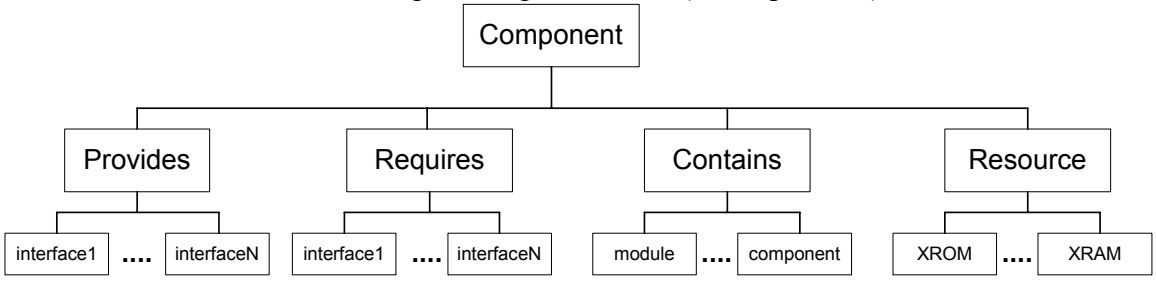


Figure 4.9: Elements of a Koala component specification

Let us consider that each component specifies its *provides* and *requires* interfaces, contains internal code modules and other components, and describes its resource demands (e.g., RAM and ROM memory). Using this knowledge about component description

elements and their hierarchy, one can easily construct an XML specification of the component. An example of such specification is shown in Appendix A.

In addition, we need to specify the static quality attributes of a component: formulas describing the resource consumption of a component (in our case, memory). This specification technique is detailed in the section below.

4.8.3 Specification of formulas

For the specification of formulas for static quality attributes in XML, we used a part of one of the existing XML-based notations for mathematical expression specification: MathML [MXML]. Let us illustrate the main principles and elements of the notation by means of a simple example (Figure 4.10).

The element <apply> is used to apply a function to a collection of arguments. The first nested element denotes the function, and the next two elements denote the arguments. The elements defining simple operations (such as <plus/>, <minus/> or </times>) are described with empty-element tags, as they have no content and attributes. The elements <ci> and <cn> specify identifiers and numbers, respectively.

```
<apply>
   <plus/>
   <cn>48</cn>   <!-- expression: 48+a -->
   <ci>a</ci>
</apply>
```

Figure 4.10: An example of formula specification

We introduced two additional elements: <cif> and <cc>. The former denotes the identifier of an interface whose diversity parameter is a part of a formula. The latter denotes the identifier of a component whose quality attribute is involved into a formula.

Let us consider a more complex example of the stack size specification (see Figure 4.11).

```
<STACK_SIZE>  <!--12 + (ires.MaxTasks - 1)*8 + mgcmx.STACK_SIZE-->
   <apply>
      <plus/>
      <cn>12</cn>
      <apply>
         <times/>
         <apply>
            <minus/>
            <apply>
               <cif>ires</cif>
               <ci>MaxTasks</ci>
            </apply>
            <cn>1</cn>
         </apply>
         <cn>8</cn>
      </apply>
      <apply>
         <cc>mgcmx</cc>
         <ci>STACK_SIZE</ci>
      </apply>
   </apply>
</STACK_SIZE>
```

Figure 4.11: A formula for specifying stack size

This formula depends on the parameter "*MaxTasks*" of the interface "*ires*" and also on the stack size of the nested component "*mgcmx*". As one can easily see, this form of representation does not explicitly show brackets, but, instead determines the order of calculation by means of nesting the elements and proper use of the <apply> element. This customized notation does not require adapting of existing XML parsers: it requires only modification of the applications above this parser. More examples for formula specification can be found in Appendix A.

4.8.4 Specification of estimation formulas constructed by linear regression

Section 4.7.2 describes how linear regression can be applied to construct estimation formulas. Using these regression-based formulas, it is not only possible to calculate an estimate of the static quality attribute of a composition, but also to provide figures about prediction accuracy (by calculating prediction intervals).

This section extends the specification approach described in section 4.8.3 to be able to describe regression-based formulas. A regression based-formula specifies a particular static quality attribute of a single component. This formula is parameterized by a few independent variables, which correspond with either particular diversity parameters or expressions over diversity parameters.

The specification approach described in section 4.8.3 has to be extended to encompass the use of regression-based formulas. It is necessary to specify how the calculations for a regression-based formula should be performed, what the regression coefficients are, what the explanatory power of the model is, etc.

Figure 4.12 demonstrates the use of XML for the description of a regression-based formula.

First, it is necessary to describe the regression coefficients of the fitted linear regression model, which underlies the regression-based formula. The regression coefficients are enumerated in the section between XML tags <coefficients> and </coefficients>. Each coefficient has a name, a value, and the p-value that indicates the significance of the coefficient. This information is described using <coeff> and </coeff> tags. For example Figure 4.12 demonstrates that the value of the coefficient alhpa1 equals 0.123, and the p-value of the corresponding t-statistic [WEI95] is 0.03. This p-value indicates that the coefficient "*alpha0*" is significant at the significance level 0.05 (as $0.03 < 0.05$) [WEI95].

Additionally, the specification needs to be attached with the value of the R^2-coefficient (the coefficient of determination) that indicates how well the underlying prediction model can predict the value of the static quality attribute. The <rsquare> tag is used for the specification of this R^2-coefficient.

Section 4.7.2D) shows that it is necessary to know the values of standard error of the residual and degrees of freedom[5] to be able to calculate the prediction intervals for the static quality attributes of a composition. We provide this information using tags <stderr> and <df>, respectively.

[5] The degrees of freedom of a certain statistic are the number of variables (or observations) that can be independently chosen to keep the statistic equal to a certain value.

```
<RegressionModel>
  <coefficients>
    <coeff> <!--Description of the alpha0 coefficient-->
      <name>alpha0</name>
      <value>0.123</value> <!--The value of the coeff.-->
      <pvalue>0.03</pvalue><!--The p-value -->
    </coeff>
    <coeff> <!--Description of the alpha1 coefficient-->
      <name>alpha1</name>
      <value>0.13</value>
      <pvalue>0.001</pvalue>
    </coeff>
  </coefficients>
  <stderr>0.045</stderr>  <!--The standard error of the residual-->
  <df>30</df>     <!--The degrees of freedoms-->
  <rsquare>0.87</rsquare> <!--The coefficient of determination-->
  <!-- The regression model P1(x1) = alpha0 + alpha1*x1 -->
  <declare type="fn" nargs="1">
    <ci>P1</ci>
    <lambda>
      <bvar>
        <ci>x1</ci>
      </bvar>
      <apply>
        <plus/>
        <ci>alpha0</ci>
        <apply>
          <times/>
          <ci>alpha1</ci>
          <ci>x1</ci>
        </apply>
      </apply>
    </lambda>
  </declare>
</RegressionModel>
```

Figure 4.12: An example of XML description of a regression-based estimation formula

Finally, it is necessary to specify how to calculate the value of the static quality attribute using the values of independent variables calculated on the basis of the values of diversity parameters. To this end, it is possible to use the MathML extension [MXML] of XML. For instance, Figure 4.12 describes a new function *P1*, which is a λ-expression over one independent variable *x1*. The value of this argument is calculated using the values of the diversity parameters. For instance, the specification of stack demand from Figure 4.11 of section 4.8.3 can be rewritten as follows (see Figure 17), considering that the *P1* formula describes the stack size for the component of interest.

```
<STACK_SIZE>
  <apply><plus/>
    <apply><ci>P1</ci>
      <apply>
        <cif>ires</cif>
        <ci>MaxTasks</ci>
      <apply>
    </apply>
    <apply>
      <cif>mgcmx</cif>
      <ci>STACK_SIZE</ci>
    </apply>
  </apply>
</STACK_SIZE>
```

Figure 4.13: Example of the use of the specification of the linear regression model

STACK_SIZE is calculated by applying the regression-based formula *P1* to the actual parameter, being the value of the diversity parameter "*ires.MaxTasks*", and by adding the result to the value of the static quality attribute of the inner component "*mgcmx*".

4.8.5 Summary

Concluding this section, we would like to focus on the essential issues once more. The XML language is considered as an ideal candidate for the specification of static quality attributes. This type of specification suits any component model and can be easily ported to another platform. It allows representing the basic component elements as well as resource consumption formulas in a clear and concise form, yet not requiring any changes of the XML syntax and XML parsers. There are only two points where some effort from the component designers is needed: a) the designers have to develop or customize the application responsible for resource consumption estimation, and b) the designers have to ensure the consistency between the functional description of the components (code) and the specifications of the static quality attributes (XML documents).

4.9 Example of method application to industrial software

This section details our experiences in applying one of the suggested approaches to the Koala component model [OLK00]. A brief introduction to Koala is also given in Section 3.2. We predicted the static memory demand of a Koala component composition in terms of required code and data spaces. We used the *IResource*-interface-based approach (see Section 4.5) to the specification of static quality attributes of the components.

4.9.1 Koala component example

To demonstrate the approach, we used a simplified version of the Infrastructure sub-system from a TV software stack. This simplified version is implemented within the "*CMgMiniInfra*" component. For the "*CMgMiniInfra*" component and all its constituents, the specification was enriched with the sizes of memory occupied by each component.

Figure 4.14 depicts the internals of the "*CMgMiniInfra*" component that consists of five subcomponents –"*CMgInit*", "*CMgXaAbstraction*", "*CMgStandardLibrary*", "*CIsCmx*", and "*CMgInterruptServer*"– and a number of modules[6]. In turn, the "*CIsCmx*" component consists of other two components: "*CMgCmx*" and "*CAdocCmx*" (not shown in the picture in order not to overwhelm it with too many details). The total number of components considered is thus 7. Depending on the target hardware platform, either "*CMgCmx*" or "*CAdocCmx*" component is included in a build. The target platform is specified via the "*plf:Platform*" diversity interface, the diversity parameters of which are set by the Development Environment.

The "*CMgMiniInfra*" propagates the "*plf:Platform*" diversity interface to all the constituent components, so that all subcomponents configure themselves for the necessary target platform. The other interfaces are normal provides and requires interfaces.

[6] In short, a module implements a part of the functionality of a Koala component. Modules are written in the C language.

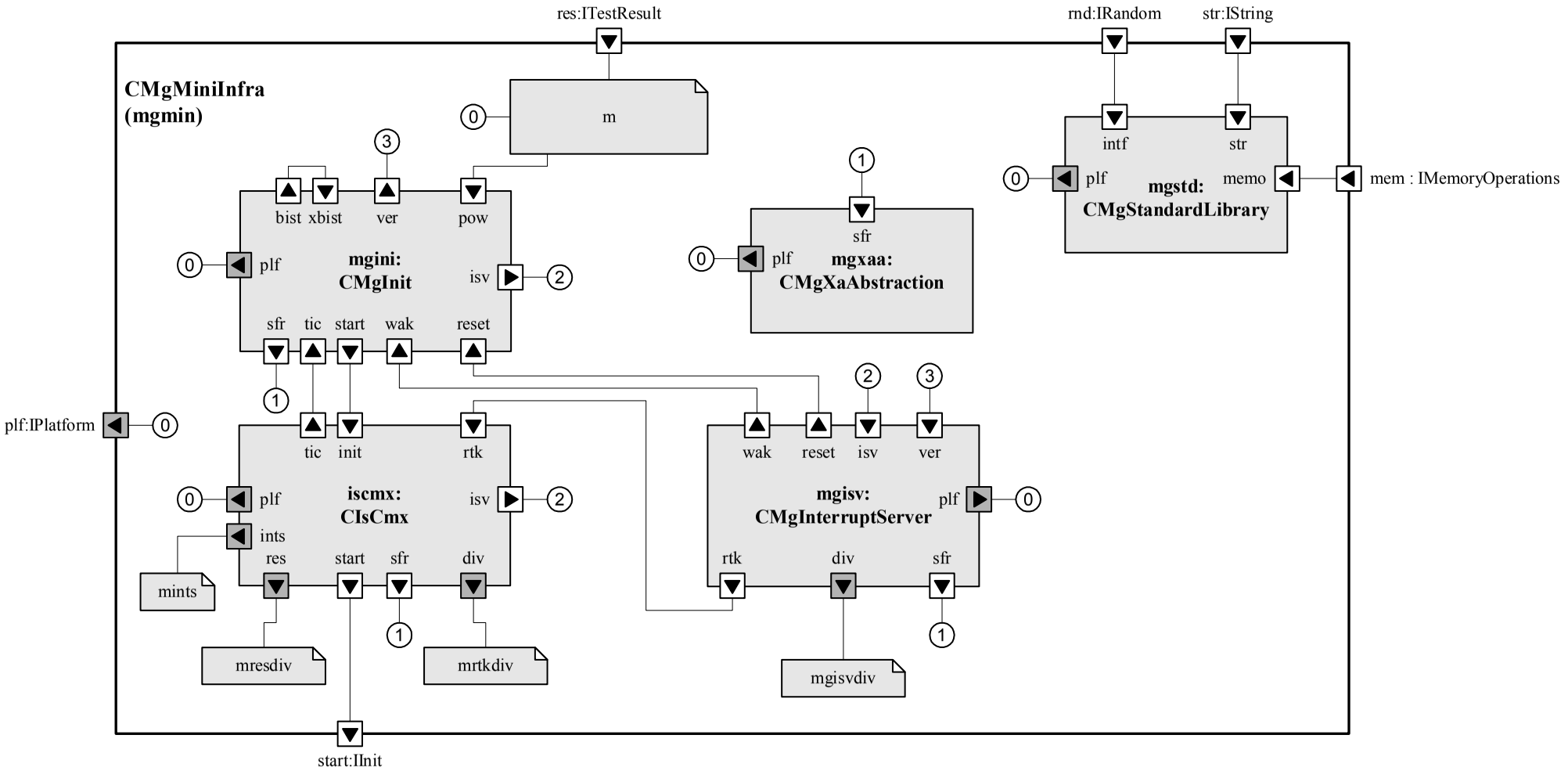


Figure 4.14: The CMgMiniInfra component. The circles with numbers inside depict connection between the corresponding *provides* and *requires* interfaces

4.9.2 Source of variability in size

The Koala component model has all sources of variability in the static quality attributes (including code and data size) that are enumerated in Section 4.3. It however introduces additional sources of variability. All sources of variability are discussed in the subsequent subsections.

A) Optional interfaces

Optional interfaces may be left unbound (see Section 3.2 or [OLK00]). Although in the example from Figure 4.14 optional interfaces are not used, they are another source of variability in the size of component code and static data in the general case. For the optional interfaces, Koala introduces a special method called "*iPresent()*". This method indicates whether methods or attributes of an optional interface are implemented by some module or component.

The influence of the *optional requires interfaces* on the size can be treated in the same way as the influence of diversity interfaces by considering the "*iPresent()*" method as a Boolean *diversity parameter*. Note that the use of *optional provides interfaces* does not directly influence the size of component code and static data. Their influence on the size of code and static data is the same as the influence of non-optional provides interfaces. For more detail the reader is referred to Section 3.2.

B) Interface binding

The Koala compiler performs a *reachabilty analysis* that evaluates the use of provides interfaces and decides whether certain components are necessary for building. All modules of a component have to be included in a build, if at least one of its *provides* interfaces is connected to other components and is implemented inside a C module. This condition is introduced for the following reason. All modules are allowed to communicate directly, i.e., bypassing interfaces, within a Koala component [OLK00]. Consider a module that needs to be included in a build. All other modules have to be also included, as they may potentially be called from the included module. The full set of rules for reachability analysis is described in Appendix B.

Notice that the analysis is performed on an entire composition. In Koala, the configuration specifies (if necessary) the values of diversity parameters and the *interface* or *function binding* to indicate how the components and modules are interconnected via interfaces.

Koala has the *switch* connection construction that controls interconnection between interfaces. A *switch,* depending on the value of its controlling expression, specifies how a set of input interfaces is connected to a set of output interfaces for each possible value of the controlling expression. Such controlling expression may include *diversity parameters*, the indication of the presence of *optional interfaces* via an "iPresent()" function, and any interface function or attribute evaluating to an ordinal value. Evaluation of these expressions, performed by the Koala compiler, may disconnect all or some *interfaces* of a component.

The results of reachabilty analysis may influence the code size twofold: (1) at the *component level* and (2) at the *module level*. The subsequent sections detail these two levels.

The component level.According to the rules of the *reachability analysis* (see section Appendix A), the building process ignores components for which all their 'provides' interfaces are not connected. All inner modules of such components are not included in a build. This exclusion may decrease the size of code and static data for the entire component composition.

The module level. This source of the variability relates to the ability of Koala to indicate if a particular *provides interface* is connected to some requires interface. This indication is implemented by a special macro-definition "*ICONNECTED*", which can be used in modules to isolate the code implementing provides interfaces. If the "*ICONNECTED*" macro evaluates false for a particular *provides* interface, the code implementing this interface and guarded with this macro is not compiled. This results in the decrease of the total size of code and static data for the entire composition.

C) Function binding

In the general case, Koala *function binding* makes it difficult to accurately predict the size of code and static data. The function binding allows the substitution of an interface function with an expression implemented using glue code and described in a "within"-section of a component description file. Processing these kinds of expressions by the Koala compiler results in the injection of some C-language expressions into the component code via macro-definitions. The accurate prediction of the code size for such components is very complex and cannot be done in practice.

4.9.3 Localization of variability in Koala components

This section briefly describes language constructs, the use of which alters the size of code and static data in Koala components (see Figure 4.15).

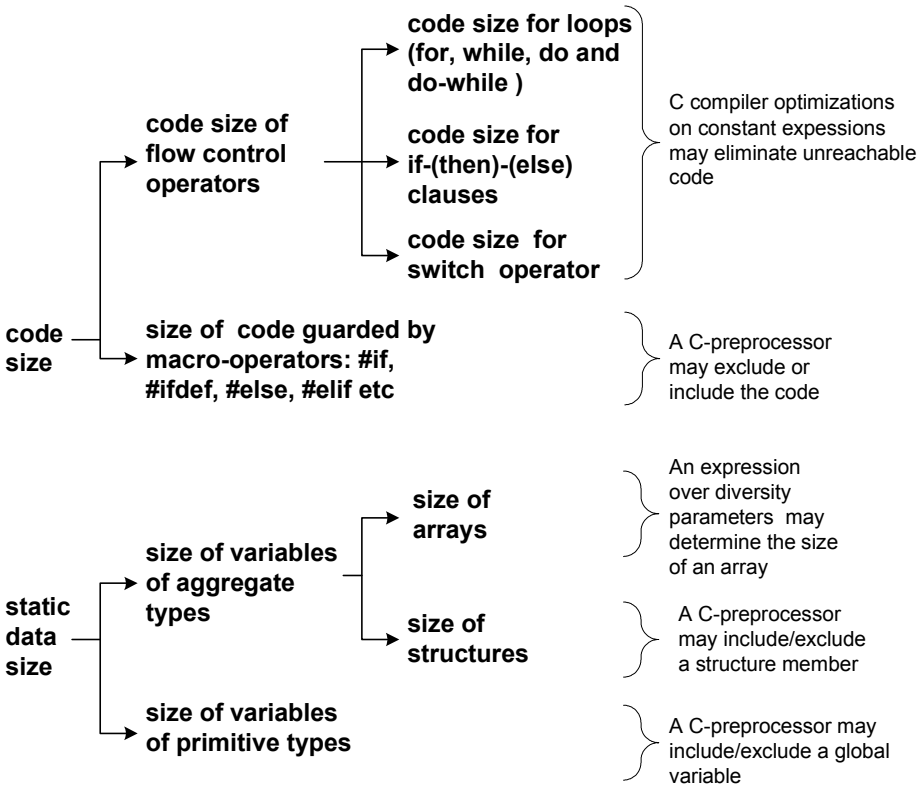


Figure 4.15: Localization of variability in Koala

All Koala components are implemented in the C language. Figure 4.15 classifies C language constructs, depending on which static quality attributes they influence and the type of the language construct. The right part of the figure indicates how the use of a particular language construct may lead to variability in memory size.

4.9.4 General formula for Koala components

In this section we shall customize Formula (4.3) from Section 4.3 to cope with Koala components. It is important to take into consideration the result of Koala's *reachability analysis* (see section Appendix B). As this reachablity analysis treats components and modules differently, it is necessary to distinguish modules and (sub)-components.

In general, the size of component code or static data can be calculated using the following formula:

$$\begin{aligned} size\left(c,\bar{D},\bar{O},\bar{P},\bar{R}\right) = & \sum_{i\in \mathrm{sub}(c)} size\left(i,F_i\left(\bar{D}\right),\bar{O}_i,\bar{P}_i,\bar{R}_i\right) + \\ & + \sum_{m\in \mathrm{mod}(c)\wedge \mathrm{reachable}(m,\bar{D},\bar{O},\bar{P})} size\left(m,\bar{D},\bar{O},\bar{P},\bar{R}\right) + \\ & + \sum_{s\in \mathrm{rtsw}\left(c,\bar{D},\bar{O},\bar{P}\right)} size(s,\bar{D},\bar{O},\bar{P},\bar{R}). \end{aligned} \tag{4.22}$$

In this formula, the following symbols are used:

- $\bar{D}$ denotes a set of the diversity interfaces of the component *c*;
- $\bar{O}$ is a set of the optional requires interfaces of the component *c*;
- $\bar{P}$ is a set of the non-optional provides interfaces of the component *c*;
- $\bar{O}_i$ is a set of the optional requires interfaces of the sub-component *i*;
- $\bar{P}_i$ is a set of the non-optional provides interfaces of the sub-component *i*;
- $F_i(\bar{D})$ is the function that maps diversity interfaces of the compound component *c* onto the diversity interfaces of the sub-component i, ;
- *sub(c)* is the set of all the sub-components of *c*;
- *mod(c)* is the set of all the modules of *c*;
- $reachable(m,d,o,p)$ denotes a predicate indicating if the module *m* of the component *c* is reachable; *d* is the diversity interfaces of *c, o* is the optional requires interfaces of *c*, and *p* is the non-optional provides interfaces of *c*;
- $rtsw(c,d,o,p)$ - the set of all the run-time switches[7] of a component *c*; *d* is the diversity interfaces of *c, o* is the optional requires interfaces of *c*, and *p* is the non-optional provides interfaces of *c*;
- $size\left(x,\bar{D},\bar{O},\bar{P},\bar{R}\right)$ is the function that calculates the size of a (sub)-component *x*, module *x*, or run-time switch *x*, taking into account the diversity interfaces $\bar{D}$, optional requires interfaces $\bar{O}$, non-optional provides interfaces $\bar{P}$, and requires interfaces $\bar{R}$[8];
- i denotes a sub-component *i* of the component *c*;
- m denotes a module *m* of the component *c*.

[7] A run-time switch occurs whenever a non-constant expression controls the switch. For more detail, the reader is referred to [OLK00].

[8] The argument $\bar{R}$ is necessary because it is possible to inject code through requires interfaces via Koala's function binding. In general, the influence of such an injection is hard to estimate.

Formula (4.22) holds both for code and static data size. However, $size\left(x,\overline{D},\overline{O},\overline{P},\overline{R}\right)$ will not depend on $\overline{R}$ for the case of static data.

4.9.5 Specification of memory consumption

We introduce an auxiliary *provides* interface *IResource* to specify memory consumption of a component. Consider an example of the "*IsCmx*" component (see Figure 4.16).

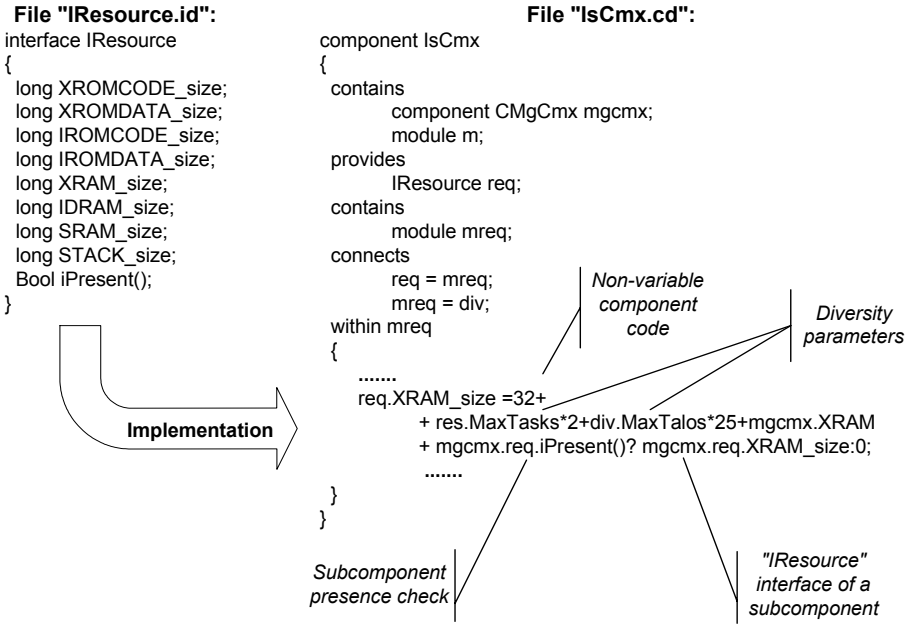


Figure 4.16: Example of an "IResource" interface

The component "*CIsCmx*" is one of components shown in Figure 4.14. It includes the module "m" and the sub-component "*CMgCmx*". Its *IResource* interface is shown in Figure 4.16. Note that this is a reflection interface as described in Section 4.5. The attributes of this interface correspond to particular types of memory. For each type of memory, the *IResource* implementation has a formula for estimating the memory size. This formula is expressed in the Koala language syntax and employs constants, expressions related to Koala features (e.g. diversity parameters), and arithmetic operations.

The formula is an expression over diversity parameters and sizes (similar formulas) of the sub-components. It also can contain some constants to denote the sizes of the inner code modules[9]. Notice that the estimation formulas were constructed using the empiric stepwise approach described in Section 4.2.

The specification of the external RAM (XRAM) size consists of the following parts:

1. *Contribution of the code module "m"*:

$$\begin{aligned} &32+res.MaxTasks\cdot 2+div.MaxTalos\cdot 2532+\\ &+res.MaxTasks\cdot 2+div.MaxTalos\cdot 25. \end{aligned} \tag{4.23}$$

This formula contains variables that depend on the diversity parameters *res.MaxTalos* and *res.MaxTasks* and some constants.

2. *Contribution of the sub-component "CMgCmx"*:

$$mgcmx.req.iPresent()\ ?\ mgcmx.req.XRAM\_size:0. \tag{4.24}$$

[9] A module is a block of code implementing the interface functions.

Note that the term above is the implementation of $in(C)$ from Formula (4.3) in Section 4.4. The expression "*mgcmx.req.iPresent()*" indicates if any of other components from the composition require interfaces provided by the component "*CMgCmx*". If "*mgcmx.req.iPresent()"* evaluates true, then the component must be included in a build, and the size of "*CMgCmx*" needs to be added to the size of "*CIsCmx*". For the component "*CMgCmx*", the interface "*req*" is also specified, and *mgcmx.req.XRAM_size* provides the size of "*CMgCmx*" to account for in the formula for "*CIsCmx*".

By using this specification technique, the memory consumption estimates for component compositions can be calculated automatically by the Koala compiler[10].

4.9.6 Evaluation

The experiment presented in this section validates both the selective and exhaustive approaches. For both approaches, formulas were constructed empirically (see Section 4.7.1). We applied these approaches to two different component compositions taken from existing TV software. The first composition consisted of seven components and was used for checking the exhaustive approach. The second composition consisted of 22 components and was used for checking the selective approach. We did not validate the two approaches on the same composition for the following reasons. On one hand, the composition of the seven components did not have a sufficient depth of component hierarchy to check the selective approach. On the other hand, applying the exhaustive approach to the composition of the 22 components would require too much effort.

The software stack for the case study was implemented for a popular micro-controller. This micro-controller uses several types of memory: external ROM (XROM), internal ROM (IROM), internal data RAM (IDRAM), static RAM (SRAM), and external RAM (XRAM). For each type of memory, the estimates were compared with the actual sizes for different sets of diversity parameters. The actual sizes were determined by compilation. The results are summarized in Table 4.5 and Table 4.6.

Table 4.5: Estimates for exhaustive approach

Type of memory	Real size (bytes)	Estimated size (bytes)	Relative error (%)
XROM Data	166	166	0.00
XROM Code	19429	19477	0.25
IROM Code	3363	3425	1.80
IROM Data	379	379	0.00
IDRAM	572	572	0.00
SRAM	145	145	0.00
XRAM	2123	2123	0.00

Table 4.6: Estimates for selective approach.

Type of memory	Real size (bytes)	Estimated size (bytes)	Relative error (%)
XROM Data	12379	12479	0.81
XROM Code	70996	71409	0.58
IROM Code	21353	21401	0.20
IROM Data	1705	1703	0.12
IDRAM	796	796	0.00
SRAM	544	544	0.00
XRAM	84471	84607	0.15

[10] The Koala compiler performs partial expression evaluation; for more details see [OLK00].

Generally, the size of static data (residing in SRAM, XRAM and IDRAM) size could be estimated with 100% accuracy. For other memory types, the relative error lies within a five-percent range, which is sufficient for many architecting and engineering appliances. The relative error achieved for the selective approach lies within the one-percent range. It was however difficult to cover the entire space of possible diversity parameters to claim that this one-percent range is achievable for all possible values of diversity parameters.

The application of the exhaustive approach took us seven days. The most time-consuming part was the stepwise formula refinement: code inspection required a number of iterations and repetitive compilation was needed to check the formulas and to calculate the sizes of the components. The implementation of the selective approach required more time (ten days), in spite of the fact that the diversity spaces of the components were reduced significantly. The reason was the mutual dependencies between components (the components defined the diversity parameters of each other). According to our experience, constructing formulas for an average component takes approximately 30-60 minutes.

4.10 Summary

In this chapter, we have described a method that allows the specification and evaluation of static quality attributes of component compositions. We have proposed two techniques for the specification of the additive static component quality attributes. The first technique advocates the use of a *reflection interface*. The model also accounts for composition aspects, such as exclusion/inclusion of components and diversity parameters. The second technique is based on building a separate specification in the XML language. The technique is illustrated by a couple of examples. This technique also supports budgeting; i.e., the expected values of the quality attributes of unimplemented components can also be accounted for. The evaluation of the resulting quality attributes can be performed with two approaches: (1) *exhaustive* and (2) *selective*. Both approaches are checked by means of estimation of the static memory size of Koala components taken from an existing TV software stack. For this purpose, the standard constructions of the Koala component definition language were used. High estimation accuracy was achieved for both approaches.

Currently, the applicability of the proposed method limited to component models for which the set of components instantiated in a composition is known before run-time. If this set of the instantiated components changes in run-time, the applicability of the method is restricted only to particular sets, that is, it will be possible to get an estimate of the quality attribute of the composition for a snapshot describing a set of components instantiated at the moment. The method also relies on propagation of diversity through the containment hierarchy. If component diversity is handled in another way, e.g. by means of a configuration database, the suggested framework must be reconsidered.

To our knowledge, only the Koala component model fully satisfies the assumptions of the proposed method. Other component models such as COM or CORBA require treating the diversity of components in a different way than the described method does.

Future research may be performed on the following directions:

1. Method validation by estimation of other static quality attributes;
2. Generalization of the method for component models with runtime binding (COM, CORBA, etc.);
3. Development of suitable languages for the specification of composition rules for non-additive static qualities.

5 Specification and evaluation of dynamic quality attributes: the APPEAR method

5.1 Introduction

Dynamic quality attributes of component-based software are essential quantities that need to be assessed as a part of early feasibility checks. Dynamic quality attributes of the software are those quality attributes that are exhibited at run-time (in opposite to static quality attributes). The examples of such quality attributes are performance, demand of dynamic memory, reliability, etc. Many architects have acknowledged the necessity of early analysis of the dynamic quality attributes. For example, Bass *et al*. [BCK98] state the following: "…You cannot get the functionality right and then go back and put in all the qualities. They all have to be designed in from the start…" The similar message is conveyed in [CN96]: "…Whether or not a system will be able to exhibit its desired (or required) quality attributes is largely determined by the time the architecture is chosen…"

In the domain of dynamic quality attributes, we limit the scope of our investigation to the analysis and prediction of software performance. Besides the timing restrictions, we had the following reasons for narrowing the scope of research: a) performance was the highest-priority issue according to the results of the investigation described in Chapter 2, and b) performance is often a critical property of embedded software.

Smith *et al*. [SW02] emphasize the importance of performance analysis and estimation at the early phase of software architecting: "…We have found that when problems are inevitable, your choice is 'pay a little now (for the scalability and performance that you need) or pay much more later.' Software metrics show that it costs up to 100 times more to fix problems in code than it does to fix problems in the architecture before code is written. The only way to determine whether problems are inevitable is to use the quantitative techniques…". Early performance estimation makes it possible to verify the feasibility of systems 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 estimating the impact of architectural decisions beforehand, in comparing architectural solutions, 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.

To date, various methods for performance modeling and estimation have been developed. Two types of methods are most frequently used: a) purely simulation-based models and b) analytical models (e.g., queuing networks, Markov chains). However, since the complexity and the diversity of software for embedded systems had grown significantly during the past decade, many existing analysis techniques have turned out to be impractical for evaluating the dynamic quality attributes (such as performance) of the entire software system. We have therefore decided to investigate how far we could push the limits of state-of-the-art methods. We did this investigation in an industrial setting, and thus we could quickly identify the following basic limitations of the existing methods (for realistic industrial cases):

1. Many simulation methods are based on an analysis of all behavioral details of the software (parameters, states, etc.). Accounting for all details usually leads to a combinatorial explosion. The high complexity of software, with hundreds of parameters influencing the software quality attributes, causes these approaches to fail.

2. Analytical methods often make too specific assumptions about the system under question. These assumptions do not hold for many systems, and thus models built using these assumptions can be both inaccurate and inadequate.
3. Analytical methods cannot adequately deal with the non-determinism of modern computing facilities— caches, pipelines, and branch predictors. Thus, it is difficult to obtain reliable estimates by analytical methods without risking serious over/underestimations.

One of the possible solutions to the first problem is the use of statistical techniques such as regression analysis. This type of analysis allows one to construct a statistical prediction model, based on measurements of the existing parts of the software, and to use this model to predict the quality attributes of newly developed or adapted parts. The use of regression techniques for software performance prediction is promising, since increasingly less software is created from scratch. There is always an initial software stack (reusable components, previous versions, etc.) that can be used for measurements and for fitting the statistical prediction model. This statistical approach abstracts from particular details of the system. However, this abstraction can cause other problems such as decreasing the accuracy of the prediction or excessive time required for the measurements. Including only the relevant details helps to shorten the time needed for constructing the prediction model, while keeping the accuracy at the desired level.

A compromise is to consider the mix of simulation and statistical techniques. This approach comprises a) modeling of the relevant behavioral details of the software, and b) use of statistical methods to abstract from irrelevant details. This mix serves as a basis for the APPEAR method (Analysis and Prediction of Performance for Evolving Architectures), described in this chapter.

The aim of the 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 components that are "sufficiently similar"[1] to the existing ones to allow the use of statistical prediction techniques. For example, the existing component responsible for viewing of medical images implements basic functionality that allows users to browse through the medical images and to view them in different modes, e.g. full-screen, four images per screen, etc. The adapted version of this component includes extra functionality: viewing images with a mask, viewing images with subtraction, etc.

The rest of this chapter is structured as follows. Section 5.2 describes the requirements for the APPEAR method. Section 5.3 introduces the essence of the method. Section 5.4 clarifies the assumptions that must hold for successful application of the method. Section 5.5 brings in the notion of signature that is the basic element of the APPEAR method. Section 5.6 describes the process of method application. Section 5.7 elaborates on the selection process of a Virtual Service Platform – another essential element of the APPEAR method. Section 5.8 details the process of signature type identification. Sections 5.9 and 5.10 are devoted to the construction of the simulation and prediction models. Section 5.11 explains the notion of prediction interval that is used for specifying the performance estimates for the adapted components. Section 5.12 summarizes the role of the APPEAR method in the architecting process.

[1] The notion of component similarity is described in Chapter 6 "Similarity of software components".

5.2 Requirements

Based on the common sense, we formulated the following list of general requirements to a performance prediction method for software components:

- The application of the method should faster than implementing the software and subsequent measurements,
- The method should be simple so that less time and fewer human resources are required for its application,
- The method should general so that it can be applied to software from various industrial domains,
- The method should be accurate enough so that it can be efficiently used for performance prediction.

We discussed these requirements with several software architects from the Professional Systems and Consumer Electronics domain and interviewed them about their requirements to a valid performance prediction method. The outcome of this interview is the following list of the requirements:

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 and their significance,
   - Construction of behavioral models (simulation models) of the components at different levels of abstraction (processes, threads, tasks, modules), and
   - Localization of performance bottlenecks.
3. Ensure a reasonable level of accuracy for performance prediction. The required accuracy level is product dependent. A 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.

5.3 Essence of the method

In this chapter, we consider the expression “performance of a component” means the performance metric for a particular performance-relevant component operation. The performance is considered in terms of CPU utilization, execution time, and end-to-end response time. In principle, the APPEAR method can also be applied for other performance metrics that relate to resource consumption (e.g., average memory demand).

The APPEAR method is based on the following view of the software stack. The software comprises two parts: (1) components and (2) a Virtual Service Platform (VSP). The components are specific for different products and can change per product version and type. To cover new functionality, the already existing components are adapted rather than creating new components from scratch. The VSP encompasses stable components, which do not significantly evolve during the software lifecycle. The VSP provides a number of services to the components (see Figure 5.1).

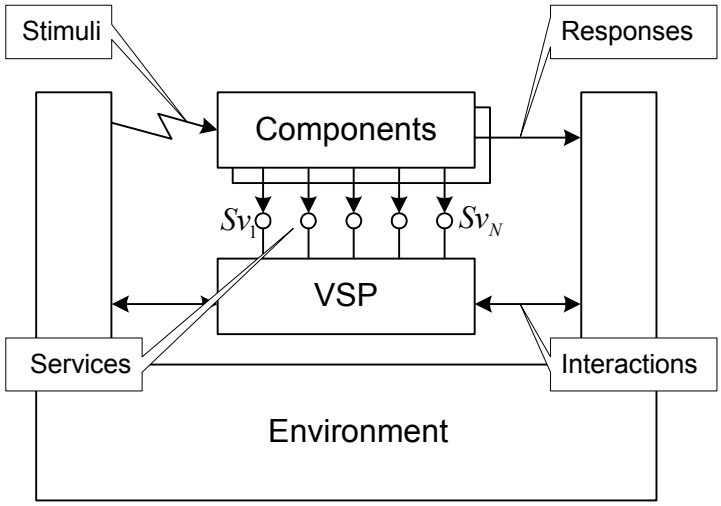


Figure 5.1: APPEAR view on the software stack

An example of VSP is a database engine that provides an application programming interface (API) for querying data from the database, managing data structures, and etc. The functions of this API are VSP services. The components comprise an application that executes on the top of this engine. In general, a VSP is easier to define for layered architectures. The criteria for definition of VSP are described in Section 5.7.

Performance estimates are obtained for a set of relevant use cases of a component. The use case describes the interaction between the component and the environment. As a result to the input stimulus, the component can call several services of the VSP to perform the functionality associated with this stimulus. After completing these calls, the component produces the response to the environment. The timing dependency between the stimulus and response can be characterised by a performance metric (e.g., response time, latency, CPU utilisation, execution time, etc.). We assume that this performance metric is correlated with the use of VSP services by components. For instance, the response time of a component that queries the database engine correlates with the number of invocations of the database query service.

The APPEAR method employs this correlation to extrapolate the performance of existing components to the performance of their adapted versions[2] This extrapolation is performed as follows. The architect needs to construct simulation models, based on the design specifications of these components. Such a simulation model should capture relevant execution parameters of the component. Those parameters are ones that have a significant impact on the performance, e.g. the most time consuming VSP services and important input or diversity parameters. These execution parameters are said to form the signature type of a component. The notions of signature type and signature instance are detailed in Section 5.5. The architect fits a prediction model to performance measurements collected for the existing components for a number of use cases representative from the viewpoint of performance. For these use cases, the prediction model relates the relevant execution parameters, obtained from a run of the simulation model of the existing component, to the measurements from the actual component. The stability of the VSP allows using this prediction model both for the existing and for adapted components. The architect can predict the performance of the adapted component as follows. First, the simulation model of the adapted is executed for the use case of interest to extract its execution parameters. Second, the prediction model is applied to these execution parameters to obtain the performance estimate of interest.

[2] We will refer the adapted versions of the existing components as adapted components in the rest of the thesis.

Notice that this chapter concerns only the performance of a single component running on the top of a VSP. Component compositions are discussed in Chapter 9. We will show how the APPEAR method can be used for modeling the processor demand and how to account for interactions between components.

5.4 Assumptions

For implementing the ideas described above in a method that can be successfully used for performance prediction, we assume the following:

1. The underlying hardware remains unchanged. Otherwise, the APPEAR method must be extended similarly to the performance prediction method described in [BK02]. The violation of this assumption will invalidate the performance measurements collected for the component and VSP. This will, in turn, invalidate the prediction model.
2. The performance of a component depends only on its internals and VSP, but not on other components. Otherwise, the prediction model must be able to capture the influence of effects such as blocking and preemption, which can occur due to interactions with other components. These effects can be hardly tackled by statistical models.
3. The services of the VSP are independent. There are no interactions that significantly influence the performance, e.g. via exclusive access to shared resources. The rationale for this assumption is similar to the rationale for the second one.
4. The order invocation of VSP services does not matter[3]. Otherwise, the prediction model fitted for particular use cases will fail to capture the performance contributions attributed to particular VSP services for other use cases.
5. The adapted and existing components are similar (see Chapter 6). If the adapted components are not sufficiently similar to the existing ones, the prediction can fail, because the observation data are not applicable anymore.
6. The amount of experimental data collected for the existing component is sufficient in the neighbourhood of the input data of adapted component to provide a robust prediction.
7. Tools for performance measurements of the components are available, e.g. tracing tools. Otherwise, significant effort may need to be spent for making proper instruments for measuring the performance.
8. The architectural information that is relevant for building a simulation model is available. This information includes documentation, architect/designers interviews, source code of the existing component, etc. Otherwise, the construction of simulation models will require significant effort, as it will be necessary to reconstruct the missing information from the code or traces of the running software.

5.5 Signature type and signature instance

The role of a signature type and signature instance is to describe the use cases of a component from a performance viewpoint. The signature type represents a vector of performance relevant parameters that a) correlate with the performance of components, and b) can be observed at the architectural level. A signature instance contains values of these parameters for a particular use case. The signature type can be expressed in terms of a) calls to VSP services, b) input and diversity parameters, c) observable component states, and etc. For example, the signature type of a hypothetical software component can look as follows:

[3] Even if the call order matters, in some cases it is still possible to use the APPEAR method and to predict the performance. More information on this topic can be found in Chapter 6.

S = (Number of memory allocation calls, Number of disk calls, Number of network calls)

The signature instance contains actual values for a concrete use case, e.g. s = (132, 57, 21).

We define the signature type $\vec{S}^{(i)}$ of a component operation *i* as follows:

$$\vec{S}^{(i)} = \left(S_1^{(i)}, ..., S_{k_i}^{(i)} \right), \quad S_j^{(i)} \subseteq \mathbb{R}. \tag{5.1}$$

In this formula, $S_j^{(i)}$ is *j*-th signature parameter of the signature type $\vec{S}^{(i)}$. The signature type $S^{(i)}$ consists of k_i signature parameters. The signature instance $\vec{s}$ is a value of the vector corresponding to a certain signature type $\vec{S}^{(i)}$:

$$\vec{s} = \left(\mathrm{s}_1, \mathrm{s}_2, ..., \mathrm{s}_{k_i} \right), \quad s_j \in S_j^{(i)}, \quad j \in 1...k_i \tag{5.2}$$

In Formula (5.2), s_j are values of signature parameters $S_j^{(i)}$.

We treat the performance $P^{(i)}$ of the component operation *i* as a function over the signature type $S^{(i)}$:

$$C = P^{(i)} \left(\vec{S}^{(i)} \right). \tag{5.3}$$

In this formula, *C* is a performance metric such as response time. In the general case, this $P^{(i)}$ function is unknown. However, the architect can apply the statistical regression techniques to construct an approximation, which we call a prediction model (See Sections 5.3 and 5.6).

The identification of the signature type presumes answering the following questions:

1. Which of the hundreds of parameters have a significant impact on the performance?
2. What is the quantitative dependency between these parameters and the performance?

The answer to the first question helps us to reduce the parameter space and to concentrate on the critical parameters only. The answer to the second question allows us to predict the performance based on the experimental data.

5.6 The description of the method

The proposed method includes two main phases: (1) calibrating the prediction model on the existing components and (2) applying this prediction model to the adapted component to obtain its performance estimate. These two phases are described in Sections 5.6.1 and 5.6.2. Sections 5.7 to 5.10 detail particular steps of the first phase.

5.6.1 Phase 1: Calibration

First, it is necessary to identify the signature type and to construct a statistically valid prediction model[4]. This can be accomplished according to the following procedure (Figure 5.2).

[4] The statistical validity of the prediction model consists in ensuring that it satisfies a number of tests that indicate the quality of the prediction. Usually, a prediction model is built with regression techniques and the criteria, in this case, are the normality of residual distribution, variance constancy, etc.

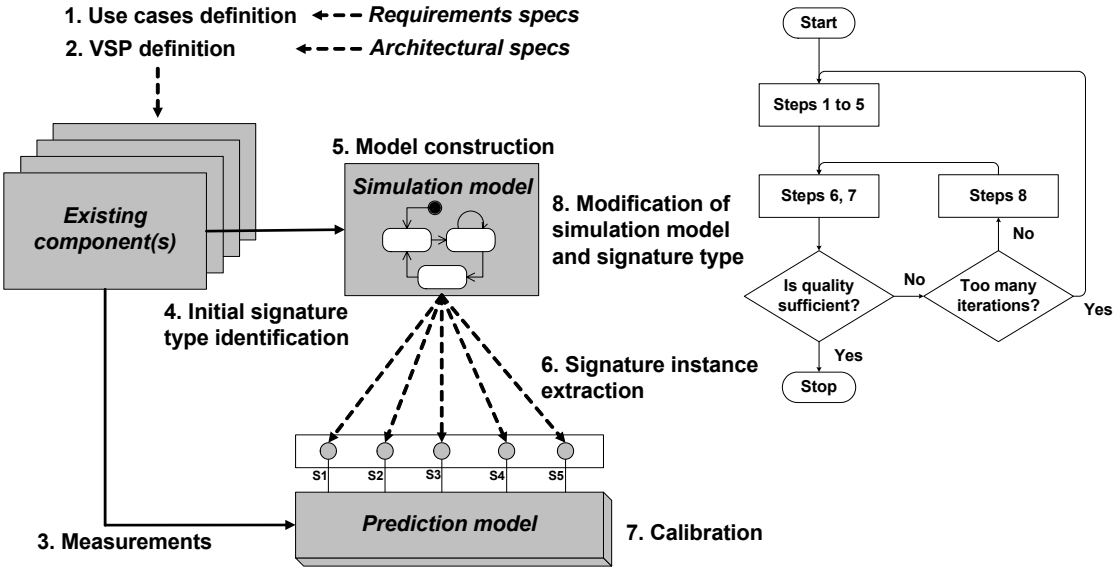


Figure 5.2: Calibration of the prediction model

The left part of Figure 5.2 shows the information flow between various entities of the APPEAR method. The information flow is depicted by arrows, and the entities by boxes and circles. The boxes and arrows are also labeled with the number of the step that concerns the corresponding entity or information flow. The order of steps of the calibration phase is described in a form of flowchart in the right part of Figure 5.2. After performing the first five steps, the prediction model is calibrated in an iterative way. The fourth step can be omitted, if the architect has sufficient preliminary knowledge about the performance-relevant parameters and can therefore estimate the initial signature type beforehand. Otherwise, the architect can identify the initial signature type by applying a regression technique as described in Section 5.8. All steps of the calibration phase are as follows:

Step 1, Use cases definition. Based on the requirements specification, a relevant set of use cases is chosen. These use cases are used for signature instance extraction, i.e. obtaining the values of the signature type.

Step 2, Virtual Service Platform identification. Based on the architectural specification, the software stack is subdivided into two parts: components and a Virtual Service Platform. The guidelines for VSP selection are described in Section 5.7.

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. The choice of the signature type determines whether the constructed prediction model calibrates well.

Step 5, Construction of the initial simulation model. Based on the available architecture description, a simulation model needs to be built to extract signature instances, i.e. to determine the values of the performance relevant parameters that can be observed at architecture level. It is important to construct a simulation model at this stage, as the prediction model should be calibrated using the signature instances generated by the simulation model, but not obtained from the traces. Otherwise, the prediction model will not be usable for the adapted components for which the traces are not available.

Step 6, Signature instance extraction. By executing the simulation model on the defined use cases, a signature instance is calculated for each use case. These values and the corresponding measurements obtained during Step 3 form a calibration dataset. The signature instances are then treated as values of input variables of the prediction model.

Step 7, Prediction model calibration. The results of Steps 3 and 6 form the input data for building a model that predicts the value of the performance measure, depending on the signature instance. Each sample of this data corresponds to a use case. By applying one of the existing regression techniques [Wei95], [KO02], it is possible to build a prediction model. This process is called calibration.

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. In this case, Steps 6-8 must be repeated until the prediction model becomes statistically valid. The signature type is thus iteratively refined during Steps 6, 7, and 8.

5.6.2 Phase 2: Prediction

As described in Section 5.3, the existing components may be adapted to account for additional functionality. For the adapted component, the design specifications are based on the design specifications of the existing components. The performance of the adapted component can be predicted by the prediction model constructed during the calibration phase (5.6.1) only if this adapted component is similar to the existing one (see Section 5.4). In Chapter 6, we will show that the architects can ascertain the similarity of the existing and adapted component by the following formula:

$$Similar = ST \wedge SI \wedge IC. \tag{5.4}$$

In Formula (5.4), *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 5.1

Table 5.1: 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 comparable for the existing and adapted components.

Chapter 6 describes these similarity criteria in more detail. If these criteria are met, the architect can use prediction intervals (see Chapter 5.11) for judging about the accuracy of the obtained predictions. Otherwise, the architect is not advised to use the obtained predictions in the general case, as they may be imprecise. Although, the architect can try to remedy this problem by taking one the “escape routes” described in Chapter 6.

The performance can be predicted for adapted components by sequentially performing Steps 9 to 11 (see Figure 5.3).

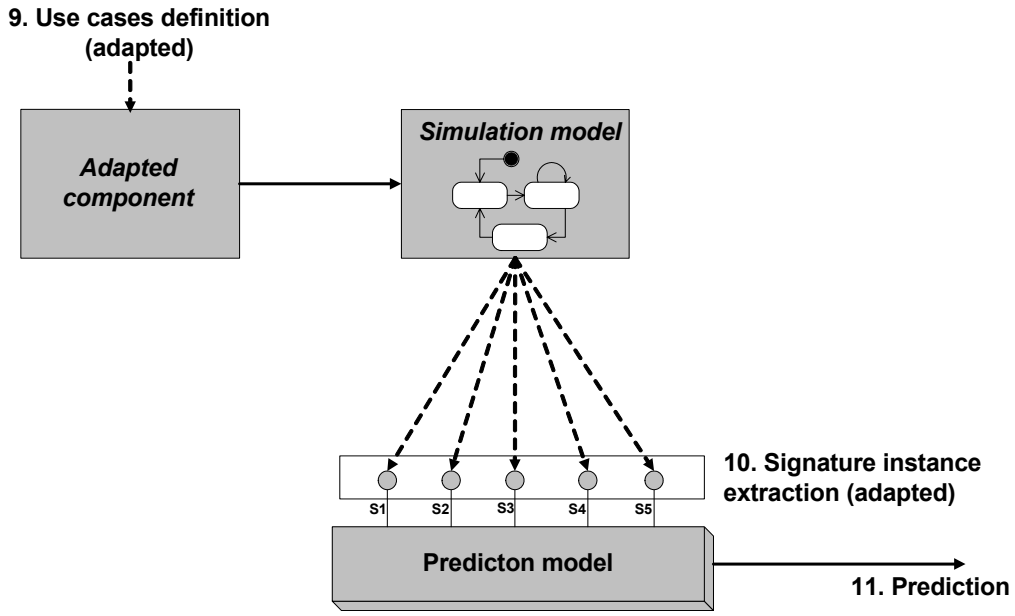


Figure 5.3: Prediction of the performance with the prediction model

Figure 5.3 shows the information flow between the entities relevant for the prediction phase. Figure 5.3 uses the same notations as Figure 5.2. The list of the steps is given below:

Step 9, Definition of use cases for adapted components. A set of use cases that needs performance prediction is determined for the adapted components. According to these use cases, the initial simulation model of the existing component is modified (if needed), with the signature type kept intact. Intact signature type is an essential condition for ensuring component similarity. The component similarity is required for the successful use of the prediction model calibrated on the existing component. Other essential conditions of the component similarity can be found in Chapter 6.

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. If the adapted component is sufficiently similar to the existing one, this signature instance can be used for predicting the performance.

Step 11, Predicting the performance of the adapted component. By applying the existing prediction model to the signature instance obtained during Step 10, the performance of the adapted component is estimated. The accuracy of the estimates can be represented in terms of prediction intervals, if the prediction model is fitted using linear regression (see Section 5.11). This estimate must further be interpreted by the architects with respect to performance requirements and obtained architectural insights.

We advise to validate the models constructed by the APPEAR method after implementing the adapted component to obtain a feedback on the modeling precision and to possibly refine the models. The adapted component is measured on the use cases for which its performance has been predicted. The predictions made at the architecting phase are compared to the measurements collected. If the performance predictions deviate from the measurements significantly, the reasons behind that should be carefully investigated. The possible reasons can be the following:

1. The signature type was chosen incorrectly.
2. The simulation model of the adapted component was incorrect.
3. The implementation of the adapted component deviated from the architecture that had been considered at the time of applying the APPEAR method; as a result the adapted component became not sufficiently similar to the existing ones.

Addressing the first two reasons may require refining the simulation model, signature type, or/and prediction model[5]. Addressing the third reason may need improvement of the implementation process.

Notice that the proposed method has an important property: during the evolution of a component, the calibration dataset grows continuously, as increasingly more component variants become available for fitting the prediction model. This growth may help in enhancing the prediction model quality, as the coverage of the measurements increases.

We validated the APPEAR method by applying it to two industrial case studies in the Consumer Electronics and Professional Systems domains. These validation experiments are described in Chapter 7 and Chapter 8, respectively.

5.7 VSP identification (Step 2)

The architect needs to determine the VSP boundary to be able to express the behavior of components in terms of calls to VSP services (see Figure 5.1). On one hand, this boundary splits the software into two parts: (1) a part that consists of components that are subject for changes, and (2) the stable part that consists of service components, which do not change over time. The latter serves a basis for the VSP. Typically, there are always relatively stable parts of a software product, and ones that need modifying or extension. On the other hand, the VSP boundary concerns also a proper abstraction level, at which the behavior of the components under consideration is described. The following aspects influence the choice of this abstraction level:

- *The choice of signature type at Steps 4 and 8 of the APPEAR method* (see Section 5.6.1). The abstraction level may need to be lowered in order to be able to fit a statistically valid prediction model. This lowering implies that the architect may need to model some of the components that belong to the stable part explicitly.
- *Explicit modeling of the performance of relevant components*. The architect needs to obtain an insight into the performance relevant behavior of particular components that belong to the stable part. This may also require lowering the abstraction level.

5.8 Identification of the initial signature type (Step 4)

If the software stack is properly instrumented, it is possible to deduce the initial signature type from the use case traces, i.e. by logging and analyzing all performance-relevant service calls. The correlation between the use of these service calls and the performance metric is analyzed by constructing an auxiliary prediction model. After the identification of the initial signature type, the auxiliary prediction model is abandoned. The parameters, relating to certain VSP calls, that calibrate this auxiliary prediction model with sufficient quality form the initial signature type.

The flowchart for constructing such an auxiliary prediction model is shown in Figure 5.4:

Step 1, Virtual Service Platform identification. The guidelines and criteria for selecting the level of the VSP are described in Section 5.7.

[5] This basically means that the adapted component is not similar to the existing one, since similar components should have the same signature type. As a consequence, the existing prediction model cannot be directly used. However, we provide a number of recommendations for obtaining performance estimates even in the cases of non-similar components (see Chapter 6).

Step 2, Use cases definition. It is vital to determine a representative set of the use cases to collect data for calibrating the auxiliary prediction model.

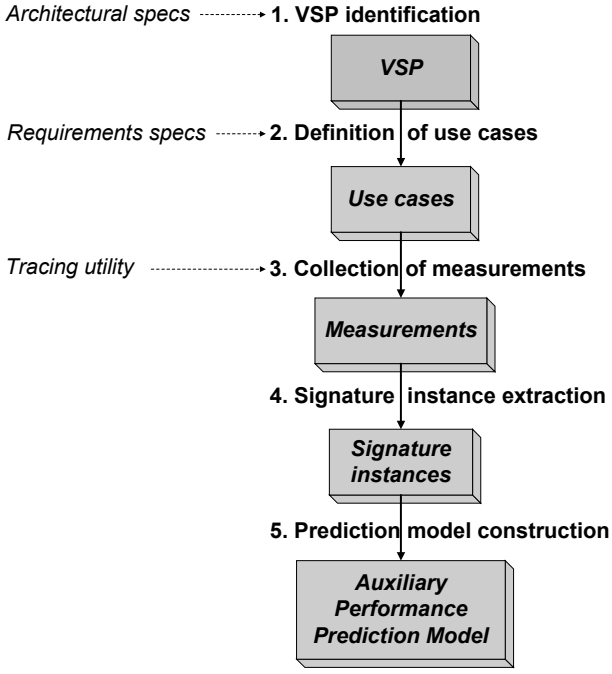


Figure 5.4: Main steps of the initial signature type identification

Step 3, Collection of measurements. The selected use cases need to be executed and traced.

Step 4, Signature instance extraction. The performance-relevant parameters can be determined from the use case traces. These parameters describe a component from a performance viewpoint and form the initial signature type.

Step 5, Construction of an auxiliary prediction model. The auxiliary performance prediction model needs to be calibrated based on the measured signature instances and corresponding performance metrics of the component.

We also advocate the use of multiple[6] linear regression for constructing the auxiliary prediction model. This model is constructed in exactly the same way as described in Section 5.10.

5.9 Simulation model construction (Steps 5 and 8)

Based on the interviews with the architects and on the study of the existing performance modeling techniques [SW02], we formulated several essential requirements that a proper simulation model should satisfy to be applicable in the context of the APPEAR method:

1. *This model should be relatively simple (at the proper abstraction level)*. Construction/modification of the model should be much faster than implementing or prototyping the software under consideration, and the model should be manageable by a single person (the architect). For the sake of comprehensibility, there should be a simple mapping between the simulation model and the architecture.

[6] The statisticians use the expression 'multiple linear regression' to refer to a linear regression technique that constructs linear regression models having multiple input parameters. Traditional linear regression concerns only a single input parameter.

2. *The model should provide insight into the execution architecture*. It should include all performance relevant details and assist in the identification of performance bottlenecks.
3. *The model should allow the extraction of signature instances*. The APPEAR method relies on the possibility to extract the signature instances.
4. *The model should preferably be executable*. It should be possible to simulate the component behavior and to observe it. This provides an additional insight into the component behavior.

It is important to consider how easy it is to modify the simulation model of an existing component so that it can also model the adapted components. On one hand, the similarity (see Chapter 8) between the existing and adapted components determines how many modifications are needed. This can only be identified on basis of the architecture of the adapted component. One has to check if the assumptions made for building a simulation model(s) remain valid. On the other hand, the techniques for building simulation models influence the speed of incorporating changes. It is useful to build such a model based on the extensive use of CASE and rapid development tools. These tools often support models that can be easily mapped onto the architecture of the component, e.g. in terms of state machines.

5.10 Prediction model calibration (Step 7)

The prediction model can be constructed by any regression technique such as multiple linear regression [Wei95], [JAI91], [MON01], MARS [Fri90], lazy learning [Bon99], etc. However, we advocate the use of the multiple linear regression for the following reasons:

- Prediction models calibrated using linear regression techniques have simple form and can be easily understood.
- Linear regression has a well-developed mathematical basis that allows extensive checking of the model quality and provides powerful mechanisms for making inferences about the model parameters and predictions, e.g. by means of confidence and prediction intervals.
- Many tools are available for performing linear regression analysis.

When applying (multiple) linear regression, the performance is assumed to depend linearly on the signature parameters. This relation can be represented as follows:

$$P_i = s_i^T \beta + \varepsilon_i. \qquad (5.5)$$

In Formula (5.5), P_i denotes the performance measurement for the *i*-th use case;

$\beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{k-1} \\ \beta_k \end{pmatrix}$ is a vector of population[7] linear regression coefficients;

[7] Here, by population coefficients we mean that these coefficients describe the linear regression model that covers the entire set of all possible performance measurements.

$s_i = \begin{pmatrix} s_{i0} \\ s_{i1} \\ \vdots \\ s_{ik-1} \\ s_k \end{pmatrix}$ is a signature instance [8] for the *i*-th use case; ε_i represents prediction errors, which must be normally distributed random variables with mean zero and constant standard deviation σ [9]. These errors must also be uncorrelated. This means that the covariance matrix E has the following form:

$$E = \begin{pmatrix} \varepsilon_{11} & \cdots & \varepsilon_{1n} \\ \vdots & \ddots & \vdots \\ \varepsilon_{n1} & \cdots & \varepsilon_{nn} \end{pmatrix} = \sigma \cdot I, \tag{5.6}$$

where *n* is the number of use cases, and *I* denotes the identity matrix.

The estimates $\hat{\beta} = \begin{pmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \\ \vdots \\ \hat{\beta}_{k-1} \\ \hat{\beta}_k \end{pmatrix}$ of the linear regression coefficients are obtained by minimizing the sum of squared errors (the least squared error method [JAI91], [Wei95], [MON01]):

$$\begin{aligned} P_i &= s_i^T \hat{\beta} + e_i, \\ \hat{\beta} &= \arg\min_b \sum_{i=1}^{N} e_i^2 = \arg\min_b \sum_{i=1}^{N} \left(P_i - s_i^T b \right)^2. \end{aligned} \tag{5.7}$$

In Formula (5.7), $\hat{\beta}$ is a vector of estimates of the linear regression coefficients β; the e_i random variable is the residual of *i*-th observation that denotes the error not explained by the prediction model. The function $\arg\min_b f(b)$ calculates the value of argument b that minimizes the function $f(b)$.

Notice that e_i must also be independently and normally distributed with mean zero and constant variance, because of the assumptions of linear regression, expressed by Formulas (5.7) and (5.6).

Linear regression allows one not only to find the estimates of regression coefficients, but also the *p*-values of the associated *t*-statistics[10] of the signature parameters (see [KO02]). These *p*-values are the probabilities that the regression coefficients for some signature parameters are zero. The greater the *p*-value, the less likely it is that the corresponding signature parameter explains the variation of the performance metric over

[8] The statistics literature traditionally uses term predictors or independent variables for referring to signature parameters. Please notice also that s_{i0} is often considered to be equal one. This is convenient for modeling the constant term (interceptor) in a multiple linear regression model.

[9] The value of σ is usually unknown.

[10] In linear regression, t-statistics are used to check the null hypothesis about the regression coefficients, i.e. to check if the regression coefficients are likely to be zero for particular signature parameters.

the set of use cases. The ultimate goal is to arrive at a list of signature parameters that are all significant, i.e. the *p*-values are below a certain threshold.

When interpreting the *p*-values, the architect has to pay attention to the variation of the values of signature parameters with large *p*-values. The values of certain signature parameters might not vary significantly for a given set of use cases. In this case, additional use cases should be traced to make sure that this is not a measurement artifact. In addition, *p*-values are only valid for a particular combination of signature parameters. If a signature parameter is omitted or added, the *p*-values of the other parameters usually change.

The architect can use the *p*-values for constructing the list of performance relevant parameters. A *p*-value threshold, usually called a significance level, can be used for determining the significant parameters. The choice of this threshold depends on the context (a typical threshold is 0.05). The parameters with p-values greater than this threshold should be excluded from the signature type. The *p*-values can also be ignored and the entire list of signature parameters is kept intact, if the architect is sure that all signature parameters are relevant for modeling the performance and none of them can be omitted. However, using statistical inferences such as confidence and prediction intervals for the performance metric is not recommended in this case, because they are likely to be incorrect [JAI91], [Wei95].

5.10.1 Validation of the prediction model

Crucial information about the quality of the prediction can be deduced from the analysis of the prediction model's residual and relative (or absolute) prediction errors. Several assumptions about the model's residual and input data set are usually made, depending on the regression technique used to construct the prediction model. The assumptions typically made for the case of linear regression are as follows:

1. The residual has a normal (Gaussian) distribution with constant variance[11] and a mean value of zero.
2. There is no serial correlation[12] between residual and dependent variable (a variable for which prediction is needed).
3. There are no outliers. An outlier is an observation that significantly deviates from the bulk of a sample.
4. There are no influential observations. These types of observations have a major effect on the coefficients of a prediction model, in comparison to other observations.
5. There is no linear correlation between independent variables (signature parameters). If there is correlation, it is said that the data is multicollinear; if the correlation is perfect, the data is singular.

Analysis of residuals helps to explain inaccurate predictions and can provide additional insight into the component behavior. Such an analysis may help in the identifying (1) missed signature parameters, (2) independent groups of measurements, and (3) insufficient amount of measurements in the neighborhood of certain points. For example, the analysis of the residual provided additional insight in the case study about the TV software and assisted in signature type selection (see Chapter 7). The analysis of the relative error helped to identify the necessity to cluster the measurements in the case study on Medical software (see Chapter 8).

[11] The phenomenon of variance constancy is often called global homoscedasity.

[12] Serial correlation within a sequence of observations means that each observation is correlated with a number of previous observations.

In addition to the analysis of residual, the quality of the prediction model can be verified by using cross-validation techniques such as leave-one-out, leave-K-out, etc. [MON01].

5.11 Prediction accuracy

Consider a prediction model built as described in Section 5.10. Let $\hat{P} = s^T \hat{\beta}$ denote the average performance estimate that corresponds to a particular signature instance s. The future observation P of the performance metric for a signature instance s can be described by a prediction interval at the confidence level[13] α, calculated as follows [JAI91], [Wei95], [MON01]:

$$\hat{P} - t_{1-\alpha/2;n-k-1} \cdot \hat{\sigma}_p \le P \le \hat{P} + t_{1-\alpha/2;n-k-1} \cdot \hat{\sigma}_p \tag{5.8}$$

In Formula (5.8), the following notations are used. The $t_{q;d}$ parameter denotes the q-th quantile of the t-distribution with d degrees of freedom; n and k are the number of use cases and signature parameters, respectively. The $\hat{\sigma}^2$ statistic denotes the standard error of fit and can be found by Formula (5.11). The $\hat{\sigma}_p$ statistic is an estimator of the standard deviation of a prediction:

$$\hat{\sigma}_p = \hat{\sigma}\sqrt{1 + s^T \left(S^T S\right)^{-1} s}\,. \tag{5.9}$$

In Formula (5.9), S is a matrix consisting of the signature instances used to fit the prediction model:

$$S = \begin{pmatrix} s_{10} & \cdots & s_{1k} \\ \vdots & \ddots & \vdots \\ s_{n0} & \cdots & s_{nk} \end{pmatrix} = \begin{pmatrix} s_1^T \\ \cdots \\ s_n^T \end{pmatrix}. \tag{5.10}$$

We will further refer to these signature instances as calibration data. The standard error of fit $\hat{\sigma}^2$ is calculated on the basis of measured performance metrics P_i and fitted performance metrics $\hat{P}_i$:

$$\hat{\sigma}^2 = \frac{1}{n-k-1} \sum_{i=1}^{n} \left(P_i - \hat{P}_i\right)^2. \tag{5.11}$$

For conciseness, we denote the lower bound of the prediction interval as *PL* and the upper one as *PU*. They are calculated by the following formulas:

$$PL = \hat{P} - t_{1-\alpha/2;n-k-1} \cdot \hat{\sigma}_p, \tag{5.12}$$

$$PU = \hat{P} + t_{1-\alpha/2;n-k-1} \cdot \hat{\sigma}_p. \tag{5.13}$$

Obviously, the following inequality holds for the future observation P and prediction interval bounds:

$$PL \le P \le PU. \tag{5.14}$$

There are a number of assumptions that must be satisfied to make the application of prediction intervals valid:

1. The distribution of the prediction errors must be normal.

[13] The confidence level is usually chosen by the user.

2. The variance σ of the errors must be constant.
3. The signature instances s_i must be uncorrelated. That is, the correlation coefficient must be low (e.g., less than 0.4). This is needed to avoid numerical problems and to ensure the decoupling of the effects of different signature parameters, such that these effects are not confused with each another.

Figure 5.5 illustrates the use of a prediction interval for a single-parameter signature type. For a particular signature instance s, the predicted value *P* should lie between *PL* and *PU*, calculated at a certain confidence level α (usually 95%).

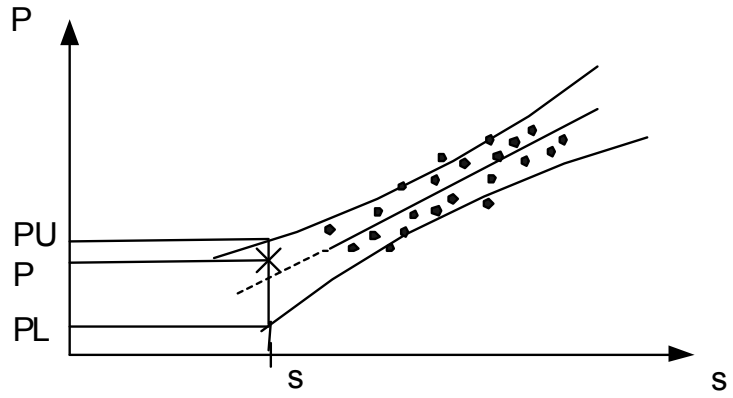


Figure 5.5: Prediction interval

The use of prediction intervals may not always be valid. This can happen because the signature instance, for which the performance prediction must be done, is located far from the calibration data. Let us consider an example, where the performance of a hypothetical system (see Figure 5.6) is depends on a single signature parameter *s*.

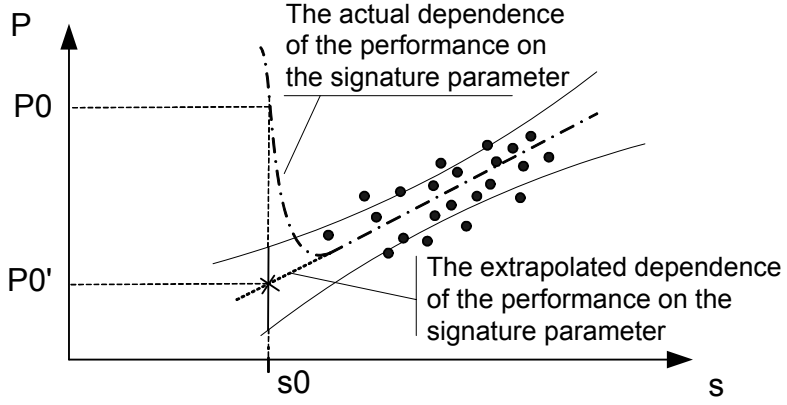


Figure 5.6: System behavior is different for remote points

A linear prediction model is constructed based only on a particular interval of the signature parameter values. However, the system can radically change its behavior for signature instances lying far from the calibration data. Therefore, the predictions outside the range of calibration data can be invalid. In Figure 5.6, the actual performance measurement equals *P0* for the signature instance *s0*, whereas the performance estimate equal *P0'*. Please notice that the actual performance lies far outside of the prediction interval for the estimate.

5.12 Scope of the APPEAR method

This section describes the scope of the APPEAR method and its role in the architecting process. The main relationships are described by the UML diagram shown in Figure 5.7.

The gray boxes and the bold arrows are the parts of the diagram that are supported by the APPEAR method.

Architects have a rationale behind the choice of a particular architecture. This rationale supports the architectural description, explains why this architecture is adopted, and clarifies main architectural concepts. To reason about the architecture, an architectural insight is required. This insight can be obtained by constructing architectural models that constitute the architecture description. The *architectural models* provide the architect with an *architectural insight*, both in terms of general architectural notions and in terms of compositional entities (e.g., connectors and subcomponents).

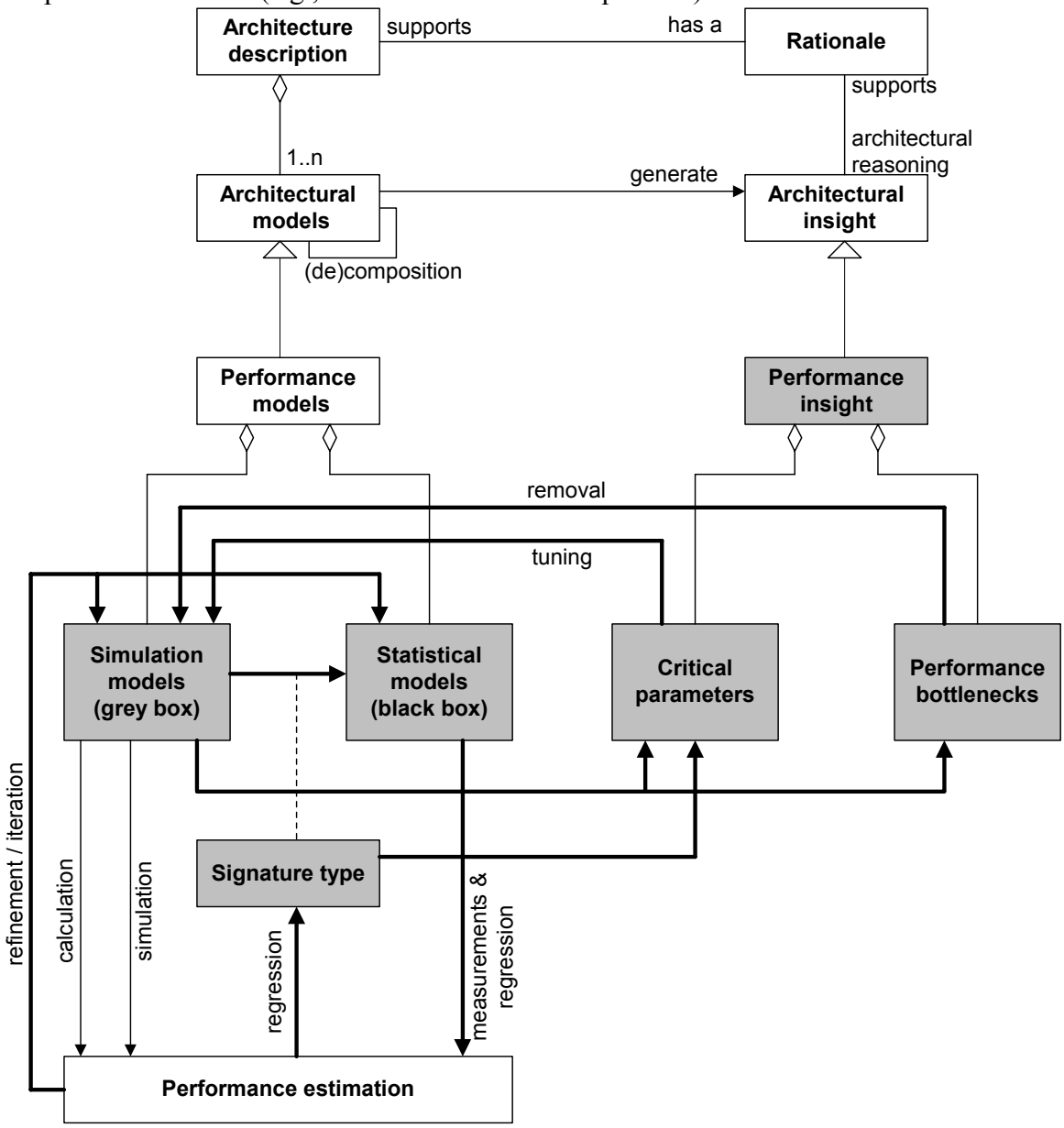


Figure 5.7: APPEAR scope

The APPEAR method provides an additional insight into the performance relevant parts of an architecture. It helps to understand why the software system exhibits particular performance and how this performance can be controlled and, probably, be optimized. *A Performance insight* concerns (1) performance *critical parameters* and (2) *performance bottlenecks*. Critical parameters are input parameters (e.g. the amount of data to process) or static architectural parameters (e.g., buffer size) that directly influence the performance. Architectural bottlenecks are architectural concepts that cause performance losses.

The basic result of applying the APPEAR method to the existing software is that architects become aware of the *bottlenecks* and *critical parameters*. During our industrial case studies, described in Chapters 7, 8, 10, and 11, we found various justifications for this fact. For instance, the performance of a software component may turn out to be limited by the underlying hardware (see Section 11.4). Besides the knowledge of performance relevant details, application of the APPEAR method can also support performance optimization (e.g., minimization of either the number of service calls or their duration). After analyzing the software by the APPEAR method the code may be modified to implement the necessary optimizations.

Performance critical parameters correlate with the performance measurement. They comprise a vector, which we call *signature type.* The *signature type* relates to service calls, input and diversity parameters, internal states, etc. It can be identified by means of statistical regression. A regression technique provides the architect with the so-called *p*-values describing the significance of the candidate signature parameters. These *p*-values show whether the signature parameters are likely to explain the observed variation of performance. Therefore, these *p*-values can be used for discovering the *signature type*.

Performance models allow the architect to gain a *performance insight*. These models show the dependencies between signature parameters, bottlenecks, architectural solutions and the performance of the software. The role of these models is to estimate the expected performance before implementing the software. Currently, such models are often constructed after the implementation, when either the software has to be extended or its performance has to be improved. The implementation effort can be reduced if the performance models are constructed beforehand.

Statistical performance models are "black-boxes": they reflect only the relation between software input parameters and the performance, but do not explore any internals of the software. These models are constructed by fitting them to measurements from the software. Using these models, the architect can estimate the performance of the software. A performance prediction model constructed by the APPEAR method is an example of such a statistical model. Statistical prediction models are useful, as they serve the following functions:

- They can provide higher prediction accuracy than simulation models, since they automatically account for unexpected overhead, deviations in execution times, implementation details, etc.
- They help identify complex run-time dependencies in the software (e.g., by means of the residual observation, analysis of contribution of each signature parameter to the performance metric, etc).
- They can help in exploring the correlation between input data, and thus diminish the number of performance relevant parameters.

However, statistical models provide little *architectural insight*, since they consider the system as a "black box". Therefore, statistical prediction models are much more useful when accompanied by simulation models.

Simulation models describe the performance relevant internals of a software system. They are usually "gray-boxes": the behavior of the software is abstracted to performance significant aspects only. These aspects can concern both the architectural solutions (e.g., a scheduling policy) and the implementation details (e.g., a search algorithm). Performance estimates are usually calculated on the basis of a number of simulation runs, each run modeling a single execution of the software system.

Simulation models play several important roles:

- They allow the architect to investigate the details that are important during software execution. Often, the performance effects of these details become visible only on the long run (e.g. jitter of a certain task),
- They can be used for modeling complex dependencies (e.g., dependencies on the history),
- They provide an opportunity to change architectural solutions or the values of particular performance relevant parameters, and to quickly observe how these changes influence the performance.

Sometimes, a *simulation model* can amount to a simple analytical model (e.g., a formula with a couple of parameters). Such a simple models can used for either simple software or for complex software for which only general *performance insight* (a rough estimate) is required. For example, such a model can demonstrate that the performance depends on a few input parameters only or that the performance is determined by underlying hardware only (e.g., disk or memory speed), with software design/implementation details being irrelevant (see Chapter 11). For such cases, the initial estimates made during the architecting phase, based on the parameters of the underlying hardware (e.g., disk speed), are often sufficient to accurately predict the performance. There is therefore no need to construct complex *simulation* and/or *statistical models*.

However, the use of such simple models also has a number of drawbacks:

- Overhead in the implemented software (e.g., communication overhead) can affect the performance and thus deteriorate the performance predictions,
- Relevant performance aspects can remain unexplored,
- External performance-relevant factors can be overlooked (e.g., non-deterministic response times when transporting data through a network).

Since modern software is rather complex, both *simulation* and *statistical models* must usually be used. *Simulation models* can be used to extract signature instances. Signature instance extraction is, in fact, a transformation of architectural concepts and solutions into performance relevant parameters. These parameters are then used as inputs for a *statistical model*. Construction of *simulation* and *statistical models* is an *iterative* process. The models are usually validated against the measurements and iteratively refined until they become sufficiently detailed and accurate.

6 Similarity of software components

In Chapter 5, we introduced the APPEAR method that can be used to predict the performance of an adapted component, based on the performance models of the existing one. The successful application of this method is only possible when its assumptions (see Section 5.4) are satisfied. Particularly, the adapted and existing components must be "similar". The purpose of this chapter is to elaborate this notion of component similarity. Note that in this chapter, as in Chapter 5, we do not consider component interactions, which are tackled in Chapter 9.

6.1 Problem statement

The APPEAR method considers the software stack in terms of components and a Virtual Service Platform (VSP). The former are described by simulation and prediction models, whereas the latter is covered by a prediction model only. The simulation model supports the estimation of the values of signature parameters as an input for the prediction model. The APPEAR method aims at reusing the prediction model, fitted to the measurements from the existing components, to assess the performance of the adapted versions of these components[1]. This allows saving a lot of effort, since performance estimates for a new version of the component can be obtained by modifying only the simulation model (Figure 6.1).

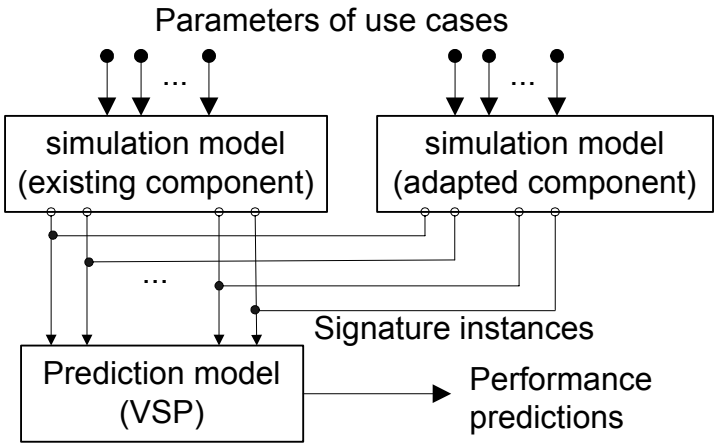


Figure 6.1: Performance prediction for adapted component

However, the accuracy and trustworthiness of such predictions is questionable for several reasons. The adapted component may have a different behavior; e.g., it may use the VSP in a different manner. As a result, the use of the prediction model for another component 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 additional means to judge the trustworthiness and accuracy of the estimates. These two properties of predictions can be represented in terms of prediction intervals, which provide a range of values where actual performance lies (see Section 5.11)

The APPEAR method can only be applied for predicting the performance of adapted components that are sufficiently "similar" to the existing ones. The purpose of this chapter is to elaborate on the notion of component similarity and to provide the architects with the means for judging when the APPEAR method can be successfully applied.

[1] As in the previous chapters, we consider that the term "performance of a component" refers to the performance-related metric of a particular component operation.

Answering the question whether the existing and adapted components are similar amounts to answering the question whether the simulation models of these components are similar with respect to the use of the same prediction model. Figure 6.2 illustrates the scope of the similarity problem in more detail.

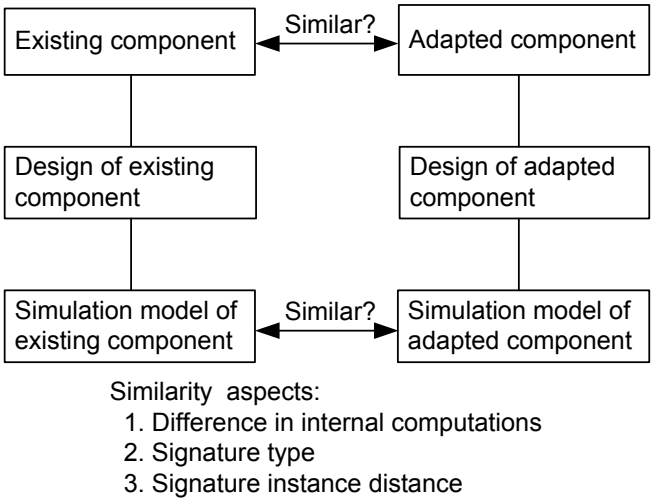


Figure 6.2: The scope of the similarity problem

The architect needs to decide whether the existing component and the adapted version thereof are similar. Both components have a particular design. These designs must reflect the use of services of the same VSP by both components so that it is possible to construct simulation models that show the use of these services. These simulation models are to be used to extract signature instances for supplying them to the input of the prediction model (see Sections 5.6.1 and 5.6.2). Already at this stage, the architect can decide that the components are dissimilar, if the adapted component uses completely different services than the existing component or if the services to be used by the adapted component are unknown. In this case, the architect has to apply performance prediction approaches other than the APPEAR method to obtain performance estimates for the adapted component (e.g., detailed simulation). In the rest of this chapter, we will consider only the case when the architect has succeeded in constructing the simulation model, for the adapted component, that can supply signature instances to the prediction model.

We use the following definition of component similarity:

$$Similar = IC \wedge ST \wedge SI \tag{6.1}$$

In Formula (6.1), *Similar* is a Boolean variable that indicates whether the adapted and existing components are similar or not. It is calculated by the conjunction of three Boolean variables corresponding to the following similarity conditions: *IC* (internal computations), *ST* (signature type), and *SI* (signature instances). The values of these variables are assigned using the rules described in Section 6.2.

The components are considered similar if all three conditions of component similarity are satisfied:

1. The internal computations (*IC*) of the adapted and existing component are identical.
2. The signature types (*ST*) of the adapted and existing component are identical.
3. The signature instances (*SI*) of the adapted and existing component are close.

For the first two conditions, we formulate some recommendations in Section 6.3. They help the architect to still be able to obtain predictions for the adapted component, when the conditions are not satisfied. Though, these predictions will have lower accuracy.

6.2 Similarity conditions

This section details the similarity conditions enumerated in Section 6.1.

6.2.1 Identical internal computations of adapted and existing component

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 computations that contribute to the overall performance. These internal computations and timing dependencies may differ for the adapted and existing component.

This problem can be addressed if an adequate model of the internal computations and timing delays are provided. These computations can be modeled either by analytical formulas or by simulation. The information about the processor demand and delays can be obtained from measurements or by budgeting.

The following factors are important when comparing the internal computations of the existing and adapted component:

- *The amount of internal computations*. The prediction model accounts not only for the computations performed in the VSP, but also for the internal computations of the existing components (see Chapter 5). The amounts of computations of both types must be alike for the existing and adapted components in order to obtain valid predictions.
- *The possible correlation between the amount of internal computations and signature instances*. Service call invocation may require a significant amount of internal computations. Thus, certain invocation patterns of the service calls may lead to the correlation between the amount of the internal computations and signature instances. It is therefore important to check that the existing and adapted components exhibit the same correlation.

The existing and adapted components must be alike with respect to both factors, which are detailed in the subsequent sections.

The information about the internal computations of the existing components can be extracted either from measurements or from the simulation model, whereas the same information for the adapted component can only be obtained from its simulation model. Section 6.4 describes and example of how the internal computations of the adapted component are modeled.

A) Comparison of the amounts of internal computations

The amount of internal computations of the existing component can be measured in terms of average values, by applying the following formula:

$$\overline{P} = \frac{1}{N}\sum_{j=1}^{N} P_j \ . \tag{6.2}$$

In Formula (6.2), $\overline{P}$ denotes the average amount of internal computations, and P_j is the amount of internal computations observed for the *j*-th use case.

Let $\overline{P}_{existing}$ and $\overline{P}_{adapted}$ denote the amount of the internal calculations of existing and adapted component, respectively. It is not correct to check if $\overline{P}_{existing}$ and $\overline{P}_{adapted}$ are equal or not, as both of them are, in fact, random variables calculated from observations. This means that any difference between $\overline{P}_{existing}$ and $\overline{P}_{adapted}$ has a statistical nature, i.e. it is observed by chance. In this case, it is valid to check if $\overline{P}_{existing}$ and $\overline{P}_{adapted}$ are statistically distinguishable. Figure 6.3 illustrates this phenomenon in more detail.

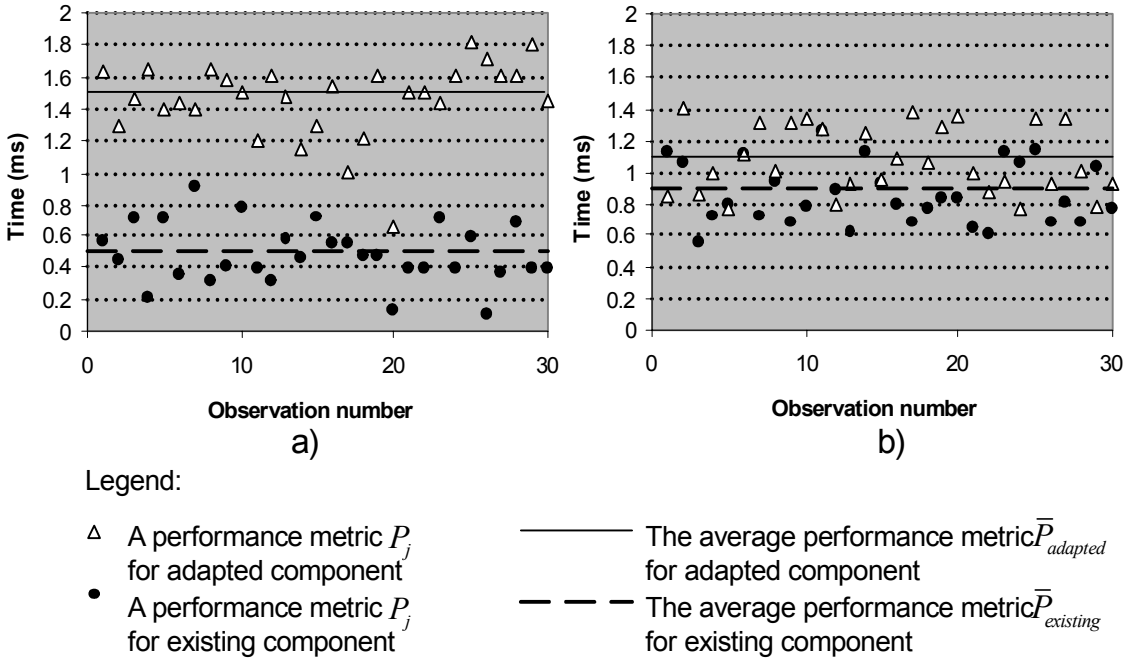


Figure 6.3: a) $\overline{P}_{existing}$ and $\overline{P}_{adapted}$ are statistically distinguishable, b) $\overline{P}_{existing}$ and $\overline{P}_{adapted}$ are statistically indistinguishable

In both cases, $\overline{P}_{existing}$ does not equal $\overline{P}_{adapted}$. Plot a) shows the case when $\overline{P}_{existing}$ equals $\overline{P}_{adapted}$ are statistically distinguishable, as the difference $\overline{P}_{adapted} - \overline{P}_{existing}$ is larger than the variations of corresponding performance metrics around the means $\overline{P}_{existing}$ and $\overline{P}_{adapted}$ for the existing and adapted components, respectively. Plot b) shows the opposite case, that is, the difference $\overline{P}_{adapted} - \overline{P}_{existing}$ is smaller than the two variations. This implies that $\overline{P}_{existing}$ and $\overline{P}_{adapted}$ will be statistically indistinguishable.

A proper approach to checking whether $\overline{P}_{existing}$ equals $\overline{P}_{adapted}$ or not is to check whether they are statistically indistinguishable or not (see Figure 6.3). The latter can be done by applying statistical hypothesis testing (e.g., the *z*-test or *t*-test [Wei95], [MR03]) or by constructing the confidence interval for $\overline{P}_{adapted} - \overline{P}_{existing}$ and then checking if this interval contains the value zero [Wei95], [MR03].

B) Comparison of the correlations between the amount of internal computations and signature instances

Certain internal computations may need to be performed whenever VSP services are invoked, with each VSP service corresponding to particular amount of internal computations. For instance, the existing component performs most internal computations when it calls a particular VSP service, e.g., a service *S1*. On the other hand, the adapted

component performs most of its internal computations, when it invokes another service, e.g., a service *S2*. These two services are described by two distinct signature parameters, the numbers of invocations of these service calls. The prediction model fitted to the measurements from the existing component will provide biased predictions for the adapted component, as the prediction model confuses the contribution of the *S2* service with the contribution of the *S1* service.

Thus, it is also necessary to compare the degree of correlation between the signature instances and the amount of internal computations for the existing and adapted components. These degrees of correlation have to be statistically indistinguishable to guarantee that the internal computations of the components are the same. In the general case, we have not found a solution to this problem.

There is however a solution for the case when the amount of internal computations does not correlate with the signature instances for both existing and adapted components. In this case, one can construct auxiliary regression models[2] (for both existing and adapted components) that relate the amount of internal computations to the signature instances. Both models must not provide significant regression, that is, all regression coefficients are not considered significantly different from zero. This can be tested by applying the F-tests for regression significance [JAI91], [Wei95], [MON01], [MR03], i.e. to test whether the internal computations of the existing and adapted components can be described only by their mean values. Finally, it is necessary to check if the residuals of the auxiliary prediction model satisfy the assumptions of the regression technique. The residuals should

- not have any structure,
- be normally distributed,
- have a constant standard deviation.

Provided that all these checks are passed and the amount of internal computations is the same (see section above), the existing and adapted components are concluded to be similar with respect to their internal computations.

6.2.2 Identical signature types of adapted and existing component

The existing and adapted components are similar only if their signature types are exactly the same, that is, their performance can be described by the same set of signature parameters. This is required as the prediction model can only input the same signature type that was used to fit the model. This signature type is derived from the existing component. If the signature types are different, the components are concluded to be dissimilar, and the recommendations from Section 6.3.2 might provide an "escape route".

6.2.3 Distance of the signature instances of adapted and existing component

As explained in Section 5.11, trustworthy predictions can only be obtained for the signature instances that are close to the ones used to fit the prediction model. It is therefore necessary to check the distance between the signature instances of the existing and the adapted components. For this purpose, a metric of the distance between signature instances is introduced. It characterizes the distance between a signature instance used for predicting the performance and the nearest *p* signature instances from the calibration data (see Figure 6.4).

[2] The model is constructed using linear regression tools, e.g. S-PLUS

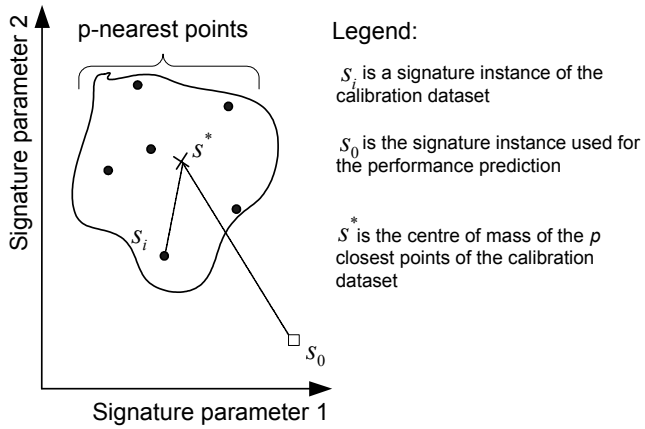


Figure 6.4: The local distance metric for signature instances

Let SA denote the set of indexes of all signature instances from the calibration data, and $SN = \{i_1, i_2, ..., i_{p-1}, i_p\} \subseteq SA$ denote the set of indexes of the p nearest signature instances. The following holds for these two sets:

$$\forall i \in SN, \forall j \in SA \setminus SN: \quad d(s_i, s_0) \le d(s_j, s_0). \tag{6.3}$$

In Formula (6.3), $d(s_l, s_m)$ is a weighted Euclidean distance between signature instances $s_l = (s_{l1}, s_{l2}, ..., s_{lk-1}, s_{lk})$ and $s_m = (s_{m1}, s_{m2}, ..., s_{mk-1}, s_{mk})$, each consisting of k signature parameters. The weighted Euclidean distance $d(s_i, s_j)$ can be calculated by

$$d(s_i, s_j) = \sqrt{\sum_{l=1}^{k} \frac{(s_{il} - s_{jl})^2}{\left(s_{\cdot l}^{(\max)} - s_{\cdot l}^{(\min)}\right)^2}}. \tag{6.4}$$

In Formula (6.4), $s_{\cdot l}^{(\max)}$ and $s_{\cdot l}^{(\min)}$ denote, respectively, the maximum and the minimum values of the l-th element of all signature instances used to fit the prediction model. The global scaling introduced by these two variables is needed to make a distance metric equally sensitive to all signature parameters. Otherwise, the distance metric would be dominated by the signature parameters that have larger values.

Let us also introduce the notion the center s^* of the nearest points of all signature instances from SN. This center is a reference point that allows determining the distance between the point s_0 (where the prediction is needed) and the p nearest points. The distance metric is determined by relating the distances between the points from SN and s^* to the distance between s_0 and s^*. The value of the center s^* of nearest points can be calculated by the following formulas:

$$s^* = (s_1^*, s_2^*, ..., s_k^*), \tag{6.5}$$

$$s_j^* = \frac{1}{p} \sum_{i \in SN} s_{ij}, \quad j \in 1..k. \tag{6.6}$$

Finally, the local distance metric L (L-metric) can be calculated as follows:

$$L = \frac{d(s_0, s^*)}{\max_{i \in \mathrm{SN}} \left(d(s_i, s^*)\right)} \tag{6.7}$$

In Formula (6.7), s_0 denotes the signature instance that needs predicting the performance, s^* is the center of nearest points of all signature instances from SN.

The larger the *L*-metric is, the further the point is located from the calibration data, and the less likely it is to obtain valid predictions. A more detailed explanation for this phenomenon can be found in Section 5.11.

The L metric has values smaller than one, if the signature instance used for the performance prediction is located in the neighborhood of the *p*-nearest points. In this case, the prediction model can be safely used to estimate the performance, as it is used for the signature instance from the calibration data [Wei95], [MON01], [MR03], [JAI91].

Otherwise, the architect has to carefully analyze the validity of the prediction: there should be a sufficient evidence of the applicability of the prediction model beyond the area that the model has been fitted in. If such guarantees cannot be provided, the architect may rely on the predictions only if the impact of the wrong decision, made on the basis on these predictions, is low. In this case, the architect is recommended either to apply other performance prediction methods, or to negotiate the risks due to the wrong decision with the stakeholders.

Please notice that the behavior of the *L*-metric and, consequently, its reliability depends on the chosen value of *p*, the number of closest points considered. Particularly, the value of *p* should be not less than the number of the signature parameters, as the latter is the minimum number of observations required to build a unique prediction model. On the other hand, choosing a significantly larger value of *p* may result in a failure of the *L*-metric to capture the distance between a signature instance and the calibration data. For instance, Figure 6.5 demonstrates the case when the calibration data space has a complex shape. Predictions for the signature instances lying equally far from both parts of the space may be incorrect, if the value of *p* equals to the number of signature instances in the calibration data. In addition, cluster analysis techniques [JAI91] can be employed for choosing the value of *p*.

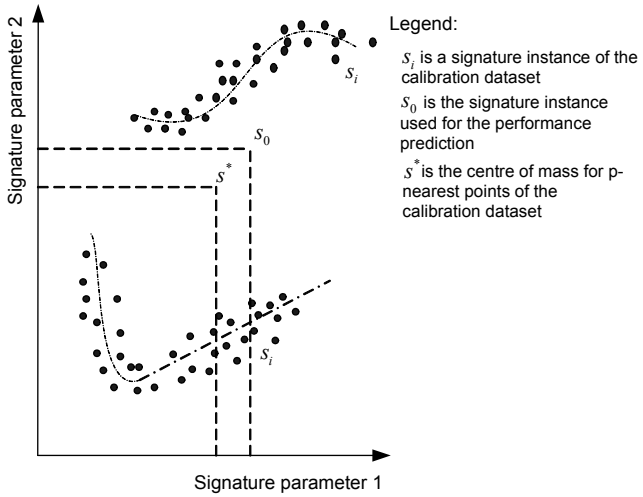


Figure 6.5: The complex-shaped signature space used for the calibration of a prediction model

6.3 *"Escape routes"*

This section describes a couple of recommendations for tackling particular types of component dissimilarity. These recommendations are described in the subsequent subsections.

6.3.1 Difference in the internal computations of components

In the general case, difference in the internal computations of the existing and adapted components makes components dissimilar. Despite this complication, the performance estimates for the adapted component can still be obtained in some cases. Addressing this type of dissimilarity requires that performance contributions of the VSP and component be treated separately for both existing and adapted components. In this case, the prediction model reflects the VSP contribution, whereas the simulation model describes the component contribution to the performance.

For constructing such a prediction model, it is sufficient to use the measurements of the VSP contribution only instead of the measurements for a complete use case. The code needs to be instrumented to support such measurements.

The simulation model must explicitly specify the internal computations of components in terms of resource demands and timing delays[3]. For the existing component, these values can be extracted from the measurements, whereas for the adapted component they can be budgeted.

Both simulation and prediction models yield performance contributions (in terms of performance metrics, e.g. execution time) (see Figure 6.6).

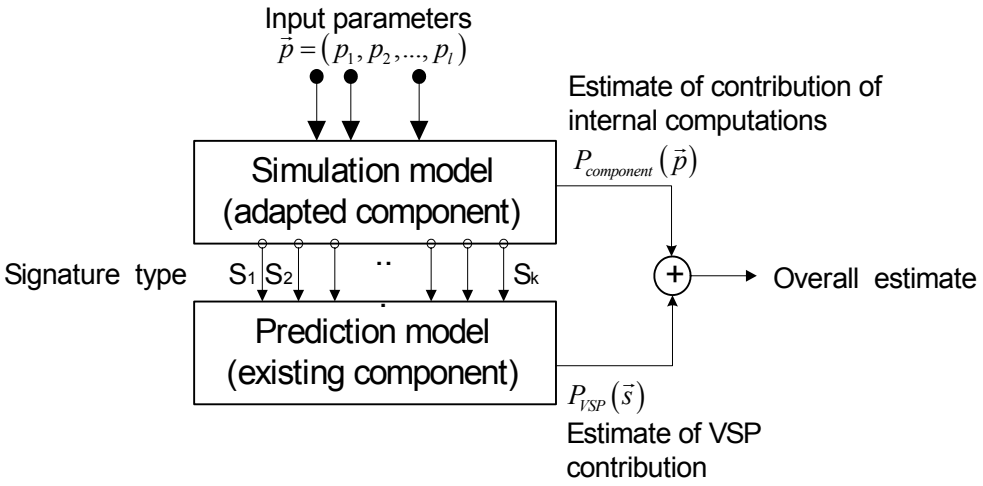


Figure 6.6: Separate performance contributions of component and VSP

In Figure 6.6, $\vec{s} = (s_1, s_2, ..., s_k)$ denotes a signature instance. The remaining variables are defined in the figure.

Using the calibration phase of the APPEAR method (see Section 5.6.1), the prediction model for the contribution of the VSP to performance is fitted. To make a prediction for a certain use case parameterized by a vector $\vec{p} = (p_1, p_2, ..., p_k)$, both VSP and component contributions need to be estimated. The former is determined by calculating the signature

[3] Note that approach differs from software prototyping, as no line of code is actually implemented, but only a high-level model where certain time-consuming operations are treated explicitly is constructed.

instance $\vec{s}$ and applying the prediction model $P_{vsp}(\vec{s})$ to it. The latter has to be calculated by the simulation model. Finally, the two estimates need to be summed up to obtain the overall estimate for the chosen use case.

In this way, the performance of the adapted component can be predicted with certain accuracy, even if this component has a significantly different amount of internal computations than the existing one. This accuracy depends on (1) the accuracy provided by the prediction model and (2) the accuracy of modeling the internal computations by the simulation model of the adapted component. The former can be described by means of prediction intervals (see Section 5.11). The latter has to be specified by the architect, preferably in the form of the distribution of expected prediction error for the internal calculations. For instance, the architect can use measurements from a prototype of the new functionality or construct a number of benchmarks to obtain figures about the internal computations of the existing component. In Section 6.4, we described an example of applying the proposed escape route and constructing the total prediction intervals for the performance of the adapted component. In the general case, the architect can expect that the accuracy of modeling the internal calculations has a major impact on the overall prediction accuracy, as the actual implementation of the adapted component may have a completely different amount of the internal calculations than the one considered within the simulation model.

6.3.2 Difference in the signature types of the adapted and existing component

The prediction model is calibrated based on a particular signature type, obtained for the existing component. As soon as the signature type of the adapted component differs from the signature type of the existing one, the use of the existing prediction model is no longer possible, without modifying the APPEAR method. However, even if this problem occurs, there can still be solutions that allow the prediction of the performance of an adapted component.

Three cases of differences in the signature types are distinguished:

- Signature type of an adapted component is wider than that of the existing one:

$$\begin{aligned} &S = \{S_1, ..., S_k\} - signature\ of\ existing\ component \\ &S' = \{S_1, ..., S_k, ..., S_n\},\ n > k - signature\ of\ adapted\ component \end{aligned} \tag{6.8}$$

- Signature type of an adapted component is narrower than that of the existing one:

$$\begin{aligned} &S = \{S_1, ..., S_n, ..., S_k\} - signature\ of\ existing\ component \\ &S' = \{S_1, ..., S_n\},\ n < k - signature\ of\ adapted\ component \end{aligned} \tag{6.9}$$

- Signature types of the existing and adapted component intersect:

$$\begin{aligned} &S = \{S_p, ..., S_n, ..., S_k\} - signature\ of\ existing\ component \\ &S' = \{S_1, ..., S_p, ..., S_n\},\ p < n < k - signature\ of\ adapted\ component \end{aligned} \tag{6.10}$$

The subsequent sections will briefly describe each case.

A) Wider signature type

An adapted component can have a wider signature type for the following three reasons (see Figure 6.7):

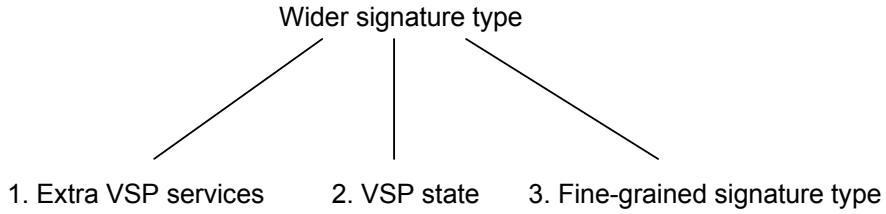


Figure 6.7: Different reasons for wider signature type

1. The adapted component uses more VSP services than the existing one,
2. The adapted component uses the VSP in a different mode or invokes service calls with a different pattern; i.e., the state of the VSP becomes influences the adapted component,
3. The adapted component invokes VSP services with different parameter values, i.e., the signature type is more fine-grained.

The second and the third cases relate to the different patterns of the use of VSP services by existing and adapted components. These different patterns ultimately result in different signature types. Two different patterns of use can be discerned: a) different call frequency and b) different call sequence. The first pattern does not have impact on signature type, as it results only in changing signature instances. The second pattern often relates to the relevance of the VSP state.

Case 1: extra VSP services. The adapted component may use more services of the VSP than the existing one does (see Figure 6.8).

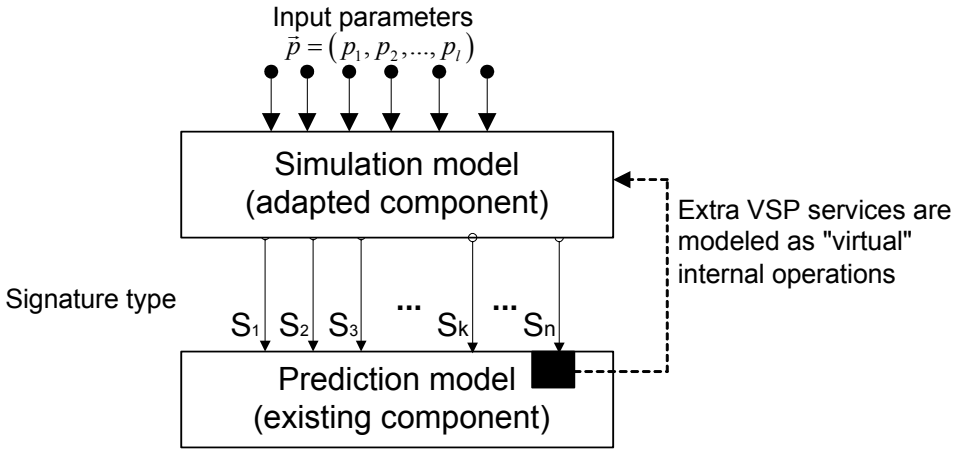


Figure 6.8: Wider signature type as a result of usage of extra VSP services

In this case, the performance contribution related to the additional signature parameters implies additional computations. These computations cannot be captured by the prediction model, as measurements for the adapted component are not available. Therefore, we propose treating these computations as internal computations of the adapted component, and represent them as "virtual" internal operations in the simulation model. The contribution related to these extra signature parameters is estimated or budgeted in advance and incorporated to the simulation model in terms of resource demands and timing delays. For the rest of the signature parameters, the existing prediction model can be used. This approach allows reducing the problem of extra VSP services to the problem of difference in internal computations, which can be solved in a manner as described in Section 6.3.1.

Case 2: VSP state. The VSP state can have a significant influence on performance, if the invocation patterns of service calls are different for the adapted and existing component (see Figure 6.9). The effect of extra VSP state(s) can be treated in the following way.

These states can be considered just as additional signature parameters. As a result, extra VSP state(s) can also be represented as “virtual” internal operations in the simulation model (see the case above). This approach allows using the solution described in section 6.3.1.

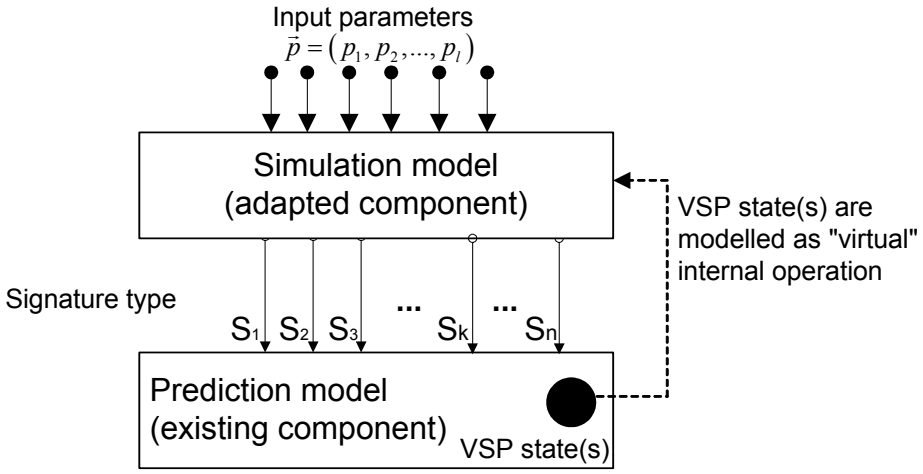


Figure 6.9: Handling a wider signature type as a result of the relevance of the VSP state(s)

Case 3: fine-grained signature type. The signature type may need to be refined, that is, certain signature parameters have to be substituted with groups of parameters. For example, additional parameter ranges of a service call have become relevant for the adapted component, whereas they had been irrelevant for the existing component. For the adapted component, the duration of this call may vary depending on the values of particular input parameters. On the other hand, no variation of the duration of this call may have been observed for the existing component. As a result, a single signature parameter cannot capture this service call any longer. Therefore, additional signature parameters must be introduced (see Figure 6.10). This leads to the same situation as in the previous case where extra VSP services are modeled as “virtual” internal operations.

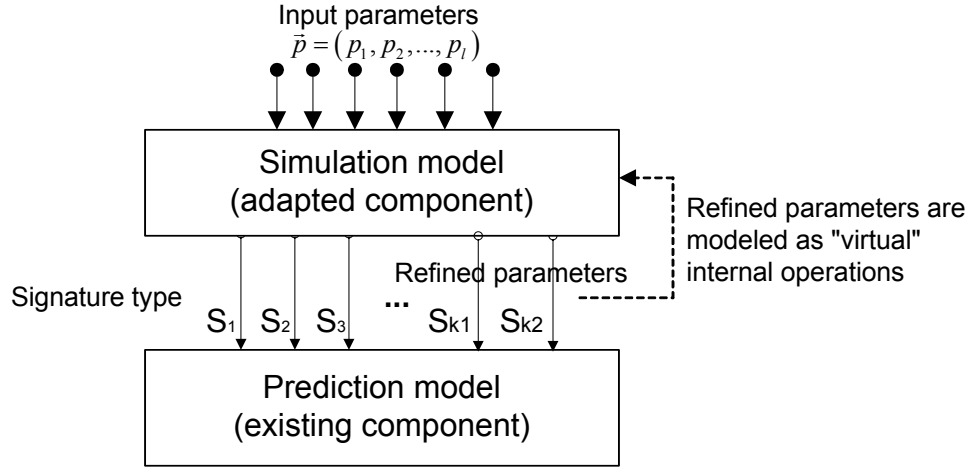


Figure 6.10: Handling a wider signature type as a result of the refinement of coarse-grained signature

B) Narrower signature type

It may be the case that the performance can be described for the existing component by fewer signature parameters than for the existing one. For example, the adapted component can use fewer service calls of the VSP (e.g., parameter S1 is not used by the adapted component in Figure 6.11).

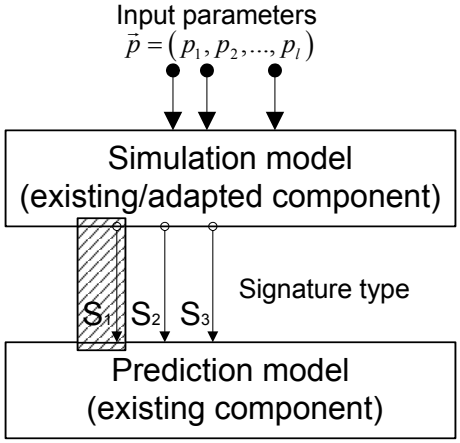


Figure 6.11: Narrower signature type

The performance prediction model for the existing component is originally fitted over all signature parameters. It has the following form for the case of linear regression:

$$P = \beta_0 + \sum_{i=1}^{n} \beta_i \cdot S_i. \tag{6.11}$$

In this formula, P denotes the performance metric, β_i are linear regression coefficients (β_0 is an intercept), and S_i are the signature parameters.

In principle, the prediction model can input only the same number of signature parameters that it has been fitted over. To be able to continue using this prediction model, it is necessary to guess the values of the signature parameters that are not necessary for the adapted component. Please notice that this guessing may require a lot of effort. When it is impossible or too costly to guess these missing values, we recommend refitting the prediction model to the performance measurements from the existing component, but using only the signature parameters that are relevant for the adapted component. Note that this approach can increase the prediction error.

C) Intersecting signature types

Finally, the case of signature type intersection is considered. Intersection of signature types means that the signature type of the adapted component includes only some signature parameters of the existing one, and, in addition, it includes a number of extra ones, not employed in the signature of the existing component.

For example, the adapted component can use fewer service calls of the VSP than existing one, and use a new service of the VSP (e.g., parameter S1 is not used by the adapted component, whereas parameter S4 is added in Figure 6.12). The solution in this case is the combination of the situations described the “Wider signature type” and “Narrower signature type” sections. Extra signature parameters can also be represented as “virtual” internal operations in the simulation model (see the cases above), and the approach described in Section 6.3.1 is applied. The problem of the unnecessary signature parameters is addressed by either guessing their values or refitting the prediction model, as described previously.

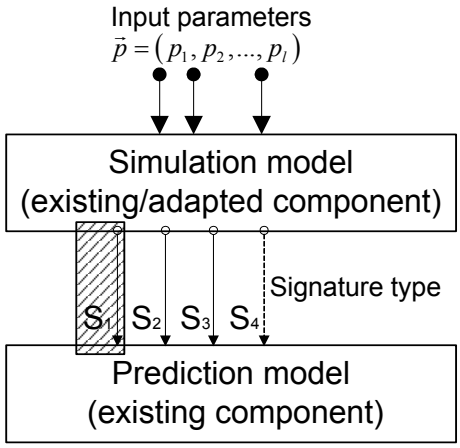


Figure 6.12: Intersecting signature types

6.4 Example of ascertaining the similarity of components

We have chosen the Medical Imaging Software Stack (MISS), described in Chapter 8 of this thesis to exemplify ascertaining the similarity of the existing and adapted components. Particularly, future versions of the "*Reviewing*" component (see Section 8.2) needed to be extended with a "*ViewSubtractedTrace*" function. This function merges a series of images of the current run[4] into a single composite image. This composite image contains a view of the entire vascular tree structure. Before merging, each image undergoes the subtraction procedure: the mask is subtracted from the image to remove irrelevant pixels in order to improve the visualization of the vascular tree structure. Note that this subtraction procedure involves CPU-intensive computations. The image subtraction procedure could not be implemented in the existing dedicated hardware of the MISS. Therefore, it was decided to implement it in software.

The architects would like to evaluate the performance of the "*ViewSubtractedTrace*" in advance, in order to verify whether these estimates match the requirements specification. The use case for predicting the response time of the adapted "*Reviewing*" component is the execution of the "*ViewSubtractedTrace*" function for various numbers of images, e.g. 8, 16, 24, and 32.

6.4.1 Similarity conditions

The similarity conditions with respect to this example are as follows:

- The amount of internal computations is different for the existing "*Reviewing*" component and the adapted version thereof. Moreover, the performance contribution of the internal computations is negligible (compared to the contribution of the VSP) for the existing component, whereas this is not true for the adapted one. This implies that we cannot use the prediction model fitted on the "*Reviewing*" component as is. Instead, we must take the "escape route" described in Section 6.3.1.
- Both the adapted and existing components have the same signature type. No additional VSP services are required to implement the "*ViewSubtractedTrace*". The image subtraction procedure is implemented within the adapted "*Reviewing*" component and is accounted for as the internal computations of this component (see the bullet above).

[4] A "run" is a series of a medical images recorded during a single acquisition procedure.

- Various distances between the signature distances of the existing and adapted component are considered. Section 8.4.3 shows that the *L*-metric (see Section 6.2.3) varies in the interval 0 to 4.96 for the number of images in the interval 8 to 32. Signature instances for which the *L*-metric is larger than 1.0 do not allow to obtain reliable predictions.

The example demonstrates how to deal with these similarity conditions to provide a trustworthy performance estimate for the adapted component in terms of prediction intervals.

6.4.2 Simulation model of the adapted "Reviewing" component

We modify the simulation model of existing component (see Section 8.3.5) to account for the introduction of the "*ViewSubtractedTrace*" function. Figure 6.13 shows the state-machine that describes the behavior of the obtained simulation model of the adapted "*Reviewing*" component. The original state-machine, described in Section 8.3.5, needs added the "*ViewSubtractedTrace* $(+T_{sub})$" transition, which models the extra functionality.

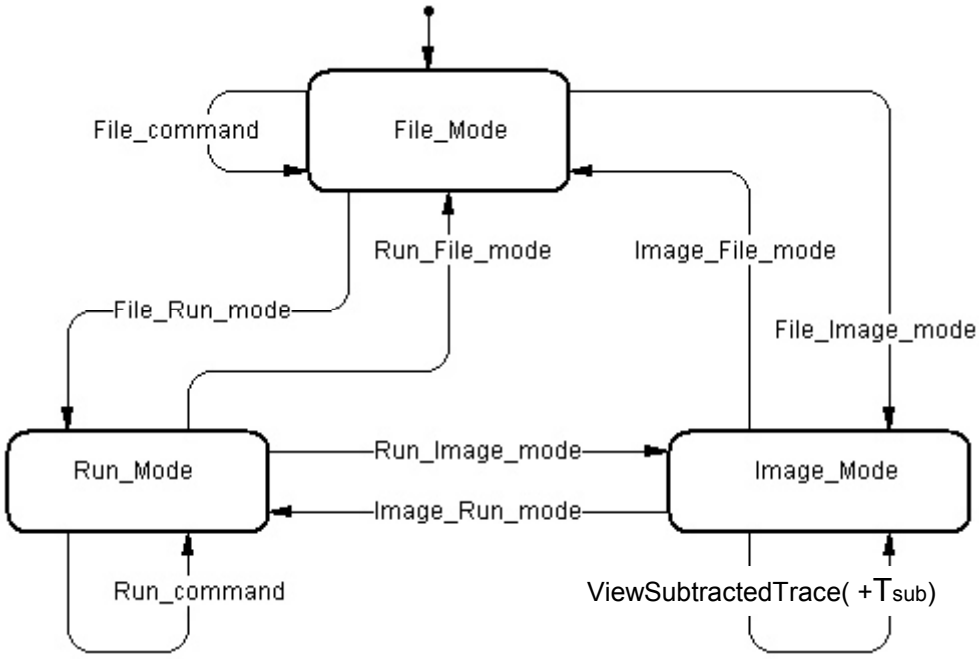


Figure 6.13: Simulation model of adapted component

The time consumed by the subtraction procedure is significant and can vary, depending on the number of images. Therefore, the simulation model of the existing "*Reviewing*" component needs to be extended to account for this new function (see the "*ViewSubtractedTrace* $(+T_{sub})$" transition in Figure 6.13). The simulation model of the adapted "*Reviewing*" component now calculates not only signature instances but also the total time T_{sub} spent in the image subtraction procedure. This total time T_{sub} can be calculated by the following formula:

$$T_{sub}(\#Images) = \sum_{i=1}^{\#Images} t_i. \qquad (6.12)$$

In Formula (6.12), $\#Images$ is the number of images, and t_i is the time required for the subtraction procedure for a single image. The time t_i is measured in advance and equals

100±15 ms (i.e., the standard deviation $\sigma = 15$). We consider that the measurements have no systematic errors, but only a normally distributed error with the standard deviation σ. Additionally, each execution of this procedure is independent from other executions.

6.4.3 Prediction of the response time of the "ViewSubstracted-Trace" function

As explained in Section 6.4.1, we cannot use the prediction model fitted on the existing "*Reviewing*" component as is. We have also to account for the significant internal calculations of the adapted "*Reviewing*" component, i.e., for the image subtraction procedure. As the internal calculations of the existing "*Reviewing*" component (see Chapter 8) were negligible, we can predict the performance of the "*ViewSubstractedTrace*" function by summing up the estimate made by means of the prediction model and the estimate obtained from the simulation model of the adapted "*Reviewing*" component (see Section 6.4.2). The first estimate P_{pred} accounts for the contribution of the VSP, whereas the second estimate T_{sub} models the total contribution of the image subtraction procedure. The response time P of the "*ViewSubstractedTrace*" function can be calculated by

$$\begin{aligned} &P\left(\#Images, \#ShortUpdate, \#LongUpdate, \#Paint\right) = \\ &\quad P_{pred}\left(\#Images, \#ShortUpdate, \#LongUpdate, \#Paint\right) + \\ &\quad T_{sub}\left(\#Images\right). \end{aligned} \tag{6.13}$$

In Formula (6.13), all arguments starting with the '#' symbol are signature parameters, which are detailed in Section 8.3.4. For conciseness, we will omit these arguments when writing P_{pred} and T_{sub}.

We aim at constructing the prediction interval for the response time P at the confidence level α:

$$PL(\alpha) \le P \le PU(\alpha). \tag{6.14}$$

In Formula (6.14), PL and PU are the lower and upper bounds of the prediction interval for $P = P_{pred} + T_{sub}$.

The calculation of the prediction interval is based on the knowledge about the statistical distribution of predictions. In our case it is possible to use the normal distribution because of the following reasons:

- The degrees of freedom for the prediction model equal 43 (see Appendix F). It is therefore possible to approximate the t-distribution, usually used for constructing the confidence intervals in linear regression, by the Normal distribution, as the degrees of freedom are larger then 30 [JAI91], [MR03].
- The errors of the prediction and simulation model are independent of each other.

Those two arguments allowed us to apply the theorem stating that the sum of a number of independent normally distributed random variables is also distributed normally [Wei95]. Thus, the formulas for calculating the bounds *PL* and *PU* of the prediction interval differ from the ones given in Section 5.11. These formulas are as follows:

$$PL(\alpha) = P_{pred} + T_{sub} - q_{z,1-\alpha/2} \cdot \hat{\sigma}_p, \tag{6.15}$$

$$PU(\alpha) = P_{pred} + T_{sub} + q_{z,1-\alpha/2} \cdot \hat{\sigma}_p. \tag{6.16}$$

In Formulas (6.15) and (6.16), $q_{z,1-\alpha/2}$ denotes the $1-\alpha/2$-th quantile[5] of the normal distribution. $\hat{\sigma}_p$ is the total standard deviation of the prediction Because of the normality and independence of errors from the simulation and prediction error, we can apply the rule for calculating the standard deviation of the sum of normally independently distributed random variables to obtain $\hat{\sigma}_p$:

$$\hat{\sigma}_p = \sqrt{\hat{\sigma}_{sim}^2 + \hat{\sigma}_{pred}^2}. \tag{6.17}$$

In Formula (6.17), $\hat{\sigma}_{sim}^2$ is the variance of the error provided by simulation, whereas $\hat{\sigma}_{pred}^2$ is the variance of the prediction model error. Notice that both $\hat{\sigma}_{sim}^2$ and $\hat{\sigma}_{pred}^2$ are functions of signature parameters.

The simulation model error is the total measurement error for the subtraction times t_i for *#Images* images. , This total variance $\hat{\sigma}_{sim}^2$ equals the sum of variances (each variance σ^2) obtained during the measurement:

$$\hat{\sigma}_{sim}^2\left(\#Images\right) = \#Images \cdot \sigma^2. \tag{6.18}$$

$\hat{\sigma}_{pred}^2$ is the estimate of the variance of the prediction model error. It is calculated by Formula (5.9) from Section 5.11:

$$\hat{\sigma}_{pred}\left(\#Images, \#ShortUpdate, \#LongUpdate, \#Paint\right) = \hat{\sigma}\sqrt{1+\vec{s}^T\left(S^T S\right)^{-1}\vec{s}}. \tag{6.19}$$

In Formula (6.19), $\hat{\sigma}$ denotes the standard error of fit. S is the matrix describing the calibration data (see Section 5.11), and $\vec{s}$ is a signature instance for which the prediction is needed. In this example, $\vec{s}$ has the following structure[6]:

$$\vec{s} = \begin{pmatrix} 1 \\ \#Images \\ \#ShortUpdate \\ \#LongUpdate \\ \#Paint \end{pmatrix}. \tag{6.20}$$

The measurements described is Section 6.4.2 and Formula (6.12) allow us to describe the total contribution of the subtraction procedure by

$$T_{sub}\left(\#Images\right) = \#Images \cdot \overline{t}. \tag{6.21}$$

In Formula (6.21), $\overline{t}$ denotes the average execution time of the image subtraction procedure ($\overline{t} = 100\ ms$).

By applying Formulas (6.17), (6.18), and (6.21), Formulas (6.15)-(6.16) give the following equation for the limits of the prediction interval:

[5] Note that we used here the quantiles of the normal (Gaussian) distribution instead of the *t*-distribution. The reason for that is that we approximate the t-distribution by the normal distribution. (This *t*-distribution is typically used in the context of constructing confidence and prediction intervals.)

[6] The first element of $\vec{s}$ equals so that the prediction model can have not only terms related to signature parameters but also the constant term (intercept).

$$PL(\#Images, \#ShortUpdate, \#LongUpdate, \#Paint, \alpha) = \#Images \cdot \bar{t} + P_{pred}(\#Images, \#ShortUpdate, \#LongUpdate, \#Paint) - q_{z,1-\alpha/2} \cdot \sqrt{\#Images \cdot \sigma^2 + \left(\hat{\sigma}_{pred}(\#Images, \#ShortUpdate, \#LongUpdate, \#Paint)\right)^2}, \quad (6.22)$$

$$PU(\#Images, \#ShortUpdate, \#LongUpdate, \#Paint, \alpha) = \#Images \cdot \bar{t} + P_{pred}(\#Images, \#ShortUpdate, \#LongUpdate, \#Paint) + q_{z,1-\alpha/2} \cdot \sqrt{\#Images \cdot \sigma^2 + \left(\hat{\sigma}_{pred}(\#Images, \#ShortUpdate, \#LongUpdate, \#Paint)\right)^2}. \quad (6.23)$$

After executing the simulation model and employing the prediction model, we applied the Formula (6.22)-(6.23) to the signature instances. The values of constants $\bar{t} = 100\,ms$ and $\sigma = 15\,ms$ are taken from Section 6.4.2. The chosen confidence level $\alpha = 0.95$, and the corresponding quantile of the Normal distribution $q_{z,1-\alpha/2} \approx 1.96$ (the values of the quantiles of the Normal distribution are tabulated). Finally, we used a linear regression tool to calculate the values of $P_{pred}(\#Images, \#ShortUpdate, \#LongUpdate, \#Paint)$ and $\sigma_{pred}(\#Images, \#ShortUpdate, \#LongUpdate, \#Paint)$ for the prediction model constructed in Chapter 8. Table 6.1 summarizes the obtained results.

Table 6.1: Predictions obtained by means of the simulation and prediction models

#Images	#Long Upadte	#Short Update	#Paint	P_{pred}, ms	T_{sub}, ms	σ_{pred}, ms	Lower pred. bound	Upper pred. bound
8	0	8	1	843.5	800	94.8	1485.5	1801.3
16	0	8	1	1098.4	1600	112.4	2518.6	2878.3
24	0	8	1	1353.6	2400	125.9	3549.8	3957.4
32	0	8	1	1608.6	3200	137.4	4579.5	5037.9

The four left columns contain the signature instances generated by the simulation model which were supplied to input of the prediction model. The next column P_{pred} enumerates the predictions obtained by applying the prediction model to the signature instances. These predictions model the contribution of VSP to the overall response time. The T_{sub} column shows the estimates for the internal computations of the adapted component. The σ_{pred} column lists the standard error of the predictions obtained by the prediction model. Finally, the two right columns enumerated the upper and lower bounds of the performance prediction intervals for the adapted component.

Notice that the *L*-metric calculated for the last two rows (see Table 8.2 in Section 8.4.3) is greater than 1.0. This means that the use of these predictions is risky. For the first two rows, the *L*-metric is smaller than 1.0, and the corresponding predictions are safe to use.

In this example, we demonstrated the use of similarity conditions described in Section 6.2. We showed that the existing "*Reviewing*" component and the adapted version thereof were dissimilar because of extra internal computations performed by the latter. Moreover, we showed that for particular signature instances, making performance predictions for the adapted component would be risky anyway, as the *L*-metric was greater than 1.0 for these signature instances (see Section 6.2.3). For those signature instances that had *L*-metric smaller than 1.0, we showed the use of one of the "escape routes" (see Section 6.3.1) to obtain performance predictions for the adapted "*Reviewing*" component.

6.5 Summary

This chapter elaborates on the various forms of component similarity. Components are considered similar if all three conditions of component similarity are satisfied such that

- internal computations of the adapted and existing component are identical;
- signature types of the adapted and existing component are identical;
- distance metric of signature instances has a value smaller than one.

For each condition, a number of criteria for judging the similarity of the adapted and existing component are given. These criteria should be satisfied to ensure that the components are similar and that the prediction model (fitted to the measurements of the existing component) can be safely used to predict the performance of the adapted component.

Existing and adapted component may not always meet the aforementioned criteria. In this case, the prediction model fitted for the existing component cannot be directly used for the performance estimation. However, we recommended a few approaches to make the performance estimation possible (see Section 6.3). These approaches are based on separation of the component and VSP contributions to the performance. This separation presumes that the simulation model of the component describes the performance-relevant behavior explicitly (in terms of timing delays and resource needs). The simulation model is executed in order to get the performance contribution of the adapted component. The overall performance estimates are obtained by adding this contribution to the VSP contribution, given by the prediction model.

Our approach for tackling the problem of component similarity was demonstrated by a simple example, where the performance of the modified version of the "*Reviewing*" component (see Chapter 8) was estimated, and the corresponding prediction intervals were constructed.

7 Application of the APPEAR Method in the Consumer Electronics Domain

In this chapter, we describe the application of the APPEAR method (see Chapter 5) to an industrial software stack. Both calibration and prediction phases of the method are demonstrated by the example of a TV Teletext decoder. The goals of this experiment are the following:

1. Checking the APPEAR method in an industrial context;
2. Predicting the average execution time needed for Teletext information acquisition;
3. Providing the architects with insight about the performance-relevant parameters and behavioral aspects of the Teletext decoder.

There are several presentation levels of Teletext data—1, 1.5, 2.5, and 3.5— that determine possible enhancements to a Teletext page. All these levels are forward- and backward compatible. For instance, a level 1.0 Teletext decoder can display 3.5 level pages (but without additional enhancement information) and vice versa. Each Teletext presentation level subsumes all the lower presentation levels. A Teletext decoder can also support two types of navigation systems: (1) First Level One Facilities (FLOF) and (2) Table of Pages (TOP). Both navigation systems use the hypertext representation of data, but the way in which the Teletext decoder transmits navigation data is different for these navigation systems.

We used a Teletext decoder that supports the presentation level 1.5 and Full Level One Facilities (FLOF) navigation system to calibrate the prediction model and to construct the simulation model. Based on these models, we predicted the average execution time, needed for acquiring and decoding the Teletext information, of a Teletext decoder that supports the presentation level 2.5 and both FLOF and Table of Pages (TOP) navigation systems. For the sake of brevity, the original decoder is referred to as the Teletext 1.5 decoder, and the modified one is called the Teletext 2.5 decoder. The figures about the accuracy of the predictions are given in comparison with the measurements collected on the implementation of the Teletext 2.5 decoder.

This chapter is structured as follows. First, a brief introduction to Teletext is given. Afterwards, the application of the calibration phase of the APPEAR method is described for the Teletext 1.5 decoder, followed by the description of the prediction phase. Finally, the discussion of the results concludes this chapter.

7.1 Overview of Teletext

This section introduces the basic notions of the Teletext and outlines the architecture of the Teletext decoder that was used for this case study.

7.1.1 Teletext broadcasting

Each Teletext transmission [Txt02] comprises packets that together can form pages. Each page has a three-digit hexadecimal number in the range 100-8FF. Pages with decimal numbers (e.g., x00-x99) need displaying, whereas the rest serve special purposes. All pages are organized in eight magazines, where the number of a magazine is the most significant digit of the number of all pages that belong to this magazine. The pages can also have sub-pages that are distinguished by sub-codes.

Some packets are not directly related to a particular page, but rather to a magazine or a broadcast service. These packets have the following numbers:

- Page header packets (packet number 0; these packets indicate the beginning of a new page and fill gaps),
- Directly displayable page data packets[1] (packet number 1-23, 24 and 25),
- Non-displayable page data packets (packet number 26-28),
- Magazine enhancement data packets (29),
- General-Purpose and Broadcast Service data packets (30, 31).

Packets with numbers greater than 24 are additionally discerned by designation codes, numbers in a range 0-15. Depending on the designation code, the function of a particular packet may change.

The transmission order of Teletext packets is determined by the transmission mode: either (1) *serial* or (2) *parallel*. In the *serial mode*, pages are transmitted sequentially. First, a header packet is transmitted to open a new page, i.e., the header packet indicates that the Teletext receiver should be ready to receive page data packets. Then all data packets of this page are transmitted. Note, that these packets should carry the same magazine number as the header packet of the new page. Finally, a next header packet closes the page and opens a new one, if this page header packet has a different page number.

In the *parallel mode*, up to eight pages can be transmitted simultaneously (one page per magazine). All page data packets with the same magazine number belong to the page that has this magazine number. The currently transmitted page is closed and a new page is opened upon the arrival of the next header packet with the same magazine number, but with a different page number. Notice that the header packets that have the same page number and sub-code as the currently opened page do not open a new page both in serial and parallel transmission mode. Such header packets serve special purposes such as updating page flags, implementing rolling headers, and etc [Txt02].

Teletext packets are transmitted during vertical blanking intervals (VBI) both for odd and even fields (see Figure 7.1).

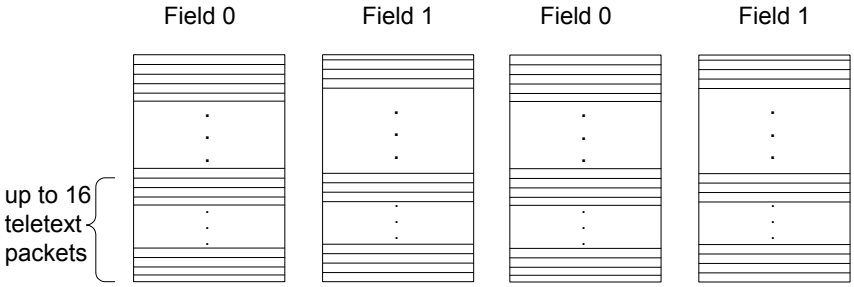


Figure 7.1: Teletext transmission

A field corresponds to half an image to be displayed and contains either odd or even lines. In each field, up to sixteen packets can be transmitted. A typical broadcaster transmits 11-14 packets per field.

Depending on the presentation level (1.0, 1.5, 2.0, or 3.5), Teletext pages are not only distinguished by numbers and sub-codes, but also by type: custom data pages, normal displayable pages, MOT pages, MIP pages, etc. For each page and packet type, page data packets (packet numbers 1-28) can be encoded using a combination of error correction codes:

[1] Such packet contains symbols that must be displayed in the row with the number that equals the packet number. For instance, the content of a packet with number 1 is displayed in the thirst row.

- The odd parity;
- The hamming 8/4 forward error correction (FEC);
- The hamming 24/18 FEC.

As mentioned above, there are two navigation systems: FLOF and TOP. The FLOF navigation requires editorial links be provided for a page. These links are transmitted in packets with number 27 and designation codes 0-3 and allow navigation to the corresponding pages. The TOP navigation uses a dedicated page that relates different topics to dedicated pages. The user can select a topic via menu, and the corresponding page will be jumped to. This navigation system is constructed using the dedicated pages that have predefined page numbers.

7.1.2 The Teletext decoder structure and behavior

The simplified structure of the Teletext decoder is sketched in Figure 7.2.

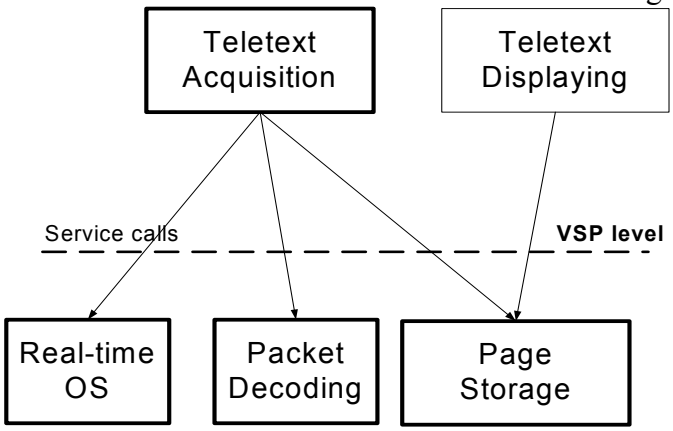


Figure 7.2: Structure of the Teletext decoder

The arrows in Figure 7.2 depict a 'uses' relationship. The dashed line corresponds to the level of service calls. The bold rectangles denote components that are relevant for the performance analysis of Teletext Acquisition component. The *Teletext Acquisition* component is a part of Teletext decoder that builds upon the virtual service platform (VSP) formed by the following components: the Real-time OS, Packet Decoding, and Page Storage.

The VSP provides the following service calls to the *Teletext Acquisition* component:

- *ClosePageWrite*, which notify the Page Store component that a currently transmitted page is complete, i.e., all its packets are received, and the packets can be moved from the local cache to the global one;
- *OpenPageWrite,* which indicates that the Page Store component needs to allocate internal structures and initialize internal cache for receiving a new page;
- *AllocatePacketMemory,* which makes the Page Store component allocate memory for new packet;
- *DecodeNone,* which makes the Packet Decoding component update the content of packet data with the data from a newly received packet;
- *DecodeOddParity,* which makes the Packet Decoding component update the content of packet data with the data from a newly received packet. Each byte of the packet is checked for errors, and it substitutes the old byte, if the new one does not have any error;
- *DecodeHamming8_4,* which makes the Packet Decoding component update the content of packet data with the data from a newly received packet. Each byte of the packet is checked for errors, and it substitutes the old byte after decoding, if the

errors in the new one can be corrected by the hamming 8/4 code

- *DecodeHamming24_18,* which makes the Packet Decoding component update the content of packet data with the data from a newly received packet. Each triplet (three consecutive bytes) of the packet is checked for errors, and it substitutes the old triplet after decoding, if the errors in the new one can be corrected by the hamming 24/18 code;
- *ProcessWSSPacket,* which makes the Teletext decoder process a Wide Screen Signaling (WSS) packet;
- *ProcessVPSPacket,* which makes the Teletext decoder process a Video Program Selection (VPS) packet;
- *ProcessPacket830,* which makes the Teletext decoder process a 8/30 packet related to the general purpose services of a broadcast.

After acquiring all data packets arrived during a single field interval, a high priority task implemented is invoked to decode and store these packets. This task will be referred to as the *Teletext field routine* in the rest of the chapter. The decoding is performed by a dedicated component (*Packet Decoding*). After decoding, the packets are stored within the *Page Storage* component in a local page cache. When all packets of a page are received, they are moved to a global page store, and the other parts of the TV software that need this page are notified that the page has arrived. For example, the UML sequence diagram depicted in Figure 7.3 demonstrates how various VSP services are used to process a simple sequence of packets received with in a single Teletext field.

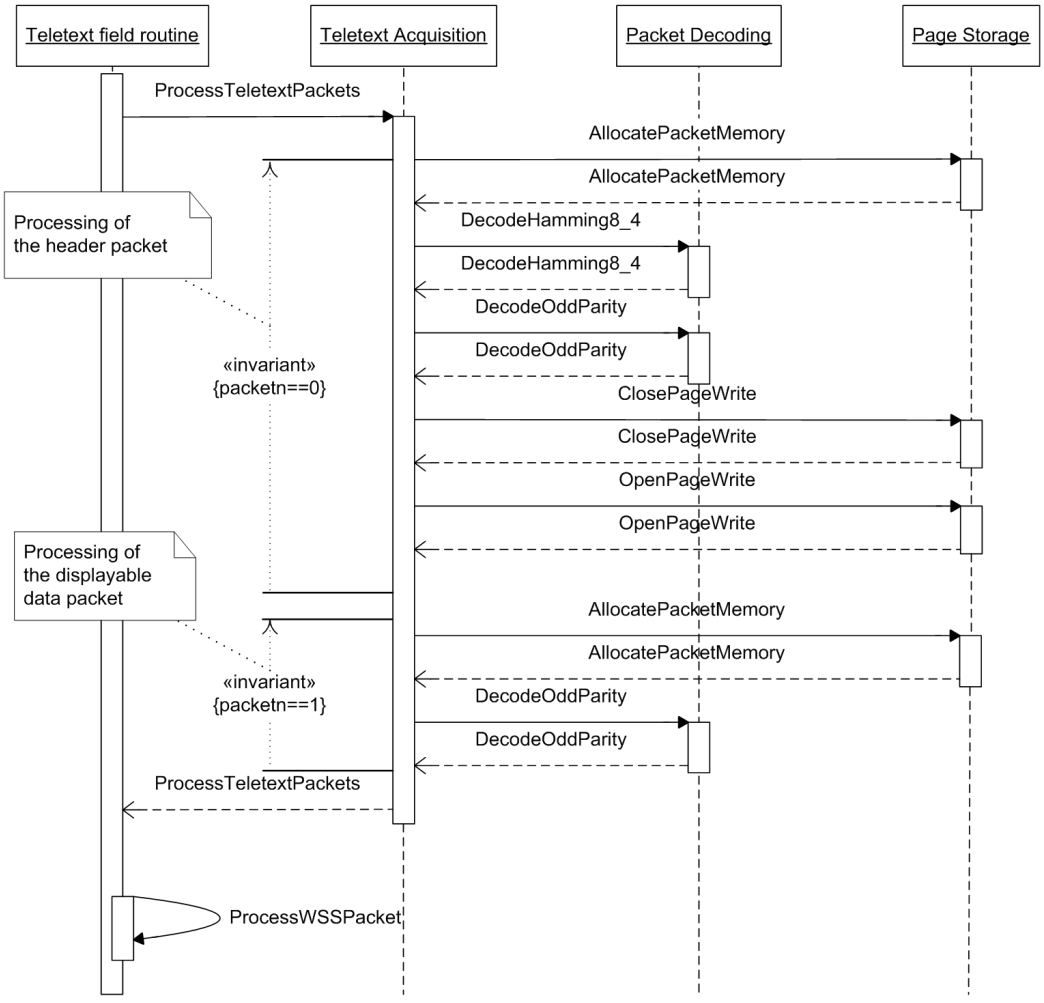


Figure 7.3: An example of the processing of packets by the Teletext field routine

Object live lines refer to the components involved in this scenario. First, the *Teletext field routine* detects that a number of Teletext has been received and invokes the *Teletext Acquisition* component to process these packets. The header of the first packet is checked and it turns out that this packet is a header packet (packet number 0). The *Teletext Acquisition* component allocates memory for the newly packet by calling the *AllocatePacketMemory* service, implemented within the *Page Store* component. This memory is used to store the decoded content of the packet. A part of the packet is decoded by invoking the *DecodHamming8_4* service, whereas the remaining part is decoded by the *DecodeOddParity* service. This header packet starts a new Teletext page. Therefore the page being received has to be completed and a new one has to be started. The completion of the page is performed by the *ClosePageWrite* service, which moves the packets stored in the local page cache to the global cache. The starting of the new page is performed by the *OpenPageWrite* service. This service call prepares the refreshes the content of the local cache in accordance with the number of the page that is stored within the received header packet. Afterwards, the *Teletext Acquisition* component starts processing the next packet. Its packet number is one, which means that it is an ordinary displayable data packet. As in the case with the header packet, some memory is allocated for the decoded content of this packet. The decoding of the packet content is performed by invoking *DecodeOddParity* service. This packet is the last Teletext packet received in the current Teletext field, and the focus of control returns to *Teletext field routine*. Finally, this routine finds out that a *WSS* packet has been also received within the current Teletext field. This packet is processed by the *ProcessWSSPacket* operation. As all the packets receive are processed, the *Teletext field routine* blocks until the next Teletext field.

7.2 Experiment scheme

The aim of this experiment was to predict the execution time of the Teletext field routine (see section 7.1.2) of the Teletext 2.5 acquisition component using the APPEAR method. It was required that the maximal absolute error did not exceed 1ms, as the Teletext field routine had a soft deadline of 20 ms.

The experiment was conducted as follows. *First*, we applied the calibration part of the APPEAR method (see Figure 7.4 and Section 5.6.1) to the Teletext 1.5 acquisition component: the prediction model was calibrated based on the simulation model and the actual implementation of the Teletext 1.5 acquisition component.

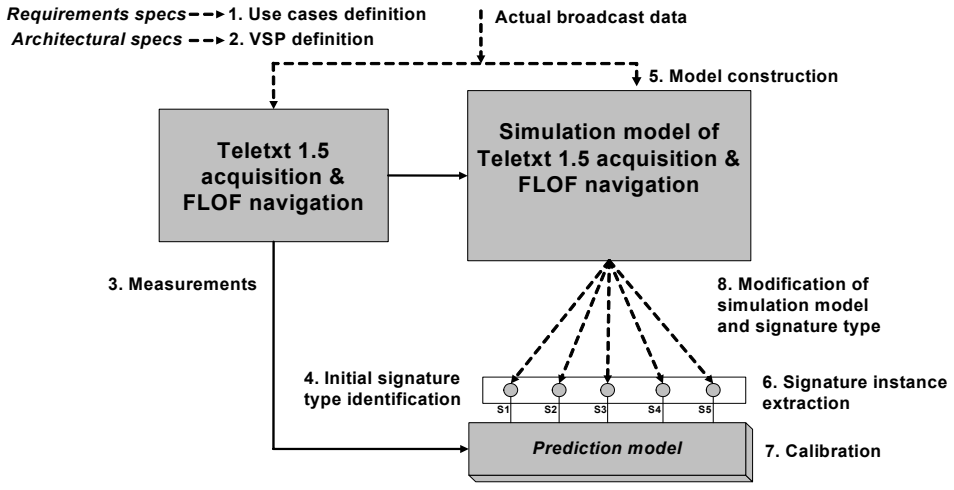


Figure 7.4: Calibration of the prediction model on the Teletext 1.5 component

The steps of the calibration phase are enumerated in the figure. Please notice that steps 6 to 8 were implemented iteratively, until the prediction model had sufficient quality.

Second, by adapting the simulation model and using the calibrated prediction model, we predicted the performance of the Teletext 2.5 acquisition component. The steps of the prediction phase of the APPEAR method are enumerated in Figure 7.5.

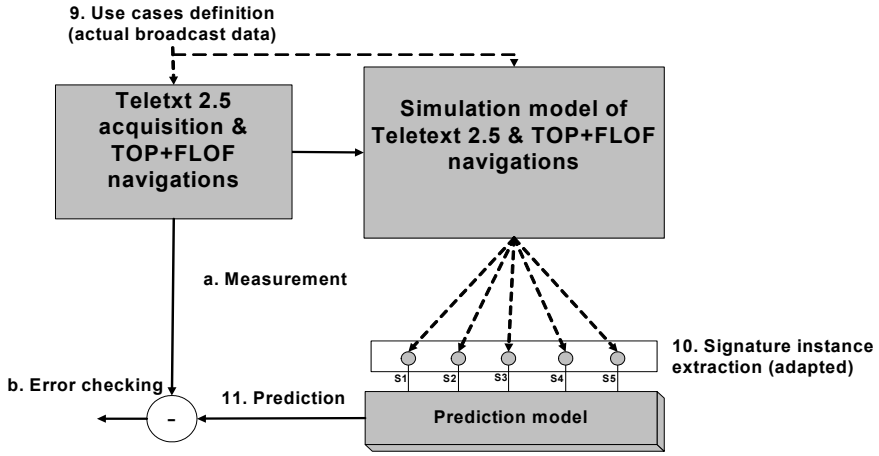


Figure 7.5: Prediction the execution time of the Teletext 2.5 component and comparison of the predictions with the corresponding measurements

Finally, we compared the obtained predictions with the actual measurements from the implementation of the Teletext 2.5 acquisition component. This comparison is reflected by steps *a* and *b* in Figure 7.5. Notice that these actual measurements were based on the same broadcasts that had been used for predicting the performance.

Note that due to technical reasons the Teletext 1.5 acquisition component was created by "downgrading" the existing Teletext 2.5 acquisition component, and not vise versa. This made these two components more similar than they would have been if we had created the Teletext 2.5 component by modifying the Teletext 1.5 component. However, this artificial similarity allowed us to check whether the APPEAR method performs well at least in an "ideal" situation.

7.3 The calibration phase of the APPEAR method

This section describes in detail the application of the calibration part of the APPEAR method to the Teletext 1.5 acquisition component (see Figure 7.4). Each step is discussed in a subsequent subsection.

7.3.1 Use case definition (Step 1)

The considered use cases were the watching of different TV channels that included Teletext information. This means, that a TV set in a steady state was collecting the Teletext data without any interferences. Thirty broadcast channels were chosen (see Table 7.1) to drive both the Teletext application and the simulation model. These channels provided enough observations to calibrate the prediction model. Most of them transmitted Teletext information in the serial mode. Only a few, e.g., BBC 1 and BBC 2, transmitted the information in the parallel mode.

7.3.2 VSP identification (Step 2)

The VSP was identified by studying the architecture and design documentation of the Teletext subsystem as well as by studying its source code. The main findings are summarized in section 7.1.2. When identifying VSP, we relied on the standard criteria for VSP selection (see Section 5.7):

- A VSP is not expected to change during the evolution of a product;
- A VSP defines a proper abstraction level for describing the behavior of the software.

Table 7.1: The broadcast channels used for calibration

Nederland 1	Nederland 2	Nederland 3	RTL 4	RTL 5
SBS 6	Yorin	V 8	Net 5/Kindernet5	UPC Infokanaal
Omroep Brabant TV	Lokale Omroep	Nieuws TV (kabelkrant)	VRT	KetNet / Canvas
National Geographic Channel	Discovery Channel	The Music Factory (TMF)	MTV Europe	ARD 1
ZDF 2	WDR 3	RTL Television	BBC 1	BBC 2
CNN International	TV 5 Europe	TRT International	Eurosport	Local VRT

7.3.3 Measurements (Step 3)

The measurements were collected as depicted in Figure 7.6.

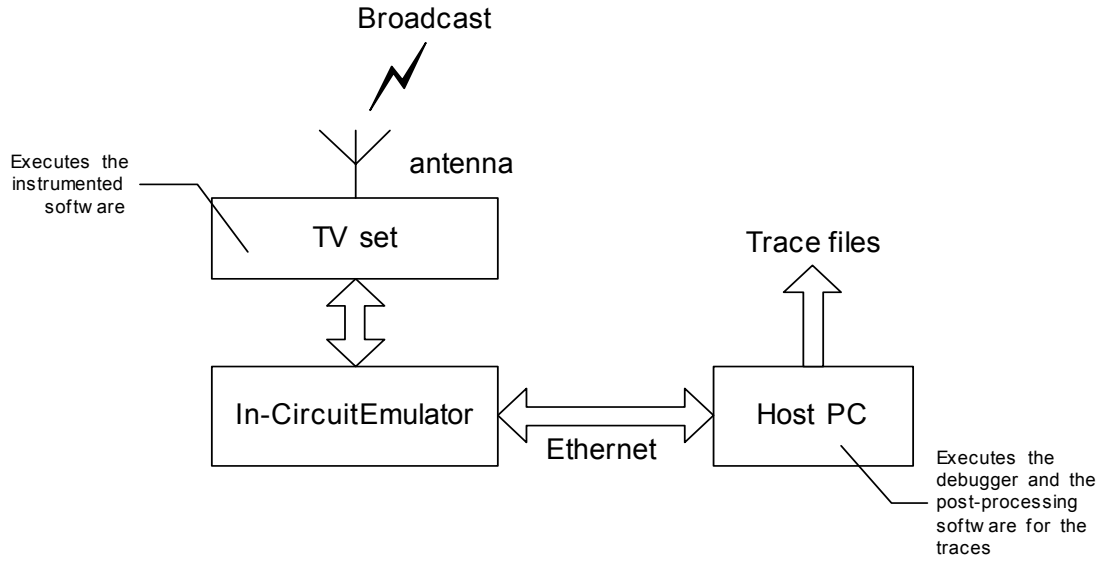


Figure 7.6: The measurement scheme

The TV's antenna input was connected to a cable that broadcasts different channels. The TV was bonded to the In-Circuit Emulator (ICE) and executed the instrumented TV software (including the Teletext decoder). The ICE was also connected via the Ethernet bus to the host PC, which executed the debugging software that also allowed the programming of the ICE for collecting the traces. The length of a trace was bounded due to the limited size of the trace buffer of the ICE (32K). The resolution of the ICE's timing analyzer module allowed collecting the data with 5 ns resolution. The code instrumentation, however, caused a tracing overhead of approximately 1-10 µs per event, which made the measurement error due to the resolution of the ICE negligible.

After filling the trace buffer of the ICE with the trace information, it was transferred to the host PC and post-processed to obtain a trace file in a readable form.

The following time-stamped events were performance-relevant and, therefore, traced:

- the arrival of a header packet that corresponds to a certain page number, sub-code, and type and carries certain control bits;
- the arrival of a data page packet (number 0-25) ;
- the arrival of an extension data packet (number 26-28, designation codes 0-16);
- the arrival of an erroneous packet;
- the arrival of a VSP and WSS packets;
- the context switch between different tasks;
- the beginning of the *Teletext field routine*;
- the completion of the *Teletext field routine*;
- the completion of the packet processing.

Figure 7.7 shows a fragment of typical trace file.

Timestamp	Thread ID	Event
3.576404925	8	OnDcuWstPacketEur(mag=0, packetn=31)
3.576524925	8	OnDcuWstPacketEur(mag=5, packetn=27)
3.576589875	8	OnDesignationCode(0)
3.576919300	8	OnDcuWstPacketEur(mag=5, packetn=1)
3.577277675	8	OnDcuWstPacketEur(mag=3, packetn=31)
3.577397875	8	OnDcuWstPacketEur(mag=5, packetn=2)
3.577752050	8	OnDcuWstPacketEur(mag=5, packetn=3)
3.578110425	8	OnDcuWstPacketEur(mag=5, packetn=4)
3.578468200	8	OnDcuWstPacketEur(mag=5, packetn=5)
3.578826375	8	OnDcuWstPacketEur(mag=5, packetn=6)
3.579184750	8	OnDcuWstPacketEur(mag=5, packetn=7)
3.579542325	8	OnDcuWstPacketEur(mag=5, packetn=8)
3.579896875	8	OnDcuWstPacketEur(mag=1, packetn=31)
3.580017500	8	OnDcuWstPacketEur(mag=5, packetn=9)
3.580375075	8	OnDcuWstPacketEur(mag=5, packetn=10)
3.580734250	8	OnDcuWstPacketEur(mag=5, packetn=11)
3.581032800	8	OnDcuEndData

Figure 7.7: An example of a trace

The leftmost column contains the timestamp of the corresponding event, which is described in the rightmost column. The middle column is the identifier of the thread on which the event is traced. This identifier allows accounting for context switches, when calculating the execution time of the Teletext field routine. In this example, it can be seen that a few page data packets have been received for a page with the number in a range of 500-5ff. Also, two general-purpose packets (with number 31) have been received.

The CPU time needed to process all Teletext packets that arrive in one field interval is chosen as a performance measure. It is calculated by subtracting the timestamp of beginning of the *Teletext field routine* from the timestamp of its completion. The fields that were affected by context switches were considered as a measurement error and were excluded.

The measured data consisted of traces of thirty broadcasts (see section 7.3.1). Each trace described about 20-80 seconds of a broadcast, depending on the number of packets transmitted per field.

7.3.4 Initial signature type identification (Step 4)

The initial signature instance (step 3) was already known beforehand from the earlier experiments [EFH02]. Hence, step 3 is not discussed in this chapter.

7.3.5 Construction and modification of the simulation model for the Teletext 1.5 acquisition component (Steps 5, 6, 8)

The simulation model for the Teletext 1.5 acquisition component was constructed to extract signature instances at step 6 of the APPEAR method (see Figure 7.4). The initial simulation model obtained at step 5 considered only the Teletext Acquisition component (see Figure 7.2) and only a part of packet types. This model calculated signature instances at step 6 (see Figure 7.4) in accordance with the initial signature type (see section 7.3.4). It was however impossible to construct a statistically valid prediction model at step 7 (see Figure 7.4) based on these initial signature type and simulation model. After a few iterations through step 6-8, we managed to obtain an adequate simulation model and sufficient signature type that allowed us to fit the prediction model well. The rest of this subsection describes this adequate simulation model and refers to it as to 'the simulation model'.

This simulation model accepted the descriptions of events that corresponded to packet arrivals in a field. It calculated a signature instance for this field, based on the packets received so far. Notice that it was important to maintain the history from the very beginning of a broadcast, i.e., starting from the very moment when a Teletext signal had become available after channel switching.

The simulation model shown in Figure 7.8 mimicked the behavior of two components: (1) the Teletext Acquisition component and (2) the Page Storage component (see Figure 7.2). Modeling the former helped to account for the short-term history, i.e., to keep track of the pages being transmitted, whereas modeling the latter helped to maintain the long-term history, which turned out to be important for fitting a sufficiently precise prediction model. This long-term history concerns the size of all the pages that had been received since the Teletext signal became available. Notice that the models of both the Teletext acquisition and the Page Storage components were implemented in the AWK language [AKW88].

A) The simulation model of the Teletext Acquisition component

The Teletext Acquisition component implements the functionality such as, decoding and storing of page data packets then assigning them to pages, handling of transmission errors, handling of repeated header packets, supporting the serial and parallel transmission modes, and preserving the consistency of pages (e.g. closing a page on timeout). Most of this functionality is implemented within the *Teletext field routine* (see section 7.1.2). Figure 7.8 presents the UML state chart that describes its behavior. This routine is fed the packets received in a particular Teletext field, and for each packet it invokes the corresponding *packet processing routine*. The *packet processing routine* is relates to the *ProcessNextPacket* composite state. The packet number and magazine number of newly arrived packets are decoded. Depending on these numbers, further processing is delegated to the corresponding state: *ProcessHeaderPacket*, *ProcessBodyPacket*, *ProcessPacket29*, *ProcessPacket830* or *DropPacket*. Note that these states correspond to the functionality in the call hierarchy above the VSP level. The invocations of VSP service calls (see section 7.1.2) are not depicted in Figure 7.8 for the sake of simplicity.

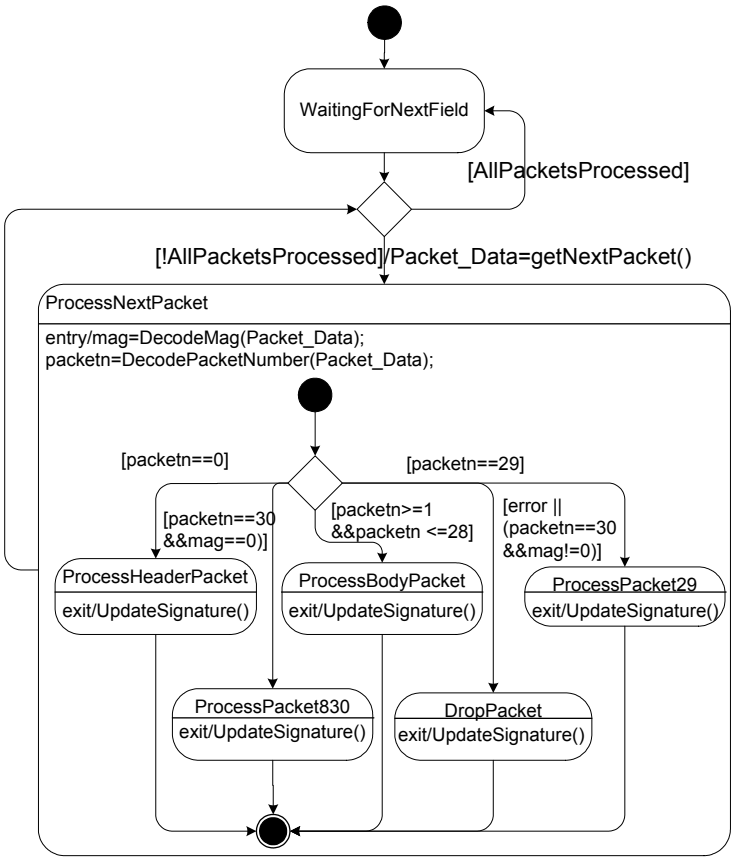


Figure 7.8: The high-level behavior of the Teletext acquistion routine

Based on the current state of the broadcast (it is stored in internal variables, which are also not shown in Figure 7.8), the contribution to the signature instance is calculated for each arrived packet. When all packets have been processed, the signature instance is generated at step 6 (see Figure 7.4) for the entire field, and the next field can be processed.

B) The simulation model of the Page Storage component

The Page Store component maintains the long-term history, i.e., it collects all pages and their content since the Teletext signal was first received.

The model of this component was simplified by assuming that the page storage has infinite capacity. However, the actual implementation had finite page storage, and a dedicated P*age aging* algorithm is used to decide which page has to be removed from the store when it is full.

Another simplification is to ignore page sub-codes. The actual implementation of the Page Storage component handles pages with sub-codes in such a way, that pages with large sub-code numbers may require significantly more processing than the ones with small sub-code numbers. The rationale behind this simplification is that there were no pages with large sub-codes in the measurements of real broadcasts.

Given these simplifications, the implementation of the Page Storage simulation model is straightforward: it is just an array indexed by the tuple {page number, sub-code, packet number}. An element of this array equals one if a page {page number, sub-code} contains packet {packet number}, otherwise it equals zero.

7.3.6 Calibration of prediction model (Step 7)

The S-Plus [KrO02] linear regression tool was used to fit the prediction model, based on the signature type described in section 7.3.7. The prediction model had the following structure:

$$\begin{aligned}\tilde{y} = \beta_0 &+ \beta_1 \cdot WssPacket + \beta_2 \cdot VpsPacket + \beta_3 \cdot NErased + \\ &+ \beta_4 \cdot DecodeNone + \beta_5 \cdot DecodeOddParity + \\ &+ \beta_6 \cdot DecodeHam8To4 + \beta_7 \cdot DecodeHam24To18 + \\ &+ \beta_8 \cdot ErrPackets + \beta_9 \cdot UpdatedPkts + \beta_{10} \cdot Packets830 \\ &+ \beta_{11} \cdot OpenedPages + \beta_{12} \cdot ClosedPages + \beta_{13} \cdot RepeatedHeaders.\end{aligned} \tag{7.1}$$

In Formula (7.1), $\tilde{y}$ is the predicted execution time needed for processing all packets received in a single field; β_i are linear regression coefficients. The remaining variables are signature parameters that are detailed in section 7.3.7.

After calibrating Formula (7.1), the following results were obtained. The multiple R^2-coefficient (the coefficient of determination) is 0.974. This means that the model explains the variability of the execution time well. All regression coefficients turned out to be significant, with a significance level of 0.05 (see Table E.1 in Appendix E). The probability density and the histogram of the residuals is presented in Figure 7.9. Please notice that the bulk of residuals (more than 98%) is concentrated within a ±1 ms interval.

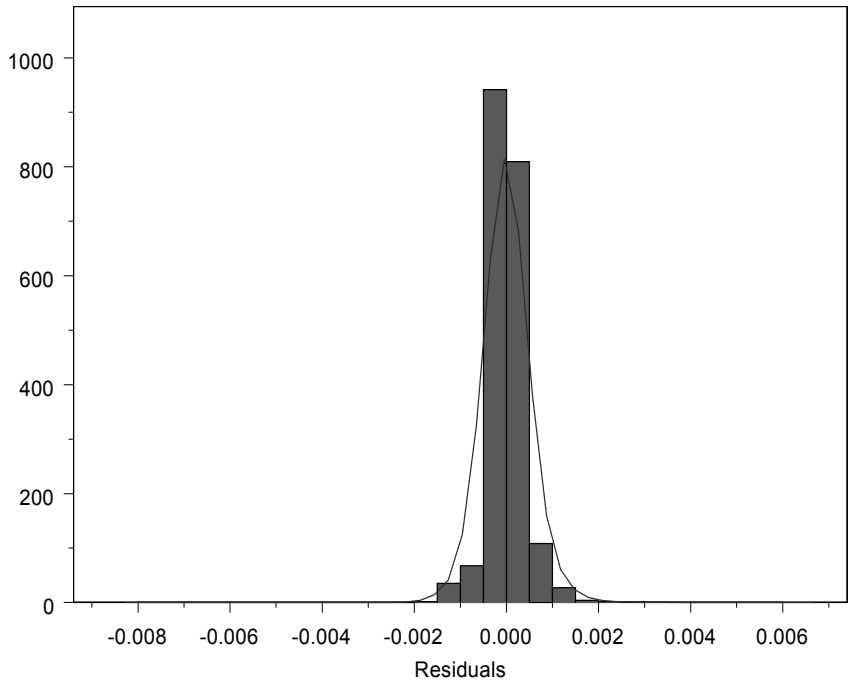


Figure 7.9: Histogram and probability density of the prediction model residual

The residual diagnostic plots (see Figure E.1 in Appendix E) indicated the presence of outliers. The investigation of the corresponding measurements showed that these outliers occurred due to the insufficient handling of packets 8/30. These packets are broadcast service data packets (see [Txt02]); they can have two formats and are discerned with designation codes 0 to 3. One broadcast channel had a number of fields where 12 or 13 packets of type 8/30 arrived. Other broadcasts had at most one such packet per field. However, there were too few fields where more than one packet 8/30 arrived to obtain sufficient calibration data. Additionally, packets 8/30 were treated too simplistically: more

signature parameters are actually needed to describe the execution time needed for these packets.

The prediction errors can be explained as follows:

- *Uncertainty of the long-term history*. The information about the pages stored in the Page Storage turned out to be important to predict the execution time accurately. This information was missing, as the start of the measurements was not synchronized with the start of Teletext data acquisition. This acquisition begins immediately after the TV software detects the Teletext signal. To partially compensate for this effect, it was decided to consider only such Teletext fields that contained either no page header packets or only those that opened or closed pages that had been received at least once. Other fields were considered as measurement error and were therefore ignored. This measure allowed us to restore the missing information, to a degree.
- *The influence of the timer interrupt*. This interrupt occurred every 10 ms and affected the execution time for some fields. The longer it takes to process all packets in a field, the more chances are that this field gets affected. The distribution of execution time of this timer interrupt has several peaks and introduces a non-normal component into the overall residual.
- *The variability of execution time needed to process packets 8/30*. It turned out that those packets require a significantly variable processing time. For instance, the largest outlier shown in Figure E.1b) in Appendix E is an example of this.

These errors are in fact the result of inadequate modeling of the aspects enumerated above. As these errors do not follow the normal distribution (assumed by linear regression), they may also explain the heavy-taildness of the distribution of the residuals.

7.3.7 Modification of signature type (Step 8)

It took us six iterations to amend the simulation model and the signature type. The final signature type accounts for different types of packets, their encoding, and the way they are stored. The following signature type was identified (each number is calculated per field):

- the number of Wide Screen Signaling (*WSS*) packets (*WssPacket*),
- the number of Video Programming System (VPS) packets (*VpsPacket*),
- the total number of bytes that have no encoding (*DecodeNone*),
- the total number of bytes that have odd parity encoding (*DecodeOddParity*),
- the total number of bytes that have Hamming 8/4 encoding (*DecodeHam*8*To*4),
- the total number of triples that have Hamming 24/18 encoding (*DecodeHam*24*To*18),
- the total number of dropped packets (*ErrPackets*),
- the total number of packets 8/30 (*Packets*830),
- the number of repeated headers (*RepeatedHeaders*),
- the number of erased pages (*NErased*),
- the number of opened pages (*OpenedPages*),
- the number of closed pages (*ClosedPages*),
- the total number of packets that have been updated in the Page Storage (*UpdatedPkts*).

Notice that the last signature parameter had to be extracted from the simulation model of the Page Storage component (a part of VSP), whereas the other signature parameters were extracted from the simulation model of the Teletext Acquisition component.

7.4 The prediction phase of the APPEAR method

This section details the application of the prediction phase of the APPEAR method for the Teletext 2.5 acquisition component. The subsequent sections detail each step of the prediction phase of the APPEAR method (see Figure 7.5 and Section 5.6.2).

7.4.1 Use case description (Step 9)

For checking the prediction phase of the APPEAR method, the same channels were used as for calibration phase. For more details, the reader is referred to section 7.3.1.

7.4.2 Measurement (Step a)

The measurements of the Teletext 2.5 decoder were performed in exactly the same way as for the Teletext 1.5 decoder (see section 7.3.3).

7.4.3 The simulation model for the Teletext 2.5 acquisition component (Step 10)

The simulation model of the Teletext 2.5 acquisition component is similar to the one for Teletext 1.5 and supports signature instance extraction (adapted) at step 10. Its main structure remains the same as for the Teletext 1.5 decoder (see Figure 7.8). The differences with the Teletext 1.5 acquisition component amount to the following:

1. *Handling of packets 1-27 of a page with a non-decimal page number* (e.g. Magazine Inventory Page, Table of Pages). The Teletext 2.5 acquisition component has to both decode and store such packets, whereas the Teletext 1.5 acquisition component needs only to store them. The further processing of the non-displayable page is delegated to other dedicated components that are executed by a lower priority task.
2. *Handling of packet 28 with a designation code greater than one or packet 27 with a designation code greater than three*. These packets carry the enhancement information of a displayable page and need to be stored and decoded only by the Teletext 2.5 acquisition component.
3. *Handling of special pages such as Magazine Inventory Pages (MIP), Dynamically Re-definable Character Set (DRCS) pages, etc*. The Teletext 1.5 decoder has no knowledge about these pages and only stores them. Moreover, it does not decode these pages. In contrary, the Teletext 2.5 decoder needs to decode the packets of such page. However, all such pages are encoded using the same set of error correction codes (odd parity, hamming 8/4, and hamming 24/18) that is used by the Teletext 1.5 decoder.

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

7.4.4 Performance prediction for the Teletext 2.5 component (Step 11 and b)

Both implementation and simulation model were traced using the same set of broadcasts, and the predictions, made as outlined in section 7.2, were compared to the measurements from the implementation. The probability densities and histograms of the prediction errors and relative prediction errors are given in Figure 7.10.

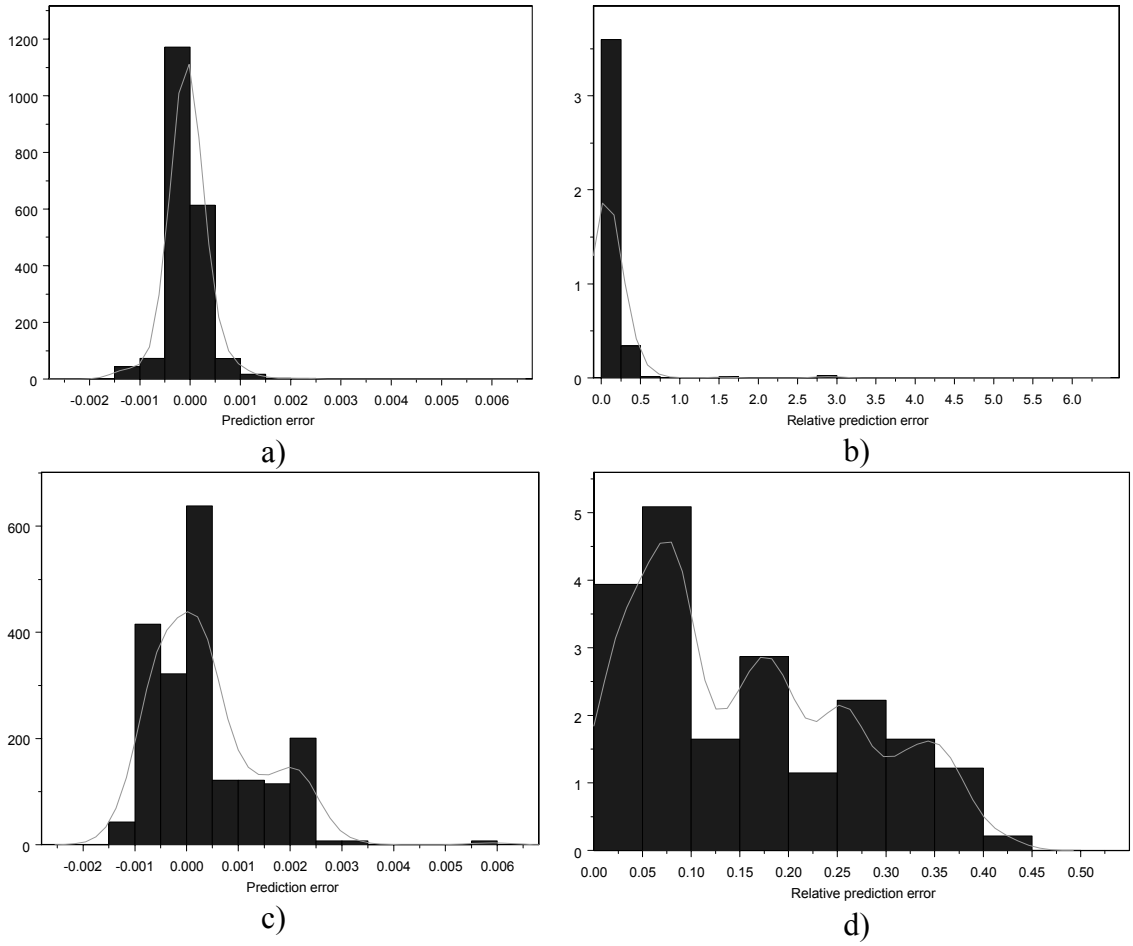


Figure 7.10: Prediction errors and relative prediction errors for the entire broadcasts (plots a and b, respectively) and for Teletext 2.5 and TOP navigation only (plots c and d, respectively)

Plots a) and b) describe these errors for the case when all packets of the broadcasts are considered. On the other hand, plots c) and d) describe the errors only for the packets that are specific to the Teletext 2.5 presentation level or TOP navigation. This subdivision is meaningful, as the functionality of Teletext 1.5 is a subset of the functionality of Teletext 2.5. The y-axis denotes the probability density in all four plots. The x-axis is the prediction error measured in seconds for plots a) and c); it is the relative prediction error (fractions) for plots b) and d).

For the entire broadcasts, the average prediction error is -5.7e-5 s, whereas the average relative error is 11%. For Teletext presentation level 2.5 and the TOP navigation only, these errors are worse: the average prediction error is 3.8e-4 s and the relative one is 16%.

The main source of the larger errors in plots c) and d) is due to the differences in signature instances generated for the entire broadcast and the Teletext 2.5 specific part thereof. The prediction model is biased by the signature instances generated for the entire broadcast, as the number of Teletext 2.5 and TOP navigation packets is very limited in comparison to the rest of the broadcast.

7.5 Similarity of Teletext 1.5 and 2.5 acquisition components

Chapter 6 introduced the following criteria for assessing whether the existing and adapted components are similar:

1. Identical internal computations of the adapted and existing component
2. Identical signature types of the adapted and existing component
3. Proximity of the signature instances of the adapted and existing component

Assessing the similarity according to the first criterion requires the measurements of the amount of internal computations for the existing component. We were not able to perform these measurements, as they were not supported by the instrumentation of the software stack. On the other hand, the difference between these two components is not large. It is detailed in section 7.4.3. This argument allows us to assume that the internal calculations do not differ significantly.

Both Teletext 1.5 and Teletext 2.5 components execute on the top of the same VSP (see section 7.1.2). Moreover, they use the same set of the service calls. This line of reasoning allows us to conclude that the same signature type is likely to suit both components.

Finally, we assessed the proximity of signature instances by calculating the *L*-proximity metric from Chapter 6 for the points covered in plots c) and d) of Figure 7.10. For calculating the values of the *L*-metric, we chose the number *p* of the closest signature instances to be equal thirteen. This choice is supported by the fact that thirteen signature parameters were identified (see section 7.3.7).

Figure 7.11 shows that the prediction errors are, on average, smaller for the points for which *L*-metric is smaller than 1.0 then for the other points. This observation can also be confirmed by applying the t-test [Wei95] for comparing the means of two independent populations.

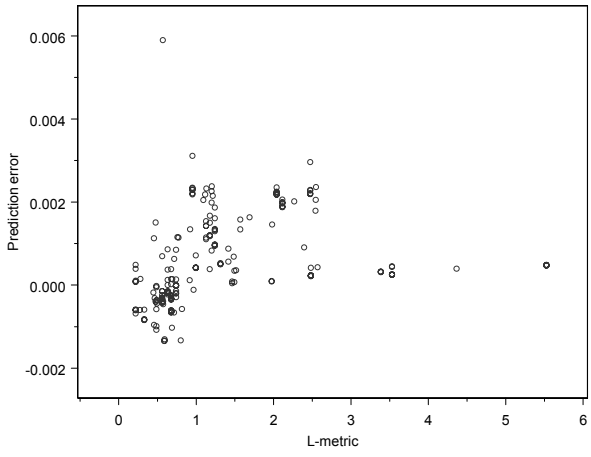


Figure 7.11: Prediction errors versus the values of the L-metric for Teletext 2.5 specific fields

We applied the *t*-test to the following two sets of points: one formed by the prediction errors for which the *L*-metric is smaller then 1.0 and another formed by the rest of the points. The distribution of the prediction errors in these two sets is far from normal. However, it is still possible to use the *t*-test, as both sets contain a large number of points (150 and 129, respectively), and the distribution of the means of these sets will be close to normal because of the Central Limit Theorem [Wei95]. The mean of the first set equals -

0.000121, whereas the means of the second one equals 0.000963. The corresponding sample standard deviations equal 0.000914 and 0.000806, respectively, which suggests us that the t-test with equal variances can be safely applied (the ratio of the sample standard deviations is less than two [Wei95]). The one-sided *t*-test for checking if the second mean is greater then the first one returned the *p*-value zero. This fact indicates that the means are indeed statistically different at the significance level of 0.05.

Both the results of the *t*-test and Figure E.1 in Appendix E indicate that the *L*-metric described in Section 6.2.3 does capture, on average, the growth of the prediction error, if its value exceeds 1.0.

7.6 Summary

This experiment shows that the required level of accuracy (1 ms according to section 7.2) was achieved for 98% of Teletext fields. However, the prediction accuracy decreases if the predictions are considered only for new parts of the Teletext 2.5 acquisition component. In addition, a number of observations about the application of the APPEAR method were made. These observations are described below.

It is important that measurements used for calibrating the prediction model contain sufficient information about all phenomena to be predicted. However, it can be the case that collected measurements tend to contain more information about one group of the phenomena than for another. This may lead to greater prediction errors for the latter group, as the prediction model is dominated by the measurements from the former group.

For example, in the conducted experiment the prediction accuracy degraded for the predictions made for fields that were processed in different ways by the Teletext 1.5 and Teletext 2.5 decoders. This degradation can be explained by the fact that the calibration data was dominated by the Teletext 1.5 and FLOF navigation data only. The data related to Teletext 2.5 and TOP navigation formed only a small fraction of the entire calibration dataset. This effectively made these data outlying observations. The predictions made using this calibration dataset were in fact biased for the fields that contained packets related to Teletext 2.5 and the TOP navigation.

It can be necessary to model a part of the VSP explicitly to be able to fit the prediction model well. This part usually reflects an internal state of the VSP that may affect the execution of the services. Note that the modeling of parts of a VSP is not in accordance with the standard APPEAR method (see Chapter 5). The VSP is defined according to two principles: (1) stability and (2) abstraction. Here, the abstraction level is lowered, while the explicitly modeled part of the VSP is still considered as stable, that is, it is not amendable for modification. The modeling of parts of the VSP broadens the scope of applications to which the APPEAR method can be applied. For example, it was necessary to introduce a simulation model for the Page Storage component in the presented case study (see section 7.3.5).

Within the context of described case study we considered the case of changing a single component only. When multiple components undergo changes, one can apply the APPEAR method to estimate the impact of these changes on the performance only if (1) the VSP remains the same and (2) these components do not interact with each another. If the first condition is violated, the architect might consider taking one of the “escape routes” described in Section 6.4. If the second condition is not met, the architect should use a hierarchical approach, described in Chapter 9, to performance prediction for component compositions.

8 Application of the APPEAR method in the Professional Systems domain

This chapter describes a case study, where the performance of the software components– parts of the Medical Imaging Software System (MISS)– was investigated by means of the APPEAR method. This case study had three main objectives:

1. Prediction of the response time of the adapted software components of the MISS.
2. Providing insight into the performance relevant aspects of the existing software components of the MISS.
3. Validation of the APPEAR method within an industrial setting.

The response time (in milliseconds) of a single component was chosen as a performance metric. The required level of the prediction accuracy, initially posed by the system architects, was 50% in terms of the relative prediction error. This chapter is structured as follows. First, we present an overview of the MISS. Second, we describe the application of the calibration phase of the APPEAR method to the MISS. Third, we explain the prediction phase of the APPEAR method for the adapted MISS components. Finally, the results are discussed.

8.1 Overview of the MISS architecture

The MISS system architecture is multi-layered (see Figure 8.1). A number of COM-components (e.g., "*Acquisition*", "*Reviewing*", etc) implement the basic functionality of the imaging system. These components are mutually independent. They use virtual drivers. Virtual drivers are software abstractions of hardware resources such as an image processing pipeline and image storage board.

We investigated only a single MISS component, "*Reviewing*" (see Figure 8.1), because, according to the MISS architects, the majority of performance claims were related to this component. This component is responsible for viewing and processing the images obtained during an image acquisition session. The "*Reviewing*" component consists of four layers: UI layer, Functional layer (implementation of high level functionality), Database layer (functionality related to patient database), and Technical Services layer (virtual drivers). The "*Reviewing*" component shares a common set of hardware resources (Image Processing, Image Storage) with other components, e.g. "*Acquisition*", but does not directly interact with the other components. The "*Reviewing*" component is detailed in section 8.2.

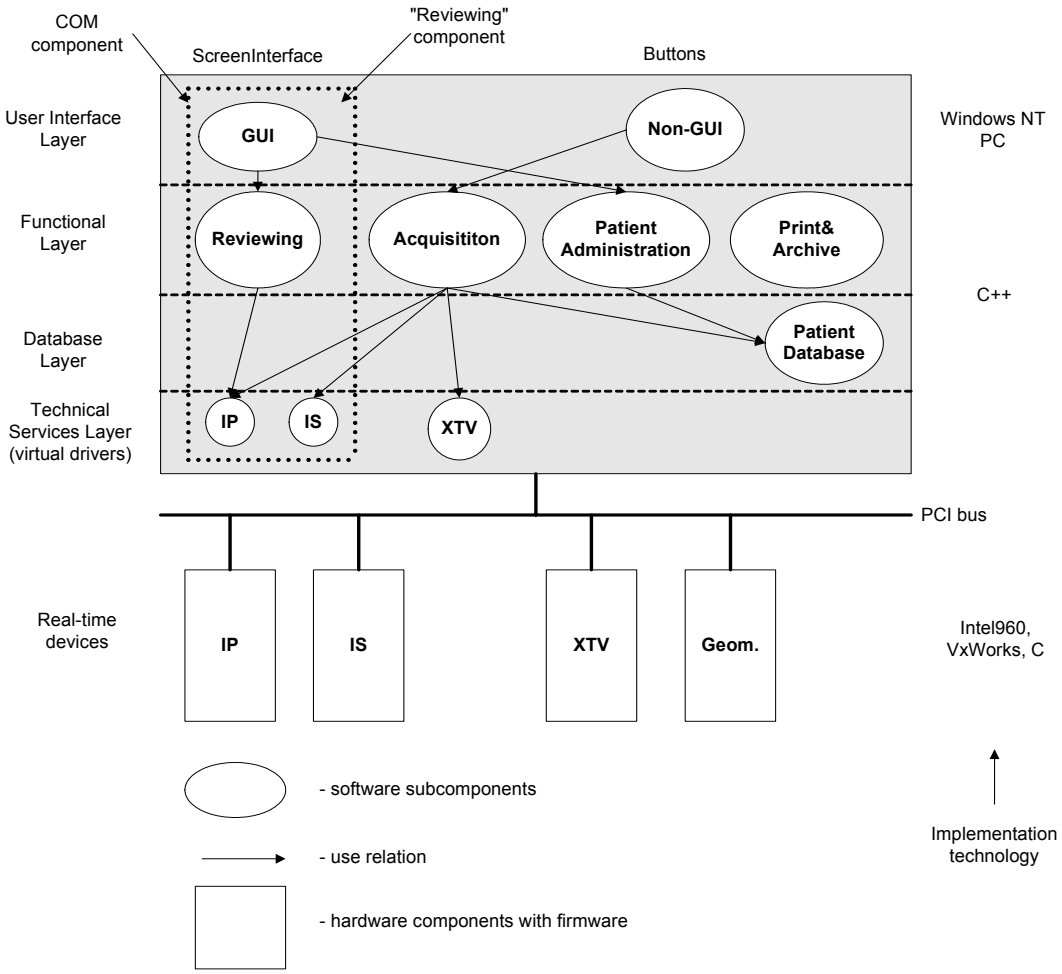


Figure 8.1: MISS system architecture

8.2 "Reviewing" component: structure and functionality

The "*Reviewing*" component contains the following subcomponents (see Figure 8.2):

1. "*GUI*", which provides the graphical user interface ("software" buttons, image windows etc.) and passes user commands to the rest of the component.
2. "*Viewing*"**,** which implements the basic image reviewing functionality.
3. "*ResLib*", which provides the necessary services to the "*Viewing*" subcomponent and schedules hardware resources. Within this component, the majority of performance-relevant services are invoked (calls to "*IPC*", and to "*Graphics*"). Scheduling of the resources has also a significant impact on performance.
4. "*Graphics*", which displays additional graphics (e.g., doctor's comments) over the images.
5. "*DataModel*"**,** which stores and handles patient data.
6. "*Image Processing Chain*" *("IPC")*, which manages the hardware of the "*Image Processor*" and "*Image Storage*" and provides the services related to this hardware to the upper components. "*IPC*" is a non-sharable, preemptable resource, which multiple MISS components contend for.

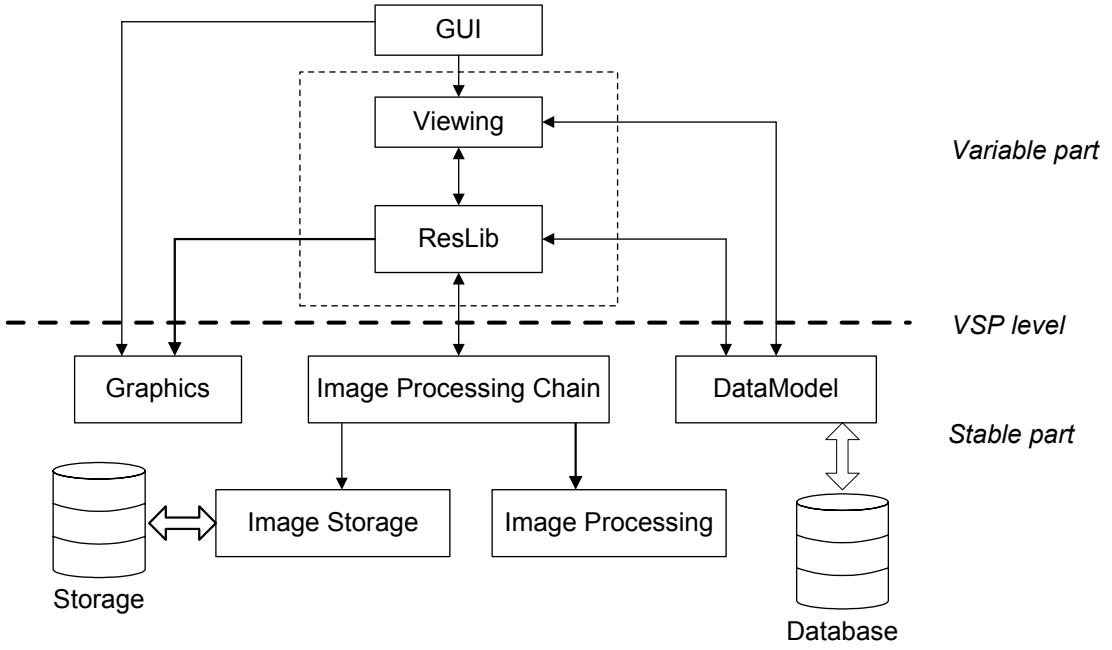


Figure 8.2: *"Reviewing"* component structure

8.3 The calibration phase of the APPEAR method

8.3.1 Use case definition (Step 1)

The MISS uses the following data entities: a file, a run, and an image. These entities form a hierarchy as shown in Figure 8.3. An image is a single medical image of a particular patient. Images can belong to a run, which consists of all images collected during a single acquisition session. There can be multiple runs for a single patient. A set of runs of the same patient forms a file, the highest entity in the hierarchy. Notice that multiple files can exist for the same patient.

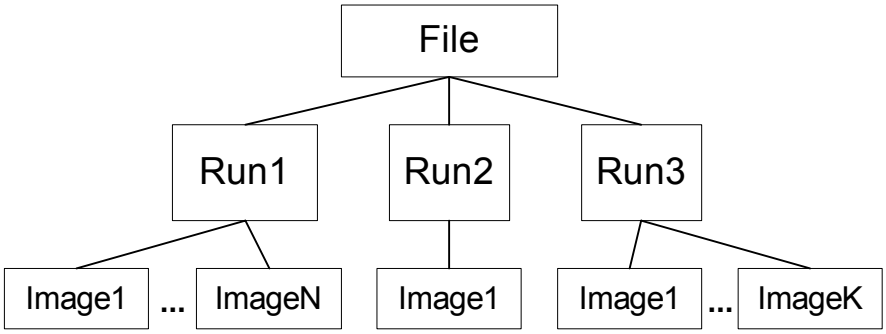


Figure 8.3: Structure of data entities in the MISS software

Images and runs can be selected, viewed and browsed. Usually, the images of a run or the runs of a file are overviewed in a mosaic mode 4 by 4, i.e. 16 images per screen. A single image can be displayed in a full-screen mode.

There are two basic types of use cases for the "*Reviewing*" component: (1) "*Image Navigation*", and (2) "*Image Processing*". They differ in the following way:

- The "*Image Navigation*" use cases are related to browsing across images, whereas the "*Image Processing*" use cases change the image presentation parameters such as brightness, contrast etc,
- The "*Image Navigation*" use cases use the image storage boards, whereas the "*Image Processing*" ones require the image processing boards (see Figure 8.2).

We selected the use cases of the "*Image Navigation*" for the first iteration of APPEAR method validation, as they a) were the most frequently invoked by users, b) exhibited a number of performance problems, and c) were easy-to-execute, to parameterize and to trace.

The navigation can be performed through different navigation objects, displayed on the screen. Each navigation object corresponds with a particular data entity from Figure 8.3. Those objects are displayed in one of the following system states: a file, run, and single image. One of the notable features of the MISS system is the dependency between the number of runs and the number of images in different system states. This dependency is depicted in Figure 8.4. The use cases can thus be partitioned into two groups: a) use cases where number of runs is variable and the number of images remains one and b) use cases where the opposite takes place. This partitioning had a significant influence on constructing the prediction model (see Section 8.3.6).

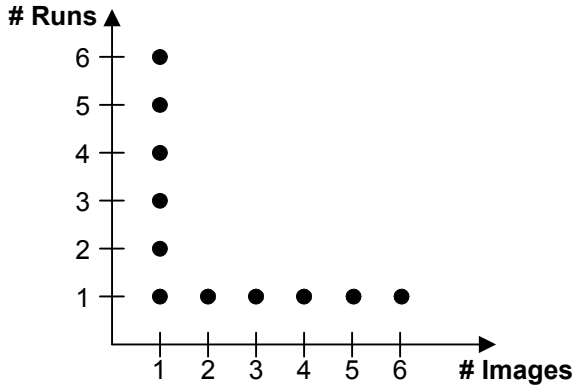


Figure 8.4: Correspondence between the number of runs and images

Table 8.1 reflects the correspondence between use cases and system states. Its rows enumerate the selected use cases of the "*Image Navigation*" type. The columns enumerate different system states. An "X" denotes that the use case from the selected row can be performed in the state denoted by the marked column.

Table 8.1: Use cases and system states

Use case/Object	File	Run	Image
StepImageForward		X	X
StepImageReverse		X	X
StepRunForward	X	X	X
StepRunReverse	X	X	X
StepPageForward	X	X	
StepPageReverse	X	X	
SelectViewPosition	X	X	
ShowSingleRunOverview	X		X
ShowFullScreenImage	X	X	
ShowFileOverview		X	X

A simple example demonstrates why the system state, in which the use case is executed, is important. If the use case "*StepRunForward*" is executed when the run is displayed in overview mode (16 images per screen), the next run in the same mode (16 images per screen) will be displayed. But, if this use case is executed when the image is displayed in full screen mode, the current image of the next run will be displayed in full screen mode. This distinction is also important when constructing the prediction models (see section 8.3.6).

8.3.2 VSP definition (Step 2)

The definition of the VSP level (see Figure 8.2) is based on the following rationales:

1. The subcomponents below this level remained stable for a long period, since they were a) specific for the underlying hardware, which had been stable, and b) common for several high-level MISS components (e.g., image acquisition).
2. The subcomponents above this level had often been improved and extended, and preliminary performance considerations on the modifications would be valuable.

Therefore, we choose a VSP consisting of the following components: "*Graphics*", "*Image Processing Chain*" and "*Data Model*". This VSP provides services to the parts of the "*Reviewing*" component located above it.

8.3.3 Measurements (Step 3)

The selected use cases were executed with various parameters and traced. This tracing allowed us to identify the signature type and to obtain the response times for calibrating the prediction model.

Three alternative tools were considered as tracing utilities: a "TraceUtility" developed at Philips Medical Systems, "Mutek Bug Trapper", "Rational Quantify". The "TraceUtility" tool was chosen because it was designed specially for the MISS software. Moreover, this tool was easily extendable with extra features. For example, the feature of tracing the termination times of functions was added by our request.

Both the tracing command and its parameters were defined by special macros, located just before the function call to be traced:

```
TRACE (TRACE_LEVEL, CATEGORY_GUID, STRING)
TRACE_TIME_START (TRACE_LEVEL, CATEGORY_GUID, EVENT_NAME)
```

Each macro belonged to a certain category. The categorization was done on a functional basis. In order to trace the information of a particular type, the corresponding macro type had to be enabled within the "TraceUtility". An example of the trace file fragment is given in Table 8.2.

Table 8.2: Trace example

Invocation time	Termination time	Thread	Category	Function call
#17:59:02.740	#17:59:02.740	0x53	ComFunc	FUNC(pIDmContext->GetIntegration())
#17:59:02.740	#17:59:02.741	0x53	ComFunc	FUNC(m_pIViewLayout->Disable())
#17:59:02.740	#17:59:02.741	0x53	ComFunc	FUNC(m_pIGraphics->SetGraphics())
#17:59:02.740	#17:59:02.741	0x53	ComFunc	FUNC(m_pIGraphics->SetGraphics())
#17:59:02.740	#17:59:02.741	0x53	ComFunc	FUNC(m_pIDmLIHData->GetFront ())
#17:59:02.740	#17:59:02.741	0x53	ComFunc	FUNC(pIUnknown->QueryInterface())
#17:59:02.741	#17:59:02.745	0x53	ComFunc	FUNC(pIRun->QueryInterface())
#17:59:02.741	#17:59:02.742	0x53	ComFunc	FUNC(pIRun->GetChannelName())
#17:59:02.741	#17:59:02.741	0x53	ComFunc	FUNC(pIRun->QueryInterface())
#17:59:02.743	#17:59:02.746	0x217	ComFunc	FUNC(pIRun->QueryInterface())
#17:59:02.744	#17:59:02.744	0x280	ComFunc	FUNC(m_pIImage->QueryInterface())

Up to 30 tracing categories could be enabled. The challenge was to find a proper balance between trace sufficiency and complexity. The sufficiency of a trace means that the trace contains enough information for identifying the signature type. A high complexity of the trace indicates that too many categories are enabled and too many function calls are traced. Tracing a huge number of irrelevant calls complicates the search

for relevant signature parameters and introduces additional overhead. An appropriate solution to this problem was to enable categories incrementally.

8.3.4 Signature type identification (Step 4)

The architects selected a number of representative performance-relevant use cases. We executed and traced these use cases to identify the initial signature type.

To automate the signature type identification and to process use cases in a joint manner, a dedicated piece of software (trace handling tool) was developed. The signature type identification was based on the use of linear regression [Wei95]. Figure 8.5 presents the information flow between different steps of an algorithm for the identification of the initial signature type. The rectangles correspond with the steps of the algorithm, whereas the arrows denote the type of information (denoted by text in bold font) and the direction of the information flow (denoted by the direction of the arrow). The algorithm consists of the following steps:

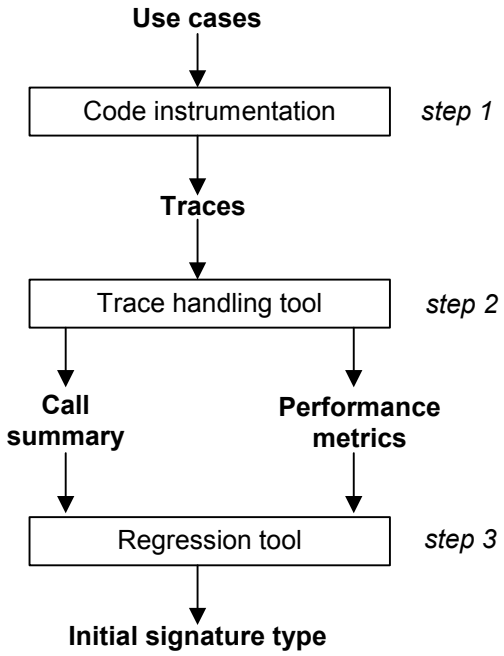


Figure 8.5: Process of signature type identification

Step 1. The code involved into the performance-relevant use cases was instrumented to enable tracing. All performance-relevant use cases were executed and traced.

Step 2. The trace files from step one were processed with the trace handling tool. First, the number of invocations of each call was calculated per use case. Second, the response time was derived for each use case. These results together with the input parameters were collected in a separate file (a part of this file is shown in Table 8.3). Because there were too many calls, their number had to be reduced. For example, the calls that occurred only once in 50 use-cases and took less than 1 millisecond were excluded.

Table 8.3: Numbers of call occurrences for different use case

Use case	Response time (ms)	# Call 1	# Call 2	# Call 3	# Call 4	# Call N
Use case 1	2200	2	100	230	210	20
Use case 2	2358	2	110	253	231	22
Use case 3	2486	2	120	276	252	24

Step 3. The output files from step two were supplied to a statistical regression tool (e.g. S-PLUS [KrO02],[SPLUS]). The prediction model was iteratively fitted on the basis of the input data. The construction of the prediction model was completed, when the model had sufficient quality (e.g., R^2 >0.9). After fitting the prediction model, the candidate calls and input parameters for the signature type were identified. In addition, the regression tool determined *p*-values for each signature parameter (see Table 8.4). These *p*-values are the probabilities that the regression coefficients equal zero for the corresponding signature parameters. The higher the value of a *p*-value, the less the probability is that the corresponding signature parameter influences the performance.

Table 8.4: Initial signature members and their significance levels.

Call	*p*-value
Call 1	0.000
Call 2	0.043
Call 3	0.000
…..	0.615
Call N	0.000

All examined use cases dealt with at least one image or one run. The execution time of many time consuming calls was correlated with the number of images of a particular run or with the number of runs, depending on the type of the use case. Some calls to the "*Image Processing Chain*" and "*Graphics*" components were also time consuming, although their execution time did not depend on the number of images or runs. Calls to these two components will be referred to as "*Update()*" and "*Paint()*", respectively, in the rest of this chapter. The former concerned updating the state of the image processing hardware, whereas the latter related to graphics that overlay medical images.

Note that the actual number of call types was greater than 100, but only two of them influenced the performance significantly. Summarizing, the signature type of the component can be represented as a vector consisting of only four parameters:

$$\textit{Signature type} = (\#\textit{Runs}, \#\textit{Images}, \#\textit{Update}, \#\textit{Paint}) \quad (8.1)$$

An example of the resulting signature instances and response time measurements is presented in Table 8.5.

Table 8.5: Examples of signature instances.

Use case	#Runs	#Images	#Update()	#Paint()	Response time (ms)
1	1	3	17	4	245
2	1	5	44	8	512
3	11	1	56	4	793
4	4	1	73	6	927

8.3.5 Simulation model (Steps 5, 6, and 8)

The simulation model mimics the high-level behavior of the "*Reviewing*" component and calculates the signature instances (see Figure 8.6). The model is activated by parameterized user commands. These parameters are the number of images and runs to be displayed, their attributes, etc. As a reaction to the user command, the model produces a part of the signature instance, i.e., the numbers of the "*Update()*" and "*Paint()*" service calls.

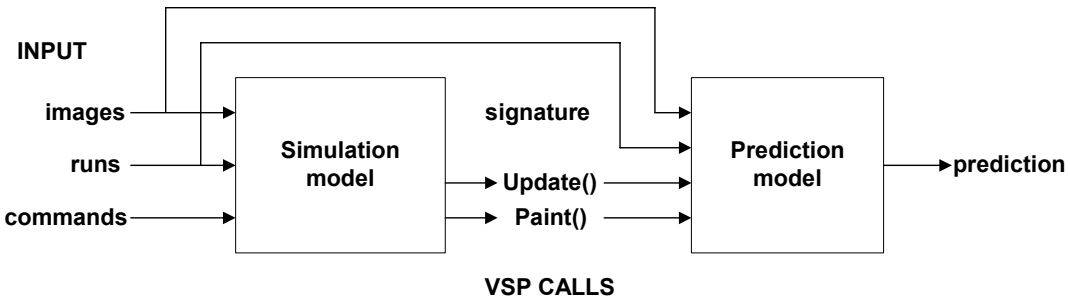


Figure 8.6: Simulation model inputs and outputs

Note that the number of runs and images were also a part of signature type, but they were directly passed to the inputs of the prediction model, bypassing the simulation model.

A) Selection of the description formalism

We decided to describe the behavior of the "*Reviewing*" component in terms of state machines for the following reasons:

4. The software was initially designed in a way that can be easily expressed in terms of state machines: states, input events, transitions and resulting actions could easily be identified.
5. There exist many tools (COVERS [COV97], AnyLogic [ANY00], UPPAAL [UPP99]) that allow the creation of executable simulation models based on state machines. Some of these tools also support automatic code generation in different languages (e.g., in Java, C++) for different platforms (Windows, Unix).
6. State machines is one of the formalisms that are intuitively understood and easily accepted by software architects and software engineers.

B) Behaviour description

The state machine representing the system behavior is depicted in Figure 8.7. The system can work in three different states: (1) "*File overview*", (2) "*Run overview*", and (3) "*Image overview*" (full screen). Each state is described by a corresponding state of the state-machine from Figure 8.7. In each state, particular invariants over the number of runs and the number of images hold (see also section 8.3.1). The user can switch between states by invoking commands of the "*Image Navigation*" type. For each command invocation, the number of service calls is calculated to obtain a signature instance.

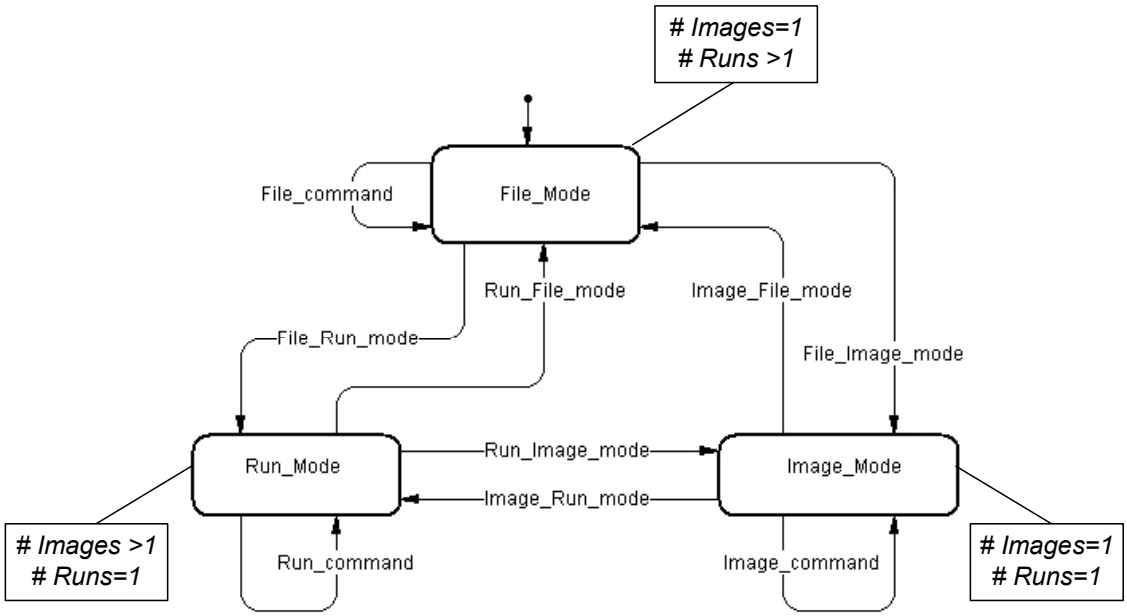


Figure 8.7: The states of the *"Reviewing"* component

C) Calculation of the signature instances

Each transition of the state machine from Figure 8.7 is associated with the invocation of a number of service calls. The number of the invoked service calls is a function of the following signature type:

$$f\,(History,\ External,\ Command) \qquad (8.2)$$

Here, the following notation is used: *History* parameters include the previous state, previous number of images/runs, previous run settings, etc.; *External* parameters are the variables that can either define current state of the system (e.g., "*File_Mode*") or global parameters (e.g., the number of monitors); *Command* is a user command that fires the transition.

The subsequent sections describe how the number of "*Paint()*" and "*Update()*" call is calculated.

D) "*Paint()*" calls

When executing a command, the user window may need to be redrawn to display the overlaying graphics. Redrawing is performed by invoking the "*Paint()*" call. The number of "*Paint()*" calls is determined by the presence of coordinate system dependent graphics (e.g., electronic shutters, pixel shifts, etc.). The number of "*Paint()*" calls always equal one, if no such graphics is attached to medical images. It becomes two, if at least on image is attached this graphics. Note that the graphics is always attributed to a single medical image.

E) "*Update()*" calls

The number of "*Update()*" calls depends on both the state and the settings of the so-called image processing nodes. An image processing node controls image attributes such as brightness, contrast, etc. and needs to be configured with the corresponding settings. These settings are specified per run and are set by the "*Update()*" call.

The number of necessary nodes can be calculated by the following formula:

$$\#Nodes = 13 + 6 \cdot \#Monitors \quad (8.3)$$

Here, *#Monitors* denotes the number of monitors of the image processing system (usually, 1 or 2). There are 13 processing nodes in the MISS system that are used independently from the number of monitors. The other six nodes are related to the number of monitors.

Table 8.6 shows the dependency between the number of the "*Update()*" calls, the state of the system, and the number of involved image processing nodes.

Table 8.6: Dependencies of the Update() calls

State	#Update()
File Overview	N_Runs*N_Nodes
Run Overview	N_Nodes
Image	N_Nodes

The dependency on the previous node settings is important for estimating the number of the "*Update()*" calls. Processing nodes must be updated with the corresponding node settings of a new run to be displayed. However, not all nodes need updating. If the node settings of the new run equal to the node settings of the previous run, the settings are not updated. In this case, the "*Update()*" call has a significantly shorter duration than for the case when node settings are updated.

So, the duration of "*Update()*" calls depends on the history, namely, on the settings of the previously displayed run. This violates the initial assumption that only the number of these calls matters. A solution to this problem was to improve the signature type by introducing two VSP calls - "*ShortUpdate()*" and "*LongUpdate()*" – instead of one "*Update()*", dependent on whether the hardware settings must be updated or not.

F) Simulation model implementation

This section details the implementation of the simulation. This model has a structure depicted in Figure 8.8. The rectangles denote the components of the model, whereas the arrows show the dataflow between the components.

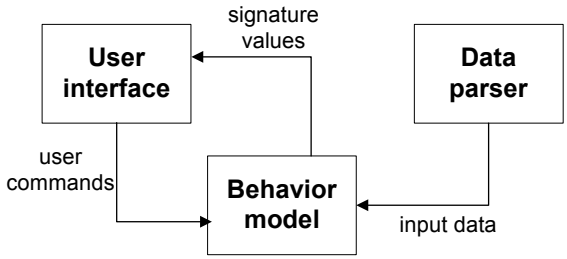


Figure 8.8: Structure of the simulation model

1. "*User interface*"– this part is responsible for interaction with the user; it implements GUI, receives user commands, and passes them to the "*Behavior model*" part.
2. "*Behavior model*"– this part implements the state machine that models the "*Reviewing*" component behavior at a high level of abstraction (see Figure 8.7): it processes user commands, and calculates signature instances.
3. "*Data parser*"– this part retrieves the input data (files, runs, images, and their settings) from the input source (file or database) and passes them to the "*Behavior model*" part.

Two versions of the model were implemented: one using the COVERS [COV97] simulation engine and another based on a conventional Windows application. The semantics of both models were the same, but input/output interfaces are different. The former model provides a better visualization, whereas the latter can display only the current state in a textual representation. However, the implementation of the former model uses obsolete libraries that cannot run on Windows platforms and are difficult to interface with a conventional GUI. The latter model does not suffer from these problems.

G) Validation of the simulation model

This section describes validation of the simulation model. This procedure consists of the following steps:

1. *Selection of a short, but sufficient, set of input parameters for the simulation model.* It is important to make the simulation model simple and easy to modify. In addition, this model must generate signature instances that can calibrate the prediction model with sufficient quality. Thus, only significant input parameters were selected, i.e. parameters that both have an impact on a signature instance and vary in many use cases. For instance, the number of parameters that determine the number of "Update" calls (settings of image processing nodes, like contrast and brightness) was decreased from 75 to 41. These parameters were further grouped.
2. *Providing the real data to the simulation model.* The simulation model had to be fed with real data retrieved from the MISS database, as it was cumbersome to generate this data artificially. However, the MISS database component could not be used "as is" as it was very complex, required the entire MISS infrastructure (image processing and storage boards, database server, etc.), and could only run on a dedicated PC. To keep the simulation model lightweight and usable, a *"Database parser"* was developed. This module was based on the existing MISS database component, but did not require the entire infrastructure. It reads a MISS database file, and translates the data into a text format that can be easily read by the internal parser of the simulation model. This data contains descriptions of the runs for a single patient: the settings of the image processing nodes, the number of images, and the number of textual annotations per image (see Figure 8.9).

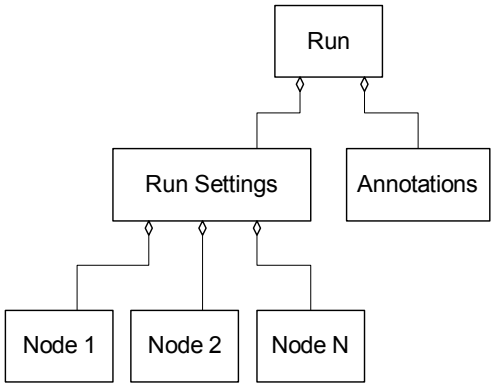


Figure 8.9: Run data

3. *Checking the quality of the simulation model (iteration).* After completing constructing the simulation model and steps 1 and 2, it was vital to check its quality. This check included the following:
    - Fitting the prediction model to the measurements from the traces based on signature instances extracted from the same traces,
    - Fitting the prediction model based on signature instances extracted from simulation model,
    - Comparison of the quality of these two models.

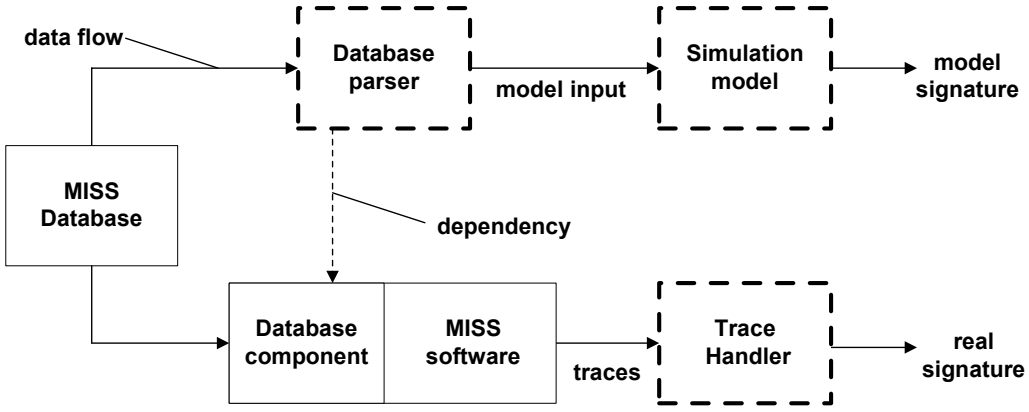


Figure 8.10: Validation of the simulation model

The validation described above was assisted by a dedicated tool chain (see Figure 8.10). This tool chain included three already existing parts (solid rectangles): the MISS software, the MISS database, and the MISS database component. Three extra parts had to be added (dotted rectangles):

1. The "*Database parser*" provided the simulation model with real input data. It retrieved the data from the MISS database and translated them into a form suitable for the simulation model.
2. The "*Simulation model*" was executed on real input data. Use cases were carried out both on real software and on the simulation model, and the corresponding signature instances are generated.
3. The "*Trace handler*" parsed the traces obtained by executing the use cases on the real software and calculated the signature instances.

Two prediction models were calibrated: one on signature instances from the measurements and another on signature instances from the simulation model. This procedure yielded the prediction models with the characteristics described in Table 8.7. More details about the construction of the prediction models and various cases can be found in Section 8.3.6.

Table 8.7: Quality of different prediction models.

Prediction model	Multiple R-squared	Average relative error
Case 1 (#Images>1, measurements)	0.9059	0.0712
Case 1 (#Images>1, simulation)	0.9386	0.0554
Case 2 (#Runs>1, measurements)	0.9893	0.0435
Case 2 (#Runs>1, simulation)	0.9907	0.0460

This table shows that both prediction models have a high quality. Two conclusions can be drawn from this fact:

1. Calibration of the prediction model on the signature instances generated by the prediction model did not worsen the prediction quality in comparison to calibration on the signature instances extracted from the traces.
2. The reduction of the number of input parameters used in simulation model did not have a severe impact on the prediction quality.

8.3.6 The prediction model (Steps 7 and 8)

This section describes the construction and validation of the performance prediction model for the "*Reviewing*" component. The prediction model was constructed with a linear regression tool (S-PLUS [KrO02]) and calibrated on the signature instances generated by the simulation model, i.e., on the number of "*Update()*" and "*Paint()*" calls. In total, 120

signature instances were used. The model was validated in the following way: 10 arbitrary points[1] were left out from the measurements, and the model was calibrated using the remaining ones. First, the quality of the model was assessed via analysis of statistical characteristics of the model: the multiple R^2-coefficient, maximal absolute error, and a few residual diagnostic plots (see Appendix F). Second, the values of response times were predicted for the initially excluded points and compared with the measured ones.

A) Construction of the prediction model

The peculiarities of the "*Reviewing*" component, e.g. the relation between the number of runs and the number of images, were described in section 8.3.1. Moreover, the measurements showed that different linear coefficients are needed to express the dependency of response time on the number of images and the number of runs (see Figure 8.11).

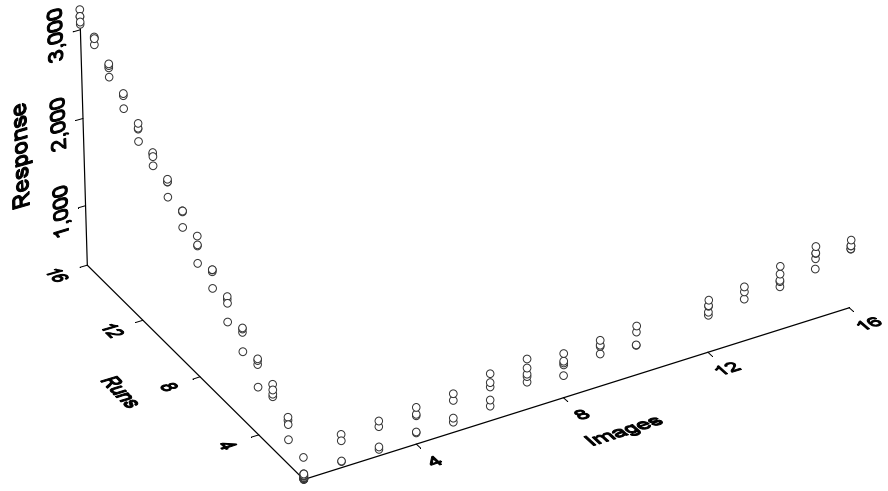


Figure 8.11: 3-D plot of the dependency of the response time on the number of runs and images

Due to these issues, the following relation between *#Images* and *#Runs* was accounted for when constructing the prediction model:

$$\begin{aligned} &\#Images = 1, \ if \ \ \#Runs \ > \ 1 \\ &\#Runs = 1, \ if \ \ \#Images \ > \ 1 \end{aligned} \tag{8.4}$$

[1] For the cross-validation, a number of points had to be chosen according to the following criteria. This number should not be "very small" (e.g. three points); otherwise the representative amount of prediction errors is not obtained. This number should not be "very large" (e.g., 100 points); otherwise there is no sufficient amount of points for fitting the prediction model. Within the interval that satisfies both criteria, any arbitrary number could be chosen, in our case, ten.

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

$$Response = \begin{cases} \beta'_0 + \beta'_1 \cdot \#Runs + \beta'_2 \cdot \#LongUpdate + \beta'_3 \cdot \#ShortUpdate + \\ + \beta'_4 \cdot \#Paint; \quad \text{for the case } \#Runs > 1, \#Images = 1 \\ \beta_0 + \beta_1 \cdot \#Images + \beta_2 \cdot \#LongUpdate + \beta_3 \cdot \#ShortUpdate + \\ + \beta_4 \cdot \#Paint + \beta_5 \cdot \#Group; \quad \text{for the case } \#Images > 1, \#Runs = 1 \end{cases} \tag{8.5}$$

In this formula, β_i and β'_i are linear regression coefficients. The preliminary analysis of the calibration data set and the quality of the initial prediction model revealed the necessity to introduce an additional signature parameter. This parameter is the *Group* categorical variable that distinguished four groups of use cases. It was necessary, as, for example, some use cases required additional interaction with the "*Database*" component (e.g., "*StepPageForward*", Group 1) whereas the others did not (e.g., "*StepRunForward*", Group 2).

Table 8.8 describes the quality of the prediction model for both cases in terms of multiple R-squared coefficient.

Table 8.8: Quality of the prediction model.

Cases	Multiple R-squared
Case 1 (#Images>1)	0.9386
Case 2 (#Runs>1)	0.9907

Figure 8.12 and Figure 8.13 show the diagnostic plots demonstrating the normal distribution of the residuals cases. Slight deviations from normality can be explained by a) insufficient amount of data that was used for model calibration and b) the presence of extra (hidden) variables. Additional data and plots describing the quality of the model (e.g., p-values of regression coefficients for the t-test for significance) are provided in Appendix F.1 and Appendix F.2.

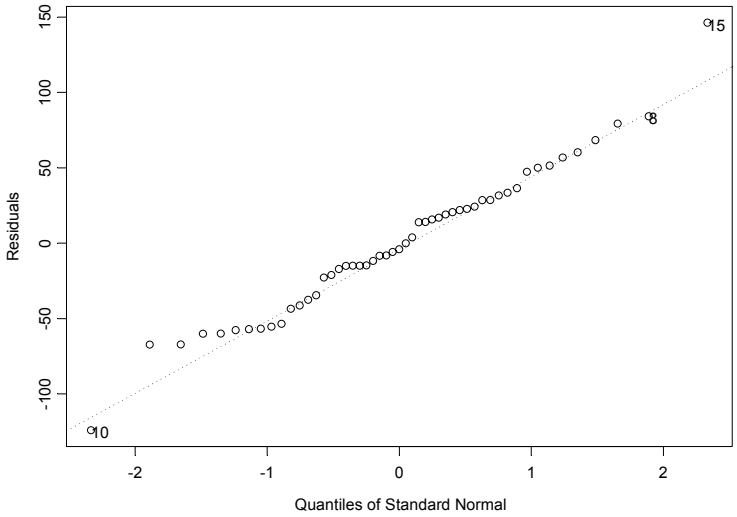


Figure 8.12: Normality of residual distribution

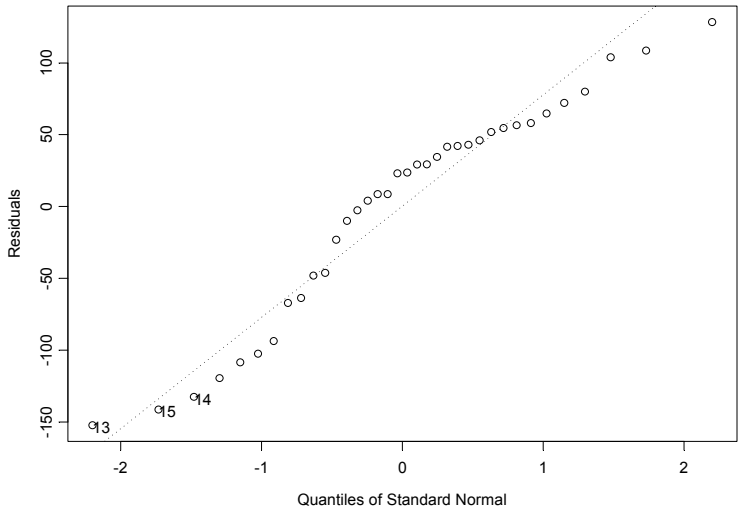


Figure 8.13: Normality of residual distribution.

B) Validation of the prediction model

The validation was performed as described above: 10 arbitrary points were excluded from the data before calibration, and the predictions were made for them.

Table 8.9 summarizes prediction errors of different types and compares prediction errors with the mean squared errors of the fitted model. Appendix F.3 explains how these errors are calculated and presents the distribution of prediction errors.

Table 8.9: Average relative errors

Cases	Average relative error	R-squared (predicted)	Average squared error (prediction)	Mean squared error (fitted)
Case 1 (#Images>1)	0.0865	0.9472	3722.97	6348.09
Case 2 (#Runs>1)	0.0151	0.9918	2358.36	5764.12

Low average relative errors, very slight differences between R-squared coefficients of the fitted model and predictions, and low average squared prediction error (in comparison with mean squared error) confirm the good prediction quality of the model.

8.4 The prediction phase of the APPEAR method

This section illustrates the use of the APPEAR method for predicting the performance of an adapted component. The adaptation consisted in adding a new function. Since this new function was not yet implemented, the simulation model was adapted based only on the design description of this new function. The performance predictions for the new function are estimated in terms of prediction intervals.

8.4.1 From design to simulation model (Steps 9 and 10)

A proper means was needed to effectively specify new features of the *"Reviewing"* component from a performance viewpoint. This specification means had to allow one (1) to describe all performance relevant aspects of the behavior of the new features, and (2) to

integrate the new features into the existing simulation model with ease. We decided to specify the new features in terms of UML Message Sequence Diagrams (MSD) because of the following reasons:

1. The MISS software developers were already using the UML MSD specification language.
2. The UML MSD notation is widely accepted and applied.
3. There existed a number of the industrial tools that supported the UML MSD specification language (e. g. Rational Rose).

An MSD was attached to each transition of the state machine of the simulation model. This MSD specified the behavior of the new feature in terms of performance relevant calls to the VSP (see Figure 8.14). When a transition was fired, the signature instance was calculated by parsing the attached MSD and counting the number of performance relevant service calls. An example of such an MSD is given in Figure 8.16.

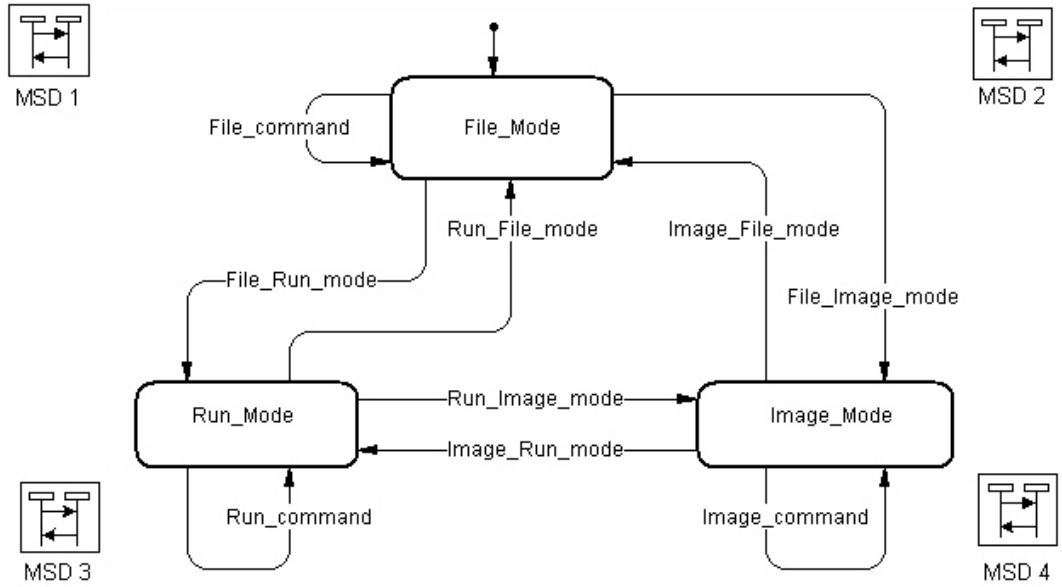


Figure 8.14: Integration of the new features into the simulation model

8.4.2 Prediction for the new function (Step 11)

As an example, the "*ViewTrace*" function was studied. This function has to be implemented in the next version of the MISS software. This function merges a series of images of the current run into a single composite image This series of images is called a *trace.* This composite image contains a view of the entire vascular tree structure, which is obtained by retaining only pixels with the minimum or maximum intensity from all images involved (see Figure 8.15).

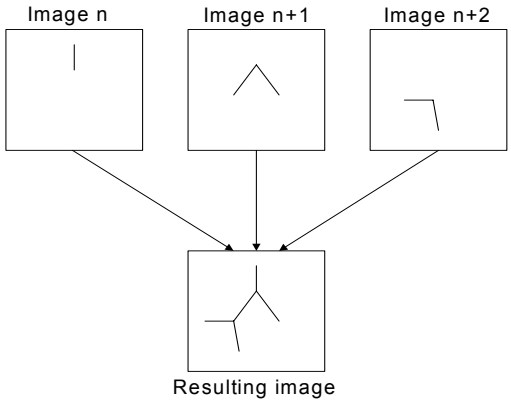


Figure 8.15: Result of the "ViewTrace" function

The UML MSD specification of the "*ViewTrace()*" function is shown in Figure 8.16.

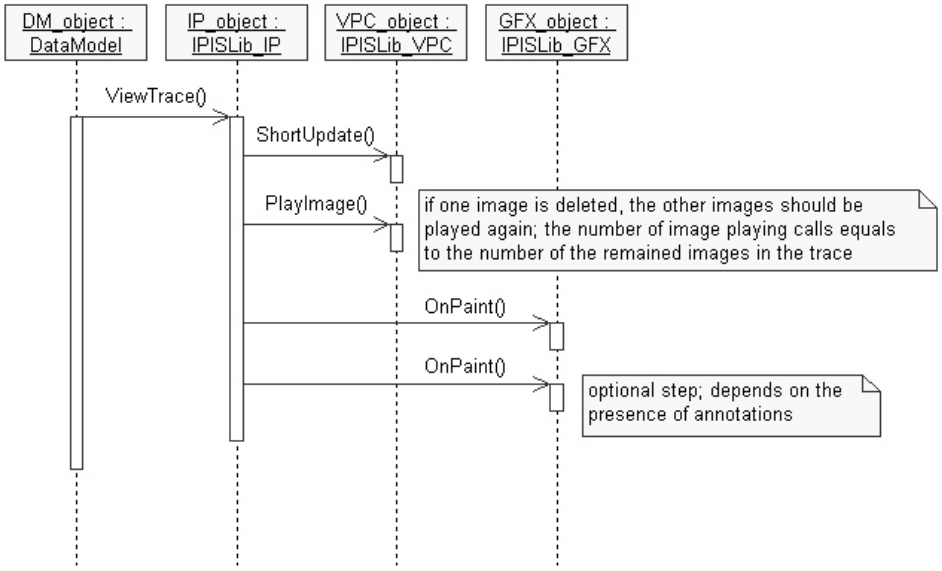


Figure 8.16: Specification of the "ViewTrace()" function

The response time was predicted for 8, 16, 24, and 32 images using the previously calibrated prediction model. The predictions and the corresponding 95% prediction intervals are shown in Table 8.10.

Table 8.10: Prediction results for the "ViewTrace()" function

#images	#"Long-Update()"	#"Short-Update()"	#"Paint()"	Predicted value [ms]	Prediction interval [ms]
8	0	8	1	843.37	(705.3; 981.5)
16	0	8	1	1098.47	(958.5; 1238.5)
24	0	8	1	1353.57	(1205.2; 1501.9)
32	0	8	1	1608.67	(1446.4; 1771.0)

The number of the "*LongUpdate()*" calls equals zero, as all actions are performed within the same run and the hardware settings do not need to be updated. The number of "*ShortUpdate()*" calls is constant since the hardware settings are fixed. Apparently, only the number of images varies.

Notice that the prediction intervals enlarge when the distance between the point to predict and the calibration set increases. Despite this fact, the prediction intervals corresponding to distant points (e.g., 24 or 32 images) remain narrow (e.g., it is ±162.30 ms for 32 images, with the predicted value being 1608.67 ms).

8.4.3 Similarity of the components

This section proves the similarity of the existing *"Reviewing"* component and the adapted version thereof, according to the similarity criteria defined in Chapter 6:

- Identical internal computations of the adapted and existing component,
- Identical signature types of the adapted and existing component,
- Proximity of the signature instances of the adapted and existing component.

In opposite to the example in Chapter 6, it is ensured by design that both existing and adapted components do not perform any internal computations. This fact satisfies the first criteria by construction.

As described in section 8.4.2, both existing and adapted components a) are implemented on the same VSP and b) use the same service calls for realizing their functionalities. This allows us to conclude that the signature types of these components are identical. This satisfies the second criteria of similarity.

The proximity of signature instances was estimated by calculating the *L*-metric (see Chapter 6). We estimated the distance between the points used for calibrating the prediction model in the first case (variable number of images) and four points, listed in Table 8.10. The number *p* of closest points was equal to the number of signature parameters (four).

Table 8.11 shows the values of the *L*-metric for the selected four points (the same points as in Table 8.10).

Table 8.11: The values of *L*-metric

#images	#"LongUpdate()"	#"ShortUpdate()"	#"Paint()"	L-metric
8	0	8	1	0.0000
16	0	8	1	0.7962
24	0	8	1	2.8572
32	0	8	1	4.9615

The value of the *L*-metric equals zero for the first point (8 images). This fact shows that it lies exactly in the center of mass of the four points closest to it. The second point (16 images) lies within the calibration dataset, but was not used for calibration. Thus, the value of *L*-metric remains lower than 1.0. For the last two points (24 and 32 images), lying beyond the calibration dataset, the value of *L*-metric is greater than 1.0.

Note that the value of the *L*-metric increases depending on the distance between prediction point and group of the points used for calibration. This increase of the L-metric corresponds with diminishing of the prediction quality and with widening of the prediction intervals shown in Table 8.10.

8.5 Summary

Besides the illustration and validation of the APPEAR method with a positive outcome, this case study explored a number of significant issues. They are discussed below.

The construction of the prediction model for two separate cases was necessary as the collected measurements corresponded to the heterogeneous use cases. Additionally, the introduction of the categorical variable that defines the type of the use case can improve the characteristics of the prediction model. This issue acknowledges the statement from Chapters 5 and 6 about the similarity of both the components and the use cases for achieving reliable predictions.

The validation of the simulation model by calibrating the two prediction models– one on the measurements and the other on signature instances generated by the simulation model– can be useful when refining the simulation model. This step can provide fast feedback on both the sufficiency of the signature type and the quality of the simulation model.

Analysis of the results obtained during the validation of the prediction model revealed several sources of prediction errors: a) the distance between the signature instances used for calibration and for prediction, b) an insufficient amount of calibration data, and c) for some points the assumption of linearity of the dependency was violated.

Also noticeable is the high relative prediction error for the points with low absolute values of the response time. These points were not considered as performance-critical by the architects, and prediction errors up to 100% were tolerable. For example, the architects were not that much interested that the relative prediction error for the case of two images sometimes was 70%, but rather interested in more time-consuming use cases, e.g. for 10-16 images. This observation demonstrated that the maximal relative prediction error is not a representative metric for the model quality, and the average relative error must be considered instead. Other representative metrics should be investigated as well in the future.

The last important remark concerns necessary conditions for the use of prediction intervals when predicting the performance of adapted components. These conditions among others include the normal distribution of the residual and constant residual variance. To ensure the reliability of the predictions, the architect has to carefully analyze the structure of the residual. The depth of the analysis depends on the expected degree of reliability and the effort affordable for the analysis. For simple cases, observation of residual diagnostic plots can be sufficient. For more complex cases (i.e., for quantifying the confidence intervals and the reliability of the predictions), thorough statistical tests checking the normality of the distribution should be executed.

9 Performance prediction for component compositions

9.1 Introduction

Component-based software architecting is becoming popular nowadays. Component-based solutions help manage the complexity and diversity of products and to reduce lead-time and development costs. However, the main advantages of this approach– reuse and multi-site development– often cause problems at the integration phase as the non-functional requirements are not satisfied even if the functional ones are met. Architects want to estimate the quality attributes of the product at the architecting phases. The early prediction of quality attributes helps in justifying design decisions early, which may save time and effort otherwise spent for the implementation of presumably poor performing software. In a component-base context, this estimation is difficult as these attributes often emerge from interactions between components.

We focus on one of the most critical quality attributes, performance, which characterizes how well the system performs with respect to the timing requirements [Smi00]. Typical performance metrics are end-to-end response time (latency), CPU utilization, etc. Let us explain the context of the problem for a component composition shown in Figure 9.1 (using Koala notation).

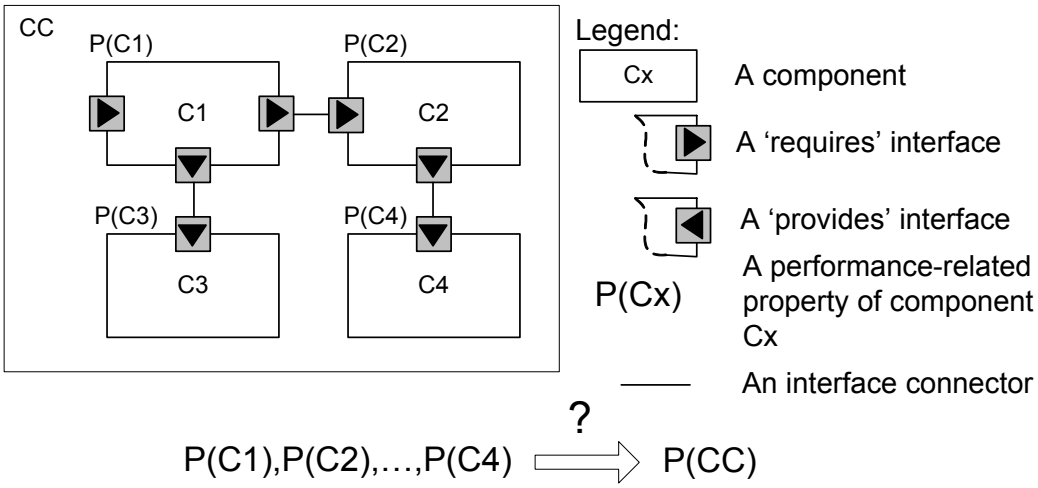


Figure 9.1: Performance prediction for a component composition

The notions are described in the legend in the right part of Figure 9.1. These notions resemble the notions used for describing Koala components and compositions thereof [OLK00]. The component composition *CC* contains four components: *C1*, *C2*, *C3*, and *C4*. The *provides* and *requires* interfaces of these components are bound by means of interface connectors. Each component has one or more performance-related properties like the processor demand of a particular component operation. The performance of the component composition is characterized by a performance-related quality attribute such as response time. We aim at finding an approach that allows the prediction of the value of the performance-related quality attribute of the composition, based on the performance-related quality attributes of the constituent components. In the general case, these attributes are not additive: they emerge not only from the performance-related properties of constituent components but also from interactions of these components. Additionally, performance estimation is hindered by the necessity to consider the scheduling of processors and access to resources.

This approach should be helpful for the architects, i.e. it should allow the quick estimation of the performance during the architecting phase. It should also support the flexible selection of an abstraction level for behavior modeling to let the architects concentrate on performance relevant details only, and to trade the estimation effort against the estimation accuracy. Additionally, the method should exploit well-known software engineering notations and must not require any additional skills from software architects.

The rest of this chapter is structured as follows. Section 9.2 discusses the essential aspects of performance estimation and introduces the relevant concepts and entities. We illustrate our approach by means of a running example. This example is a car navigation system (CNS), which is described in Section 9.3. Our approach is summarized in Section 9.4. Subsequent Sections 9.5, 9.6, and 9.7 detail this approach and exemplify it by means of the CNS example. Finally, Section 9.8 discusses the important points of the approach.

9.2 Description of component compositions

Let us introduce important entities for describing our. These entities concern various static and dynamic aspects of a composition.

9.2.1 Static aspects

Figure 9.2 presents a UML diagram that describes component composition from a structural viewpoint.

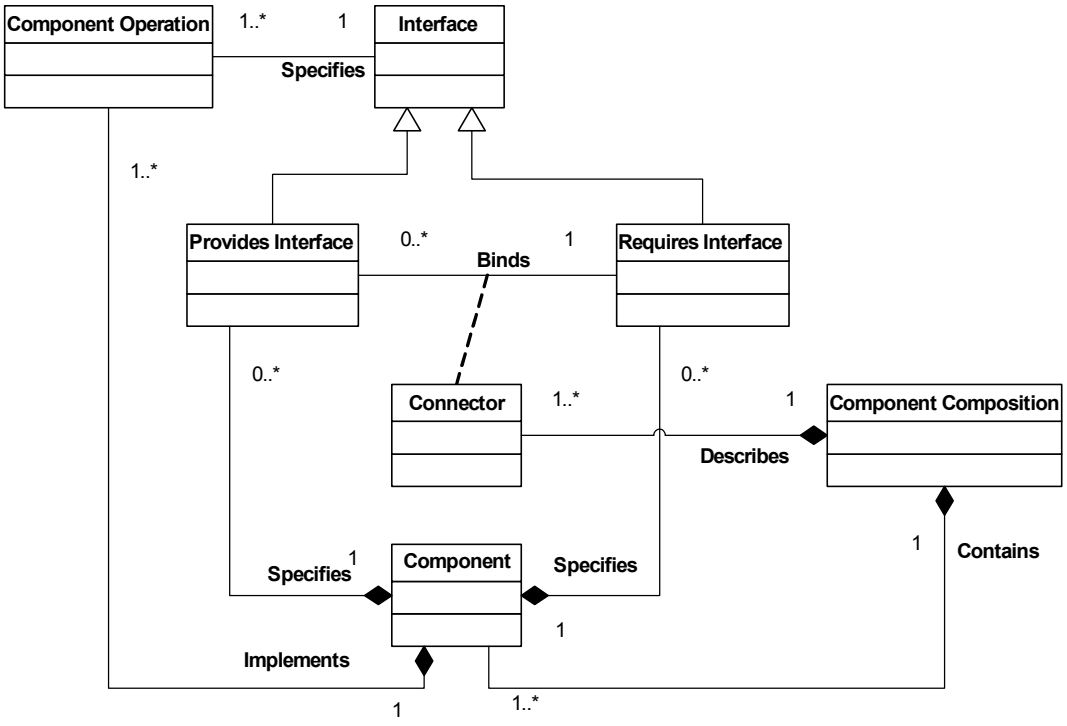


Figure 9.2: Static entities

A *component composition* contains a set of *components*. It also describes the connection between the *provides* and *requires* interfaces of the components. A *provides* interface of a component describes the functionality provided by this component to other components. The component has a number of *component operations*, which implement the functionality specified by the *provides* interfaces. The component also requires some functionality from other components. This is specified by means of *requires* interfaces. The *provides* interfaces are bound to *requires* interfaces by means of *connectors*. A

connector characterizes the way that the component operations of the *provides* interfaces are invoked through the *requires* interfaces. In this chapter, we consider only synchronous connectors. The reason is that this type of connectors is supported by many existing component models, including COM [Szy98] and Koala [OLK00], which were used in our case-studies performed.

In addition, we assume that the binding between the interfaces is known at the composition time, which is before run-time. This assumption is needed to make it possible to reconstruct activities (see Section 9.2.2), which are important for the performance analysis of component composition.

9.2.2 Dynamic aspects

Figure 9.3 shows a UML diagram of the run-time entities that are relevant for modeling the performance of component compositions.

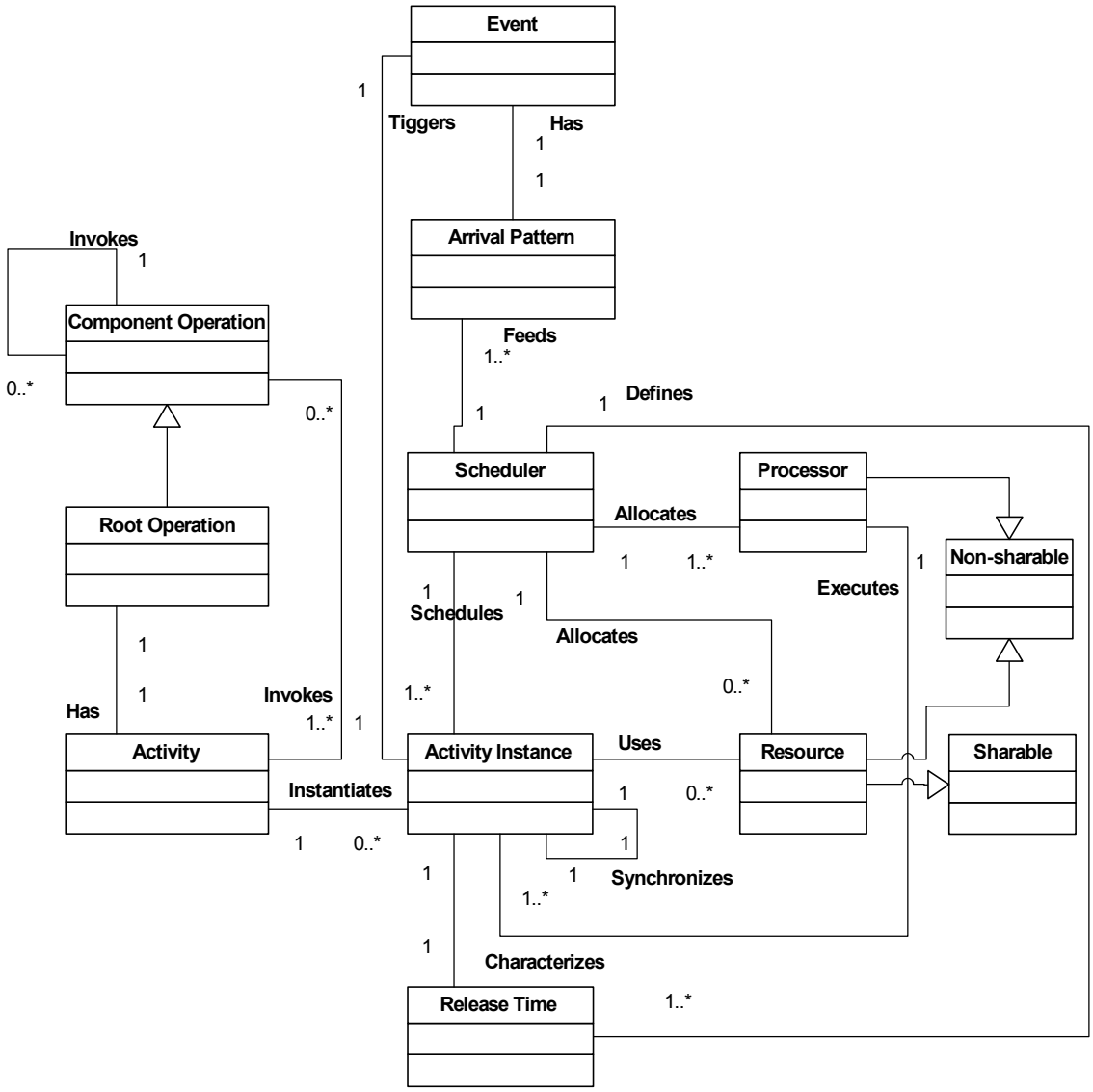


Figure 9.3: Runtime-related instances

An *activity*[1] describes a part of the behavior of the component composition under consideration. It has a *root operation*, which is invoked every time the activity is released. This root operation is an (possibly internal) operation of a particular component. Component operations may invoke each other. As mentioned in Section 9.2.1, we consider only synchronous invocation of operations. Other types of communication will need to be considered in the future.

The activity instantiates a number of *activity instances*, the units of concurrency. Activity instances are triggered by events with a certain precedence relationship and arrival pattern. The former enforce precedence constraints among a set of activity instances, whereas the latter define the instants at which the activity instances are released. An activity instance is considered to be ready to execute, when it is released. Activity instances may also synchronize with each other, for instance, on the access to shared resources.

An *arrival pattern* describes the relationship between release times of different instances of the same activity. We consider the same types of arrival patterns as the real-time literature usually does [Liu00]:

- Periodic arrivals (the inter-arrival period is specified),
- Sporadic arrivals[2] (the minimum inter-arrival interval is specified),
- Burst arrivals (the maximum number of arrivals per time unit is specified),
- Stochastic arrivals (they follow a particular probability distribution).
- A-periodic arrivals (there are no restrictions related to inter-arrival periods).

Often, several activity instances need to be executed at the same time. These activities have to be allocated to *processors* to execute upon; they can also contend for non-sharable *resources* (see [Liu00]). Processors are active entities that execute activity instances. Examples are CPUs, I/O co-processors, etc. Sharable and non-sharable resources are passive entities: an activity may need a number of resource units to be able to execute. Examples of resources are finite buffers, locks, mutexes, semaphores, etc.

The number of processors and resources is usually limited. This means that several activity instances that are eligible to execute at the same time will contend for the resources and processors. To resolve this conflict, a dedicated manager, the *scheduler*, is needed. Its role is to allocate processors and resources to activity instances. If the available amount of a particular resource is sufficient to fulfill the resource demands of all activity instances at the same time, sharable resources do not need to be treated explicitly, as there will be no resource contention. Appendix H describes processor and resource scheduling in more detail.

9.3 Description of the Car Navigation system

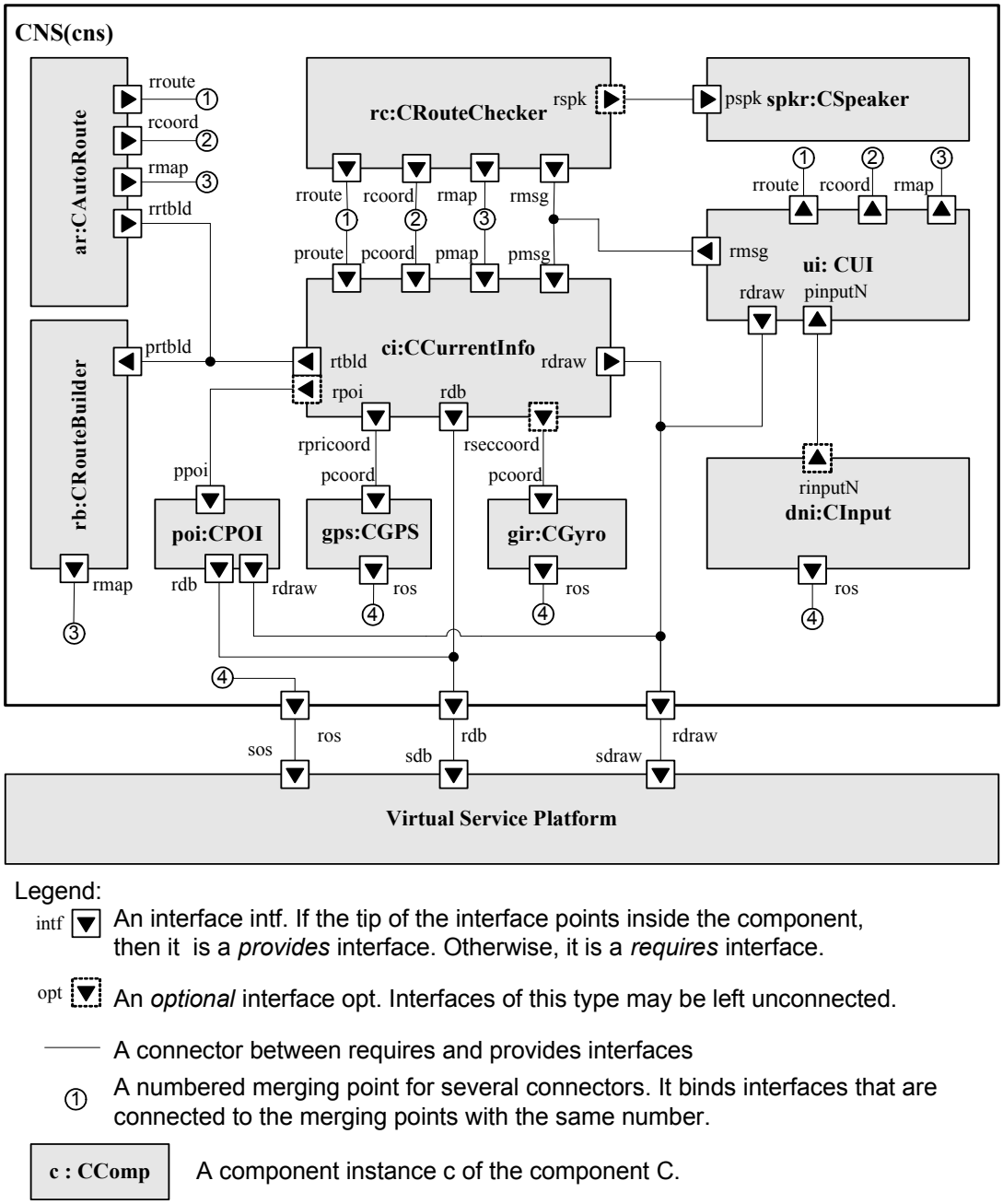
We demonstrate our approach to the prediction of the performance of component compositions by applying it to a hypothetical car navigation system (CNS). Our goal is to predict the average response time of particular activities. A car navigation system assists the driver both in constructing and in maintaining a certain route. The basic functions of this system are the following:

1. *Acquisition of route information*. Hereby, the CNS obtains the coordinates of the current geographical position from a GPS receiver or from an internal gyroscopic device. The route is constructed for the data stored in a geographical database.

[1] In the literature, also the term “logical process” or simply “process” or “task” is often used.
[2] The sporadic activities are also treated as activities that have soft deadlines.

2. *Constructing and controlling the route*. After the user specifies the departure and destination points, the system constructs a route between those points. The CNS checks if the car maintains the planned route.
3. *Interfacing with the driver*. The CNS allows the driver to choose the route in an interactive fashion. The system is responsible for displaying the map, route and current position of the car during the trip. Additionally, the system notifies the driver by means of dedicated messages (picture and/or voice) about approaching the obstacles and turns that he or she should follow. Messages are displayed in advance for the turns and crossings, where the driver should execute a maneuver.

The structure of the CNS software is shown in Figure 9.4.



Legend:

intf — An interface intf. If the tip of the interface points inside the component, then it is a *provides* interface. Otherwise, it is a *requires* interface.

opt — An *optional* interface opt. Interfaces of this type may be left unconnected.

A connector between requires and provides interfaces

① A numbered merging point for several connectors. It binds interfaces that are connected to the merging points with the same number.

c : CComp A component instance c of the component C.

Figure 9.4: The structure of the CNS

Figure 9.4 uses the same notations as the Koala component model [OLK00]. The main ones are explained in the legend part of Figure 9.4. The CNS consists of the following components:

- The "*CCurrentInfo*" component, which stores the current coordinates of the car, the current map, and the current route. This storage is implemented as shared memory and is a non-sharable resource;
- The "*CrouteChecker*" component, which observes the current coordinates and route and decides if the driver needs to be given a signal about approaching a turn;
- The "*CUI*" component, which implements the user interface with the driver. This component is responsible for presenting the relevant information to the driver, for reacting to driver's commands such as calculating the route, etc.;
- The "*CInput*" component, which implements an API for for handling input devices such as buttons;
- The "*Cspeaker*" component, which plays back a voice message to the driver, when she or he needs to make a maneuver;
- The "*CAutoRoute*" component, which checks whether the car is off the route and recalculates the route if necessary;
- The "*CRouteBuilder*" component, which constructs the route between a departure and destination point;
- The "*CPOI*" component, which manages a database of points of interests (POI) (e.g., restaurants, museums, etc) and supports the drawing of POI's on the map;
- The "*CGPS*" component, which collects the information obtained from a GPS receiver and maintains the currents coordinates;
- The "*CGyro*" component, which determines the current coordinates of the car based on the information collected from a gyroscopic device.

The components above execute on top of a Virtual System Platform (VSP). This VSP provides a number of services through the interfaces (1) "*sos*" for general-purpose operating system API, (2) "*sdb*" for storing the database with geographical information, and (3) "*sdraw*" for graphics primitives such as a '*DrawPolygon*'.

We consider the details of the CNS system that are relevant for a single use case only. The rest is omitted for the sake of brevity. The use case concerns following the route already selected by the driver. The CNS instructs the driver about upcoming maneuvers. The driver does not issue any commands to the CNS. The following activities are relevant for this use case:

- The "*DrawDispl*" activity, which is responsible for drawing the relevant information on the display. This information includes the current map, the current position, the route, messages to the driver, and etc;
- The "*RouteChecker*" activity, which periodically checks that the car follows the selected route;
- The "*UpdateCoord*" activity, which periodically updates the coordinates from the *primary* or *secondary* coordinate source. The primary source of the coordinates is represented by the "*CGPS*" component, whereas the secondary by the "*CGyro*" component. When both sources are available the primary source has the preference;
- The "*AutoRouteCalc*" activity, which recalculates the route if the car is off the route;
- The "*PlaybackSndMsg*" activity, which plays back a voice message to the driver about approaching a point where a maneuver is required;
- The "*PollKbd*" activity, which periodically polls the keyboard to check if any keys have been pressed.

These activities are started by particular components, according to Table 9.1.

Table 9.1: The mapping between components and activities. A symbol 'S' at a particular cell means that the component from the cell's row starts the activity from the cell's column. A symbols 'C' means that the activity call calls operations of the corresponding component.

	Draw-Displ	*Route-Checker*	*Update-Coord*	*Auto-Route-Calc*	*Replay-SndMsg*	*Poll-Kbd*
CCurrentInfo	C		S			
CRouteChecker		S				
CInput						S
CUI	S					
CSpeaker					S	
CAutoRoute				S		
CRouteBuilder				C		
CPOI	C			C		
CGPS			C			
CGyro			C			

Table 9.2 describes the arrival patterns of the relevant activities.

Table 9.2: The arrival patterns of the CNS activities

Activity	**Arrival Pattern**
DrawDispl	Periodic (100 ms)
RouteChecker	Periodic (200 ms)
UpdateCoord	Periodic (50 ms)
AutoRouteCalc	Sporadic (min. period 2s)
PlaybackSndMsg	Sporadic (min. period 200ms)
PollKbd	Periodic (50 ms)

This arrival pattern described a pre-calculated schedule according to the time-driven architectural pattern. The instances of the two sporadic activities— "*AutoRouteCalc*" and "*PlaybackSndMsg*"— are triggered by particular instances of the "*RouteChecker*" activity. Please notice that not more than one activity instance of these sporadic activities may be active at the same time. If the previous activity instance has not completed when the next instance is triggered, it is forced to terminate.

Some of the activities may contend for shared resources. The shared resources are all implemented within the "*CCurrentInfo*" component. These resources are shared memory objects, which describe the information about the current map, the current coordinates, the current route, and the current message being displayed. These objects must be used in a mutually exclusive manner. Therefore, the access to these objects via the "*pmap*", "*pcoord*", "*proute*", and "*pmsg*" provides interface of the "*CCurrentInfo*" component is serialized. Table 9.3 shows the interfaces through which the various activities access particular shared memory objects.

Table 9.3: The use of shared resources by particular activities

Interface/ Activity	*DrawDispl*	*RouteChecker*	*UpdateCoord*	*AutoRouteCalc*
pmap	X	X		X
pcoord	X	X	X	X
proute	X	X		X
pmsg	X	X		

9.4 Performance modeling of component compositions

This section presents our hierarchical view for the modeling of component compositions (see Figure 9.5). The complexity of the resulting performance model increases towards the top of the pyramid.

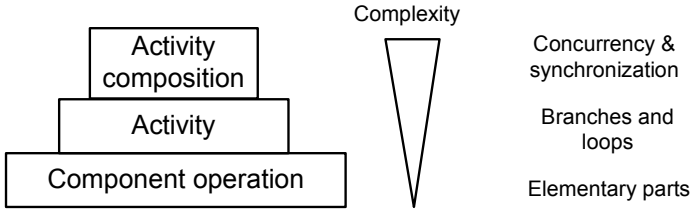


Figure 9.5: Building composition models

The construction of the performance model of a component composition starts with performance models for the component operations. These operations are described by models for predicting the processor and resource demand. These prediction models are constructed by means of the APPEAR method (see Chapter 5). Section 9.5 details this process.

Activities call component operations to implement their functionality. Each activity is modeled by an activity control flow graph[3] (CFG). The activity models are used to estimate the processor and resource demand of activity instances. At this level of hierarchy, the processor and resource demand is considered to be independent from interactions with other activity instances. Section 9.6 describes the modeling of individual activities in more detail.

Finally, at the top of the pyramid, we build a model of the activity composition. This model describes the concurrency and synchronization between different activity instances. The model not only combines the resource and processor demands of all activities running concurrently, but also accounts for effects of scheduling such as blocking and preemption. Section 9.7 details the modeling of the concurrent execution of activities.

For isolated component operations and activities, we consider the performance in terms of processor and resource demands. For the composition of activities, the performance is estimated in terms of usual performance measures like response time, processor utilization and throughput.

9.5 Modeling of component operations

The processor and resource demand of a component operation is estimated by means of a prediction model constructed by the APPEAR method (see Chapter 5). This model reflects the correlation between the resource demand and particular performance-relevant parameters of the component operation. These parameters can be the input parameters of the operation, the diversity parameters of the component, and etc. We call the vector of these parameters a *signature type*. The values of this vector are *signature instances*.

As Chapter 5 explains, the prediction model P_o is a function over the signature type $\vec{S}_o$:

$$P_o = f\left(\vec{S}_o\right) \in \mathbb{R}. \tag{9.1}$$

[3] Control flow graphs are notions similar to execution graphs and UML activity diagrams.

It returns a Real value, which is interpreted as a value of the resource or processor demand of the component operation *o*. The prediction model is fitted to the performance measurements of the component implementation. For this purpose, one of many existing regression techniques can be applied. The fitting process is detailed in Sections 5.6.1 and 5.10.

In Chapter 5, we also explain that a component operation may invoke services of the virtual service platform (VSP). For instance, the VSP services needed by the "*DrawMap*" component operation (e.g. the "*DrawPolygon*" service) are made via the "*pdraw*" interface, which is connected to the VSP (see Section 9.3). We consider the processor/resource demand of the VSP as a part of the processor/resource demand of the operation. This view is possible because of the stability of the VSP.

Parameters characterizing the interactions between the component operation and the environment[4] of the component are a good candidate for defining the signature type. For instance, a signature type may consist of the number of times a particular service is executed during the invocation of the component operation.

In addition to the prediction model, a simulation model may need to be built for extracting signature instances for the prediction model. The role of the simulation model is to characterize use cases of the component operation in terms of *signature type*. These use cases characterize how the operation is used by its environment, particularly, which parameters are inputted to the operation by its callers. Let us assume that the signature type of the operation relates to the use of VSP services. In this case, the simulation model represents that part of the behavior of the component operation that maps the use cases to these service calls (see Section 5.3 and 5.4). In other cases, the signature can also be related to the internal calculations of the component operation. This however may require a "white-box" model of component internals, which may involve too many implementation details.

9.5.1 An example of the simulation model for a component operation

Let us demonstrate how such a simulation model may look like by constructing this model for the "*DrawMap*" component operation. This component operation has the following input parameters:

- "*DrawingMode*", which selects one of two drawing modes: "*Fast*" or "*Normal*". In the former mode, the polygons that represent various map objects such as buildings, roads, etc. are drawn without filling. In the latter mode, the polygons are filled with a particular color, depending on the type of the object that the polygon represents. The filling requires additional processor demand.
- "*Map*", which describes the current map. The "*Map*" object contains a topographically[5] sorted list of map objects. Each map object represents an entity such as a road, building, a segment of a river or lake, etc. The map object contains certain geographical information (e.g., a name) and the description of the polygon that represents this object. The map object contains also information about the region it covers in terms of geographical coordinates.

[4] The environment of a component is other components that are bound to this component through interfaces.
[5] By topographically sorting we mean the following. Map objects can be nested: for instance, a village map object may include a number of house map objects. The larger map objects, such as the village one, must be drawn before the smaller ones. For this reason, these objects are stored in the beginning of the list. This arrangement of location of map objects is called the topological sorting.

- "*Region*", which describes a rectangular part of the map to be drawn on the screen. It specifies the geographical coordinates of the left-top and right-bottom corner.

Drawing many polygons may require a substantial processor demand. To reduce this demand, the "*DrawMap*" operation caches the previously drawn polygons in an off-screen buffer (see Figure 9.6). It is not necessary to redraw the polygons that are still visible in the new region (specified by the "*Region*" parameter). Instead, the already drawn part can be directly copied from the off-screen buffer, accounting for a possible shift of the region.

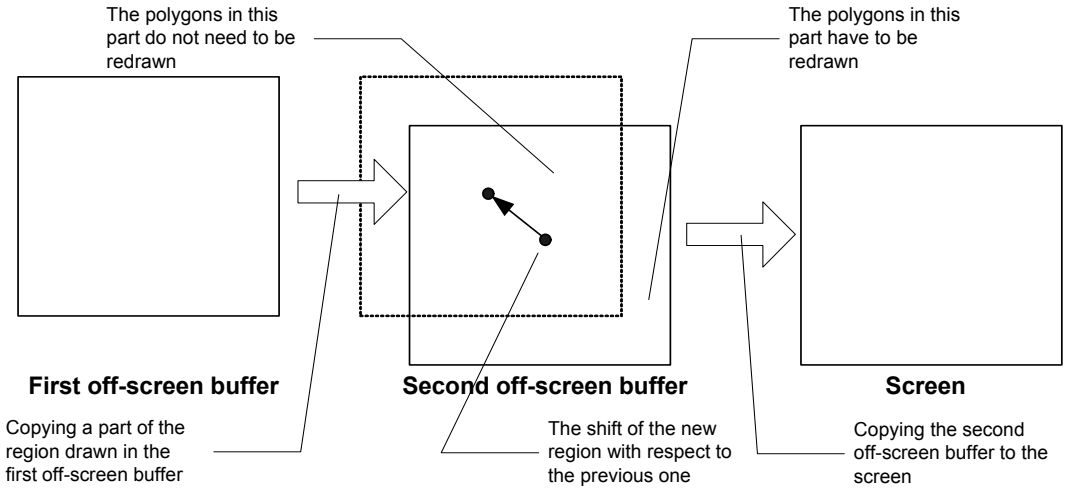


Figure 9.6: Copying the off-screen buffers to the screen

If this shift is too large, copying a part of the previous region may be not efficient anymore. The "*DrawMap*" operation displays then all polygons from the new region. Displaying all polygons may also be required, if the drawing mode, specified by the "*DrawingMode*", for the new region differs from the drawing mode of the previously displayed region.

Summarizing, the variation of processor demand of the "*DrawMap*" operation is primarily determined by the following factors:

1. The number of polygons to be drawn ("*#Polygons*");
2. The drawing mode (*Mode*=0 means the "*Fast*" drawing mode, whereas *Mode*=1 means the "*Normal*" drawing mode);
3. The possible copying of the previously drawn polygons from the first off-screen buffer to the second one (*ShiftAndCopy*=0 means that no copying takes place, whereas *ShiftAndCopy*=1 indicates the fact of copying).

The signature parameters are given for each of these factors in brackets.

The "*DrawMap*" operation maintains two sets (lists) of the polygons: "*PrevPolygons*" and "*NewPolygons*". The former contains the description of the polygons that has been drawn by the previous invocation of the "*DrawMap*" operation. The latter comprises the polygons that need to be displayed for the new region. Figure 9.7 presents the simulation model of the "*DrawMap*" operation in form of a state machine.

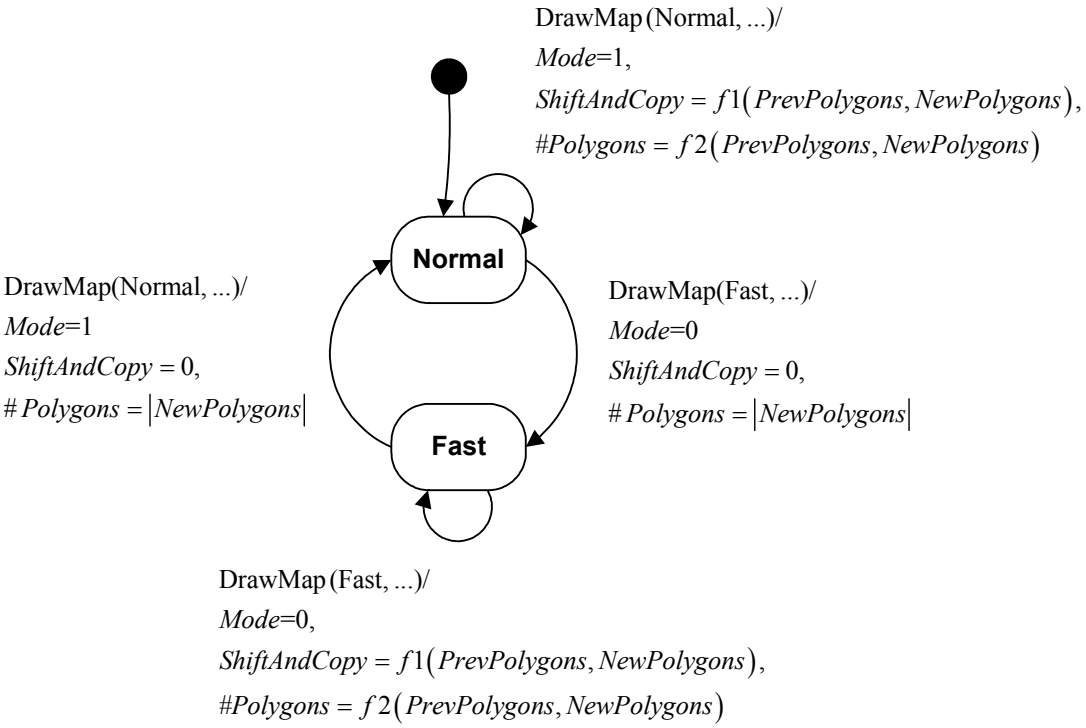


Figure 9.7: The simulation model of the '*DrawMap*' component operation

The states reflect which value of the "*DrawingMode*" parameter was passed to the "*DrawMap*" operation during its previous invocation. The transitions are marked with pairs of input events and actions. The input events correspond to the invocation of the "*DrawMap*" operation with various values of its "*DrawingMode*" input parameter. The actions specify the expressions for calculating values of the signature parameters: "*Mode*", "*ShiftAndCopy*", and "*#Polygons*". The operator $|x|$ denotes the cardinality of a set *x*, that is, the number of elements in this set.

Function $f1$ calculates whether the copying of previously drawn polygons is needed or not and is defined as follows:

$$f1(prev, new) = \begin{cases} 1, & if\ |new \cap prev| \geq Threshold; \\ 0, & otherwise. \end{cases} \tag{9.2}$$

Function $f2$ calculates how many polygons need to be drawn and is defined as follows:

$$f2(prev, new) = \begin{cases} |new \setminus prev|, & if\ |prev \cap new| \geq Threshold; \\ |new|, & otherwise. \end{cases} \tag{9.3}$$

In Formulas (9.2) and (9.3), "*prev*" and "*new*" denote the sets of polygons that need to be displayed at the previous and the current invocation of the "*DrawingMap*" operation. The "*Threshold*" constant defines the critical number of polygons, such that drawing these polygons by means of the "*DrawPolygon*" service is slower than copying and shifting the previously drawn polygons from the off-screen buffer.

"*PrevPolygons*" and "*NewPolygons*" depend on the information from the map. This information must either be guessed by the architects or extracted from the actual map. The latter requires the modeling of a part of the software responsible for parsing map information. This part must be included into the simulation model to be able to calculate precise signature instances.

9.5.2 An example of the prediction model for a component operation

We use the example of the "*DrawMap*" operation to demonstrate a prediction model that is constructed by applying the APPEAR method. The signature type of the "*DrawMap*" operation is explained in Section 9.5.1. As we assume a linear dependence between the processor demand of the "*DrawMap*" operation and the signature parameters, the prediction model has the following form:

$$D = \alpha_0 + \alpha_1 \cdot ShiftAndCopy + \alpha_2 \cdot \#Polygons + \alpha_3 \cdot \#Polygons \cdot Mode. \qquad (9.4)$$

In Formula (9.4), D denotes the processor demand of the "*DrawMap*" operation. α_i are regression coefficients. The remaining variables are signature parameters. Please notice that the α_3 regression coefficient corresponds to the interaction term $\#Polygons \cdot Mode$. This interaction term accounts for additional processor demand required for filling each polygon. To estimate the values of the α_i regression coefficients, prediction model needs to be fitted and validated in the way explained in Sections 5.6.1 and 5.10.

9.6 Modeling of activities

The processor/resource demand of an activity depends on the following factors:

1. The processor/resource demands of the component operations that are invoked by this activity,
2. The control flow of the activity. Depending on input parameters, states and diversity parameters of particular components, and etc., the activity may invoke various component operations with different values of input parameters for different activity instances.

The first factor was addressed in Section 9.5. This section focuses on the second factor. To model the control flow within of an activity, we adopt a slightly modified notion of control flow graph (CFG). Traditional CFGs are summarized in Appendix I, whereas a modified version thereof is detailed in Section 9.6.1.

9.6.1 Description of the control flow of component operations

We aim at describing only the performance-relevant control flow information of a component operation. Traditional CFGs (see Appendix I) contain too much information: considering all basic blocks may easily lead to combinatorial explosion. Moreover, the prediction model constructed for the component operation already covers its processor/resource demand. This fact implies that the information about the resource demands of basic blocks is redundant. However, it is important to preserve the information about the invocation of other component operations, as they may significantly influence the total resource demand of the entire activity. This goal can be achieved by keeping the following information from the CFG of the operation under consideration:

- *Invocation nodes* (basic blocks), which describe the invocation of component operations,
- Nodes and edges, which describe the control flow that may lead to the the invocation of other component operations. We call these nodes *branching nodes* or *loop headers*,
- *Entry* and *exit blocks* (see also Appendix I).

We call a graph obtained this way the *reduced CFG* of an operation. Each component operation needs to be attached such a reduced CFG to specify how often and which operations of other components it invokes. The *reduced CFG* can be constructed automatically, based on the analysis of the source code of the component (see Appendix G).

Figure 9.8 shows an example of such a reduced CFG for the "*DrawMap*" component operation, provided through the "*rmap*" interface of the "*CCurrentInfo*" component (see Figure 9.4).

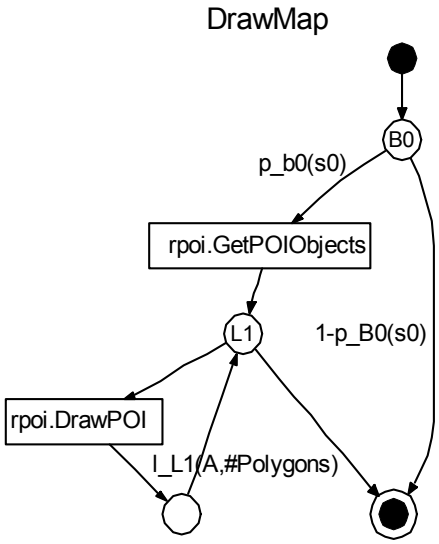


Figure 9.8: The CFG of the "*DrawMap*" component operation

The circles with the "*L1*" and "*B0*" labels are a loop header and a branching node, respectively. Finally, the rectangles with a label inside indicate the invocation of the component operation with the respective identifier. The identifier of the callee consists of two parts. The first part, before the point, is the name of the *requires* interface through which the callee is invoked. The second part is the name of the operation. The "*DrawMap*" operation invokes other operations through the "*rpoi*" interface.

The decision at the "*B0*" branching depends on whether the optional "*rpoi*" interface is connected. If this is the case, the control flow follows the left branch. Otherwise it follows the left one.

The "*GetPOIObjects*" operation returns a list of POI objects (see Section 9.3). The loop "*L1*" iterates through this list, calling the "*DrawPOI*" operation for each POI object to display it. The "*DrawPOI*" operation also calls a number of VSP services, but they are not depicted in this CFG, as their resource/processor demand is covered by the prediction model (see Section 9.5).

In Figure 9.8, the arrows denote the edges of the CFG, which describe the possible control flow between a pair of the CFG nodes. Consider a *branching node*. Its outgoing edges are assigned branching probabilities, depending on the respective signature type. The construction of a prediction model for branching probabilities is explained in Section 9.6.2. The sum of the probabilities at all outgoing edges of a branching node must equal 1, indicating that the control flow will pass one of the edges for sure. The labels that specify the formulas for calculating are attached to the corresponding edges. For instance, these probabilities are "*p_B0(s0)*"and "1-*p_B0(s0)*" for the "*B0*" branching node from Figure 9.8, with *s0* being a single signature parameter for this branching node.

Now consider a *loop header*. This type of node is used to model repetitive control flow structures such as various types of loops. The loop header is the first node being passed at

the first iteration of a loop. The incoming edge of this loop header is the loop's back-edge. This back-edge indicates the direction of control flow to complete the loop. The label assigned to the back-edge is the iteration count that is described by a formula. In Figure 9.8, the loop starts from the "*L1*" loop header and iterates for "*l_L1(A,#Polygons)*" times, with *A* and *#Polygons* being a signature parameter for the loop. The construction of a prediction model for loop counts is explained in Section 9.6.2.

Figure 9.9 shows another example of a reduced CFG. It describes the "*DrawDisplay*" operation implemented by the "*CUI*" component (see Figure 9.4). The operation is responsible for displaying the content of the entire screen.

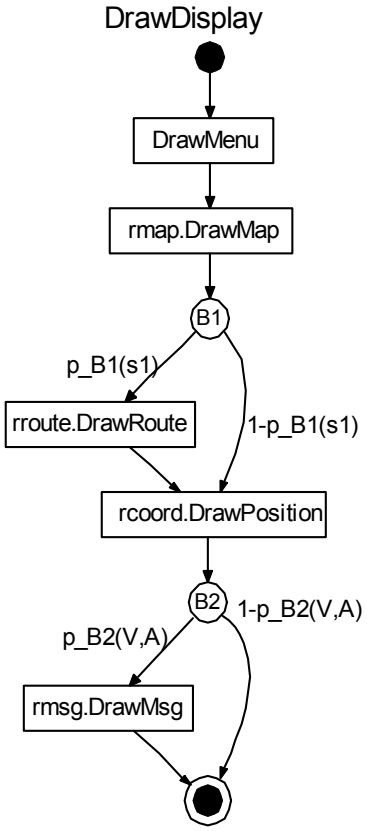


Figure 9.9: The CFG of the '*DrawDisplay*' operation

In Figure 9.9, the same notations are used as in Figure 9.8. The "*DrawDisplay*" operation invokes one internal component operation "*DrawMenu*" and four component operations—"*rmap.DrawMap*", "*rroute.DrawRoute*", "*rcoord.DrawPosition*", and "*rmsg.DrawMsg*"—through the corresponding 'requires' interfaces. Notice that the "*DrawRoute*" and "*DrawMsg*" operations will be invoked not for all invocations of the "*DrawDisplay*" operation, as they are located at the "*B1*" and "*B2*" branch, respectively. The corresponding branching probabilities equal "*p_B1(s1)*" and "*p_B2(V,A)*", respectively. The *V*, *A*, and *s1* arguments are signature parameters.

9.6.2 Prediction model for branching probabilities and iteration counts

In Section 9.6.1, we showed that the control flow of a component operation can be modeled by a *reduced CFG*. The control flow may follow different execution paths the for the following reasons:

- The input parameters of the component operation;
- The diversity parameters of the component;

- The return values of the operations of other components called via the *requires* interfaces.

The path is defined by the results of expressions over the items described in the list above.

We introduce branching probabilities and loop counts to describe the possible variation of the control flow. We treat branching probabilities and loop counts as functions over particular signature types. The signature parameters are supposed to capture the influence of the items from the list above on the control flow and, consequently, on the branching probabilities and loop counts. Please notice that this signature type may be completely unrelated to the use of service calls.

In case this influence is deterministic, we suggest the construction of an analytical formula. For example, consider the "*B0*" branching node of the reduced CFG of the "*DrawMap*" operation (see Figure 9.8). The left branch is taken when the "*rpoi*" optional requires interface is connected. In the opposite case, the right branch is taken. Figure 9.4 shows that the "*rpoi*" interface is connected. Therefore, we may consider that the left branch will be always taken and that

$$p\_B0\left(\vec{s}_{B0}\right) = 1. \tag{9.5}$$

In Formula (9.5), p_B0 denotes the branching probability for the "*B0*" branching node, and $\vec{s}_{B0}$ is the corresponding signature instance.

Another example of a trivial formula is the branching probability of the "*B1*" branching node of the reduced CFG depicted in Figure 9.9. The branching decision is determined by the state of the "*CUI*" component that indicates if the current route needs to be displayed. In the context of this chapter, we analyze only the case when the route is always displayed. This implies

$$p\_B1\left(\vec{s}_{B1}\right) = 1. \tag{9.6}$$

In Formula (9.6), p_B1 denotes the branching probability for the "*B1*" branching node, and $\vec{s}_{B1}$ is the corresponding signature instance.

In practice, the influence of the factors mentioned at the beginning of this section will be non-deterministic due to the unknown values of particular parameters. In this case, a prediction model for branching probabilities and loop counts can be constructed by means of the APPEAR method (see Chapter 5). The application of the APPEAR method implies that a simulation model is constructed for extracting the signature instances. Based on the values of various parameters, the simulation model calculates the values of the signature parameters for the prediction model.

As an example of a complex branching decision, let us consider the "*B1*" branching node of the "*DrawDisplay*" operation (see Figure 9.9). At this node, a decision is taken whether a message about approaching a turn needs to be displayed. The moment of displaying the message depends on the current speed and the distance to the next turn.

The prediction model for the "*B2*" branching node calculates the probability p_B2 of taking the branch with or without calling the "*DrawMsg*" operation through the "*rpoi*" operation. The prediction model is fitted over a signature type:

$$\vec{S}_{B2} = \left(V, A\right). \tag{9.7}$$

In Formula (9.7), V denotes the current speed of the car, and A is the type of area where the car is being driven. The second parameter is categorical, and it can have two values: "*City*" or "*Highway*". On one hand, the higher the speed, the faster the car can reach the turn that

it needs to take. On the other hand, it is more likely to encounter the turn in the city than on the highway.

As the branching probability for "*B2*" branch is a binary variable and the corresponding branching probability "p_B2" must lie in the interval from 0.0 to 1.0, it is impossible to construct a prediction model for "p_B2" by means of traditional linear regression. (Traditional linear regression constructs models that can supply values beyond the interval 0.0 to 1.0, but the probabilities must always lie within this interval.) For the regression of prediction models with binary responses that have a distribution from the exponential family (Poisson, binominal, etc), one can, however, use generalized linear regression [MN97]. This regression introduces the so-called logit link function η_1

$$\eta_1\left(\vec{S}\right) = \log\left(p\left(\vec{S}\right) \Big/ \left(1 - p\left(\vec{S}\right)\right)\right) \tag{9.8}$$

to model binary responses. In Formula (9.8), $p\left(\vec{S}\right)$ denotes the probability of success, which is the probability of taking a particular branch in our case. $\vec{S}$ is the signature type for the branching probability under consideration.

The prediction model for the branching probability "p_B2" can be described now by the following formulas:

$$\eta_1\left(A,V\right) = \alpha_0 + \alpha_1 \cdot V + \alpha_2 \cdot A; \quad \eta_1(A,V) = \log\left(\frac{p\_B2(V,A)}{1-p\_B2(V,A)}\right). \tag{9.9}$$

In Formula (9.9), α_i are regression coefficients; V and A are signature parameters. Note that this model linear in regression coefficients, although in signature parameters it is not linear.

There exist a number of tools that support generalized linear regression, e.g., the S-Plus tool [KO02]. These tools may be used to fit the prediction model defined by Formula (9.9). These tools not only calculate the values of the regression coefficients, but also give figures about the overall quality of the regression.

As mentioned above, loop counts can also by described by means of a prediction model over particular signature type. The construction of a prediction model in the form of an analytical formula is performed in the same way as for branches. Otherwise, the prediction model has to be constructed by using statistical regression techniques. Also in this case the generalized linear regression can be applied. However, the logit link function cannot be used as for branching probabilities. Instead, another link function has to be used. Response variables such counts often follow the Poisson distribution, and we use the η_2 Poisson link function

$$\eta_2\left(\vec{S}\right) = \log\left(\overline{L}\left(\vec{S}\right)\right). \tag{9.10}$$

In formula (9.10), $\overline{L}\left(\vec{S}\right)$ is the average number of loop iterations, which coincides with the parameter of the underlying Poisson distribution, and $\vec{S}$ is the signature type for the loop under consideration.

For example, let us consider the "*L1*" loop header (see Figure 9.8) for the "*DrawMap*" component operation. We assume that the average number of iterations of this loop relates to the type *A* (see above) of area where the car moves and to the number of the surrounding geographical objects. The number of these geographical objects depends on the signature parameter "*#Polygons*" explained in Section 9.5.1. It is more likely to move

across a point of interest in an area with many buildings than in an area with only few ones. Summarizing, the signature type $\vec{S}_{L1}$ of the iteration count of the "*L1*" loop has the following form:

$$\vec{S}_{L1} = \left(A, \#Polygons \right). \tag{9.11}$$

The prediction model $l\_L1\left(A, \#Polygons \right)$ for the "*L1*" loop is described by the following formula:

$$\begin{aligned} &\eta_2 \left(A, \#Polygons \right) = \alpha_0 + \alpha_1 \cdot A + \alpha_2 \cdot \#Polygons + \alpha_3 \cdot A \cdot \#Polygons; \\ &\eta_2 \left(A, \#Polygons \right) = \log\left(l\_L1\left(A, \#Polygons \right) \right). \end{aligned} \tag{9.12}$$

In Formula (9.12), α_j denotes the regression coefficients. The rest of variables are explained above. This prediction model can be fitted by any statistical tool that supports generalized linear regression with the Poisson link function (e.g. the S-Plus regression tool [KO02]).

9.6.3 Description of the control flow of an activity

For estimating the total processor/resource demand of an activity, its control flow of an activity needs to be described. The estimation process is further explained in Section 9.6.4. The component model described in Section 9.2 implies that the control flow of an activity is only known after composition time, when all interfaces are bound. The control flow of the activity is defined by the control flows of all the callee component operations (see e.g. Figure 9.10). Note that it is necessary to treat only the component operations that are implemented by the involved components. The contribution of VSP services are covered by the prediction models of individual component operations (see Section 9.5) and do not need to be represented explicitly in the context of the control flow modeling.

As mentioned before the execution of an activity starts from a particular (possibly internal) operation called the *root operation* of the activity. This root operation may invoke a number of component operations through the 'requires' interfaces of the component that implements this operation. These invoked operations may also call other component operations, and so on and so forth. At a certain depth of the call hierarchy, no operation will invoke other component operations. In practice, the modeling of call graph might stop earlier as the lowest level operations usually are small and have a low processor/resource demand. Their effect can be covered by the regression model.

We define the control flow graph of an activity as a hierarchical graph and refer is as an *activity CFG*. An example of such an activity CFG is shown for the "*DrawDispl*" activity (see Section 9.3) in Figure 9.10.

The *root operation* of this activity is the "*DrawDisplay*" operation, which is described in Section 9.6.1. The "*DrawDisplay*" operation invokes one internal component operation "*DrawMenu*" and four component operations— "*rmap.DrawMap*", "*rroute.DrawRoute*", "*rcoord.DrawPosition*", and "*rmsg.DrawMsg*"— through the corresponding 'requires' interfaces. Only the "*rmap.DrawMap*" operation invokes operations of other components. The remaining operations invoke only VSP services such as reading the data from the geographical database or drawing graphics. The use of these services is addressed by the APPEAR prediction models of the remaining operations. For these reasons, the activity CFG for the "*DrawDispl*" activity includes the reduced CFGs of the "*DrawMap*" and "*DrawDisplay*" operations only.

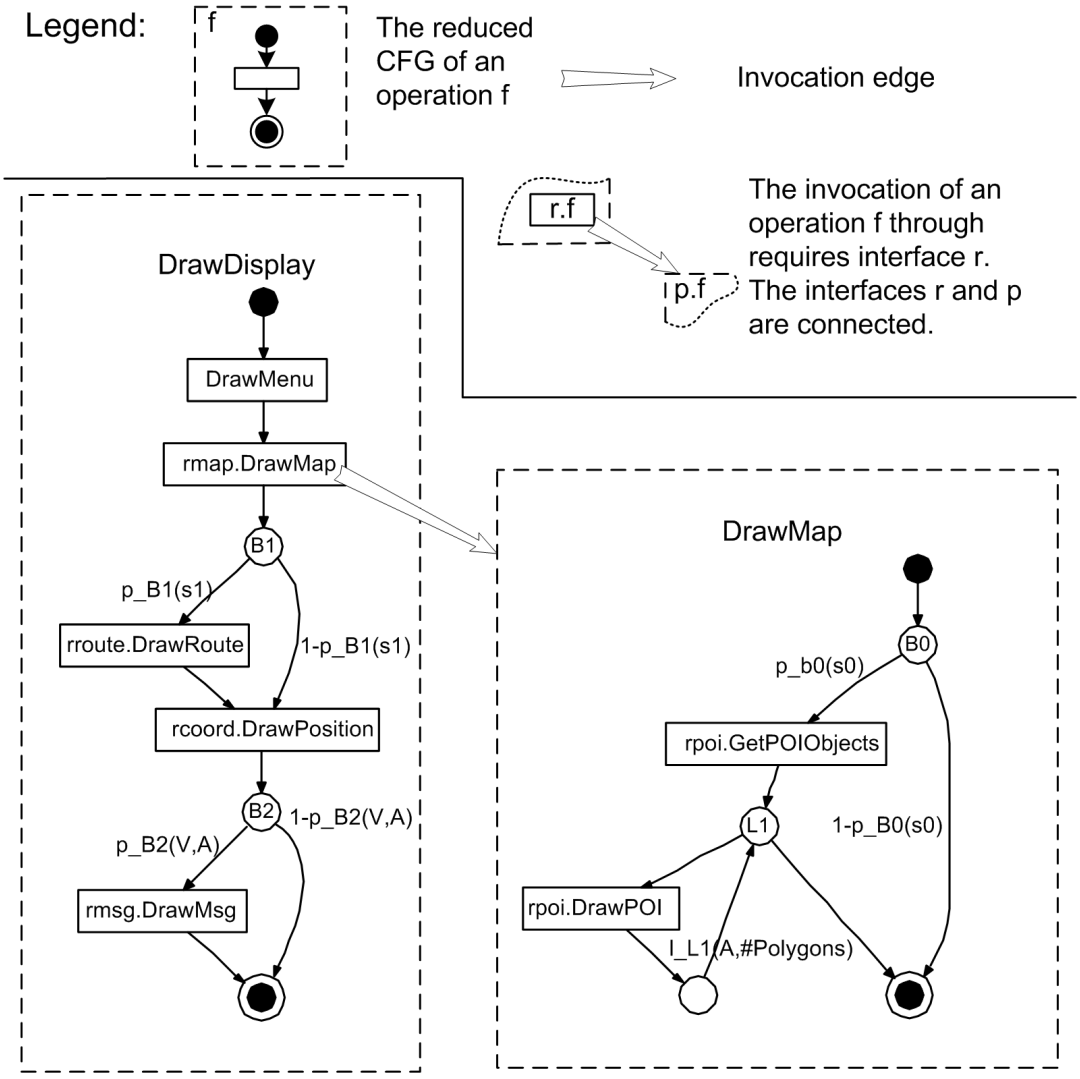


Figure 9.10: The activity CFG for the "*DrawDispl*" activity

A more extended example of an activity CFG is presented in Appendix J.

9.6.4 Estimation of the resource/processor demand of an activity

Considering the dependencies between different nodes and edges of the activity CFG (see Section 9.6.3), different modeling techniques can be used to estimate the resource demand of an activity (see Figure 9.11).

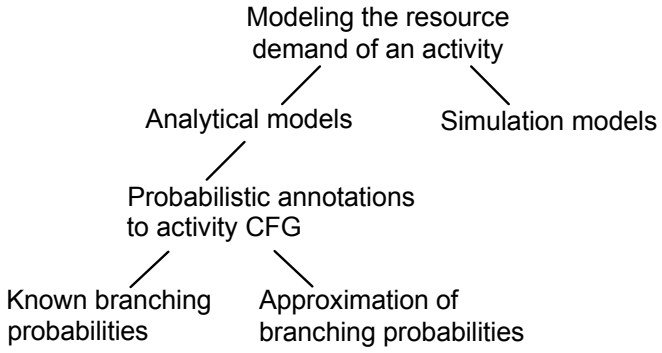


Figure 9.11: Modeling of the control flow

The simulation of the CFG of an activity may be required, when it is impossible to capture in an analytical way the complex dependencies of the processor/resource demand of the activity on various performance-relevant factors such as input parameters, component states, etc. The simulation may also require the attachment of additional annotations to the edges of the activity CFG. These annotations may include guards for branches and loops, expressions over input parameters, etc. The simulation of an activity CFG is only feasible for a small subset of activities, as the construction of the simulation models is a big effort.

The construction of analytical formulas to estimate the resource/processor demand of an activity is also based on the activity CFG. We make the assumption of statistical independence between the processor/resource demands of the constituent component operations, branching probabilities, and iteration counts to be able to construct simple estimation formulas. If this assumption is violated, simulation has to be used to estimate the resource demand of the activity instance. In the literature (e.g. [MR03]), approaches for checking statistical independence are discussed.

Under this assumption, the estimation formulas can be constructed automatically from the activity CFG. The idea behind the algorithm is as follows. The activity CFG of an activity *A* is traversed starting from its *root operation o,* which may invoke a number of component operations through the *requires* interfaces. For each invoked operation *i*, we define a sub-activity A_i, which has the operation *i* as its root operation. In these terms, the processor/resource demand of the activity *A* is defined by the processor/resource demand of the root operation *o* and the processor/resource demands of the defined sub-activities A_i. However, their processor/resource demands must be considered with respect to the control flow context: if a particular component operation *i* is invoked from a branch or a loop, its processor/resource demand must be multiplied by the corresponding branch probability or loop count. Summarizing, the formula is as follows:

$$D(\vec{s}_A) = D_o(\vec{s}_o) + \sum_{i \in Callees(o)} \left(D(\vec{s}_{A_i}) \cdot \prod_{j \in Branches(i)} p_j(\vec{s}_j) \cdot \prod_{k \in Loops(i)} l_k(\vec{s}_k) \right). \qquad (9.13)$$

In formula (9.13), $\vec{s}_A$ denotes a signature instance for estimating the processor/resource demand $D(\vec{s}_A)$ of the activity *A*, including all its sub-activities. Similarly, $\vec{s}_{A_i}$ are signature instances estimating the processor/resource demands $D(\vec{s}_{A_i})$ of the sub-activities A_i. Branching probabilities and loop counts are denoted by p_j and l_k, respectively, with the corresponding signature instances being $\vec{s}_j$ and $\vec{s}_k$. The signature instance $\vec{s}_A$ includes the signature instance $\vec{s}_o$ of the root operation *o*, the signature instances $\vec{s}_{A_i}$ of the sub-activities, and the signature instances $\vec{s}_j$ and $\vec{s}_k$ of branches and loops:

$$\vec{s}_A = (\vec{s}_o, ..., \vec{s}_{A_i}, ..., \vec{s}_j, ..., \vec{s}_k, ...). \qquad (9.14)$$

The processor/resource demand D_o is calculated by a prediction model, constructed by the APPEAR methods, as described in Section 9.5. *Callees(o)* denotes the set of callee operations of the *o* operation *o*. *Branches(i) and Loops (i)* denote sets of branching nodes and loop headers, respectively, that define the control flow context of the callee operations *i*. *Callees(o)*, *Branches(i),* and*, Loops(i)* can be deduced automatically from the reduced CFG of the operation *o*.

Formulas (9.13) and (9.14) need to be applied recursively to each sub-activity, starting from the *root operation* of the activity. The processor/demand of this root operation,

accounting for all its callees, coincides with the processor/demand of entire activity. The *simulation model of an activity* consists of the simulation models of all operations (see Section 9.5).

We applied Formula (9.13) to the CFG of the "*DrawDisplay*" activity (see Figure 9.10) and obtained the following formulas. The processor demand $D_{DrawMapA}$ of the "*DrawMapA*" sub-activity with the "*DrawMap*" root operation, (see Figure 9.8) is as follows:

$$\begin{aligned} D_{DrawMapA}\left(\vec{s}_{L1},\vec{s}_{B0},\vec{s}_{\text{DrawMap}},\vec{s}_{\text{DrawPOI}},\vec{s}_{\text{GetPOIObjects}}\right) = D_{\text{DrawMap}}\left(\vec{s}_{\text{DrawMap}}\right) + \\ l\_L1\left(\vec{s}_{L1}\right)\cdot p\_B0\cdot\left(\vec{s}_{B0}\right)\cdot D_{\text{rpoi.DrawPOI}}\left(\vec{s}_{\text{DrawPOI}}\right) + \\ p\_B0\left(\vec{s}_{B0}\right)\cdot D_{\text{rpoi.GetPOIObjects}}\left(\vec{s}_{\text{GetPOIObjects}}\right). \end{aligned} \tag{9.15}$$

In Formula (9.15), $l\_L1\left(\vec{s}_{L1}\right)$ is the iteration count variable for the "*L1*" loop, and $\vec{s}_{L1}$ is a signature instance for this loop. $p\_B0\left(\vec{s}_{B0}\right)$ denotes the branching probability for the "*B0*" branching node; $\vec{s}_{B0}$ is the corresponding signature instance. $D_{\text{rpoi.DrawPOI}}$ and $D_{\text{rpoi.GetPOIObjects}}$ are the processor demands of the "*DrawPOI*" and "*GetPOIObjects*" operation, respectively, and their callees. $\vec{s}_{\text{DrawPOI}}$ and $\vec{s}_{\text{GetPOIObjects}}$ are signature instances for these operations. Finally, $D_{\text{DrawMap}}\left(\vec{s}_{\text{DrawMap}}\right)$ is the resource demand contribution of the "*DrawMap*" operation itself. This contribution is calculated by the prediction model, constructed by means of the APPEAR method. This prediction model is explained in Section 9.5.2.

The processor demand $D_{\text{DrawDispl}}$ of the "*DrawDispl*" activity equals to the sum of the processor demands of its root operation (see Figure 9.9) and the other operations that it invokes.

$$\begin{aligned} D_{\text{DrawDispl}}\left(\vec{s}_{B2},\vec{s}_{B1},\vec{s}_{\text{DrawDisplay}},\vec{s}_{\text{DrawMsg}},\vec{s}_{\text{DrawPos}},\vec{s}_{\text{DrawRoute}},\right. \\ \left.\vec{s}_{L1},\vec{s}_{B0},\vec{s}_{\text{DrawMap}},\vec{s}_{\text{DrawPOI}},\vec{s}_{\text{GetPOIObjects}}\right) = \\ D_{\text{DrawDisplay}}\left(\vec{s}_{\text{DrawDisplay}}\right) + \\ D_{\text{DrawMapA}}\left(\vec{s}_{L1},\vec{s}_{B0},\vec{s}_{\text{DrawMap}},\vec{s}_{\text{DrawPOI}},\vec{s}_{\text{GetPOIObjects}}\right) + \\ p\_B1\left(\vec{s}_{B1}\right)\cdot D_{\text{rroute.DrawRoute}}\left(\vec{s}_{\text{DrawRoute}}\right) + \\ D_{\text{rmap.DrawPosition}}\left(\vec{s}_{\text{DrawPos}}\right) + \\ p\_B2\left(\vec{s}_{B2}\right)\cdot D_{\text{rmsg.DrawMsg}}\left(\vec{s}_{\text{DrawMsg}}\right). \end{aligned} \tag{9.16}$$

In Formula (9.16), $DT_{\text{DrawDisplay}}$ is the processor demand of the "*DrawDispl*" activity. $D_{\text{rmsg.DrawMsg}}$, $D_{\text{rmap.DrawPosition}}$, and $D_{\text{rroute.DrawRoute}}$ are processor demands of the respective component operations. The corresponding signature instances are $\vec{s}_{\text{DrawMsg}}$, $\vec{s}_{\text{DrawPos}}$, and $\vec{s}_{\text{DrawRoute}}$. Functions $p\_B2\left(\vec{s}_{B2}\right)$ and $p\_B1\left(\vec{s}_{B1}\right)$ are branching probabilities for the "*B2*" and "*B1*" branching nodes, respectively. $\vec{s}_{B2}$ and $\vec{s}_{B1}$ denote the corresponding signature instances. The D_{DrawMapA} is the processor demand of the sub-activity "*DrawMapA*" with the root operation "*DrawMap*" (see also Formula (9.15)) Finally, $D_{\text{DrawDisplay}}\left(\vec{s}_{\text{DrawDisplay}}\right)$ is the processor demand of the "*DrawDisplay*" operation itself and the internal operation

"*DrawMenu*". This processor demand depends on the corresponding signature instance $\vec{s}_{\text{DrawDisplay}}$ and is calculated by a prediction model constructed by the APPEAR method.

9.7 Modeling of concurrent activities

Concurrent activity instances are scheduled for processors and non-sharable resources. Depending on aspects such as scheduling policy, resource access control protocols, activity arrival patterns, and the way the activity instances synchronize, accounting for the concurrency may require the use of various techniques (see Figure 9.12).

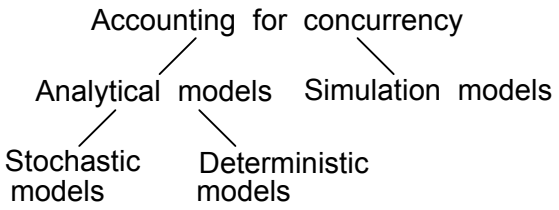


Figure 9.12: Accounting for concurrency

Analytical models can only be applied when they can be constructed with reasonable effort. For instance, when activity instances arrive according to a Poisson stream and only restricted resource contention occurs, it is possible to construct a queuing model for predicting the average response time of the activity. Stochastic analytical models are usually built by techniques such as queuing systems/networks [Kle76], [Kle96], stochastic or timed Petri Nets [KiP00], etc. These sorts of techniques are based on strict assumptions about underlying distributions[6] for the processor/resource demands and inter-arrival times of activities. In very simple cases, it is even possible to build a deterministic analytical model, described by a formula. An example of such simple formula is described in Section 10.3 and in Section 11.6.

However, for many software-intensive systems the assumptions of the existing analytical techniques are severely violated. In this case, a simulation model can be used. Section 10.5 exemplifies the use of a simulation model to predict the CPU utilization. The rest of this section describes a simple analytical model for our example of the hypothetical CNS (see Section 9.3).

Figure 9.13 shows a periodic schedule for the activities of the CNS constructed according to Table 9.2.

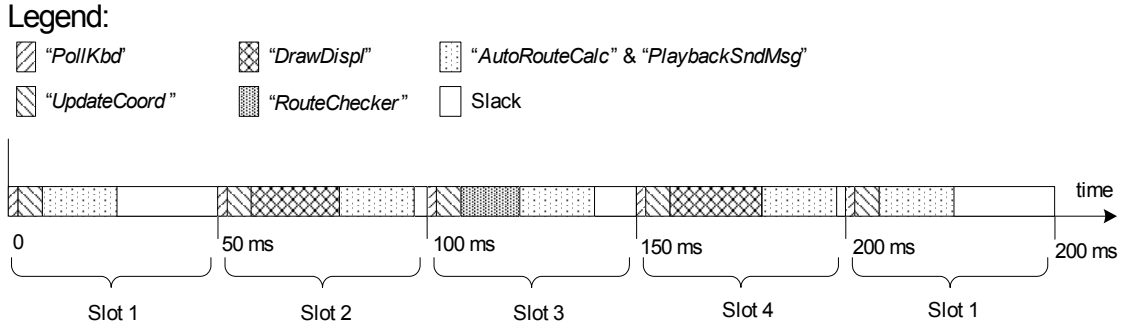


Figure 9.13: Construction of a schedule for the periodic activities of the CNS

This schedule is calculated at design-time and enforced by a *cyclic scheduler* at run-time. Since resource conflicts are resolved by the serialization of activity instances, this corresponds to a time-triggered architecture.

[6] In most cases, these distributions are considered to be from the exponential family.

A super period of 200 ms is partitioned into four time slots. Each time slot occupies 50 ms. The "*PollKbd*" and "*UpdateCoord*" activity executes in every slot. The "*DrawDispl*" is assigned to every second and fourth slot, whereas "*RouteChecker*" runs only every fourth one. Each slot may have a slack, the remaining processor time not consumed by the activity instances executing in this slot. The slack is used for executing the non-periodic activities "*AutoRouteCalc*" and "*PlaybackSndMsg*".

The construction of the cyclic schedule shown in Figure 9.13 requires the knowledge of the worst-case processor demand of the involved activities. We choose for a time-triggered architecture for the following reasons. First, a run-time schedule would require the same knowledge, as periodic activities have the deadlines that coincide with the respective periods. Second, a time-triggered schedule automatically resolves resource conflicts by searilization.

The estimates of worst-case resource demands of the activities can be derived from the models obtained by applying the first two steps of our method (see Sections 9.5 and 9.6). The procedure is as follows. First, we identify the worst-case values of signature parameters for the activities of interest. For instance, the *#Polygons* signature parameter of the "*DrawDispl*" activity cannot exceed 1000.

These worst-case values are passed to the prediction models of the activities, and the corresponding 95% prediction intervals (see Section 5.1) are calculated. For example, to estimate the processor demand of the "*DrawDispl*" the worst-case signature instances are passed to Formula (9.15) and (9.16). We consider the upper bound of a prediction interval as a tight estimate of the worst-case processor demand. To make this estimate safe, we add a safety margin of 10% to the calculated upper bound. Table 9.4 summarizes the worst-case processor/resource demand of the various activities obtained this way.

Table 9.4: The average processor/resource demands of the various activities

Activity	Upper bound of the prediction interval for the processor demand (ms)	Estimate of the worst-case processor demand (ms)
"*DrawDispl*"	20	22
"*PollKbd*"	1	1.1
"*UpdateCoord*"	5	5.5
"*RouteChecker*"	14	15.4
"*AutoRouteCalc*"	15	16.5
"*PlaybackSndMsg*"	2	2.2

The data from Table 9.4 and the enforced cyclic schedule shown in Figure 9.13 allow us to estimate the worst-case response time of each activity. For example, the worst-case response time $T_{DrawDispl}$ can be found by the following formula:

$$T_{DrawDispl} = W_{UpdateCoord} + W_{PollKbd} + W_{DrawDispl}. \quad (9.17)$$

In Formula (9.17), W_A denotes the worst-case processor demand of the activity A.

By substituting the data from Table 9.4 to Formula (9.17), we calculate the worst-case response time of the activity "*DrawDispl*":

$$T_{DrawDispl} = 22 + 1.1 + 5.5 = 28.6. \quad (9.18)$$

The obtained worst-case response time implies that 21.4 ms slack remains in every second and fourth slot. In this timeframe, also the sporadic activities "*AutoRouteCalc*" and "*PlaybackSndMsg*" may execute. The actual slack is thus 2.7 ms.

Summarizing, the time-triggered architecture supports the analysis and construction of the schedule. For an event-triggered architecture, we would most likely have to construct a simulation model to account for resource contention. This example demonstrates that the analyzability and predictability of a system can be achieved by limiting the design freedom. An approach to the construction of such a simulation model is described in the subsection below.

9.7.1 Construction of the simulation model

The information flow in a simulation model for predicting the performance of activities executing concurrently is described by means of an UML activity diagram shown in Figure 9.14.

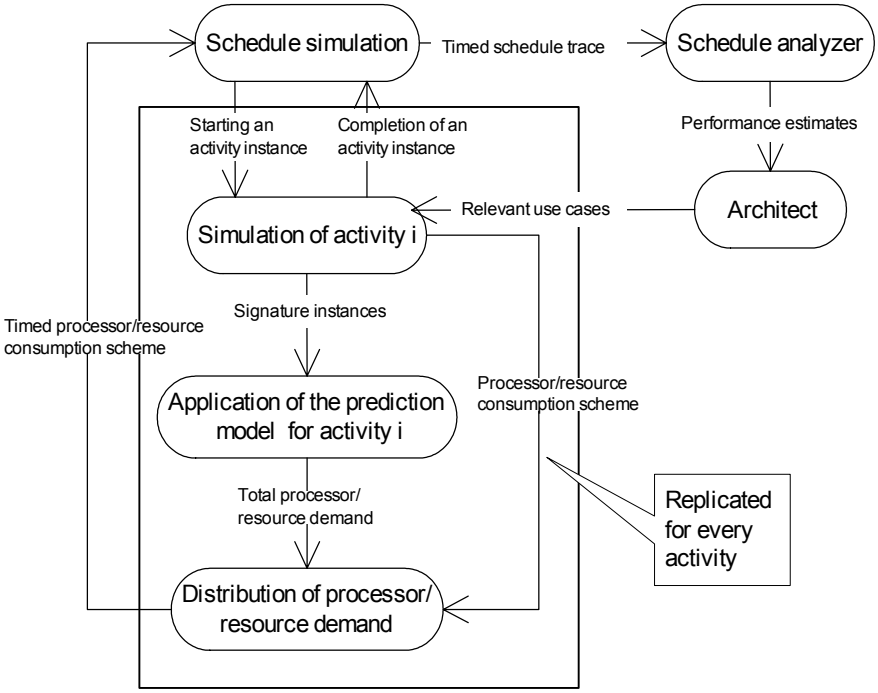


Figure 9.14: The information flow within a simulation model that accounts for activity concurrency

The architect devises the use cases and their parameters for which the performance prediction must be performed. These data are passed to the simulation models of the activities. The simulation model of an activity is formed by the simulation models of individual component operations that are invoked by this activity (see Section 9.5 and 9.6). The simulation models of activities calculate signature instances, which characterize the performance of the activity instances. In addition, these simulation models also generate schemes in which the processors and resources are consumed by activity instances. For example, the activities of the CNS (see Section 9.3) may be blocked on the access to the "*CCurrentInfo*" component via the "*pmap*", "*pcoord*", "*proute*", and "*pmsg*" provides interfaces. Since this component contains shared memory, it is necessary to provide processor/resource consumption schemes for all activities that may invoke component operations through these interfaces. For the "*DrawDispl*" activity, the reduced CFG of the "*DrawDisplay*" operation can be used (see Figure 9.9) to deduce such a scheme, as it describes the order of invocation of component operations through these interfaces. Depending on the scheduling algorithm under consideration, the level of processor/resource consumption granularity may vary. The calculated signature instances are passed to the activity prediction models (see Section 9.6.4) to estimate the total processor/resource demands of the activity instances. The obtained demand estimates are

then partitioned according to the processor/resource consumption schemes to obtain schemes that contain also timing information.

The constructed timed processor/resource consumption schemes are used to simulate the schedule on the basis of the model of the scheduler. The model of the scheduler must emulate the scheduling policy and resource access protocols (see Section 9.2.2 and Appendix H). It controls the execution of the simulation models by sending events to start an activity instance model and by receiving the events about the completion of an activity instance model. The role of these events is to coordinate the work of the simulation models of various activities. The construction of simulation models of various types of schedulers are widely described in the literature (see e.g. [FM03]).

The simulation of the schedule produces a trace, which contains timed information about the consumption of the processors and resources by the activities under consideration. By analyzing these traces it is possible to derive the required performance estimates and to report them back to the architects.

9.8 Summary

In this chapter, we have presented an approach to performance prediction for component compositions. A component composition is considered not only in terms of components but also in terms of concurrent activities that invoke a number of component operations. The approach is hierarchical: first, performance models are constructed for component operations; then the models are built for activities executing in isolation; finally, the concurrent activities are considered. The approach employs the basic principles and techniques of the APPEAR method to abstract from the implementation details of component operations and to construct prediction models for branches and loops. The approach is illustrated by an example of a Car Navigation System that demonstrated the important steps of our approach.

Some of the existing approaches (e.g., see [DMM03], [WMW03]) use only analytical techniques such as Layered Queuing Networks, timed/stochastic Petri nets etc to model the scheduling. Other approaches (e.g., [BMW04], [BWC04]) consider simulation. An important feature of our hierarchical approach is the possibility to choose between analytical and simulation modeling techniques at each step. The architect can select the appropriate technique based on (a) satisfaction of the assumptions of the techniques by the system under consideration, (b) the goals of the analysis, (c) the required accuracy, and (d) the timing budget available for the estimation.

At the moment, the existing approaches tend either to ignore the parameters of component operations (e.g. [BMW04], [WMW03]) or to treat them in simplistic manner [HAT03]. We suggest accounting for these parameters by means of signature types.

In the future work, the following aspects should be thoroughly studied:

1. *Fragmentation of information needed for predicting performance.* The partitioning of the software components usually does not correspond with the run-time architecture (e.g., activities may pass through many components). The performance is however determined by the run-time architecture. It is cumbersome to reconstruct the necessary models from the pieces related to different components. In the light of the proposed approach, this fragmentation means the growth of the number of models (both prediction and simulation) and the number of reduced CFGs for component operations. On the other hand, many simulation models, constructed by the APPEAR method, can provide more architectural insight.
2. *Complex control flow and data dependences between components.* Components usually interact through functional interfaces, which may have operations with

many input parameters. Calling such operations introduces complex control flow and data dependencies between the operations of a caller component and the ones of a callee component. These complex dependencies must often be modeled to obtain sufficiently precise performance estimates for the activities (compositions) that involve those operations. This problem is the inherent problem of functional interfaces. One must carefully consider what parameters are passed to and returned from component operations through the interfaces. The larger the number of parameters and the larger their variation is, the more severe the problem of complex control flow is.

10 Performance prediction for component compositions in the Consumer Electronics domain

This chapter demonstrates the application of the approach to the performance prediction of component compositions, described in Chapter 9, to a component-based TV software stack. The case study aims at predicting the CPU utilization of the TV software in a steady state. A TV is said to perform in a steady state, if it receives a broadcast and is not interrupted by any means, e.g., a local keyboard, remote control, etc.

The software of the TV that we have investigated is composed from Koala components. Some of these components start a number of periodic activities. The activities interact with one another via shared resources.

The two main objectives of this case study were the following:

1. *Validation of the approach for the prediction of the performance of component compositions.* Chapter 9 describes a hierarchical approach to the performance prediction for component composition. We aimed at validating this approach as follows. First, we construct a model for predicting the average CPU utilization of the TV software. Second, we compare the predictions obtained by this model to the measured CPU utilization. The average relative prediction error should not exceed 5%
2. *Analysis of the behavior of the TV software in steady state.* The architects were interested in obtaining an insight in to the complex behavior of the TV in steady state to be able to assess the extensibility of the TV software in the light of performance.

The chapter is structured as follows. First, the goals and requirements of the case study are explained. They are followed by an overview of the TV software stack. Then, we demonstrate the use of a simple analytical formula for the prediction of the CPU utilization. A more detailed analysis of the run-time architecture of the TV software is given further. Based on this analysis, we show how to construct a simulation model for predicting the CPU utilization. We conclude this chapter by discussing the results of predicting the CPU utilization by means of an analytical formula and simulation.

10.1 TV Software overview

This section describes the components and subsystems that are relevant for the analysis of the average CPU utilization of the TV software.

10.1.1 Component interaction mechanisms

The TV software is built of Koala components [OLK00]. The components form a containment hierarchy. At the highest abstraction level, the TV software is assembled from subsystems, being groups of logically related components. Each subsystem handles certain TV functionality or feature such as Teletext, Electronic Programming Guide (EPG), etc.

A component can start one or more activities, which can invoke operations of other components. The control flow of such activity can cross many components and form non-trivial execution paths.

The TV software employs two types of communication: (1) synchronous method invocation and (2) asynchronous message passing. The former is implemented as conventional calls through functional interfaces (which are standard for Koala), whereas the latter is based on the use of a dedicated activity for processing a message.

Activities perform asynchronous and/or timed actions. An activity is implemented as a cyclic logical thread, which processes messages from its message queue. Such logical thread with a queue is called a *pump* [Omm03]. The pump has a function that handles messages from the queue. An activity instance is released whenever a message (event) is placed in the message queue. The possible sources of messages are the following:

1. Methods of the provides interfaces of the component that owns the pump. These methods can be invoked by other pumps;
2. The pump itself (for performing repetitive actions);
3. An interrupt service routine.

A component that starts an activity uses a special *requires* interface, "*pmp*". This interface has a number of operations: the creation of a pump, sending messages (events) to a pump (both timed and immediate), and etc. We say that an component owns a pump, if this component has created it. Notice that it is impossible to send a message, by using the "*pmp*" interface, from one pump to another pump. The only way to send a message to a particular pump from other pumps is though the *provides* interfaces of the component that owns this pump.

A message may need to be delivered as soon as possible or after some delay. The former helps implement asynchronous message passing, whereas the latter facilitates the construction of timed activities.

Once a *pump* starts processing a message, it can invoke operations of the *requires* interfaces of the component that owns this pump. The components that implement the corresponding *provides* interfaces may in turn call operations of its own requires interfaces. In this way, the control flow may pass through several components before the processing of the message completes.

A *pump* executes on some physical thread (called a *pump engine* [Omm03], [OLK00]). All physical threads are scheduled by a fixed priority pre-emptive scheduler and consequently have priorities. In the contrary, the pumps that share the same physical thread cannot preempt each other. Please notice that all *pumps* that execute on the same *pump engine* share the same message queue. This sharing enforces the FIFO order of processing the messages by these *pumps*.

The mapping of activities (pumps) onto physical threads is made explicit by using a dedicated diversity interface (see [OLK00]), usually called "*pen*". This mapping is performed at composition time. The TV software stack has a dedicated component ("*isfib*" in Figure 10.1) that provides physical threads on which the other components can map their activities. This component creates a few physical threads with different priorities during initialization time. The identifiers of these threads are provided to other components through the *pen* interfaces (dotted lines in Figure 10.1). This allows a flexible choice of threading structure at composition time.

10.1.2 TV software structure

Figure 10.1 depicts the most relevant subsystems and components for analyzing the steady-state behavior of a TV, that is, only these components perform CPU-intensive tasks. Each subsystem from Figure 10.1 is described in the subsequent sections.

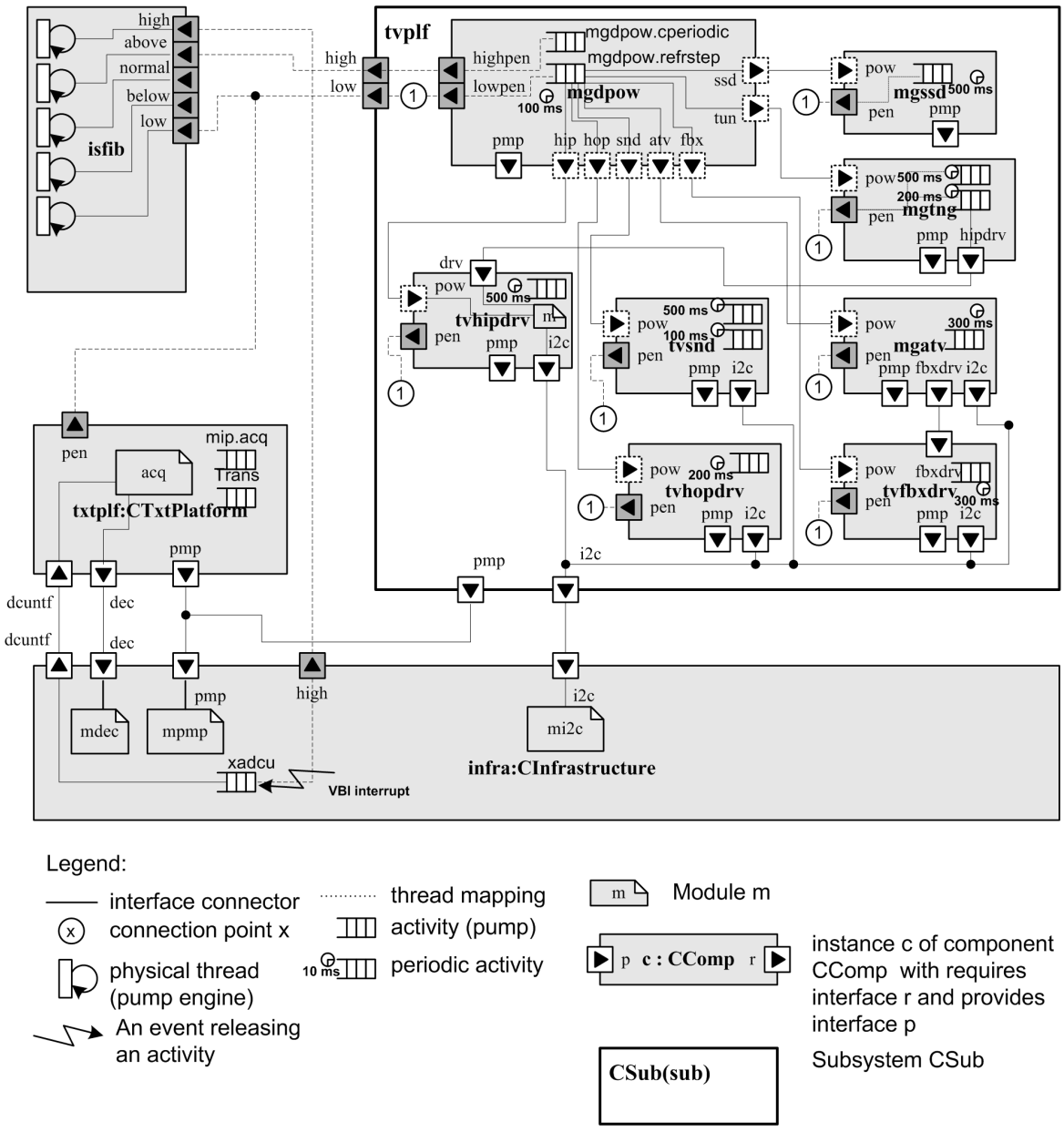


Figure 10.1: Components relevant for the steady-state behavior of a TV

Please notice that the presented component composition structure is a simplification, as it represents only the subsystems and the immediate subcomponents thereof. Presenting the entire hierarchy is not necessary for understanding the component composition structure.

Further, the following simplifications are also made in the figure:

- Binding between the "*pmp*" interfaces (see Section 10.1.1) is omitted in Figure 10.1 to reduce the number of lines. The "*pmp*" requiresinterface of each component needs to be connected to the "*pmp*" providesinterface of the enclosing component (e.g., "*tvplf*").
- The internal structure of inner components is omitted: the internal modules, their interconnections, etc are not drawn.

The multi-tasking of the TV software is based on a dedicated real-time kernel. This real-time kernel supports the typical primitives of a RTOS: events, semaphores, message boxes, timers, etc. On the top of this kernel, a dedicated API is provided for flexible threading (this API is implemented by means of the "*pmp*" interface described in Section 10.1.1). The structure of the physical threads is made explicit in Figure 10.1 by using

Koala interface binding: the "*pen*" interfaces of components that own a pump are connected to a physical thread of the "*isfib*" component.

A) Infrastructure ("infra")

The Infrastructure subsystem provides basic RTOS functionality to other subsystems. A part of this functionality is implemented by a group of components depicted as module *mpmp* in Figure 10.2.

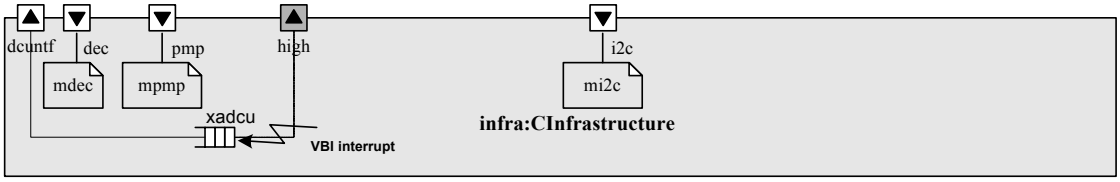


Figure 10.2: Outline of the Infrastructure subsystem

The Infrastructure also implements drivers for on-chip hardware devices located in the microcontroller that executes the TV software. In steady state, the performance relevant devices are the Data Capture Unit (DCU), Data Decoding unit (DDU), and I^2C-bus controller.

The DCU allows collecting all data arrived during a VBI (e.g., Teletext, WSS, and VPS). The DDU is a hardware decoder for hamming 8/4 and hamming 24/18 error correction codes. Both devices are used for implementing the Teletext subsystem. The DCU collects Teletext data packets from a broadcast, whereas DDU is used to decode the parts of these packets that are encoded by hamming 8/4 and 24/18 codes.

Upon arrival of a packet, the DCU generates an interrupt. The interrupt handling routine then sends a message to the "*xadcu*" activity that handles the further processing of the packet data. If the arrived packet is a Teletext packet, this activity invokes an operation of a dedicated callback interface provided by the Teletext acquisition component. The packet is further processed within the Teletext subsystem.

The driver of the DDU is represented by module "*mdec*" in . The functionality of the DDU is accessible via the "*dec*" interface. The operations of this interface are invoked by the "*xadcu*" activity.

The I^2C-bus is a shared resource, which can be used in a mutually exclusive manner only. The Infrastructure implements an interface ("*i2c*") for sending and receiving data. A group of the components that implement the I^2C functionality is depicted as module "*mi2c*" Figure 10.2. The activity performing an I^2C transaction can block on waiting for transaction completion. Note that the entire physical thread also blocks, with the other activities that share this physical thread being also blocked.

The mutual exclusion of the I^2C-bus is implemented using a semaphore with the Highest Locker protocol to limit priority inversion. According to this protocol, the priority of the physical thread is boosted to a certain priority ceiling when this physical thread acquires the semaphore. The priority ceiling is chosen to be higher than the priority of any physical thread in the investigated version of the TV software.

B) Teletext subsystem ("txtplf")

In steady state, only activities that relate to the Teletext data acquisition play a role. The important ones are implemented by three activities:

1. The "*xadcu*" activity, processing all information extracted during Vertical Blanking Interval (VBI) lines;
3. The "*mip.acq*" activity, decoding the magazine inventory pages (MIP);
4. The "*TransPump*" activity, decoding the content of a page when its type is extracted from the magazine inventory page.

Note that the "*xadcu*" activity is owned by the Infrastructure subsystem, but most of its processing is done within the Teletext subsystem. However, some of the processing[1] (e.g., WSS) is also performed within the TV Platform subsystem ("*tvplf*").

The Teletext acquisition is implemented by a few dedicated components that are depicted as a module "*acq*" in Figure 10.3. These components are executed on the "*xadcu*" activity for each packet received during a VBI interval.

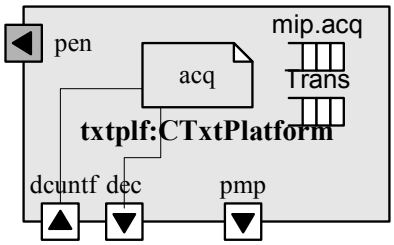


Figure 10.3: Outline of the Teletext subsystem

The simulation model of Teletext acquisition has been constructed in the previous case studies (see chapter VII). Thus, we will not further detail the Teletext acquisition component in this chapter.

C) TV Platform subsystem ("tvplf")

The TV Platform is responsible for maintaining and modifying the audio and video (A/V) signal chain. Both types of signals are handled by dedicated hardware chips. In software, these chips are represented by *conceptual devices* and *device drivers*.

A conceptual device is a Koala component that imitates the behavior of a certain hardware chip from a signal connectivity point of view: each input and output signal pin of the hardware chip has a corresponding, input and output, pair of interfaces of the conceptual device. The binding between the input pair of interfaces of one conceptual device and the output pair of another conceptual device emulates the actual wiring between the input and output signal pins of the corresponding hardware chips.

By using a dedicated architectural style, the *Horizontal Communication* ("*HorCom*") protocol [Omm03], it is possible to change a signal path between different hardware chips and to propagate signal properties and notifications about their change. The use of this style allows the uniform development of TV sets that have different features. However, the "*HorCom*" architectural style is usually used only when transient behavior takes place, i.e., a channel is being switched. It can thus be ignored for the analysis of the steady-state behavior.

Device drivers support interactions with the hardware chips. Most chips are connected to the CPU via an I^2C-bus. By using the I^2C-bus driver implemented in the Infrastructure subsystem, each device driver can read from or write to device registers that control the corresponding hardware chip. A typical device driver implements the following functionality:

[1] This is not shown in Figure 10.1

- It periodically reads the values of the corresponding hardware chip registers and notifies the conceptual device about changes in the values since the last read operation (the "*ForceReadRegisters()*" function from Section 10.4.4).
- It periodically checks for erroneous situations and notifies the conceptual driver if they are found (the "*CheckPeriodicErrors()*" function from Section 10.4.4).
- It checks, if a protection situation takes place, that is, a situation that requires the power of the TV be switched down immediately.
- It periodically updates the values of write-only registers.

Each driver has a dedicated activity that periodically polls the status registers (the read-only registers of a chip). This activity also checks for erroneous situations. The thread notifies the conceptual driver component about the changes of register values and erroneous situations. Notice that the activities of different drivers share the same physical thread, although the polling periods can be different (100-500ms).

To check for protection situations and to write to write-only registers, a driver implements two operations that are invoked by the "*mgdpow*" power management component. The "*mgdpow*" component manages the power of the entire TV set. It supports different power modes: normal, standby, protection, etc. This component can switch up or down the power of each hardware chip individually by using the "*pow*" interface of the corresponding device driver. This power interface also has functions to poll a device driver about erroneous situations and to write the values of registers from an in-memory cache into the chip. These two activities are implemented by two activities, "*mgdpow.cperiodic*" and "*mgdpow.refrstep*", respectively. Notice that the "*mgdpow.cperiodic*" activity, being of high importance, is mapped onto a physical thread with a high priority, whereas the "*mgdpow.refrstep*" is mapped on the same physical thread as the other activities of the TV platform subsystem.

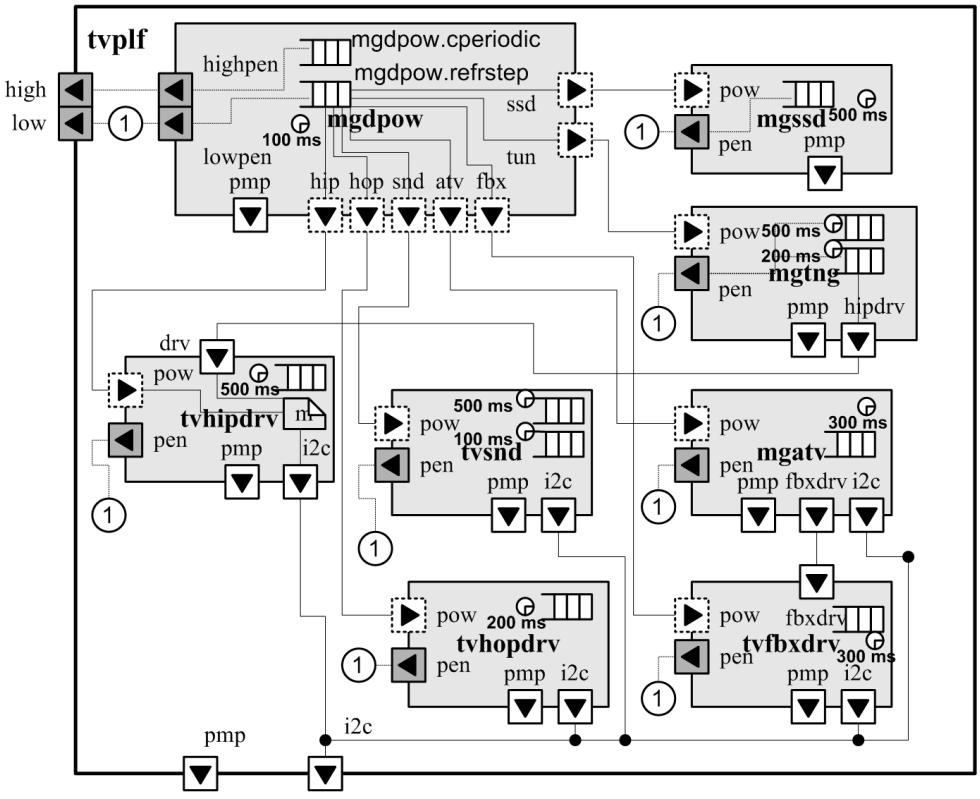


Figure 10.4: Outline of the TV Platform subsystem

Figure 10.4 does not show the internal details of device drivers: modules and their interconnections. To illustrate how the internals of a typical driver look like, the internals of "*tvhipdrv*" component are shown. It contains a module *m* that implements the

functionality of the driver (provides interface "*drv*") and the functions provided through the "*pow*" interface, namely, refreshing the values of write-only registers and checking for a protection situation. Module *m* also creates an activity that periodically polls the read registers about changing their values and erroneous conditions. The other device drivers have a similar structure.

10.1.3 Major contributors to the CPU utilization

The following subsystems are active when the TV performs in steady-state: (1) Infrastructure ("*infra*") with its DCU activity, "*xadcu*", (2) the Teletext Platform ("*txplf*") that implements most of the functionality executed by the DCU activity, and (3) the TV Platform ("*tvplf*") that implements device drivers and handles the A/V signal chain processing.

The measurements performed by us showed that the major contribution to CPU utilization is regarded to the activities of the Infrastructure and TV Platform subsystems (see Figure 10.1, Figure 10.2, Figure 10.3, and Figure 10.4):

- the "*xadcu"*activity that acquires Teletext data,
- the "*tvsndsys.pmp"* activity that polls the sound chip,
- the *"mgdpow.cperiodic"* activity that checks protection conditions for certain devices,
- the *"mgdpow.refrstep"* activity that periodically updates the content of write registers of hardware chips connected to the host CPU via the I^2C the bus,
- the "*tvfbxdrv.mon*" activity that periodically polls the feature box chip,
- the "*mgtng.afc*" activity that implements the "Automatic Frequency Control" feature,
- the "*mgatv.ctr*" activity that dynamically tunes certain displaying parameters (depending on the ambient light),
- the "*tvhopdrv.hop*" activity that periodically polls the Hardware Output Processor (HOP) chip,
- the "*tvsndqp.wait*" activity that automatically adjusts sound volume,
- the "*tvhipdrv.mon*" activity that periodically polls the Hardware Input Processor (HIP) chip.

The measurements also revealed that these activities contributed more than 95% of the total average CPU utilization of a TV in steady state.

All these activities, except for *"mgdpow.refrstep"* are quasi-periodic. The behavior of such a quasi-periodic activity is shown in Figure 10.5. After receiving a message, the activity executes certain functionality, then sends itself a timed message, and suspends. A particular period is assigned to this timed message. Only after the expiration of this period, the delivery of this message resumes the activity again. Note that the actual period of such activity may be longer than the period assigned to the timed messages because of (1) preemption by higher priority activities and (2) blocking on waiting for the completion of activities that were scheduled to release earlier but did not complete in time. We will call this actual period the effective period of the activity.

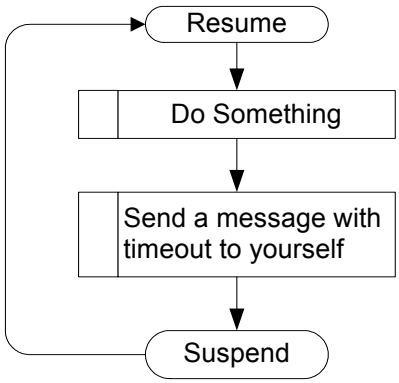


Figure 10.5: Typical behavior of a quasi-periodic activity implemented by a pump

10.2 Calculation of the average CPU utilization of a composition

We consider the component composition that includes the components and subsystems shown in Figure 10.1. Some of these components start activities, whereas others do not. The letter are used by the started activities.

Over time interval *T*, the CPU utilization of a set of activities is defined by the following formula (e.g., see [MAD04], [Jai91]):

$$U = \frac{E}{T}. \qquad (10.1)$$

In Formula (10.1), *U* denotes the CPU utilization of the set of activities; *E* is the time during which the CPU executes these activities (processor demand).

The performance (including the CPU utilization) of a component assembly is defined not only by the resource and processor demands of the components, but also emerges from the interactions between these components. As explained in Chapter 9, we suggest a hierarchical approach for treating the performance of the component composition. In accordance with the three levels of the approach, three types of contributions to the performance are distinguished:

- A part relating to the resource consumption at the component-operation level. For example, an operation of a component can consume a certain amount of CPU cycles, or it can occupy a bus for a certain amount of time;
- A part relating to the resource consumption at the activity level. This part concerns the processor and resource consumption of an activity executed in isolation;
- A part attributing to the interactions between activities and synchronization on shared resources. This part accounts for possible delays due to blocking and pre-emption.

We treat these three parts separately. The subsequent subsections describe the corresponding steps of our hierarchical approach.

10.2.1 Modeling of component operations

Please notice that we skipped the first step of the hierarchical approach[2] for the following reasons. The Koala components used in the TV software are defined on the low

[2] Consequently, we did not calculate the contribution of the first type.

abstraction level. They are not independently deployable and have a restricted reuse scope. The components implement a number of operations, which are too low level to be a concern of the architects. These operations are only interesting in the context of the activities that they are invoked by. Therefore, it does not have much sense to consider these operations individually. Moreover, considering the contributions of all individual component operations separately is cumbersome, as this involves complex control flow analysis and intensive measurements.

10.2.2 Modeling of activities

We started applying the hierarchical approach from the second step, which constructs a model for predicting the processor and resource demand of an activity. As we decided to treat each activity as a whole (without considering all the individual component operations that the activity invokes), we used the APPEAR method, described in chapter V, to predict the resource and processor demand at the activity level. For this purpose, we consider the activity as an operation of a hypothetical component that includes all component operations of the Koala components that are invoked by this activity.

Using the APPEAR method, we may predict the processor demand E_i of an activity *i* during a particular, sufficiently long, observation interval *T*. The considered use case is the execution of a number of instances of activity *i* that are started and completed within the observation interval *T*. As we consider an activity as an operation of a hypothetical component, the notions of signature type and signature instances have to be introduced. Let us denote the signature type and instance of activity *i* as S_i and s_i, respectively. Please notice that one signature instance may cover not only a single activity instance but also a sequence thereof. Signature instances of an activity may encompass the effects of the interactions with other activities.

In the simple case, the signature type of an activity may be defined by a one-element vector, the number of the activity instances that execute during the observation time *T*. This number may depend on the interactions with other activities.

The simulation model of an activity calculates signature instances that describe the execution of a single activity instance or a sequence thereof. The simulation model may input particular parameters that are related to the performance relevant input parameters of the activity. In addition, it may indicate the order of resource consumption and the CPU and resource demand by a single activity instance. This order and demand may be required for modeling the scheduling.

Finally, we fit a prediction model over signature type S_i to the measured processor demand for each activity *i*.

10.2.3 Modeling of activity composition

Over a sufficiently long interval T, the total utilization *U* of a composition of activities can be estimated using the following formula (see also Formula (10.1)):

$$U = \frac{\sum_{i=1}^{N} E_i(s_i)}{T}. \qquad (10.2)$$

In Formula (10.2), E_i estimates the average utilization and total execution time of activity i as explained in Section 10.2.2. N is the number of activities that contribute to the utilization significantly. The s_i vector denotes the signature instance of activity *i*.

Both the direct interactions between different activities and the use of shared resources may influence the values of signature instances s_i. Making particular assumptions about these interactions and the use of shared resources, the values of signature instances s_i can be calculated by an analytical formula (in Section 10.3). Less strict assumptions made us use simulation (see Section 10.5).

10.3 Prediction of the average CPU utilization of the TV software by means of an analytical formula

Let us consider the use case of watching a particular broadcast channel. The analysis of the measurements of the TV software in steady state showed that each of the most CPU consuming activities (see Section 10.1.3) executed at least few dozen times during each observation interval. In addition, these activities behave in a repetitive manner. These facts allow us to fit prediction models E_i in the following form:

$$E_i\left(s_i\right) = \overline{e}_i \cdot s_i . \tag{10.3}$$

In Formula (10.3), s_i denotes a signature instance of activity *i*, which equals the number of times it executes, and $\overline{e}_i$ is a regression coefficient, which coincides with the average CPU demand of a single instance of activity *i*. The values of $\overline{e}_i$ are given in Appendix K. Notice that Formula (10.3) does not have a constant addendum. The reason is that that the CPU demand must equal zero, if no activity instances execute.

In order to calculate signature instances s_i, we had to assume the following:

- All activities under consideration are periodic. This assumption is supported by observations made in Section 10.1.3.
- The effects of direct or indirect interactions between these activities are negligible. This assumption allows constructing a simple analytical formula.

Section 10.1.3 shows that the first assumption is not satisfied only for a single activity. This activity behaves however in a repetitive manner and can therefore be considered as a periodic one on the long run. Each periodic activity has an effective period, which may be longer than the nominal period of the activity attached to timed messages. Let $\overline{T}_i$ denote the effective period of activity *i*; it accounts not only for the nominal period T_i of activity *i*, but also for a delay $\overline{\Delta}_i$ due to blocking and preemption. The effective period $\overline{T}_i$ can be calculated by the following formula:

$$\overline{T}_i = T_i + \overline{\Delta}_i . \tag{10.4}$$

The nominal periods T_i could be found in the design specification and source code of the TV software. The delays $\overline{\Delta}_i$ turned out to depend on the scheduling and were difficult to guess analytically. Section 10.4 illustrates this difficulty in more detail.

The quasi-periodicity of the activities allows us to calculate the signature instances as follows:

$$s_i = \left\lfloor \frac{T}{\overline{T}_i + \overline{e}_i} \right\rfloor. \quad (10.5)$$

In Formula (10.6), T denotes the observation period for which the average CPU utilization needs to be calculated. Function $\lfloor x \rfloor$ returns the greatest integer that is less or equal x.

The second assumption allows us to consider that the $\overline{\Delta}_i$ equals zero. As result, Formula (10.5) can be rewritten as follows:

$$s_i = \left\lfloor \frac{T}{T_i + \overline{e}_i} \right\rfloor. \quad (10.6)$$

By considering n broadcast channels and using Formulas (10.2), (10.3), and (10.6), we construct the following formula for predicting the CPU utilization of the most CPU consuming activities:

$$U_{form}^{(j)} = \frac{\sum_{i=1}^{N} \overline{e}_i \cdot \left\lfloor \frac{T^{(j)}}{T_i + \overline{e}_i} \right\rfloor}{T^{(j)}}, \quad 1 \le j \le n. \quad (10.7)$$

In Formula (10.7), $U_{form}^{(j)}$ denotes the CPU utilization estimated for the j-*th* broadcast channel. $T^{(j)}$ is the observation interval for the *j*-th channel. There are *n* channels in total. The remaining variables are described in Section 10.3.

The *N* activities from Formula (10.7) are taken from a validation activity set. This validation activity set includes all most CPU consuming activities describe in Section 10.1.3, except for the "*xadcu*" activity. The rationale is as follows:

1. We already constructed a detailed prediction model for the processor demand of the "*xadcu*" activity (see chapter VI). This model is fine-grain and allows predicting the CPU demand of an instance of the "*xadcu*" activity with the average prediction error of only 8.5%. On a long observation interval, the total CPU demand of this activity can be estimated with much higher accuracy, as prediction errors for single activity instances will cancel out. The accurate modeling of this activity is however not the goal of the present experiment.
2. The "*xadcu*" activity executes on the physical thread (*pump engine*) that have the highest priority and cannot be therefore preempted. Thus, its effective period equals the nominal period, and the signature instances can be calculated precisely.

As a result, the contribution of the "*xadcu*" activity to the CPU utilization can be estimated with high precision. It is therefore not worthwhile to consider this activity in Formula (10.7).

10.3.1 Experiment scheme

In the experiment, we considered the prediction of the CPU utilization for a number of broadcast channels. Each channel was unique with respect to the type of transmitted audio and video information, Teletext data, etc.

The experiment was performed according to the following algorithm:

1. The TV software was traced in steady state for 30 different broadcast channels. These channels provided various workloads for the Teletext subsystem. This variability in Teletext workloads led to various total CPU utilizations. Though, the variability in the CPU utilization of the validation activity set was significantly less.
2. Based on the measurements collected at step 1, we fitted the prediction models $E_i(s_i) = \overline{e}_i \cdot s_i$ for the processor demand of each activity of the validation activity set. Please notice that we used all 30 channels for fitting the prediction models. This was possible as for each activity from the validation set, its activity instances exhibited similar processor demand on the average.
3. For each of the 30 broadcast channels traced at step 1, we predicted the average CPU utilization of the validation activity set by using Formula (10.7) and compared the obtained estimates to the actual CPU utilization. The comparison details are described in Appendix K, whereas its summary described in section 10.3.2.

10.3.2 Experiment results

Appendix K shows that the simple analytical formula overestimates the actual CPU utilization calculated from the measurements by 0.0084 on the average, which means the average relative prediction error of 5.04%. This conclusion is supported by the paired one-sided t-test at the significance level of 0.05.

This obtained average relative prediction error is very close to the required level of 5%. However, we carefully investigated the reason for the prediction errors.

The first assumption turned out to be violated for the "*mgdpow.refrstep*" activity. This activity is not periodic. For each broadcast channel, we traced the nominal periods of releasing the "*mgdpow.refrstep*" activity instances and calculated the average nominal period. Figure 10.6 shows the box-and-whiskers plot [KO02] that summarizes the observed average nominal period for various broadcasts.

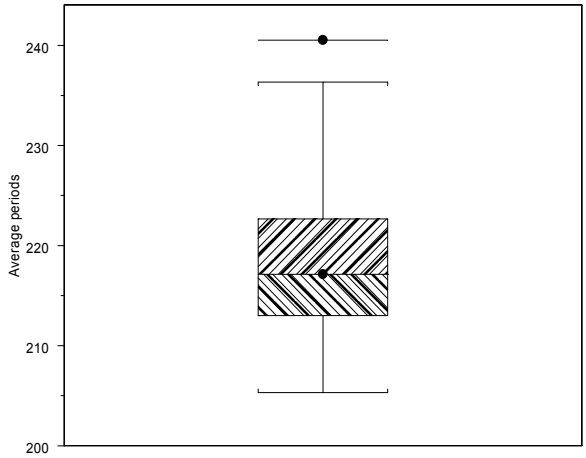


Figure 10.6: The nominal periods of the *"mgdpow.refrstep"* activity for various broadcast channels

The width of the box is equal to the *interquartile range*, or IQR, which is the difference between the third and first quartiles of the data. The IQR indicates the spread of the distribution for the data. Whiskers extend from the edges of the box to either the extreme values of the data, or to a 1.5 IQR distance of from the median (217 ms), whichever is less. Data points that fall outside of the whiskers may be outliers, and are therefore indicated by additional lines.

Figure 10.6 shows that the bulk of data lie in a range 212 to 223 ms. However, for few channels the periods may be as small as 205 ms and as large as 240 ms. These facts invalidate the first assumption for the "*mgdpow.refrstep*" activity.

Measurements also showed that for particular activities (e.g., the "*tvfbxdrv.mon*" activity) the effective period was longer than the nominal period. For instance, the nominal period of the "*tvfbxdrv.mon*" activity equals 300ms, whereas its effective period exceeded 340ms for many channels. This difference invalidates the second assumption described in Section 10.3.

Achieving higher accuracy required a deeper analysis of phenomena that resulted in violating the assumptions. To this end, we analyzed the run-time architecture of the TV software in steady state. The results of this analysis are presented in Section 10.4. Moreover, it will be shown that certain phenomena could not be modeled by an analytical formula and that simulation had to be applied.

10.4 Run-time architecture analysis

Various aspects of TV software run-time architecture turned out to be important for modeling the average CPU utilization. The subsequent sections discuss these aspects in more detail. Particularly, Section 10.4.1 demonstrates that activities might suffer from preemptions and blocking on the access to the shared resources. In Section 10.4.2, we show that the time services used in the TV software can introduce inaccuracy in the release times of activity instances. Finally, Sections 10.4.3 and 10.4.4 explain that particular I/O devices might influence the scheduling dramatically. The issues turned out to be hard to describe analytically. Therefore, we decided to use simulation to account for the effects of these issues. The use of simulation for predicting CPU utilization is described in Section 10.5.

10.4.1 Analysis of the scheduling

As mentioned earlier, the TV software builds on a fixed-priority preemptive scheduler. Each task corresponds to a physical thread (*pump engine*) that can execute a few activities (*pumps*). The mapping between groups of activities and physical threads as well as priorities of physical threads is chosen at the composition time. The following mapping is used in the considered composition (see Table 10.1):

Table 10.1: The mapping between activities and physical threads

Physical thread priority	Activities
"*High*"	"*xadcu*" (for acquisition of Teletext)
"*Above Normal*"	"*mgdpow.cperiodic*"
"*Low*"	"*tvsndsys.pmp*", "*mgdpow.refrstep*", "*tvfbxdrv.mon*", "*mgtng.afc*", "*mgatv.ctr*", "*tvhopdrv.hop*", "*tvsndqp.wait*", "*tvhipdrv.mon*"

The TV software is designed in such a way that only physical threads with "*High*", "*Above Normal*", and "*Low*" priorities are active in the steady state. The physical threads that have the remaining priorities are activated only in transient states such as channel switching.

Shared resources are guarded by semaphores. The semaphores use the Highest Locker protocol to prohibit unbounded priority inversion. An example of such a resource is the driver of the I^2C-bus.

The behavior of each *pump engine* is described by the statechart shown in Figure 10.7.

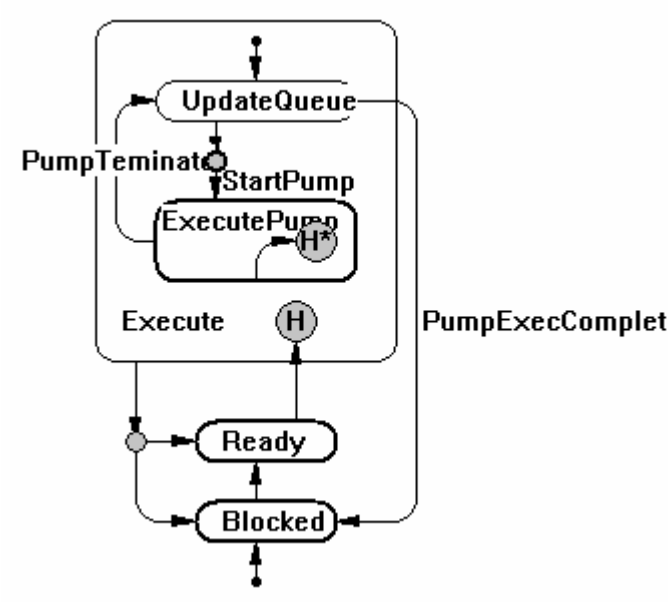


Figure 10.7:The behavior of a *pump engine*

Each *pump engine* is assigned the priority in accordance with Table 10.1. The *pump engine* object can be in one of the following states: "*Blocked*", "*Ready*", and "*ExecutePump*". Only a single *pump engine* is allowed to be at the "*Execute*" state at each moment. The "Execute" hyper-state is partitioned into two sub-states: "*UpdateQueue*" and "*ExecutePump*". In the former state, it is checked if the message queue contains a message that needs processing by the corresponding *pump*. If such a message is found, the *pump engine* switches to the "*ExecutePump*" state and starts the corresponding *pump* by firing the "*StartPump*" transition. When the current *pump* completes, the *pump engine* switches back to the "*UpdateQueue*" sub-state. If the message queue is empty, it switches to the "Blocked" state, and the CPU is assigned to the next lower priority *pump engine*.

When the message queue becomes non-empty, the *pump engine* unblocks and switches from the "*Blocked*" state to the "*Ready*" state. From this "*Ready*" state, the *pump engine* can switch to the "*Execute*" state, if it has the highest priority. In this case, the *pump engine* that presently executes and in the "*Execute*" state is preempted and forced to switch to the "*Ready*" state. When the highest priority *pump engine* completes execution of all its *pumps*, the lower priority one may resume and switch back to the sub-state of the "*Execution*" state where it has been before preemption.

Finally, the *pump engine* may be blocked and have to switch from the "*Execution*" state to the "*Blocked*" state, if it tries to access a shared resource owned by another *pump engine*. The *pump engine* resumes and switches back to the "*Execution*" state when it acquires the freed resource.

Please notice that preemption and blocking introduce an additional delay and increases thereby the effective period or response time of activities.

10.4.2 Time services

Time is measured in terms of timer ticks, each tick being 10 ms. The implied message communication and the precision of time representation causes the degradation of timing accuracy when scheduling activities in a timed fashion. Figure 10.8: illustrates this degradation in more detail.

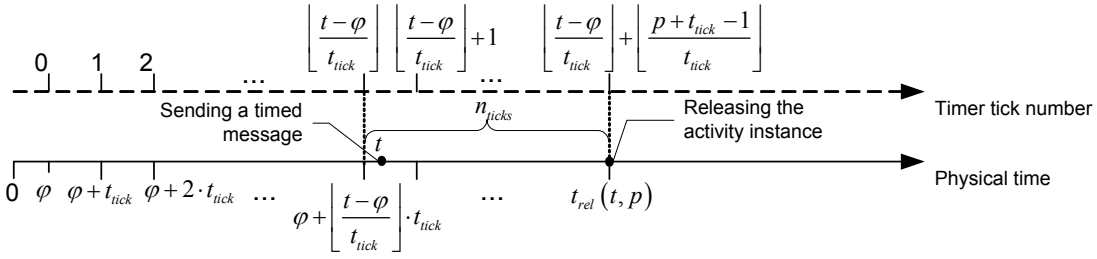


Figure 10.8: Illustration of the degradation of time accuracy

The physical time is continuous. It is depicted by the solid arrow, with the arrow tip indicating the direction of time increase. Timer ticks are discrete values. They mark the dashed arrow that shows the direction of the increase of timer ticks.

Suppose that the timer that counts ticks with a period t_{tick} starts at the moment φ of physical time. We call this moment the phase of the timer. At certain time t (also in terms of physical time), an activity is sent a timed message, which is assigned period p. From a viewpoint of the timer, this event happens at the $\lfloor (t-\varphi)/t_{tick} \rfloor$-th tick[3]. Starting from this tick, the TV software calculates, in terms of ticks, the moment of releasing the activity instance corresponding to the timed message. For this purpose, the period p is rounded to ticks in the following way:

$$n_{ticks} = \left\lfloor \frac{p + t_{tick} - 1}{t_{tick}} \right\rfloor. \tag{10.8}$$

In Formula (10.8), n_{ticks} denotes the number of ticks after which the activity instance will be released. The formula was constructed based on the analysis of the TV software implementation. Note that the rounding described by Formula (10.8) differs from the traditional rounding.

Finally, the actual release time can be estimated by the following formula:

$$t_{rel}(t, p) = \varphi + \left\lfloor \frac{t-\varphi}{t_{tick}} \right\rfloor \cdot t_{tick} + n_{ticks} \cdot t_{tick} = \varphi + \left(\left\lfloor \frac{t-\varphi}{t_{tick}} \right\rfloor + \left\lfloor \frac{p + t_{tick} - 1}{t_{tick}} \right\rfloor \right) \cdot t_{tick}, \tag{10.9}$$

$$t_{tick} = 10ms.$$

Note that Formula (10.9) does not account for possible timer drift, that is, when the actual timer tick slightly differs from 10 ms.

Depending on the period, the phase of the timer, and the moment of time, an error is introduced into the release time of an activity instance. This error lies within an interval of two ticks (-10ms, 10ms).

10.4.3 Analysis of the I²C transactions

The I²C bus is a shared resource that can be used by several activities. They relate to the chip drivers, protection functionality, automatic TV functionality, etc. The host CPU may communicate with the following chips: FBX, HIP, HOP, SCAVEM and the tuner.

The use of this bus may influence the schedule dramatically. For instance, due to the long response time of the feature box (FBX) chip, a delay of 20-60ms is introduced to all

[3] Operator $\lfloor x \rfloor$ returns the integer part of x.

activities that share the same physical thread (*pump engine*) and have the same inter-arrival period of 300 ms.

Each I^2C transaction is performed by two parties [I2C03]. One is a master device, which initiates the transaction. The other is a slave device, which may be either read from or written into by the master device.

The analysis of the execution traces has shown that the time needed to complete an I^2C transaction may be influenced by the following factors (see Appendix O):

1. The I^2C-bus is inherently non-deterministic due to the possibility of slave devices to control the data flow by holding the clock line.
2. Each hardware chip can have a non-deterministic latency for responding to write and/or read commands. This latency is determined by the internals of the chip.
3. Sometimes transmission errors can occur; retransmission is then needed.

The I^2C protocol allows a slave device to stretch clocks to slow down the dataflow. This clock stretching prolongs the total length of an I^2C transaction. The duration of this stretching is determined by the slave device. Each hardware chip of a TV chassis uses this clock-stretching feature in its own manner. Thus, the time needed for a transaction depends not only on the traffic, but also on the chip to be communicated with.

Appendix O shows that the time needed for I^2C transactions can be modeled separately for each transaction type. The transaction type determines which chip is communicated with and how many bytes are read from or written to the chip. For each chip, except for the FBX chip, the transaction duration is modeled by the mean time calculated from the measurement. For FBX chip, a more detailed model is needed. This model is described in Appendix O.

10.4.4 Analysis of the device drivers

Each device driver owns an activity that periodically polls hardware chips to check if there is a change in status (read-only) registers or an erroneous situation. A typical activity of a device driver can be represented by the following pseudo-code:

```
[ 1] CheckPeriodicAll()
[ 2] {
[ 3]    if(mode==operational)
[ 4]    {
[ 5]        ForceReadRegisters();
[ 6]        CheckPeriodicErrors();
[ 7]        if(recover)
[ 8]        {
[ 9]            pmp_PmpSendAfter(self, PERIOD_SMALL);
[10]        }
[11]        else
[12]        {
[13]            CheckPeriodicEvents();
[14]            pmp_PmpSendAfter(self, PERIOD);
[15]        }
[16]    }
[17]    else
[18]    {
[19]            pmp_PmpSendAfter(self, PERIOD_SMALL);
[20]    }
[21] }
```

Figure 10.9: The behavior of a typical driver

The refreshing of write registers and protection checking is implemented on the activities of the "*mgdpow*" component of the TV platform subsystem. This section only describes modeling of the activities that owned by the device driver itself.

Each device driver can be in a certain state defined by the values of the mode and recovery variables. The mode can equal "*virgin*", "*off*", and "*operational*". The virgin mode corresponds to the state, when the device has not been initialized yet. The "*off*" mode means that the device is not yet active, but has already been initialized. The "*operational*" mode indicates that the device is active. The behavior of device driver can be described be the following state-machine (see Figure 10.10).

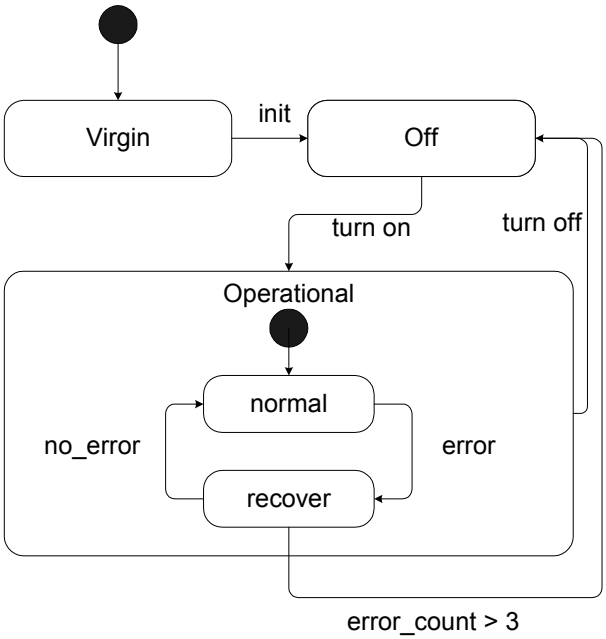


Figure 10.10: The power states of a device driver

After powering up the TV set, the driver is in the "*Virgin*" state. Then, it performs initialization and switches to the 'Off' state. At a certain moment, the "*mgdpow"* component (see Figure 10.4) switches the power of the driver up, and the driver toggles to the "*Operational*" (normal) state. In this state, it executes lines 5-15 of the pseudo code shown in Figure 10.9 with a period "*PERIOD*".

The driver switches to the "*recover*" state and starts to poll the hardware chip more frequently (with a period "*PERIOD_SMALL*"), if an error occurs. If the error is not indicated anymore, the driver returns to the normal operational mode; if the error persists, the driver deactivates itself and notifies the "*mgdpow*" component that the TV set has to be powered down.

The prediction models that describe the CPU demand of the device driver activities are described in Appendix N.

A) Feature box driver

The FBX driver periodically checks the values of FBX chip registers. These values are read by performing an I^2C-transaction. The periodic checks are implemented using a quasi-periodic activity with a period of 300 ms. Each instance of this activity makes three transactions sequentially, each transaction reading 6 registers.

The total time spent within the I^2C-transactions of a single instance of the FBX activity may vary significantly in a range [8ms, 52ms], because the FBX chip can send data over the I^2C-bus only at certain instants. These moments correspond to a VBI (20ms). The

measured period of the FBX activity was approximately 340 ms, except for 7 outliers from 31 observations in total (see Figure 10.11).

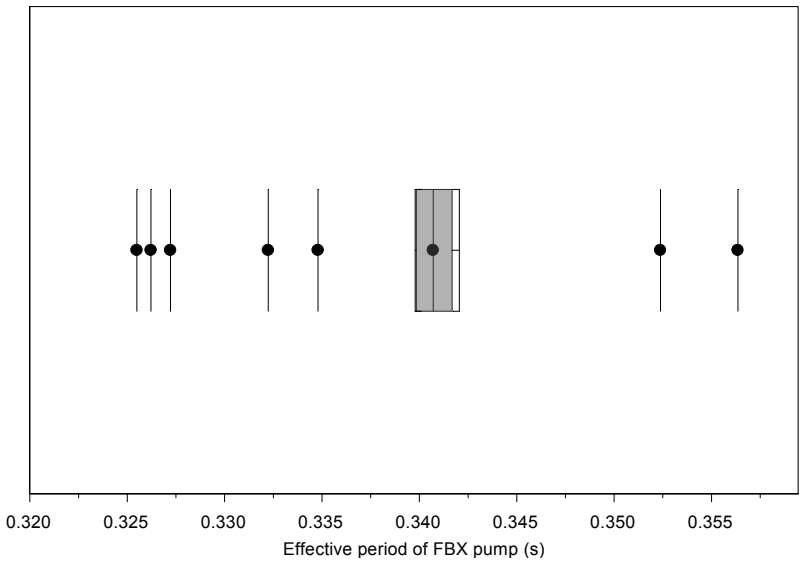


Figure 10.11: Measured period of the FBX activity

This figure shows that the bulk of measured periods are about 341 ms. There are however two groups of outliers: two in the right part of the figure and five in the left part. The former are explained by the arrival of the instances of the computationally intensive activity that decodes a magazine inventory page (MIP) received by the Teletext decoder. This activity shares the physical thread with the "*FBX*" activity and thus can block it. The outliers in the right part of Figure 10.11 can be explained as follows:

The pre-emption scheme for the channels that exhibit the five outliers can differ from the rest of the channels in dynamic equilibrium, as the average CPU utilization of the Teletext acquisition is low (<20%) for these channels. Also, the effective period is shorter as the "*FBX*" activity is pre-empted for shorter times and less frequently.

1. The implicit synchronization between the timer tick server, Teletext field routine and "*FBX*" activity has a different pattern than for other channels. This synchronization occurs as (1) the timer tick (10 ms) is a multiple to the VBI length (20 ms) and (2) the "*FBX*" chip seems to be able to send data over I^2C-bus only at a certain subinterval of each VBI (see Appendix O).
2. The effects of rounding errors described in Section 10.4.2 play a role, as for these five channels the phase shift between timer tick server and Teletext fields may differ from other channels.

10.4.5 Analysis of the refreshing and protection activities

Both refreshing and protection activities are started by the "*mgdpow*" component (see Section 10.1.2). These activities periodically invoke dedicated operations provided by the components that need refreshing or protecting. An example of such components is device drivers.

A) Refreshing activity (*"mgdpow.refrstep"*)

The refreshing activity updates the content of the write registers of the TV chassis hardware chips via the I^2C-bus. This activity is implemented on one of the activities of the "*mgdpow*" component (see Section 10.1.2).

The refreshing process is subdivided into a number of steps in order not to overload the CPU and the I^2C-bus for too long. During initialization time, each conceptual device provides an operation for performing a refreshing action to the "*mgdpow*" component. The conceptual device also provides the total number of phases that are needed to completely update the entire set of write registers.

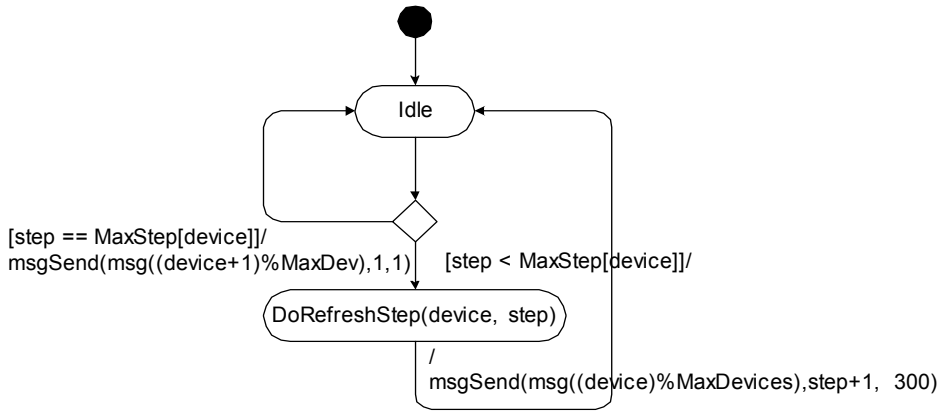


Figure 10.12: UML statechart of the refreshing activity

At each step, the refreshing activity (see Figure 10.12) checks if all registers of a conceptual device have been updated. If there are some registers left, it invokes the corresponding operation and sends itself a message with a 300 ms timeout. Otherwise, the next conceptual device is chosen to be refreshed, and the refreshing activity sends itself a message with a 1 ms timeout. This results in a-periodic behavior.

The refreshing operation of a conceptual device is parameterized by the current refreshing step number. Based on this parameter, it is decided which registers need to be written via the I^2C-bus.

B) Protection activity (*"mgdpow.cperiodic")*

The protection activity periodically checks the condition of parts of the TV hardware. The TV needs to be powered down, if some dangerous conditions occur. This activity is started by the "*mgdpow*" component (see Section 10.1.2) and is mapped onto the physical thread with the second highest priority (the highest priority is given to Teletext acquisition with a 20 ms deadline).

The protection activity is subdivided into a number of phases, which are scheduled every 100 ms. During each phase, a few *protection checks* are performed, such that the CPU and I^2C-bus load is balanced amongst different phases. This load balancing is ensured by constructing a proper protection schedule during the initialization of the TV software. Each component (e.g., a conceptual device) registers a protection operation that checks if certain dangerous conditions occur. The component also specifies the frequency of protection checks in terms of the numbers of phases.

10.5 Prediction of the CPU utilization of the TV software by means of simulation

This section describes the construction of the simulation model and its role in predicting the CPU utilization of the TV software in steady state (see Figure 10.13).

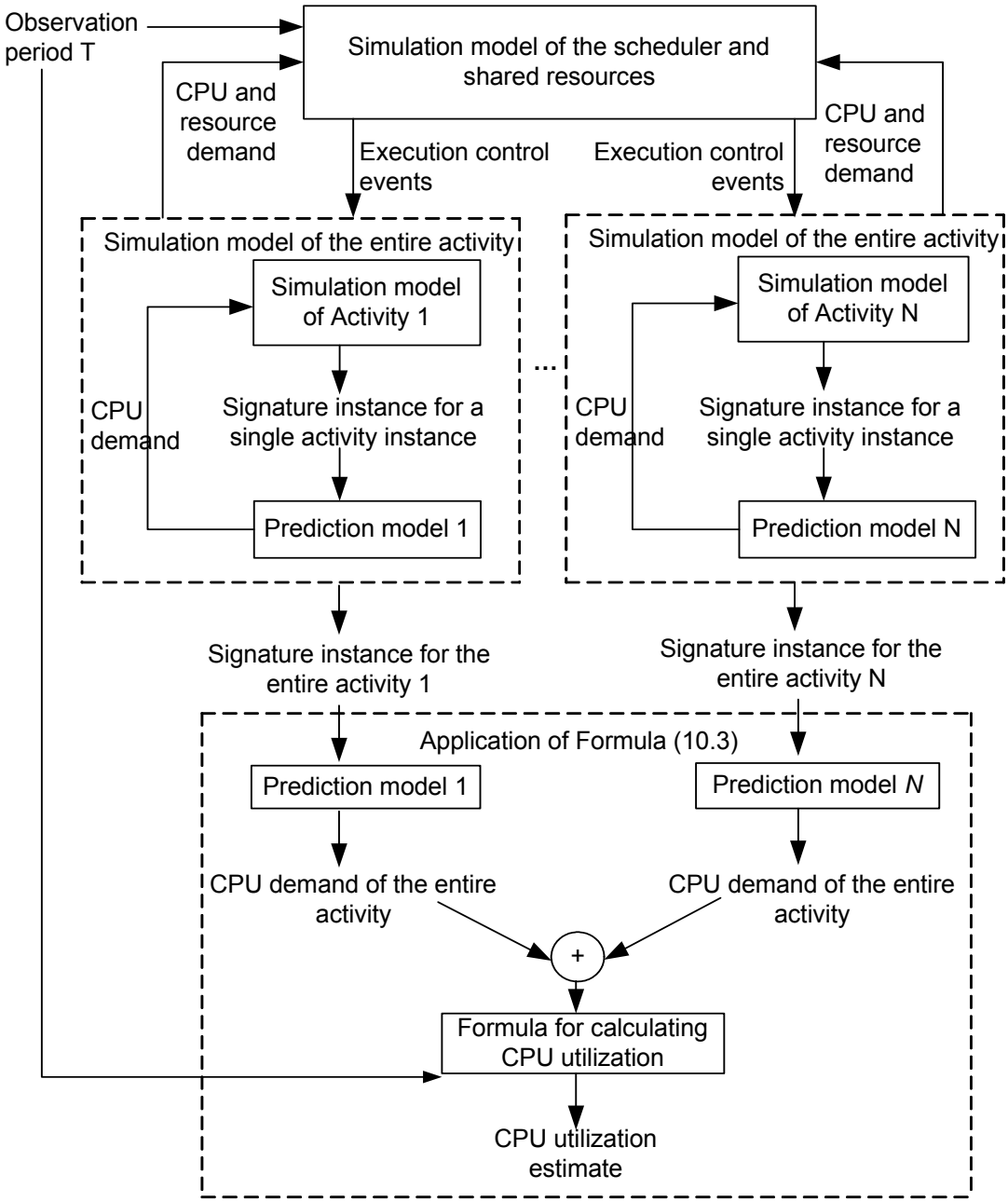


Figure 10.13: The prediction of CPU utilization by simulation

The boxes denote particular entities processing input information and producing output information. Examples of these entities are simulation and prediction models. Arrows describe the information flow. The arrow is attached text that explains what information is passed. For instance, the simulation model of activity 1 produces signature instances for each activity instance. Theses signature instances are then passed to the prediction model 1. The circle with a plus symbol inside is an adder, which calculates the sum of all numbers that are inputted to it.

The simulation model consists of the following items:

1. The simulation model of the scheduler;
2. The simulation model of the shared resources;
3. The simulation models of activities.

Please notice that the first two items are represented in Figure 10.13 by a single box. The simulation models of activities are depicted with a separate box each. We use these simulation models twofold. First, they calculate signature instances for predicting the CPU

and resource demand of a single activity instance. Second, these models also generate signature instances for calculating the CPU demand of the sequence of activity instances over the observation interval *T*. This second way of using the simulation models is depicted by means of dashed boxes.

We use simulation to estimate the CPU utilization as follows. The simulation models of the activities together with the corresponding prediction models estimate the CPU demand of each activity instance. In addition, the simulation models indicate how the resources are used by the activity instances (if necessary). The obtained CPU and resource demands of the activities are used by the simulation model of the scheduler and shared resources to model the schedule. In turn, the scheduler simulation model controls the execution of the simulation models of activities by sending particular events.

The simulation models of activities calculate signatures instances that are inputted to the prediction model, after the simulation completes, to calculate the total CPU demand of the activity during the observation interval *T*. The obtained total resource demands of all activities are summed up and then used to calculate the total CPU utilization by using Formula (10.1).

Notice that the same prediction models, constructed by the APPEAR method, are used to estimate the CPU and resource demands[4] of both separate activity instances and a sequence of activity instances. The possibility to use the same prediction models for these two purposes is ensured by the construction of the prediction models in such a way that their coefficients have an interpretation in terms of CPU demands of an activity instance or its parts. In addition, the prediction models are linear in signature parameters and in coefficients. This linearity allows us to add up the contributions of individual activity instances both in the space of the signature parameters and in the space of the CPU demands. The prediction models are detailed in Appendix N. They are incorporated in the simulation models of the corresponding activities.

For the same reasons as for the case of the prediction of the CPU utilization by means of an analytical formula (see Section 10.3), we did not build a detailed model of the Teletext acquisition activity ("*xadcu*"). Instead, we considered the "*xadcu*" activity as a periodic one with the period of 20ms (VBI length). The processor demand of a single instance of this activity was considered as being equal to the mean demand calculated from the trace of a particular broadcast. This approximation is sufficient as all variations of the processor demand of the actual "*xadcu*" activity cancel out anyway on the long run.

The validation of the prediction of the CPU utilization by means of simulation is detailed in Appendix P. The rest of this section describes the construction of the simulation model in more detail.

10.5.1 Structure of the simulation model

The simulation model of the behavior of a TV in steady state implements the following functionality:

- The fixed-priority preemptive scheduler, including multiple access to shared resources (critical sections and semaphores with the Highest Locker protocol) and blocking on performing I/O (see also Section 10.4.1);
- Physical threads (*pump engines*) that can execute activities (*pumps*) (see Section 10.1.3);
- An API for inter-activity communication (corresponding to the "*pmp*" interface from Section 10.1.1 and Figure 10.1);

[4] Here, the CPU and resource demand is considered in terms of execution times and resource consumption times, respectively.

- The time services (see Section 10.4.2);
- An API for manipulating shared resources and I/O devices;
- A framework for collecting measurement data from a run of the simulation model (e.g., average effective period, etc).

Particular items from the list above are detailed in the subsequent sections.

The simulation model was constructed using the COVERS even-driven simulation engine [BKR97], which allows the user to model activities in terms of Harel's statecharts [Ha87].

The COVERS tool supports the visual construction of an executable model. Such an executable models contains a number of active objects. Active objects are entities that have autonomous behavior, which is usually specified in form of a statechart. The transitions, guards, and states of this statechart are then annotated with C++ statements and expressions. The obtained code is compiled and linked to an executable that can be run to simulate the behavior.

The structure of the simulation model is presented in Figure 10.14.

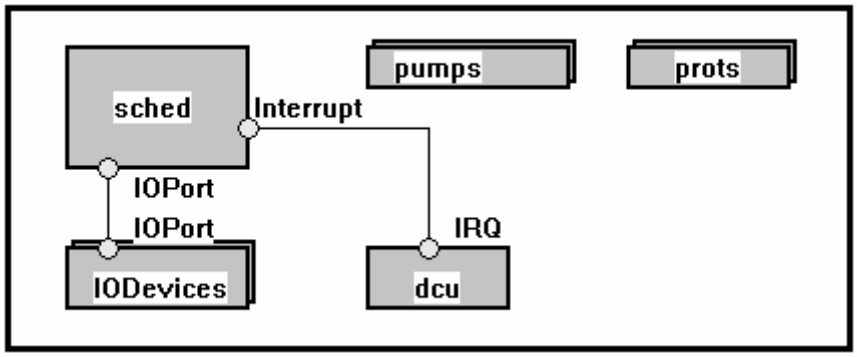


Figure 10.14: The structure of the simulation model

The gray rectangles represent active objects. The circles at the borders of active objects denote ports that are used for asynchronous message passing between active objects. The line between a pair of ports means that the active objects that own these ports can communicate. For instance, the "*sched*" and "*dcu*" active objects can communicate via the "*Interrupt*" and "*IRQ*" ports, respectively. Please notice that COVERS also supports synchronous method invocation between the active objects, but only in the C++ domain. Although the developers of COVERS discourage this type of communication, it can be used when simulation efficiency is a concern. We used the synchronous method invocation to model communication between the "*sched*", "*pumps*", and "*prots*" active objects.

The most CPU consuming activities (see Section 10.1.3) are modeled as a single or a number of COVERS active objects (the "*pumps*" active objects in Figure 10.14). Each activity model (1) specifies the moments when the shared resources (e.g., the I^2C-bus) and how much of them are used by the activity instance, and (2) generates signature instances for inputting to the prediction model. In total, the simulation model contains about a dozen of such simple active objects. In addition, the *"mgdpow.cperiodic"* activity (see Section 10.1.3) is modeled by the "*prots*" active objects. Each of them is responsible for mimicking the behavior and CPU/resource demand of a single *protection operation*.

The model of the scheduler, represented by the "*sched*" gray rectangle is executed together with the models of the various activities to simulate the long-term behavior of the TV software, and to extract the signature instances of separate activities. These signature instances are then used to predict the total CPU demand of all activities during simulation (see Section 10.5).

The "*IODevices*" group of active objects represents the hardware chips connected to the CPU via the I^2C-bus. The I^2C driver permits only a single device to be communicated with at a given moment of time. This means that the "*IODevices*" (as well as the I^2C driver) can be considered as shared resources with mutually exclusive access. The mutual exclusiveness is implemented by binary semaphores with the Highest Locker resource access protocol.

The "*dcu*" active object emulates the interrupts from the Data Capture Unit (DCU) that collects the Teletext information from every VBI (20 ms). This active object notifies the scheduler active object that the Teletext acquisition activity needs to be scheduled.

Figure 10.15 illustrates a screenshot of the execution of the simulation model.

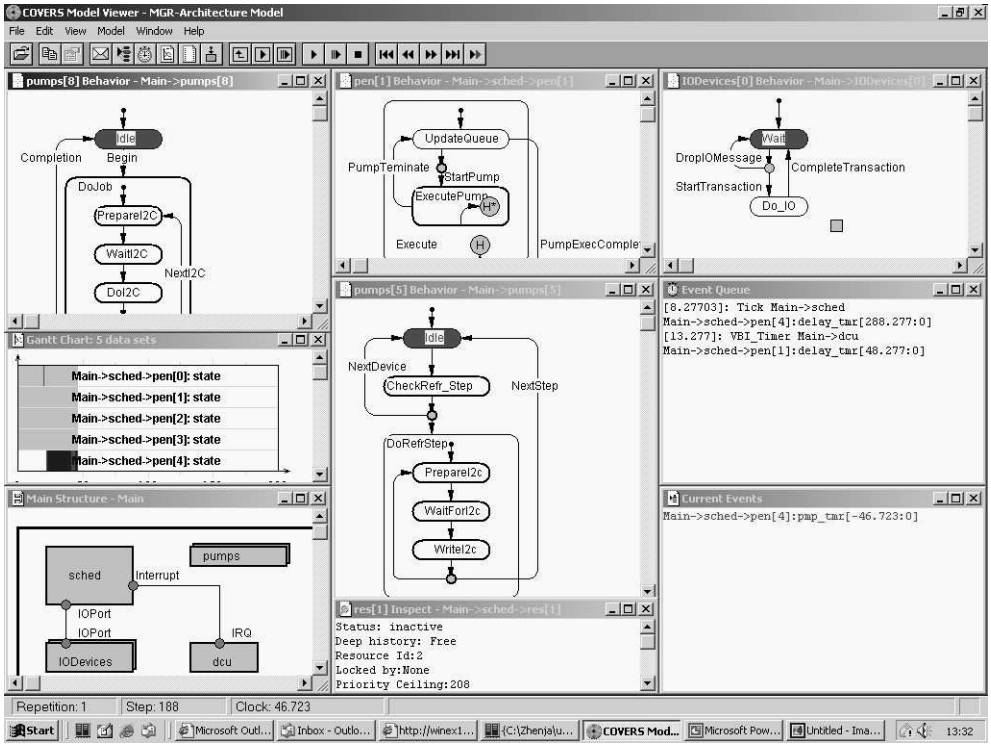


Figure 10.15: A screenshot of the simulation model

To give a flavor of how the described active objects look like, the subsequent sections detail various simulation models that comprise the overall simulation model.

10.5.2 Simulation models of activities

The simulation model of each activity was not only used to calculate signature instances for estimating the CPU demand, but also to estimate how much I^2C-bus time was consumed by each instance of the activity. This data is needed to simulate the scheduling. The CPU demand is calculated by applying the prediction model to the generated signatures (see Appendix O).

The simulation models of all relevant activities were implemented as a part of the entire simulation model. These models interact also with the simulation model of the scheduler (see Section 10.5.3).

Figure 10.16 demonstrates the behavior of the active objects that models the device driver polling activities (see Section 10.4.4). These active objects belong to the "pumps" active objects described in Section 10.5.1.

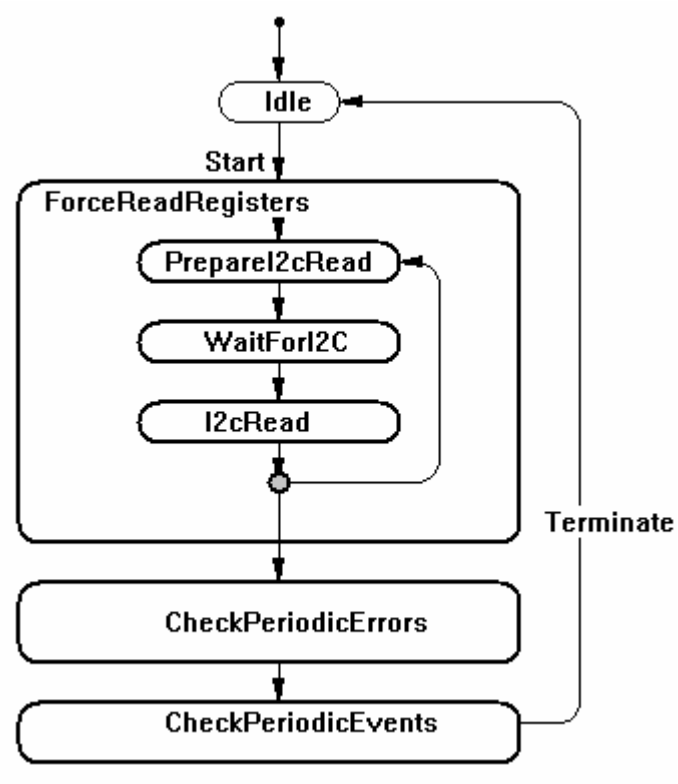


Figure 10.16: The statemachine of a device driver activity

The active object can be either in the "*Idle*" state or in one of the resource consumption states, which model the consumption of the CPU or I^2C-bus. In the "*Idle*" state, it waits until the "*sched*" active object (see Section 10.5.3) sends it a message to trigger the "*Start*" transition and to switch to the "*ForceReadRegisters*" hyper-state. In this hyper-state, the active object simulates the behavior the "*ForceReadRegisters()*" function (see Section 10.4.4). It sequentially switches between the following inner states: "*PrepareI2cRead*", "*WaitForI2c*", and, "*I2cRead*".

In the "*PrepareI2cRead*" state, the device driver active object notifies the "*sched*" active object about how much CPU must be consumed to prepare the data for an I^2C transaction. Being in the "*WaitForI2c*" state models the waiting for the semaphore that guards the I^2C driver. When the I^2C is acquired, the active object switches to the "*I2cRead*" state and models the reading a number of bytes via the I^2C bus. When all data is read, the active object proceeds to the "*CheckPeriodicErrors*" state. Otherwise, the active object switches back to the '*PrepareI2cRead*' state to begin preparing the next transaction.

The "*CheckPeriodicErrors*" state is responsible for modeling the consumption of the CPU by the "*CheckPeriodicErrors()*" function (see Section 10.4.4). After this consumption is completed, the active object switches to the "*CheckPeridoicEvents*" state to model the CPU consumption by the "*CheckPeridoicEvents()"* function. As we investigated the TV software only in steady state, we did not observe any events that could lead to interactions with other components by means of the *HorCom* architectural style (see Section 10.1.2). It was therefore sufficient to model only the CPU consumption of the "*CheckPeridoicEvents()"* function.

Finally, the active object models the suspending of the device driver polling activity by triggering the "*Terminate*" transition and switching back to the "*Idle*" state. At this transition the next invocation of the active object is scheduled such that it models the scheduling of the next activity instance.

Note that the CPU consumption budgets were assigned to the various states of the active object in an arbitrary way. However, the total CPU demand of all states where the

CPU consumption is modeled equal to the value provided by the respective prediction model (see Appendix N).

Other activities are modeled in a similar way.

10.5.3 Simulation model of the scheduler

The simulation model of the scheduler is represented by the "*sched*" active object of a class "*TScheduler*" (see Figure 10.14 and Figure 10.17). The "*TScheduler*" class is a singleton, i.e., only one instance (the "*sched*" object) of this class exists.

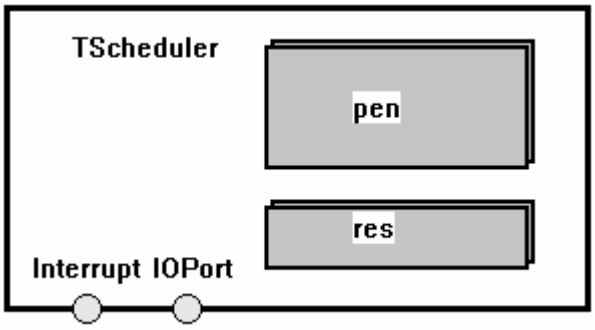


Figure 10.17: The structure of the scheduler active object

The "*sched*" active object models the behavior of the fixed-priority scheduler (see Section 10.4.1) used in the TV software. The "*sched*" active object contains a number of "*pen*" and "*res*" active objects. The "*pen*" active objects model physical threads ("*pump engines*") with a message queue, whereas the "*res*" objects model shared resources such as a communication bus. The "*pen*" active objects interact with the "*pumps*" active objects (see Figure 10.14) to model the scheduling of various activities represented by these "*pumps*" active objects. The *"IOPort"* port is used to communicate with the models of hardware chips (the "*IODevices*" active objects from Figure 10.14). The "*Interrupt*" port is used to deliver messages from the "*dcu*" active object (Figure 10.14). These messages model interrupts from the DCU for processing Teletext packets.

10.5.4 Modeling I^2C transactions

As described in Section 10.4.3, the TV chassis contains the following hardware chips: FBX, HIP, HOP, SCAVEM, and the tuner. The host CPU controls these chips via the I^2C bus. The analysis of the use of the I^2C bus is given in Appendix O. Figure 10.18 presents the statechart that describes the behavior of an active object that models one of these hardware chips. This active object belongs to the "*IODevices*" group (see Figure 10.14).

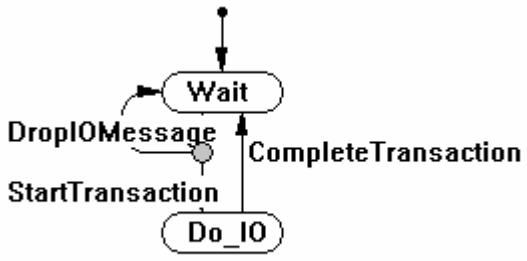


Figure 10.18: The behavior of an active object modeling a hardware chip

The "*Wait*" state models the monitoring of the state of the I^2C bus by a hardware chip. Performing an I^2C transaction is modeled by sending a message to the "*IOPort*" of each "*IODevices*" active objects (see Figure 10.14). One of the active objects recognizes that

this message is sent to it and switches to the "*Do_IO*" state by firing the "*StartTransaction*" transition. Others ignore the message by firing the "*DropIOMessage*" transition and switching back to the "*Wait*" state.

In the "*Do_IO*" state, the addressee active object starts a timer, which models the consumption of the time for performing the I^2C transaction. The duration of the transaction being modeled depends on the hardware chip that the active object models. The duration is calculated on the basis of the analysis of the timing behavior of the actual hardware chips (see Appendix O).

The expiration of the timer indicates the completion of the I^2C transaction. The active object returns to the "*Wait*" state by firing the "*CompleteTransaction*" transition.

10.5.5 Modeling physical time

It was necessary to introduce the physical time in the simulation model to be able to model activity interactions accurately and adequately. Particularly, the following had to be modeled:

- The durations for which activity instances consume CPU and other resources. Different instances of the same activity may have a different duration. For the sake of simplicity, we modeled only the average durations.
- The order of the consumption of resources of different types. Some activities consumed not only the CPU but also the I^2C bus. The order of consumption was kept intact in the simulation model, but the partitioning of consumption durations was chosen arbitrarily, with the total resource and CPU demands being equal to the corresponding average demands (see point above).
- The rounding of event times. All timed events (e.g., sending messages) are bound to the real-time operating system timer ticks. When scheduling a particular timed event, the same time rounding was implemented as in the actual TV software (see Section 10.4.2).

Notice that we deliberately chose not to model the overhead due to interrupts and context switches, as it was negligible.

10.5.6 Experiment scheme

To validate the simulation based approach, we used the same validation activity set as for the case of an analytical formula (see Section 10.3). The experiment was performed according to the following algorithm:

1. The TV software was traced in steady state for 30 different broadcast channels. These channels provided various workloads for the Teletext subsystem. This variability in Teletext workloads led to various total CPU utilizations. The measured average CPU demand of the "*xadcu*" activity instances was used to model the workload of the Teletext acquisition in the simulation model (see Section 10.5).
2. Based on the measurements collected at step 1, we constructed the prediction models for the processor demand of each activity of the validation activity set. The obtained prediction models are described in Appendix N.
3. For each of the 30 broadcast channels traced at step 1, we predicted the average CPU utilization of the validation activity set by simulation (see Section 10.5) and compared the obtained estimates to the actual CPU utilization. The comparison details are described in Appendix P.

Appendix P shows that the simple analytical formula overestimates the actual CPU utilization calculated from the measurements by 0.00237 on the average, which means the average relative prediction error of -1.43%. This conclusion is supported by the paired one-sided *t*-test at the significance level of 0.05. This obtained average relative prediction error is less the required level of 5%.

10.6 Modeling results

Appendix K and Appendix P detail the prediction of the CPU utilization of the most CPU consuming activities by means of a simple formula and simulation, respectively.

We have shown that the simple analytical formula overestimates the actual CPU utilization on the average by 0.0084, which means the average relative prediction error of 5.04% (at the 0.05 significance level). The reason for this overestimation is the wrong assumption that the effective periods of the activities under consideration equal to their nominal periods. This assumption results in the overestimation of the signature instances, related to the number of times that the activities execute. The CPU demand of activities is linear with the signature instances, which results in the overestimation of this CPU demand.

On the other hand, the simulation underestimates the actual CPU utilization on the average by 0.00237 on the average, which means the average relative prediction error of -1.43% (at the same significance level). Those figures allow us to conclude that the simulation provides on the average more accurate results than the simple formula does. Moreover, one can except that the prediction accuracy of the simple formula will degrade further with the growth of the CPU utilization, as the simple formula does not account for the interactions of the activities. The simulation does take into account these interactions, and no severe degradation of the prediction accuracy is expected.

The construction of the simulation model took us much more effort than the prediction of the CPU utilization by means of the simple formula. The former required about four man months, whereas the latter required only about two man weeks. The major effort needed for implementing the simulation approach is as follows:

1. Analysis of the run-time architecture (1.5 man month);
2. Construction of the simulation model of the scheduler (0.5 man month);
3. Construction of the simulation models of the individual activities (0.75 man month);
4. Construction of the prediction models of the individual activities (0.5 man month);
5. Construction of the simulation models of the hardware chips (0.25 man month);
6. Debugging the overall simulation model (0.5 man month).

A significant effort (about two to three man month) was needed for instrumenting the TV software and performing the measurements.

Qualitatively, the comparison of the two techniques is summarized in Table 10.2.

Summarizing, we can recommend the use of analytical formulas for obtaining performance predictions fast. On the other hand, the simulation-based approach may be needed, if the architects need higher prediction accuracy and/or they want to gain an insight into the performance relevant behavior of the software.

Table 10.2: Comparison of estimation via simulation and estimation using a simple formula

	Advantages	Disadvantages
Estimation using a simple formula	1. Simple 2. Fast to perform	1. Less accurate; always overestimates 2. Gives no insight about interactions
Estimation via simulation	1. Gives insight about scheduling and interactions 2. Higher accuracy	1. Many runs may be needed 2. Needs significant effort to construct the simulation model

10.7 Summary

In this chapter, we presented a case study (in the Consumer Electronics domain) aimed at verification of our hierarchical approach for predicting the performance of component compositions (see Chapter 9). We demonstrated the approach by predicting the average CPU utilization of the most CPU consuming activities from the TV software. These activities execute when the TV performs in steady state.

The Rate Monotonic Analysis (RMA) [KRP93] turned out to be not suitable for the case under consideration, as it concerns the worst-case CPU utilization and worst-case execution times. We however aimed at predicting the CPU utilization for the typical case. On the other hand, the constructed analytical formula resembles the formula from RMA quite well.

We had to skip the first step of the hierarchical approach: the modeling of component operations. The reason was the excessive measurement effort and the restricted reusability of Koala components from the TV software (see also Section 10.2).

Two approaches were considered for predicting the CPU utilization of the activities: (1) a simple analytical formula and (2) simulation. The simulation approach relies on the simulation and prediction models constructed for each activity by means of the APPEAR method. The simulation models are built not only for the individual activities, but also for the scheduler and shared resources.

Quantitatively, the average relative error provided by applying the simulation (-1.43%) is better than by using the simple analytical formula (5.04%). On one hand, using the simple analytical formula provided the results that were sufficiently accurate with respect to the required level of accuracy (5%). This kind of approach can be easily applied, when a fast guess is needed. We however expect that the simple-formula approach will provide significantly poorer results for heavily loaded systems. On the other hand, the simulation-based approach required the substantial effort of four man months for constructing the necessary models (see Section 10.6).

The performed analysis of the run-time architecture and the constructed models allowed us not only to predict the average CPU utilization, but also to gain an insight.

11 Performance prediction for component compositions in the Professional Systems domain

11.1 Objectives

The validation of the APPEAR method has already been described for isolated software components (see Chapter 7 and Chapter 8). This chapter describes a case study for the validation of our hierarchical approach (see Chapter 9) for the performance prediction of component compositions. The case study was performed within the MISS (Medical Imaging Software System) software stack for the composition of two components that influence each other via shared resources. Since both components had already been implemented, their cooperative performance could be directly measured and used to validate the results provided by the approach to performance prediction for component compositions. The two main objectives of this case study were:

1. *Validation of the approach for the prediction of the performance of component compositions.* The approach, described in Chapter 9, was applied for constructing the performance prediction model of component composition:

$$P_{compos} = f(P_1, ..., P_n). \quad (11.1)$$

   In this formula, P_{compos} is the performance of the composition; P_n are the performances of the component operations of the constituting components. The response time was chosen as a performance metric.
2. *Analysis of the cooperative execution of the MISS components.* We investigated the composition of two MISS components– "*Archiving*" and "*Reviewing*"– in order to gain architectural insights into their concurrent behavior, to study its impact on performance, and to suggest performance improvements (when possible).

This chapter is structured as follows. Section 11.2 tailors the main steps of the approach to performance prediction for component compositions to the context of the case-study. Section 11.3 gives an overview of the MISS software stack, its components and resources. Then, we apply the steps of the approach. Section 11.4 demonstrates the construction of the APPEAR models for the "*Archiving*" component. Section 11.5 considers the issues related to the composition of activities. Section 11.6 describes the performance prediction experiment for the validation of the approach. Section 11.7 and 11.8 suggests certain improvements of the performance of the composition under investigation and draws the conclusions on the case study, respectively.

11.2 Performance prediction for component compositions

Chapter 9 presents an approach to the estimation of the performance of a component composition. This approach tackles the estimation problem in three steps:

1. Estimation of the processor and resource demand of the operations of components in isolation.
2. Estimation of the processor and resource demand of the activities, consisting of several component operations, in isolation.
3. Estimation of the performance of activities in the composition. This step accounts for the interactions between activities and blocking and pre-emption delays due to the scheduling of shared resources.

We consequently perform the steps of the approach in the following way. During the first step, we construct the performance prediction model for the operations of the "*Archiving*" component in isolation. The similar models for the "*Reviewing*" component are described in Chapter 8. Note that each component operation corresponds to a particular use case.

During the second step, we model two activities. In our case, each activity contains only one operation of each component: *"Reviewing"* and *"Archiving"*. Thus, modeling of branches and loops is not needed, and this step is omitted.

During the third step, we analyze the contention of these two activities for shared resources. We estimate the response time of the *"Archiving"* activity in the composition with the *"Reviewing"* one. (The former is affected by the latter.)

It is important to realize that as for the individual components as for the components in composition, the same use cases are considered. More information about these use cases can be found in Section 11.6.2 and in Chapter 8.

11.3 Overview of the MISS components

11.3.1 MISS software structure

The structure of the MISS software is presented in Figure 11.1.

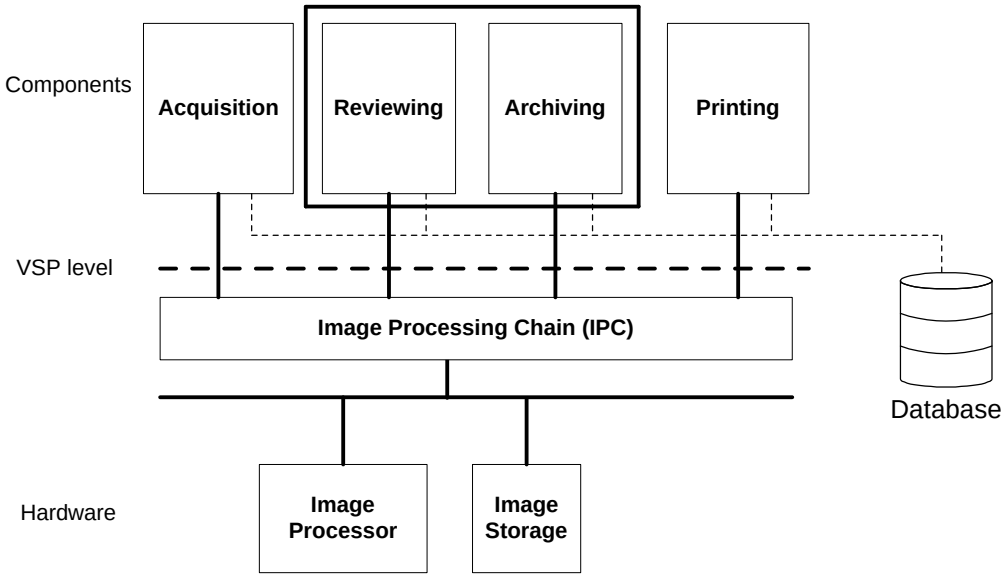


Figure 11.1: MISS software structure

We investigated the concurrent execution of the two MISS components: *"Reviewing"* and *"Archiving"* (see rectangle in Figure 11.1). These components execute on a common platform that provides a number of resources. One of these resources is the shared resource *"Image Processing Chain"* (*"IPC"*), which manages the *"Image Processor"* and *"Image Storage*" (see also Chapter 8). The scheduling of this resource is pre-emptive and priority-based. The priorities are enumerated in Table 11.1.

Table 11.1: Priorities of the MISS components

Component	Priority
Acquisition	High
Reviewing	Medium
Archiving, Printing	Low

11.3.2 Representation of "IPC" resource for components

Both the "*Reviewing*" and the "*Archiving*" components contend for the "*IPC*" resource. This resource is used to set the processing settings (attributes like brightness and contrast) of images before the images are displayed on the monitor or archived. The "*IPC*" resource can be in different states. The time required for acquiring the resource by a component depends on the state of the resource.

From a viewpoint of any component, the behavior of the "*IPC*" resource can be represented as follows (see Figure 11.2).

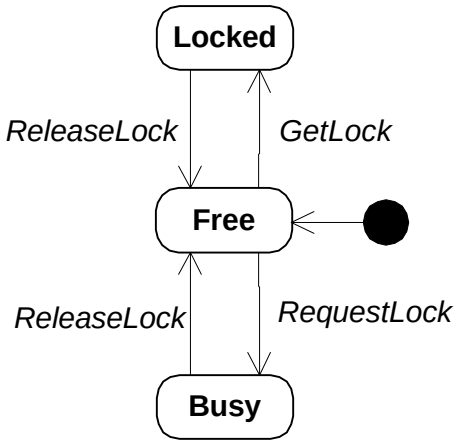


Figure 11.2: *"IPC"* behavior

The "*IPC*" resource can be owned by only one component at a certain moment of time. If the component does not need the resource, the resource becomes "*Free*". If the resource is owned by another component, it is in the "*Busy*" state. Before using the resource, a component has to lock it. It is important to distinguish between "*Busy*" and "*Free*" states, as it takes less time to lock the free resource (approximately 100 ms) than to acquire and to lock the resource in "*Busy*" state, i.e. owned by another component (approximately 500 ms).

11.4 APPEAR models for individual components (Step 1)

This section demonstrates the first step of our compositional approach. During this step, performance prediction models for the component operations in isolation are constructed (by means of the APPEAR method).

The models of the *"Reviewing"* component are described in Chapter 8. The models of the *"Archiving"* component are presented in this section.

The *"Archiving"* component is responsible for packing runs (collections of images) and transferring them to a remote archiving server.

For the performance study, the most frequently used performance-relevant use case was "*ExportToNetworkNode*". This use case allows the user to archive images on a remote or local server. Note that this use case corresponds to a particular component operation with the same name, i.e. the operation, which is invoked when this use case is executed.

Initial analysis of the component design and the execution traces showed that more than 90% of the total response time was consumed by the following four service calls[1]:

1. "*GetImage*" (78%). This call was used to obtain image pixel data from the Image Storage.

[1] Notice that these calls can only be invoked after component has locked the IPC resource

2. "*GetNextFrame*" (10%). This call was used to obtain image data (brightness, contrast, etc.) from the database.
3. "*Construct_IPC*" (2.5%). This call was used to create new "*IPC*" objects for each run.
4. "*Destruct_IPC*" (2.5%). This call was used to delete the "*IPC*" objects of the archived run.

However, further analysis showed that only two input parameters are relevant for the prediction model: 1) the number of images, and 2) the number of runs. The reason is that the number of first two service calls equals to the number of images, and the number of second two service calls equals to the number of runs. As a result, the signature type is

$$S = (\#Images, \#Runs) \tag{11.2}$$

It is important to realize that the performance relevant service calls and aspects have to be indicated and documented, as they still provide architectural insight into the performance relevant parts of the system, even if they are not directly used as signature parameters for the prediction model.

In this case, the component performance can be described by the prediction model only. The simulation model is not needed for the following reasons:

1. More than 90% of the response time is determined by the input parameters, and the number of service calls was equal to these parameters.
2. More than 90% of the response time is determined by the underlying hardware (covered by VSP), but not by the design/implementation details of the software.

The prediction model was constructed with the S-PLUS tool [4]. The resulting linear regression model had the following form:

$$Response = 432.22 \cdot \#Images + 1740.84 \cdot \#Runs - 4905.77 \tag{11.3}$$

In this formula, *Response* is the response time in milliseconds; *#Images* is the number of images to archive; *#Runs* is the number of runs to archive.

The p-values of the parameters of the regression model are presented in Table 11.2:

Table 11.2: P-values of the prediction model for *"Archiving"* component

Regression model parameter	*P*-values
#Images	0.0000
#Runs	0.0009
Intercept	0.0084

Practical rules of the use of linear regression [Wei95] usually suggest that the significance level of 0.05 is appropriate for most engineering appliances (see also Chapter 8). In this respect, the probabilities from Table 11.2 showed that all the parameters of the model were significant (as all *p*-values are smaller than the significance level of 0.05).

The high quality of this model is also confirmed by the following:

1. High values of the multiple regression coefficient: $R^2 = 0.9968$
2. Low average relative prediction error: $\overline{E} = 3.72\%$
3. Low maximal relative prediction error: $E_{\max} = 13.56\%$

In addition, the residual diagnostic plots are presented in Appendix Q.1.

11.5 Activity composition (Step 3)

In our case, each component operation is triggered by a separate activity. Thus, we skip the second step of our approach (modeling the activity control flow) and consider the activity composition only.

The activities compete for the "*IPC*" resource in the following way (see MSD in Figure 11.3).

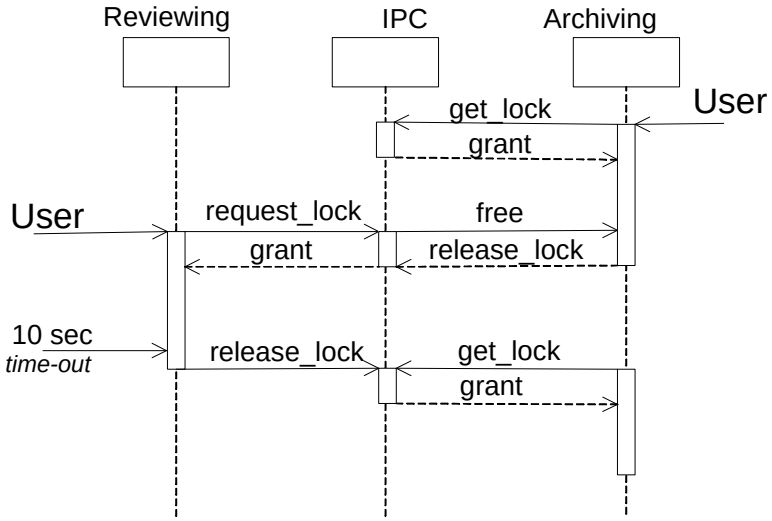


Figure 11.3: Pre-emption of the *"IPC"* resource

The "*Reviewing*" component can always pre-empt the "*Archiving*" component, as the former has a higher priority than the latter. This pre-emption includes releasing the "*Image Processing Chain*" by "*Archiving*" and locking it by "*Reviewing*". At the same moment the "*Reviewing*" component operation also occupies the CPU, and for this period the "*Archiving*" component has to be suspended. As soon as "*Reviewing*" completes its actions, it waits for a pre-defined time (time-out of 10 seconds) and releases the acquired resources. When the "*Archiving*" component is notified about the availability of the resource, it can lock the resource and resume execution.

In the case of multiple consecutive actions, "*Reviewing*" does not release the resources after each action, and a time-out of 10 seconds occurs only after the last action.

If necessary, "*Archiving*" can access the database during this 10 second time-out, (see Figure 11.4). Otherwise, "*Archiving*" remains blocked until the time-out expires.

We used the response time of the "*Archiving*" component as performance metric for the entire composition, as the "*Reviewing*" component has a higher priority and the response time of the "*Reviewing*" component cannot be affected by the "*Archiving*" component.

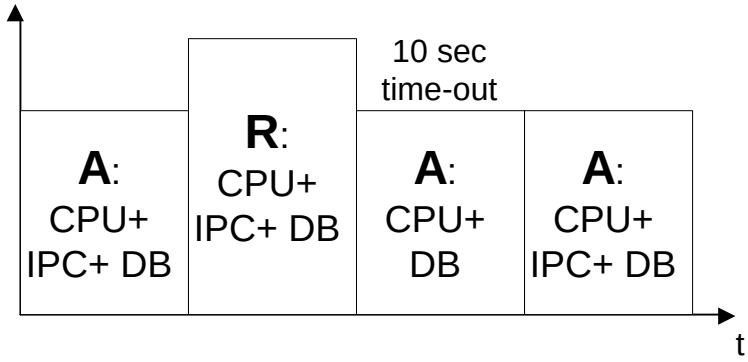


Figure 11.4: Example of resource usage by *"Archiving"* (A) and *"Reviewing"* (R)

Concluding all above, there are the following performance-relevant issues for the activity composition:

1. Priority-based pre-emption of the "*Archiving*" component by the "*Reviewing*" component.
2. Time required for acquiring the resources.
3. Time-out of 10 seconds that increases the response time of "*Archiving*" significantly.
4. The ability of the "*Archiving*" component to use the database during the 10 second time-out.
5. The invocation patterns of the "*Reviewing*" component.

The next chapter describes an experiment that validates our approach to performance prediction for component composition. The formula for the estimation of the response time of the "*Archiving*" component is constructed and confirmed by the experimental results.

11.6 Experiment description

11.6.1 Initial formula for performance prediction

Our hypothesis was that a simple analytical formula could be sufficient to obtain accurate performance estimates for the composition of the activities. Thus, we followed the following steps.

First, an initial analytic formula for the prediction of the response time of the component composition was constructed on the basis of the input from the architects, design documentation, system description (see above), and initial measurements. This formula relates the response times of the components executed in isolation to the performance-relevant factors, enumerated at the end of section 11.5:

$$T_{compos} = T_A + N \cdot T_R + N'' \cdot (T_{res} + 10) \qquad (11.4)$$

In Formula (11.4), T_{compos} denotes the response time of the "*Archiving*" component operation in composition; T_R is the execution time of the "*Reviewing*" operation; N is the number of "*Reviewing*" operations; T_A denotes the execution time of the "*Archiving*" component operation; N'' is the number of long intervals between the "*Reviewing*" operations, when "*Archiving*" operation locks the resources; T_{res} represents the time required for resource overtaking, and 10sec is the time-out before releasing the resources by "*Reviewing*" operation.

Second, this formula was verified against the measurements collected during the concurrent execution of the activities that invoke operations of the "*Reviewing*" and "*Archiving*" components. Numerous use cases (about 30) were executed to generate a sufficient amount of data. The comparison of measured data and estimations made by this formula had to confirm the following:

1. Formula (11.4) had no missing parameters that had significant impact on the performance of the composition.
2. The accuracy of formula (11.4) was 97.87% that was far above the requirements of the architects (50%).

11.6.2 Use case selection and execution

For the measurements, a number of performance-relevant use cases had to be chosen for both activities.

For the activity that uses "*Archiving*" component operation, the "*ExportToNetworkNode*" use case was selected (the same as for the calibration of the prediction model). For this experiment, the number of images was equal 100, which is a typical number for archiving procedures. This use case was chosen out of many others for the following reasons:

- This use case is time consuming and, thus, performance-relevant.
- The execution of this use case is usually pre-empted many times, which is a representative behavior.
- This use case can be executed in different configurations (different servers, network, local archiving, etc.).
- This use case is typical for a hospital setting: while archiving the patient images in the background, the doctor can perform a number of reviewing actions. The performance of the archiving action is important, since the doctor is interested in patient throughput, i.e., in completing the "*Archiving*" operations as soon as possible.
- This use case is easy to execute and to trace.

For the activity that uses the "*Reviewing*" component, two use cases were selected: "*ShowFileOverview*" and "*ShowRunOverview*" (that correspond to the same operations of the *"Reviewing"* component). During these use cases, the user can browse through the runs of a file or through the images of a run, respectively. These use cases were chosen for the following reasons:

- These are time-consuming use cases.
- These use cases are frequently performed by the users.
- During these use cases, the "*Reviewing*" component operation requires the "*IPC*" resource and, thus, pre-empts the "*Archiving*" component operation.
- These use cases are easy to execute and to trace.
- Because of the specifics of the MISS functionality, these use cases are coupled, i.e., it is not possible to continuously execute the "*ShowFileOverview*" use case only, but the two use cases have to be interleaved.

The use cases were executed for both components (see Figure 11.5). A single long archiving operation (marked with "A" in the figure; approximately 200 seconds) was pre-empted by multiple (30-50) short reviewing operations (marked with "R"; approximately 3 seconds).

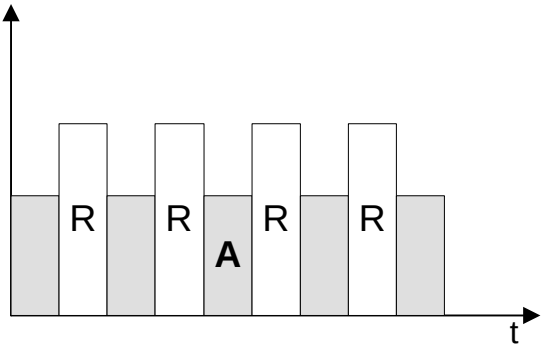


Figure 11.5: Execution of *"Reviewing"* and *"Archiving"* operations

The "*Reviewing*" use cases can have different invocation patterns: a) there are different time intervals between starting the "*Archiving*" use case and starting the "*Reviewing*" use cases and b) there are different time intervals between two subsequent "*Reviewing*" use cases.

During the experiment, the following sequence of steps was performed:

1. A single "*Archiving*" operation was started.
2. 20 "*Reviewing*" operations were performed with different intervals[2] between them: 3 seconds (short interval) or 12 seconds (long interval).
3. The "*Archiving*" operation was completed.

For the "*Archiving*" and "*Reviewing*" component operations in isolation, execution times were determined in advance by means of the APPEAR method (see section 11.4 and Chapter 8). The response time of the "*Archiving*" operation and the characteristics of "*Reviewing*" invocation patterns were stored in trace files.

11.6.3 Experiment results

The obtained trace files were used to extract the values of the response time of the "*Archiving*" action. The histogram of distribution frequency[3] of these response times is presented in Figure 11.6.

The response time T_{compos} lies in the following interval:

$$T_{compos} \in (140.000; 240.000) \tag{11.5}$$

Afterwards, we tried to restore the Formula (11.4) using the measurements above and the measurements of component operations in isolation. The measured response time for this use case required by "*Reviewing*" and "*Archiving*" component operations in isolation was the following:

$$T_A^i + 20 \cdot T_R^i = 51898 + 14000 = 65898\ ms \tag{11.6}$$

In Formula (11.6), T_A^i and T_R^i are the measured execution times of the "*Archiving*" and "*Reviewing*" component operations in isolation, respectively.

[2] We wanted to study the dependency of performance of the composition on the duration of the intervals between consecutive "*Reviewing*" operations. As the standard duration of the time-out was known beforehand (10 seconds), we purposefully chose for two types of intervals: shorter and longer than 10 seconds.

[3] Distribution frequency m describes the number of points p within particular intervals so that the following expression holds: $p = m \cdot N$, where N is the total number of points.

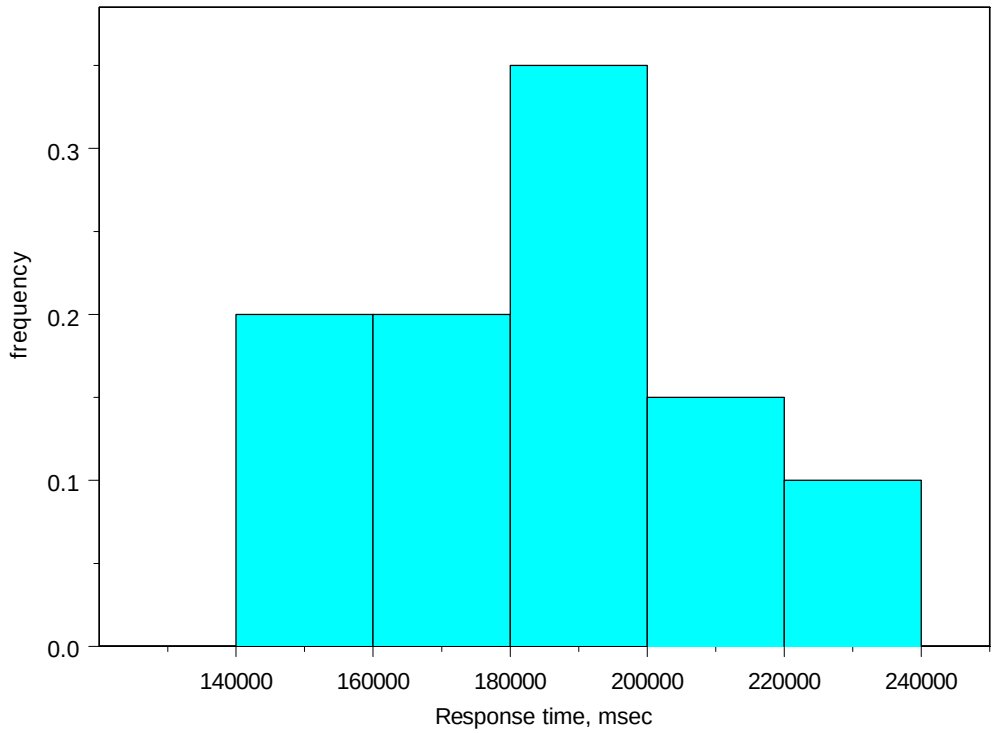


Figure 11.6: Histogram of response time of the *"Archiving"* operation in composition

Apparently, there was a difference between the measured values of the response time (see Formula (11.5)) and the sum of the response times of the component operations in isolation (see Formula (11.6)). The factors that provoke this difference needed to be identified. These factors determine the time difference $T_{unknown}$ that can be calculated as follows:

$$T_{unknown} = T_{compos} - T_A^i - 20 \cdot T_R^i . \qquad (11.7)$$

The input and configuration parameters of both components did not change during the experiment. It was decided to analyze the intervals between consecutive "*Reviewing*" actions, since this was the only varied parameter, and to check whether the moments of action invocations influenced the response time.

There were two types of intervals: a long interval (12 seconds) and a short interval (3 seconds). As the number of "*Reviewing*" operations was fixed and equaled 20 (see section 11.6.2), the following equation was always satisfied for the numbers of long and short intervals:

$$N_{long} + N_{short} = 20 \qquad (11.8)$$

Since N_{long} and N_{short} are strongly linearly correlated (see Formula (11.8)), both of these variables could not be an independent variable when calibrating a prediction model. It was thus decided to analyze the dependency of $T_{unknown}$ on the number of long intervals N_{long}. The prediction model describing the relation between $T_{unknown}$ and N_{long} was constructed with the S-PLUS tool [KO02] using linear regression techniques. The data used for the calibration is presented in Appendix Q.2.

The general form of the linear regression model was as follows:

$$T_{unknown} = a \cdot N_{long}, \quad with\, a = 10455.27 \tag{11.9}$$

The obtained value of a can be explained by two contributions: (a) the time-out of 10 seconds in the MISS software, associated with the long intervals between actions and (b) the average time required for resource acquisition. This explanation ensured that the formula (11.4) was a correct approximation, and no significant performance factors were overlooked. Both the explanatory power and the prediction quality of the model were high:

- A high value of the multiple regression coefficient: $R^2 = 0.9958$;
- A low values of the average relative error: $E = 5.68\%$.

The residual diagnostic plots, describing the quality of the model, are provided in Appendix Q.2. Table 11.3 demonstrates the quality of the prediction model built for $T_{unknown}$. It contains the measured values of $T_{unknown}$, predicted values of $T_{unknown}$ and relative prediction errors.

Table 11.3: Measured and predicted values of $T_{unknown}$

$T_{unknown}$, measured [ms]	$T_{unknown}$, predicted [ms]	Relative error
74473	52285	0.30
125904	125484	0.0033
150521	156855	0.042
102462	83656	0.18
132258	135941	0.028
113505	115027	0.013
96963	94113	0.029
90284	83656	0.073
115813	115027	0.0069
132872	135941	0.023
171164	167312	0.023
123629	125484	0.015
82057	73199	0.11
123689	125484	0.015
123460	125484	0.016
169063	177769	0.051
150751	156855	0.040
141751	146398	0.032
80215	73199	0.087
98666	94113	0.046

The values of the response time T_{compos} calculated by Formula (11.4) were also compared with the values of T'_{compos} obtained by means of the following formula (derived from the Formula (11.7)):

$$T'_{compos} = T_A^i + 20 \cdot T_R^i + T_{unknown} \tag{11.10}$$

The results and relative errors are presented in Table 11.4. The maximal relative error of the prediction was $E = 2.13\%$.

Table 11.4: Response times calculated by Formulas (11.4) and (11.10)

T_{compos}, formula, [ms]	T'_{compos}, formula, [ms]	Relative error
119098	118183	0.0077
193258	191382	0.0097
224858	222753	0.0094
150058	149554	0.0033
205498	201839	0.018
184858	180925	0.021
162938	160011	0.018
151338	149554	0.012
182938	180925	0.011
204218	201839	0.012
235178	233210	0.0084
195178	191382	0.019
140378	139097	0.0091
195178	191382	0.019
192618	191382	0.0064
246778	243667	0.012
225498	222753	0.012
214538	212296	0.010
141018	139097	0.014
161018	160011	0.0063

11.7 Suggestions for improvement

Based on the architectural insight gained during this case study, we made a number of suggestions for the performance optimization of the MISS software. The following modifications would decrease the total response time of the "*Archiving*" operations[4]:

1. *More flexible use of the database by the "Archiving" component.* Currently, the "*Archiving*" component can use the database during the 10 second time-out only in a certain state: when it retrieves the attributes of the images and runs from the database, but not when it retrieves image data from the "*Image Storage*". If data retrieval from the database were made state-independent, "*Archiving*" would be able to access the database during the 10 second time-outs, even if the "*IPC*" is occupied.
2. *Adaptive time-outs*. The fixed time-out of 10 seconds was introduced for better user perception, i.e., to avoid delays between successive "*Reviewing*" actions. During this time-out, the "*Archiving*" component is blocked, and its response time increases significantly. The value of the time-out could be adaptable, based on the history of the user behavior. This history can be used to predict the next user action and to choose the proper value for the time-out.
3. *Multiple access to the "IPC"*. Some operations of the "*IPC*" could be executed in parallel by two components, since none of the components needs all hardware resources at every moment. However, it should be investigated whether this modification does not affect the performance of the "*Reviewing*" component.

[4] Unfortunately, so far it is hardly possible to quantitatively estimate the factor of performance improvement for each modification. Moreover, there are no means to validate these estimates.

11.8 Summary

In this chapter, we presented a case study aimed at verification of our hierarchical approach for predicting the performance of component compositions. Two components from the MISS software were selected for the experiment. Both APPEAR models – simulation and prediction - were constructed for these components beforehand. The activities, using the operations of these components, were executed on the same VSP and influenced each other via shared resources.

The goal of the case study was to identify performance-relevant factors and to predict the response time of the composition. Based on the architectural knowledge, a simple analytical formula was found for the prediction of the response time. This formula was experimentally validated. Based on the measurements from the experiment, a linear prediction model was built. This model predicted the performance of the composition with high accuracy (average relative prediction error E=5.68%), and confirmed also the analytical formula.

We discovered a severe dependency of the response time of the "*Archiving*" component on the invocation patterns of the "*Reviewing*" actions. However, the knowledge of the invocation patterns does not provide much insight, as they are determined by user behavior and not by the architectural solutions.

The rigid resource scheduling policy and fixed time-outs drastically deteriorate the performance of the composition. These relevant factors should however be changed carefully, since they determine the user perception of the system.

The performance of the MISS components is also hardware bound, thus modeling the component internals is not needed. Instead, explicit modeling of the platform can be useful. However, this violates the APPEAR assumptions.

An approximate amount of effort for applying the approach turned out to be rather low and took 2.5 months in total. The major effort-consuming tasks were the following:

- Construction of the performance prediction model for an operation of the "*Archiving*" component (3 weeks),
- Analysis of the resource scheduling policy of the MISS software and deriving the initial formula for performance estimation (3 weeks),
- Collecting measurements of the concurrent execution of two activities (2 weeks),
- Validation of the initial formula for performance estimation (2 weeks).

One of the main findings of this experiment is that the performance of the component composition is far beyond optimal, despite the possibility to assess it by a simple formula with sufficient accuracy.

The future work can consist of two major parts: 1) validation of the approach for more complex component compositions, e.g. where simulation models of components and scheduler are needed for achieving an appropriate level of accuracy, and 2) investigation of the possibilities to optimize the MISS software according to the improvements suggested in section 11.7.

12 Conclusions

This chapter surveys the most relevant achievements and summarizes the results presented in this thesis. Our work contributes to the theory and practice of the modern software architecting. Software architecting is widely used nowadays, as it is an effective means for tackling the complexity of software-intensive products. The wide adoption of software architecting can be accounted to a number of aspects, crucial for the development of competitive software products: 1) enhanced communication between stakeholders, 2) reuse by supporting component frameworks and product families, and 3) making justified design decisions early. The first and the second aspects relate the organizational part of software architecting. Dealing with the third aspect, however, concerns technical solutions, and presumes two questions: which decisions should be taken, and what will be the impact of these decisions on the architecture, in terms of qualities, resource demands, etc. Early estimation of this impact is essential for reducing the development effort, increasing the predictability of the product quality attributes and by this shortening time-to-market, and for ensuring the success of a particular product-market combination. This hot issue is addressed by our research.

Nowadays, large software-intensive industrial systems are often assembled from hundreds of software components (see Chapter 1). The quality attributes often cannot be attributed to specific components, as they usually emerge from the cooperation between them. This phenomenon makes it difficult to reason about these emergent quality attributes in a compositional way, i.e. to derive the quality attributes (QA) of a composition from the QA's of separate components.

We claim to deliver not only scientific basis for assessment of QA's, but also to provide software architects with practical and feasible approaches to be applied in an industrial context. For this purpose, we started by observing the state-of-the-art problems in two domains of industrial software and identifying the most important ones. Our survey review identified the two instances of static and dynamic quality attributes: memory consumption and performance. These quality attributes were ranked as the most important ones by the majority of the architects of the investigated software projects. Quantitative estimates of memory consumption and performance were desperately needed to judge the feasibility of the software and to make appropriate design decisions. As a consequence, our research was focused on early estimation and prediction of the two aforementioned quality attributes.

In the subsequent sections, we summarize the research questions related to the estimation of quality attributes. We also describe the approaches that we developed to answer these questions, discuss the results of the application of these approaches to the industrial software, and present the contribution of our approaches. Afterwards, we present a number of lessons learnt during application of our approaches in industry. Finally, we propose a number of problems for the future research.

12.1 Research questions and answers

Our research concerned the estimation of additive static quality attributes and performance for component-based software. Here, we describe how we addressed these questions.

1. Specification and quantitative estimation of additive static quality attributes

Assessment of additive static QAs appeared to be crucial, as it helps architects to decide whether a particular component composition does not violate the resource constraints. For example, the architects want to check beforehand if a new feature can be introduced in a TV without changing the underlying hardware.

So far, there are only few approaches concerned with estimation of static QAs (e.g., memory consumption). This is due to the fact that, in general, the estimation of additive static QAs was considered easy. However, for component-based embedded software with exponentially growing complexity, strict resource constraints and high diversity, this problem became complex and therefore deserves being investigated.

As we aimed at delivering the approach applicable in industry, we quickly discovered the drawbacks of the existing approaches for the estimation of static quality attributes. These approaches (e.g., [USL00]) were developed for ideal software systems. Absence of the means for dealing with complex input and diversity parameters and the necessity to have the entire code of the component composition minimized the applicability of these approaches in industrial context. These methods also use complex mathematical theories that are not easily accepted in industry (for more details see Section 3.3).

We developed an industrial-strength approach that offsets above-mentioned disadvantages and accounts for the current needs of software architects. There are three most important features of this approach. First, the approach analyses the critical aspects influencing the QAs – component input/diversity parameters and component binding. Second, the approach allows the architect to specify QAs as a part of component description (e.g., by means of a reflection interface). Third, this approach estimates the QAs of the component composition based on the component specifications, diversity, and binding.

The approach was validated in the Consumer Electronics domain. We estimated the static memory consumption of TV software. The estimates had the maximum relative error of 1.80% that was more than sufficient to meet the accuracy requirements of the architects. The approach and the experiment are described in Chapter 4.

2. Specification and quantitative estimation of the performance of software components

We successfully tackled the problem of early performance estimation of adapted versions of software components. This estimation assists the architects in deciding on the performance characteristics of future functionalities and on the hardware resources to be required. As a result, less effort will be spent on the development of unfeasible products.

So far, the following approaches to early performance estimation were the most popular ones: simulation, queuing networks, and statistical modeling. The limitations of these methods, such as too strict mathematical assumptions, tremendous amount of modeling details, requirements for sufficient amount of measurements or entire code of the software, etc. do not allow architects to apply them directly in the domain of complex component-based software. They either provide architects with unreliable and inaccurate results or fail at all (for more details, see Chapters 3 and 5).

To meet the requirements from the industry and to compensate for the disadvantages of the current techniques, we developed the APPEAR (Analysis and Prediction of Performance for Evolving Architectures) method. This method combines the best elements of two existing estimation techniques: simulation and statistical modeling. A simulation model is used to describe evolving performance-relevant components that are not yet implemented. As a result of modeling only performance relevant details, the architect can

avoid to be overwhelmed by too many details. Statistical methods are used for abstracting from details that are not performance-relevant. Abstracting from irrelevant details helps one to reduce the complexity. The statistical approach is also employed to model those parts of a system that remain unchanged for a long time during the evolution of components, the so-called Virtual Service Platform (VSP) on which the investigated components execute. The proposed mix is supported by the fact that fewer and fewer software-intensive systems are currently being developed from scratch.

The APPEAR method works as follows. For the existing components, a simulation model is built. This model describes performance-relevant details only and allows the determination of performance-critical parameters. These parameters are used as inputs for the statistical performance prediction model. The prediction model is based on the measurements on the existing components. This model reflects the correlation between the performance metric of interest and performance-relevant parameters of the existing components. For the adapted component, only the simulation model needs to be adapted, and time-consuming and expensive implementation is not required. This model also calculates the performance-critical parameters. The prediction model for the existing components can, under certain assumptions[1], be fed with these parameters and used to extrapolate the performance of adapted components during the architecting phase.

Because of its flexibility, the APPEAR method can be used for the prediction of any quantitative QA. The limitations of the method are related to the satisfaction of the APPEAR assumptions (see Chapter 5) by the software system in question. For example, the APPEAR method does not work for components developed from scratch, as prediction model cannot be built in this case. Another important drawback of the method concerns the extensive measurements for construction of the statistical models.

The APPEAR method was validated in the Consumer Electronics and Professional Systems domains. In the first case, we applied the method to predict the average execution time of Teletext software. In the second case, we predicted the average response time of the viewing software for medical images. Although these domains differ, the APPEAR method allowed us to obtain predictions with a relative prediction error of less than 20%, which was sufficient according to the expectations of the architects.

3. Specification and quantitative estimation of the performance of component compositions

The goal of our research was to provide the software architects with an approach for the early prediction the performance of component compositions. This helps in justifying design decisions early, which may save considerable amounts of time and effort otherwise spent for the implementation of presumably poor performing software.

In principle, all the drawbacks, described in the second paragraph of the previous section, of the existing performance prediction approaches remain valid for the case of component compositions. Moreover, due to the more complex problem, these approaches require more effort and a longer learning curve. Thus, they are hardly suited for industry. The increased complexity of the problem can be attributed to the following factors: a) reconstruction of the performance model from the descriptions of separate components, b) complex interactions and data dependencies between components, and c) accounting for the effects of scheduling of the CPU and other resources.

We developed a hierarchical approach, based on the APPEAR principles, for predicting the performance of component compositions. This approach considers the major factors influencing the performance of component compositions: (1) component

[1] These assumptions relate to component similarity and are described in Chapter 6.

operations, (2) activities being executed in isolation, and (3) activities being executed concurrently. Also, the performance model of the entire system is built hierarchically. First, the contribution of component operations to performance is modeled by means of the APPEAR method. Then, the performance models of activities are specified. Finally, the parallel composition of activities is considered. During each analysis step, various models– analytical, statistical, simulation– can be constructed to specify the contribution of each factor to the performance of the composition. The architects can flexibly choose which model they use at each step, depending on their effort budget, accuracy requirements, and the performance insight to be obtained.

This hierarchical approach was also validated in two industrial projects from two domains. In the first case, the average CPU utilization of the TV software was estimated (see Chapter 10). As the amount of performance relevant details was large, a simple estimation formula did not provide us with the required accuracy. As a consequence, the simulation model for the schedule was built. In the second case, the average response time of the composition of two components dealing with medical images was predicted (see Chapter 11). As interaction and resource scheduling of the components were simple, an analytical formula was sufficient for the performance prediction. For both cases, the relative prediction error did not exceed 10%.

All proposed approaches (both for static and dynamic quality attributes) have an important feature: they allow the architect to flexibly trade estimation effort against estimation accuracy. This means that architect can choose the level of modeling in accordance with the accuracy requirements and timing budget he can afford.

12.2 Contribution of the developed approaches

This section summarizes the added value of the approaches described in this thesis in comparison to the existing methods.

1. **Specification and quantitative estimation of additive static quality attributes**
    1. Our approach is innovative as it aims at the estimation of additive static QAs of *component-based* software at the architecting phase. So far, there are only few approaches to solve this problem. An example is [USL00] for the memory size estimation. However, this method requires the entire code and is not suitable for component-based system. This is due to the fact that, in general, the estimation of additive static QAs was considered easy. However, for complex component-based software with resource constraints and high diversity, this problem becomes very complex and therefore deserves further investigation. Additionally, our approach does not require the entire source code, and allows also budgeting of static QAs. As the validation experiment showed, our approach is applicable for component-based software where composition is constructed in form of hierarchy, and static QA of the composition is determined by binding rules and diversity parameters.
    2. This approach proposes to specify static QAs as a part of the component description. As a result, the *consistency* of the component code and the specification of its QAs is easier to manage, in comparison to specifying the component and its static QAs separately. If the task of creating the QA specification is performed by the component designers, the architects can perform the evaluation with *less effort*. Also, the QA specifications are more precise, as the component designers usually know much more about the QA relevant details of component than the architect.
    3. This approach allows the practitioner to *trade estimation effort against estimation accuracy*. The approach flexibly chooses diversity and input parameters,

components, etc. to account for their impact on the QAs. This *saves a lot of effort* for the specification and estimation of QAs. This feature was missing in the approaches existing so far (e.g., [USL00],[ZG94]).

2. Specification and quantitative estimation of the performance of software components

1. The APPEAR method allows the architect to *flexibly choose the abstraction level* of modeling and to concentrate the performance-relevant details only. The application of many existing approaches that are based on simulation often ends up with a combinatorial explosion of details, as they tend to include all behavioral details into the model (see, e.g., [LS99]).
2. The APPEAR method lets the architect also *abstract from the details* that remain stable during the lifecycle of software. This is a crucial feature of the method to be applied to *modern software product families* where major parts often remain unchanged.
3. When applying the APPEAR method, the architects can obtain *insight* into the run-time architecture (via the simulation model) and the performance of the existing versions of a software system. Moreover, this insight can be described in terms of executable models that are *easier accepted* by software designers than complex mathematical expressions.
4. The prediction models of APPEAR *account for the information* about the performance of previous versions of a software system. This feature a) helps *not to miss performance-relevant parameters* that were discovered earlier, b) *gradually enhances* the prediction model with the knowledge about already implemented functions, and c) *decreases the modeling effort* when new functions are gradually added.
5. The APPEAR method allows one to make reasonably *accurate performance predictions* based on the simulation model of future versions of a software system *without implementing it*. The existing approaches that are based on statistical modeling, require implementation/prototyping to perform predictions for future versions (see, e.g., [BK02]).
6. The users of the APPEAR method are not required to obtain a deep knowledge about the techniques they use. They need *basic skills* in behavior simulation and a limited knowledge of statistical regression techniques. Many other existing approaches (see e.g., [ABI00]) often imply a long learning cycle of the techniques to be applied.
7. As simulation model of the APPEAR method is built for variable part of the software, and concentrates on performance-relevant details only, it is *simpler and requires less effort* than many conventional approaches to software simulation (see e.g. [JAI91]).
8. The APPEAR method allows the user to *trade estimation effort against estimation accuracy*. The approach flexibly chooses the abstraction level of modeling and performance relevant parameters in accordance with requirements on insight, accuracy, etc. This *saves a lot of effort* yet allowing the user to obtain predictions with the required level of accuracy.

3. Specification and quantitative estimation of performance of component compositions

This approach for the performance prediction of component compositions inherits the majority of the APPEAR method principles for single components. In addition, the hierarchical approach, has the following important advantages:

1. By applying the APPEAR method to each component in isolation, the approach *significantly reduces the effort and modeling complexity.* The existing approaches (see e.g., [HAT03]) cannot cope with the rapidly increasing number of details.
2. Our approach *accounts for the relevant* input parameters of components, branches, and loops. This means that the influence of these parameters on the performance is reflected either in the formula or in the simulation model or in the prediction model. The state-of-the-art approaches (see e.g., [BMW04], [SW02]) ignore this issue although it is often relevant in the context of complex industrial software systems and product families.
3. The architect can *flexibly choose the modeling technique* for each step of the approach. This is not the case for many existing approaches (like [MAD04], [SW02]). In addition, he can *choose the most adequate* method for each step.

12.3 Lessons learned

This section summarizes our findings when applying our approaches in the industrial context. Many findings are illustrated by examples from the case studies performed by us, and references to the corresponding chapters are provided.

12.3.1 Balancing the modeling effort and prediction accuracy

It is important to balance the modeling effort against the prediction accuracy. The accuracy is limited by measurement errors, modeling artifacts, and the regression technique used to predict performance. The closer the needed accuracy to these limits, the more effort has to be taken to further increase the accuracy. This effort will eventually become too high.

Maintaining this balance becomes especially important in the case when the system response time has a strict deadline or when the uses cases have many input parameters that have an impact of the performance. In the former case, the prediction accuracy requirements become tighter. In the latter case, the effort is greater than for simpler use cases, as these input parameters may in effect lead to the decomposition of the high-level use case into a number of simpler use cases. Both cases can be demonstrated by our experiences with the application of the APPEAR method to the Teletext acquisition subsystem (see Section 7.3.4 and Section 7.3.5). The constructed prediction model had to concern Teletext packets of a dozen different types, which led to 13 input parameters. Additionally, a soft deadline of 20 ms exists for the processing of all Teletext information received during a single VBI. This deadline makes the requirements on the prediction accuracy stricter. As a result, a lot of time was spent in accurate modeling the Teletext acquisition component.

12.3.2 Incorrect expression of dependencies by a statistical model

Linear regression tools help to construct and calibrate prediction models. The objective is to achieve the sufficient quality of the model in terms of absolute error, R^2-coefficient, etc. In some cases, the model can be perfect in statistical terms, but irrelevant in implementation terms (see example below). The possible reasons for that are a) the insufficient amount of measurements, b) large measurement errors, and c) inadequacy of linear regression with respect to the modeled phenomena. If the prediction model should reflect the implementation correctly, additional information (limitations, conditions, etc.) is required.

Example. Let us consider one of the linear regression models built for the MISS performance prediction (see Section 8.3.6).

$$Response = 409.90 + 31.89 \cdot \#Images - 31.43 \cdot \#LongUpdate + 77.85 \cdot \#Paint \quad (12.1)$$

In the Formula (12.1), some regression coefficients are negative. This happened because we calibrated the regression model on a particular set of measurements, and tried to achieve the best prediction quality (e.g., the lowest absolute error) for this particular set. The presence of negative coefficients can lead to a wrong impression about the dependencies in the system. It might seem that increasing the number of "*LongUpdate*" calls would decrease the response time, but this is of course nonsense.

12.3.3 Disjoint measurement clusters

A performance prediction model can provide accurate estimates only for the signature instances located within the measurements that were used to calibrate the model (see Section 5.11). Sometimes, the signature instances of particular use-cases can constitute several clusters located at a noticeable distance. Figure 12.1 shows an example for two signature parameters (S2 and S1).

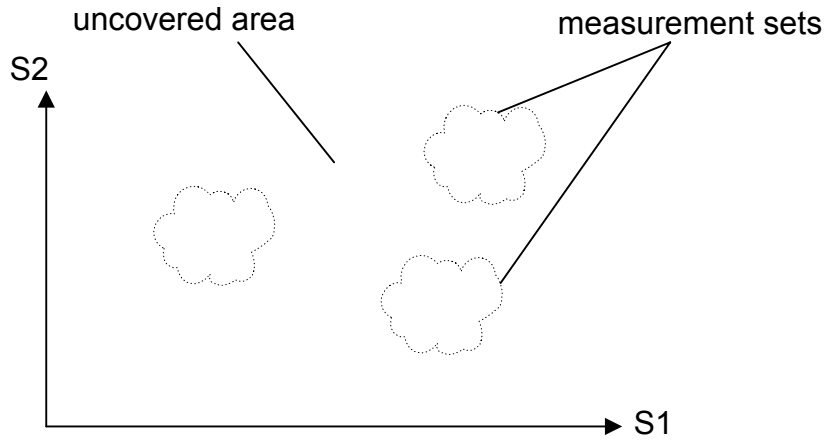


Figure 12.1: Decoupled groups of measurements

The rest of the signature space remains uncovered, and the predictions for points from this uncovered space can be inaccurate. Therefore, the calibration dataset should be analyzed before predicting the performance for new points. When dealing with this effect, two issues should be considered:

1. Selection of use cases that cover the entire range of signature parameters. Note that it is not always possible to find such use cases.
2. If the point to be predicted does not belong to one of the clusters, the prediction accuracy can decrease with the distance between this point and the covered signature instances (for more details, see Section 6.2.3).

12.3.4 Relevant choices

For the application of the APPEAR method during the case studies, it was necessary to choose between several alternatives. The choices usually traded effort needed for applying the method against the usefulness and accuracy of the results.

1. Use case (for example, see Sections 7.3.1 and 8.3.1).
2. VSP level (for example, see Sections 7.3.2 and 8.3.2).
3. Signature type (for example, see Sections 7.3.4 and 8.3.4).

4. Abstraction level of the simulation model (for example, see Sections 7.3.5 and 8.3.5).
5. Description formalism (for example, see section 8.3.5).
6. Tracing depth selection (for example, see Sections 7.3.3 and 8.3.3).
7. Validation strategy for the simulation model (for example, see section 8.3.6).

12.3.5 Use cases and input data for the simulation model

The simulation model is an abstraction of the software in question. Thus, it should be ensured that it adequately mimics its behavior, and generates the same values of the signature instances as the existing software. There are two issues to be considered when constructing a simulation model for the APPEAR method:

1. The same use cases must be possible to execute both on the existing component and on its simulation model. The prediction model must be calibrated for signature instances calculated by the simulation model to enable prediction for the adapted components. The validation of the model presumes comparison of the results generated by the actual component and the simulation model (see Section 8.3.5). Thus, it should be ensured beforehand that exactly the same set of use cases can be executed in both cases.
2. The same input parameters must be supplied both for the existing component and the simulation model thereof. This can also be difficult due to the large number of input parameters (for an example, see Section 8.3.5). The possibility and strategy of data retrieval should be considered in advance. Adapting of the tools for data retrieval (e.g., database parsers) can require unforeseen effort.

Example. In the MISS case study, all measurements were done on the test software configuration using a dedicated PC. This dedicated PC was available for experiments and measurement only for very short time. Therefore, the simulation model was executed on another computer. The MISS database was transported from the dedicated PC to the simulation PC. This database contained huge amount of input parameters used by the MISS software but not performance-relevant. Thus, we invested a significant amount of time in selecting the relevant parameters and modifying the database-parsing tool to feed the simulation model with the same input data as the existing software under consideration (see Section 8.3.5).

12.3.6 Performance prediction for adapted components

When predicting performance for the adapted component, three essential problems should be considered:

1. Proper specification of the new functionality. This requires the selection of a proper specification technique that must satisfy certain criteria. This technique must be expressive enough so that the specification can be easily depicted, intuitively understood and used for the efficient communication between designers, architects and performance experts. The specification technique should be acceptable for the designers that produce specification of the function, e.g. it should be widely known and applied for the other types of functional specification. There should also exist tools that enable automatic parsing of the specification to quickly and efficiently integrate the new function into the simulation model.
2. Extension of the simulation model. This extension concerns integrating the new functionality to the simulation model. The simulation model should be designed in such a way that its extension does not take too much effort.
3. Possibility to verify the performance predictions for new functions, although no implementation exists yet. In our case studies, we used the statistical prediction

intervals to judge on the prediction quality for the new functions (for example, see Section 5.11).

Example. In the MISS case study, Message Sequence Diagrams (MSD) were selected for the specification of new functions (see Section 8.4.1). The simulation model could be easily extended (see section 8.4.1) due to the automatic parsers of the MSD specifications. For the verification of predictions, the prediction intervals were used (see Section 8.4.2).

12.3.7 Problems with modeling the performance of component compositions

We collected the major problems that complicate modeling the performance of a component composition. These problems are confirmed by our case studies (for example, see Chapter 10).

1. *Fragmentation of information needed for predicting performance.* The performance is determined by the run-time architecture, as activities may pass through many components. It is cumbersome to reconstruct necessary models from the pieces related to different components. It is especially difficult if run-time architecture is not component-based (it can be monolith or there can be a complex mapping between components and tasks, etc).
2. *Complex control flow, and data dependences, and diversity parameters dependencies between components*. In component models widely applied in industry (COM, Koala, etc.), components usually interact through complex interfaces, which may have operations with many input parameters. Calling such operations introduces complex control flow dependencies between the operations of a caller component and the ones of a callee component. These complex dependencies must be modeled to obtain sufficiently precise performance estimates for the activities that involve those operations.
3. *Accounting for the scheduling of resources and processors.* Performance can be affected by delays due to blocking and preemption, occurring as artifacts of the scheduling of processors and other resources. These delays may be difficult to estimate (if possible at all).

12.3.8 Estimation of the accuracy of QAs of component compositions

Assessing the accuracy of the quality attribute estimate is not straightforward for the reason explained below. As stated in Section 4.7.2.2, for a single component, the accuracy of the estimate of a static quality attribute can be indicated by means of a prediction interval. It is therefore necessary to calculate a prediction interval for a sum of estimates of the static quality attribute. In statistical terms, this task amounts to estimating a prediction interval for the sum of estimates, each calculated by a linear regression model.

The literature survey showed that this is not at all simple. Moreover, standard books about statistics such as [WEI95], [MON01], and [MR03] do not cover this subject.

A more detailed analysis showed that this problem is similar to what in the statistics world is known as the Behrens-Fisher problem. The Behrens-Fisher problem is the problem of comparing the means of two normal populations with different and unknown standard deviations. Up to now, no good solution exists for this problem. All existent solutions can be criticized from the viewpoint of inference: (1) classical Neyman-Pearson inference with its confidence intervals, (2) Fisher's fiducial inference with its fiducial intervals, and (3) Bayesian inference with its credibility intervals. From a practitioner's

viewpoint, intervals of all three types may be used to judge about the likelihood that the true value of the difference of means lies with a particular interval. However, the aforementioned three types of inferences suggest their own statistical interpretation of this likelihood, see e.g. [Fis35], [Sal98], and it is questionable if these techniques are usable in practice.

12.4 Future research

This section sketches important issues for future research in the field of early prediction of quality attributes of component-based software.

1. Specification and quantitative estimation of static quality attributes

In the domain of the prediction of static quality attributes, the future research is necessary in the following directions:

1. *Validation of the suggested approach by applying it to other additive quality attributes*. A single experiment with the prediction of memory consumption is definitely an insufficient evidence for a broad application of the approach. Additional experiments with different component compositions, as well as with other additive QAs will help to check the applicability of the method and to explore its weak and strong points deeper.
2. *Extension of the approach with the composition rules for non-additive static quality attributes*. We delivered the approach (see Chapter 4) that was based on the assumption that static QAs are additive. However, there could exist other static attributes that are not necessarily add up when estimating the QA of the composition.

2. Specification and quantitative estimation of the performance of software components

The following investigations are important for the extension of the applicability of the APPEAR method:

1. *Weakening of the assumptions*. The applicability of the method is limited by a number of assumptions (see Section 5.4). Some of these assumptions are rather strict (e.g., the independence of the VSP services). It can be useful to check how critical the violation of the assumption is for the applicability and accuracy of the method. Development of the means and guidelines allowing the architects to obtain indications on the reliability of the predictions even when the assumptions are violated may be quite relevant (similar to the “escape routes” described in Section 6.3).
2. *Validation of the notion of component similarity by means of more experiments*. We have validated the notion of similarity of two components for a simple case only. For more complex components the similarity definition and the related conditions may need refinement.
3. *Further elaboration of the statistical part of the APPEAR method*. We applied only linear regression for calibrating the prediction model. Despite a sufficient prediction quality we achieved, research about the suitability of other regression techniques (e.g., non-linear) can be relevant. Additionally, a number of guidelines related to statistical modeling (e.g., how to improve the handling of outliers) should be added to the APPEAR method.
4. *Dealing with the evolution of the platform.* This issue is also related to weakening of the APPEAR assumption about stability of the VSP (see Section 5.4). In the general case, not only components, but also VSP can evolve (e.g., the hardware

platform is upgraded). The change of the VSP (both hardware and operating system) may have profound consequences on the software that runs on the top of this platform. Thus, we propose the construction of the APPEAR models (behavioral and statistical) of the existing and new platforms. The cooperative use of these models and the models of software components can provide architects with an estimate of the impact of platform change. These estimates are essential when, for example, a costly decision should be taken about porting a software stack to the next generation of hardware platforms.

5. *Derivation of design constraints.* The signature parameters that have an interpretation at the architecting level, e.g. the number of service calls of a certain type, can be used to establish design constraints. The a-priori assignment of values to these parameters may provide a more accurate way to prescribe resource consumption limits than the budgeting of the total CPU consumption of an application, a method that is presently common among the architects. The constraints for the unassigned, free signature parameters can be found by solving the inversed task to prediction, i.e., given the desired performance estimate values, what are the allowed values for the free signature parameters are:

$$\begin{aligned} & f(s_i^*, s_j) = p, \quad i, j \in [1, N] \wedge i \neq j \\ & p \in [p_{\min}, p_{\max}]. \end{aligned} \tag{12.2}$$

In formula (12.2), s_i^* are a-priori assigned signature parameters; s_j are free ones; p is the desired performance measure; $p_{\min}$ and $p_{\max}$ are the lower and the upper limits of the desired performance; $f(s_i^*, s_j)$ denotes the equation to be solved to obtain a set of values for the free signature parameters.

The steps of this approach are shown in Figure 12.2:

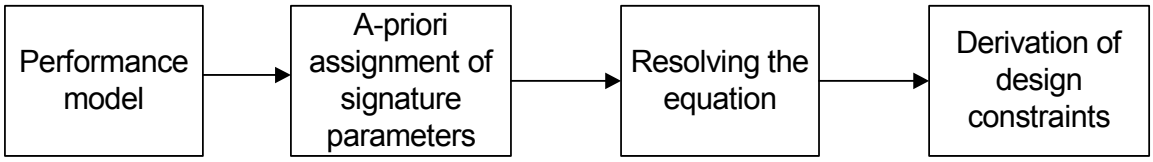


Figure 12.2: Deriving design constraints for desired performance

By solving the above equation, it might be possible to estimate the set of possible values of s_j, which describes desired design options.

3. Specification and quantitative estimation of performance of component compositions

The future work can be performed in the following directions:

1. *Validation of the approach by means of more industrial case studies*. The approach proposed in Chapter 9 was only partially validated by two case studies (see Chapters 10 and 11). In the first case study, a detailed simulation model had to be built to achieve the required accuracy. In the second cases study, a simple formula turned out to be sufficient due to simple scheduling of the resources. For demonstration of all the steps of the approach (e.g., activity modeling), thus, additional case studies are required.
2. *Elaboration on rules and guidelines for the selection between analytical, statistical and simulation approaches*. This would allow architects to flexibly select between these techniques based on their experiences and preferences. In addition, a set of

guidelines describing in which case a particular approach suits best should be developed.

3. *Elaboration on the construction of prediction models for component operations, branches, and loops*. In our hierarchical approach (see Chapter 9), the prediction models for branches and loops were constructed based on the strict assumptions such as exponential-family distribution of branching probabilities and loop counts. Also, the models for loops and branches were constructed and calibrated for relatively simple cases only. More experimental validation for more complex cases is required to weaken these assumptions and to evaluate the feasibility and reliability of the prediction models for branches and loops.

Appendix A. Relevant quality attributes in the Consumer Electronics and Professional Systems domains

A.1 Questionnaire for interviews with the architects

1. Stakeholders-related questions:
1.1 What relevant stakeholders can your remember?
1.2 What were their primary concerns?

2. What were the main contradictions within stakeholders concerns?
3. What compromises were accepted?

4. Component model questions:

4.1 What were the most significant requirements for the component model?
4.2 What were the possible alternatives for the component model?
4.3 How did the significant requirements reflect in adopted model?

5. What functional features were crucial for the product success?

6. What non-functional attributes were relevant to satisfy?

6.1 Were there any resource constraints? Which ones?
6.2 Was there necessity of legacy code support/reuse?
6.3 Were there any middleware (OS) preferences? Why?
6.4 Were there any subcontractors in the product? What was their contribution?
6.5 What were the important x-abilities (scalability, portability etc)?

7. Development process:

7.1 How the development process was organized (subdivision, coordination, interaction, communication)?
7.2 What tools were used during development (languages, environments),
7.3 What were the project deliverables?
7.4 What were the deployment units?
7.5 What was the role of architects during the initial stages of development? How did it change afterwards?

8. What risks and problems occurred?

A.2 Definition of quality attributes

Table A.1 presents the definitions for the QA's from Table 2.1.

Table A.1: Definitions of quality attributes.

QUALITY ATTRIBUTE	DEFINITION
Performance	The performance of a system characterizes how well the system performs with respect to the timing requirements [SW02]. It is measured as either the time required to respond to specific event or the number of events processed in a given interval of time [BKL95]
Timeliness	The ability of a system to perform a task at a correct time [Lar04]
Diversity	The diversity of a system relates to its ability to be tuned for a particular context of use by means of postponed design decisions (variation points)
Configurability	The ability to configure a subsystem or component for a particular environment. It is often called "technical diversity"
Reliability	A measure of the ability of a system to keep operating over time. A measure of the rate of failure in the system that renders the system unusable [BKL95]
Safety	The absence of the catastrophic consequences to the environment (user) [BKL95]
Availability	A measure of system's readiness for use. It is measure as the limit of the probability that the system is functioning correctly at time t, as t approaches infinity [BKL95]
Maintainability	The aptitude of a system to undergo repair and evolution [BKL95]
Extensibility	The ability for adding new functionality [Lar04]
Portability	The degree to which the software is independent from underlying middleware, operating system, and hardware
Scalability	The ability of a system to continue to meet its performance objectives as the size or functionality of a system significantly increase [SW02]
Reusability	The extent to which a software component or a subsystem can be reused within other (sub-) systems [Lar04]

Appendix B. Example of a XML specification

This section contains the entire text of the XML specification of a component "*IsCmx*" from Section 4.9.1, including the DTD and the specification of the formulas of the component resource (memory) consumption.

```
<?xml version="1.0"?>
<!DOCTYPE ComponentDefinition
  [
    <!-- Elements for the definition of a component -->
    <!--   Binding between modules, interfaces, and internal components -->
    <!--   is intentionally omitted -->
    <!ELEMENT ComponentDefinition (provides,requires,contains,resource)>
      <!ATTLIST ComponentDefinition name CDATA #REQUIRED path CDATA #REQUIRED>
    <!ELEMENT Prefix (#PCDATA)>
    <!ELEMENT provides (interface)+>
    <!ELEMENT requires (interface)+>
    <!ELEMENT interface (name,definitionName)>
      <!ATTLIST interface path CDATA #REQUIRED>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT definitionName (#PCDATA)>
    <!ELEMENT contains (module | component)+>
    <!ELEMENT module (name)>
    <!ELEMENT component (name)>
      <!ATTLIST component path CDATA #REQUIRED>
    <!-- Extensions for the specification of resource demands -->
    <!ELEMENT resource (XRAM_SIZE,XROM_SIZE,STACK_SIZE)>
    <!ELEMENT XRAM_SIZE (apply)>
    <!ELEMENT XROM_SIZE (apply)>
    <!ELEMENT STACK_SIZE (apply)>
    <!-- Declarations for used MathML Content elements and some extensions -->
    <!ELEMENT apply (
                (times, (apply | cn) , (apply | cn)+) |
                (plus, (apply | cn) , (apply | cn)+) |
                (minus, (apply | cn) , (apply | cn)) |
                (cc, ci) |
                (cif, ci) |
                ci |
                cn)>
    <!ELEMENT plus EMPTY>       <!-- Plus n-ary operator -->
    <!ELEMENT minus EMPTY>      <!-- Minus binary operator -->
    <!ELEMENT times EMPTY>      <!-- Multiplication n-ary operator -->
    <!ELEMENT cc (#PCDATA)>     <!-- Component name -->
    <!ELEMENT ci (#PCDATA)>     <!-- Identifier name  -->
    <!ELEMENT cn (#PCDATA)>     <!-- Number -->
    <!ELEMENT cif (#PCDATA)>    <!-- (Diversity) interface name -->
  ]
>

<ComponentDefinition name='CIsCmx' path='c:\Components\CIsCmx.xml'>
<provides>        <!-- The list of all provides interfaces -->
  <interface path='c:\Components\IInit.xml'>
    <name>iinit</name>
    <definitionName>IInit</definitionName>
  </interface>
  <interface path='c:\Components\IRtk.xml'>
    <name>irtk</name>
    <definitionName>IRtk</definitionName>
  </interface>
</provides>
```

```
<requires>        <!-- The list of all requires interfaces -->
  <interface path='c:\Components\IDiv.xml'>  <!-- A diversity interface of the CIsCmx component -->
    <name>idiv</name>
    <definitionName>IDiv</definitionName>
  </interface>
  <interface path='c:\Components\IRes.xml'>  <!-- Another diversity interface of the CIsCmx component -->
    <name>res</name>
    <definitionName>IRes</definitionName>
  </interface>
  <interface path='c:\Components\ITic.xml'>
    <name>itic</name>
    <definitionName>ITic</definitionName>
  </interface>
  <interface path='c:\Components\IIsv.xml'>
    <name>iisv</name>
    <definitionName>IIsv</definitionName>
  </interface>
  <interface path='c:\Components\ISfr.xml'>
    <name>isfr</name>
    <definitionName>ISfr</definitionName>
  </interface>
  <interface path='c:\Components\IStart.xml'>
    <name>istart</name>
    <definitionName>IStart</definitionName>
  </interface>
  <interface path='c:\Components\IInts.xml'>
    <name>iints</name>
    <definitionName>IInts</definitionName>
  </interface>
  <interface path='c:\Components\IPlf.xml'>
    <name>iplf</name>
    <definitionName>IPlf</definitionName>
  </interface>
</requires>

<contains>        <!-- Inner modules and components -->
  <module>
    <name>mcmx</name>
  </module>
  <component path='c:\Components\CMgCmx.xml'>
    <name>mgcmx</name>
  </component>
  <component path='c:\Components\CPlfCmx.xml'>
    <name>plfcmx</name>
  </component>
</contains>

<resource>            <!-- Specification of static memory demands -->
  <XRAM_SIZE>     <!-- 32+res.MaxTasks*2+div.MaxTalos*25+mgcmx.XRAM_SIZE -->
    <apply>
      <plus/>
      <cn>32</cn>
      <apply>
        <times/>
        <apply>
          <cif>res</cif>
          <ci>MaxTasks</ci>
        </apply>
        <cn>2</cn>
      </apply>
      <apply>
```

```
          <times/>
          <apply>
            <cif>div</cif>
            <ci>MaxTalos</ci>
          </apply>
          <cn>25</cn>
        </apply>
        <apply>
          <cc>mgcmx</cc>
          <ci>XRAM_SIZE</ci>
        </apply>
      </apply>
    </XRAM_SIZE>
    <XROM_SIZE>    <!--48 + plfcmx.XROM_SIZE-->
      <apply>
        <plus/>
        <cn>48</cn>
        <apply>
          <cc>plfcmx</cc>
          <ci>XROM_SIZE</ci>
        </apply>
      </apply>
    </XROM_SIZE>

    <STACK_SIZE>    <!--12 + (res.MaxTasks - 1)*8+mgcmx.STACK_SIZE-->
      <apply>
        <plus/>
        <cn>12</cn>
        <apply>
          <times/>
          <apply>
            <minus/>
            <apply>
              <cif>res</cif>
              <ci>MaxTasks</ci>
            </apply>
            <cn>1</cn>
          </apply>
          <cn>8</cn>
        </apply>
        <apply>
          <cc>mgcmx</cc>
          <ci>STACK_SIZE</ci>
        </apply>
      </apply>
    </STACK_SIZE>
  </resource>
  </ComponentDefinition>
```

Appendix C. Koala reachability rules

Reachability analysis strictly follows the rules below:

1. A module marked `present` is reachable.
2. If a module is reachable, then an interface bound with its base to the module is reachable.
3. If an interface is reachable, then an interface bound to its tip (directly or via a *switch*) is reachable.
4. If an interface is reachable, then a module bound to its tip (directly or via a *switch*) is reachable *only* if it contains any elements that are (or should be) implemented in C and can therefore not be compile-time determined by Koala. (*`inline` code* is considered to be-equivalent to C code and therefore not compile-time determinable by Koala.)
5. A component is reachable if at least one of its modules is reachable.
6. A module is reachable if its container component is reachable.
7. The reachability analysis stops when reaching an *`optional` interface*, unless it is connected to a switch.

Appendix D. Confidence and prediction intervals of the sum of predictions given by linear regression models

As motivated in Section 4.7.2, it must be possible to estimate prediction intervals for the sum of predictions obtained by linear regression models. This appendix addresses that problem. First, the necessary concepts and formulas of linear regression are introduced. Then, a solution to the problem is given based on these concepts and formulas.

D.1 Some facts about linear regression models

For the sake of brevity, it is more convenient to rewrite Formula 4.19 from Section 4.7.2 in a matrix form:

$$Y = X \cdot \hat{\beta} + E,$$
$$Y = \begin{pmatrix} y_1 \\ \dots \\ y_n \end{pmatrix}, X = \begin{pmatrix} 1\, x_{11} & \dots & x_{k1} \\ \vdots & \ddots & \vdots \\ 1\, x_{1n} & \dots & x_{kn} \end{pmatrix}, \hat{\beta} = \begin{pmatrix} \hat{\beta}_0 \\ \dots \\ \hat{\beta}_k \end{pmatrix}, E = \begin{pmatrix} e_1 \\ \dots \\ e_n \end{pmatrix}. \quad \text{(D.1)}$$

In these formulas, the following symbols are used:

- Y is the $n \times 1$ column vector of observed values of the dependent variable y;
- X denotes the $n \times (k+1)$ matrix that contains values of independent variables;
- $\hat{\beta}$ is the column vector of regression coefficients;
- E is the column vector of errors;

Notice that all elements of the first column of the matrix X equals one. This fact is convenient for modeling the constant term β_0 of a regression model.

The mean standard error of residual s_e can be found by the following formula:

$$s_e = \sqrt{\frac{\sum_{i=1}^{n} e_i^2}{n-k-1}}. \quad \text{(D.2)}$$

Note that s_e is an unbiased estimator of population standard deviation σ of errors. The random variable $\frac{s_e^2 \cdot (n-k-1)}{\sigma^2}$ has χ^2-distribution with $n-k-1$ degrees of freedom [MR03], [Jai91]. [WEI95].

Consider a particular point $x_p = (1, x_{p1}, ..., x_{pk})^T$ (symbol T denotes the transpose operator) in the space of independent variables. The mean value $\overline{y}_p$ is the mean value of the dependent variable is represented by the following formula:

$$\beta = \begin{pmatrix} \beta_0 \\ \vdots \\ \beta_k \end{pmatrix}, \quad \text{(D.3)}$$
$$\overline{y}_p = x_p^T \cdot \beta.$$

In Formula (D.3), β denotes the $(k+1)\times 1$ column vector of true regression coefficients (see Formula 4.18 from Section 4.7.2).

The estimate $\hat{y}_p = x_p^T \cdot \hat{\beta}$ of $\overline{y}_p$ is obtained by linear regression, the estimate $s_{\hat{y}_p}$ of its standard deviation is calculated by the following formula:

$$s_{\hat{y}_p} = s_e \sqrt{x_p^T \left(X^T X\right)^{-1} x_p}. \tag{D.4}$$

From the statistical literature [MR03], [Jai91], [WEI95], the random variable $t = \dfrac{\hat{y}_p - \overline{y}_p}{s_{\hat{y}_p}}$ is known to have the t-distribution with n-k-1 degrees of freedom. This fact is employed for calculating the confidence interval for $\overline{y}_p$ at confidence level α:

$$\hat{y}_p - s_{\hat{y}_p} \cdot t_{1-\alpha/2;n-k-1} < \overline{y}_p < \hat{y}_p + s_{\hat{y}_p} \cdot t_{1-\alpha/2;n-k-1} \tag{D.5}$$

In Formula (D.5), $t_{1-\alpha/2;n-k-1}$ denotes the $1-\alpha/2$-th quantile of the t distribution with $n-k-1$ degrees of freedom. Using the similar reasoning, a prediction interval for the new observation y_p at point x_p is described by the following formulas:

$$\begin{aligned} &s_{y_p} = s_e \sqrt{1 + x_p^T \left(X^T X\right)^{-1} x_p}, \\ &\hat{y}_p - s_{y_p} \cdot t_{1-\alpha/2;n-k-1} < y_p < \hat{y}_p + s_{y_p} \cdot t_{1-\alpha/2;n-k-1}. \end{aligned} \tag{D.6}$$

D.2 Confidence and prediction intervals for the sum of predictions of several linear regression models

Let us now consider that there are N multiple linear regression models. Each model corresponds with an estimation formula described in Section 4.7.2. The models are fitted in the following manner:

$$\begin{aligned} &Y^{(j)} = X^{(j)} \cdot \hat{\beta}^{(j)} + E^{(j)}, \\ &\overline{y}^{(j)} = x_p^{(j)T} \cdot \beta^{(j)}, \\ &\hat{y}^{(j)} = x_p^{(j)T} \cdot \hat{\beta}^{(j)} \qquad \forall j \in 1..N. \end{aligned} \tag{D.7}$$

In Formula (D.7), the index in round brackets enumerates through the prediction models of interest. Except this index, the formula uses the same notations as in formulas from section above.

We aim at deriving a confidence interval for the sum of predictions *N* linear regression models $\hat{y} = \sum_{j=1}^{N} \hat{y}_p^{(j)} = \sum_{j=1}^{N} x_p^{(j)T} \cdot \hat{\beta}^{(j)}$. The prediction intervals can be derived in the similar way.

To our best knowledge, this question is not yet covered in the literature. We however suggest an approach to solving this problem. The sum of predictions of dependent variables can be described by the following formula:

$$\hat{y} = \sum_{j=1}^{N} \hat{y}_p^{(j)} = \sum_{j=1}^{N} \left(\overline{y}_p^{(j)} + e^{(j)}\right) = \sum_{j=1}^{N} x_p^{(j)T} \cdot \beta^{(j)} + \sum_{j=1}^{N} e^{(j)}. \tag{D.8}$$

Here, $e^{(j)}$ denote predictions errors[1] obtained for each prediction model j. According to the results of linear regression, the random variables $e^{(j)}$ must have the normal distribution with mean zero and standard deviations $\sigma_j \cdot \sqrt{x_p^{(j)T}\left(X_j^T X_j\right)^{-1} x_p^{(j)}}$. The values of the standard deviations σ_j are however unknown.

It is necessary to distinguish the cases of equal and unequal population standard deviations of errors of all the linear regression models.

D.2.1 Equal standard deviations

As the prediction errors have the normal distribution and are independent of each another, the random variable e that denotes the prediction errors for the sum of linear regression models is also normally distributed with mean zero and standard deviation $\sqrt{\sum_{j=1}^{N}\sigma_j^2\left(x_p^{(j)T}\left(X_j^T X_j\right)^{-1} x_p^{(j)}\right)}$.

Suppose that $\sigma_j = \sigma \quad \forall j \in 1..N$. It is easy to derive the confidence interval for the total dependent variable y. First, we normalize the error term e:

$$z = \frac{e}{\sqrt{\sum_{j=1}^{N}\sigma_j^2\left(x_j^T\left(X_j^T X_j\right)^{-1} x_j\right)}} = \frac{\hat{y} - \overline{y}}{\sigma\sqrt{\sum_{j=1}^{N}\left(x_j^T\left(X_j^T X_j\right)^{-1} x_j\right)}}. \qquad \text{(D.9)}$$

Notice that the variable z has the normal distribution with mean zero and standard deviation one. Unfortunately, the actual value of standard deviation σ is unknown, and Formula (D.9) cannot directly be used to construct the confidence interval. However, it is possible to provide an unbiased statistic to estimate σ:

$$s_e = \sqrt{\frac{\sum_{j=1}^{N} s_{e_j}^2 \cdot \left(n_j - k_j - 1\right)}{\sum_{j=1}^{N}\left(n_j - k_j - 1\right)}}. \qquad \text{(D.10)}$$

In Formula (D.10), $s_{e_j}^2$ denotes the standard error of fit for the j-th prediction model. The variables n_j and k_j denote the number of observations made for the j-th model and the number of independent variables of this model, respectively.

It is easy to show that the statistic $\frac{s_e^2 \cdot \sum_{j=1}^{N}\left(n_j - k_j - 1\right)}{\sigma^2}$ has χ^2-distribution with $\sum_{j=1}^{N}\left(n_j - k_j - 1\right)$ degrees of freedom. Let us substitute the unknown parameter σ in Formula (D.9) with its estimate s_e:

[1] Notice that errors e_{ji} of i-th observation of the j-th regression model are normally distributed with zero mean and constant standard deviation σ_j, according to the assumptions of the linear regression.

$$t = \frac{\hat{y} - \overline{y}}{s_e \sqrt{\sum_{j=1}^{N} \left(x_j^T \left(X_j^T X_j \right)^{-1} x_j \right)}}. \tag{D.11}$$

In this formula, the variable t is known to have the t-distribution with $\sum_{j=1}^{N} \left(n_j - k_j - 1 \right)$ degrees of freedom. This fact can be used to calculate the confidence intervals with the confidence level α:

$$\begin{aligned} \hat{y}_u &= \hat{y} + s_e \sqrt{\sum_{j=1}^{N} \left(x_j^T \left(X_j^T X_j \right)^{-1} x_j \right)} \cdot t_{\left[1-\alpha/2; \sum_{j=1}^{N} \left(n_j - k_j - 1 \right) \right]}, \\ \hat{y}_l &= \hat{y} - s_e \sqrt{\sum_{j=1}^{N} \left(x_j^T \left(X_j^T X_j \right)^{-1} x_j \right)} \cdot t_{\left[1-\alpha/2; \sum_{j=1}^{N} \left(n_j - k_j - 1 \right) \right]}, \\ \hat{y}_l &< \sum_{j=1}^{N} \overline{y}^{(j)} < \hat{y}_u. \end{aligned} \tag{D.12}$$

In this formula, $\hat{y}_l$ and $\hat{y}_u$ denote the lower and upper limits, respectively, for the confidence or prediction interval, and $t_{[q;m]}$ is the q-th quantile of t-distribution with m degrees of freedom.

D.2.2 Unequal standard deviations

When the population standard deviations σ_i of errors are not equal for different regression models, it is impossible to use the same transformation as at the right side of Formula (D.9). In this case, the problem of finding the confidence interval of the sum of linear predictions becomes similar to what in the statistics world is known as the Behrens-Fisher problem. The Behrens-Fisher problem is the problem of comparison of two means of two normal populations with different and unknown standard deviations. To date, no solution exists to this problem such that this solution is not criticized from a viewpoint of different types of inference: (1) classical Neyman-Pearson inference with its confidence intervals, (2) Fisher's fiducial inference with its fiducial intervals, and (3) Bayesian inference with its credibility intervals. From a practitioner's viewpoint, intervals of all three types may be used to judge about the likelihood that the true value of the difference of means lies with a particular interval. However, the aforementioned three types of inferences suggest their own statistical interpretation of this likelihood, e.g. [Fis35], [Sal98].

For each prediction model, let us consider a variable t_j, which is calculated as follows:

$$T_j = \frac{\hat{y}^{(j)} - \overline{y}^{(j)}}{s_e^{(j)} \cdot \sqrt{x_p^{(j)T} \left(X_j^T X_j \right)^{-1} x_p^{(j)}}} = \frac{x_p^{(j)T} \cdot \hat{\beta}^{(j)} - \overline{y}^{(j)}}{s_e^{(j)} \cdot \sqrt{x_p^{(j)T} \left(X_j^T X_j \right)^{-1} x_p^{(j)}}}. \tag{D.13}$$

As stated in Section D.1, the variable T_j has the t-distribution with $n_j - k_j - 1$ degrees of freedom. Its cumulative distribution function is defined as follows: $F_{T_j}(t) = P(T_j < t), \ \forall t \in R$.

It is also easy to show that

$$P\left(\overline{y}^{(j)} > x_p^{(j)T}\cdot\hat{\beta}^{(j)} - t\cdot s_e^{(j)}\cdot\sqrt{x_p^{(j)T}\left(X_j^T X_j\right)^{-1}x_p^{(j)}}\right) = F_{T_j}(t), \tag{D.14}$$

or equivalently

$$P\left(\overline{y}^{(j)} < x_p^{(j)T}\cdot\hat{\beta}^{(j)} - t\cdot s_e^{(j)}\cdot\sqrt{x_p^{(j)T}\left(X_j^T X_j\right)^{-1}x_p^{(j)}}\right) = 1 - F_{T_j}(t). \tag{D.15}$$

Let us denote $x_p^{(j)T}\cdot\hat{\beta}^{(j)} - t\cdot s_e^{(j)}\cdot\sqrt{x_p^{(j)T}\left(X_j^T X_j\right)^{-1}x_p^{(j)}}$. Formula (D.15) can then be rewritten as

$$F_{\overline{y}^{(j)}|\hat{\beta}^{(j)},s_e^{(j)}} = P\left(\overline{y}^{(j)} < v\right) = 1 - F_{T_j}\left(\frac{x_p^{(j)T}\cdot\hat{\beta}^{(j)} - v}{s_e^{(j)}\cdot\sqrt{x_p^{(j)T}\left(X_j^T X_j\right)^{-1}x_p^{(j)}}}\right). \tag{D.16}$$

Formula (D.16) describes the conditional distribution of the unknown parameter $\overline{y}^{(j)}$ in the light of the observed data (represented by the statistics $\hat{\beta}^{(j)}$ and $s_e^{(j)}$ calculated upon it). This distribution is called the fiducial distribution of the parameter $\overline{y}^{(j)}$ and can be used to make inferences about values of this parameter, by considering it a random variable.

So, according to the Fisher's fiducial argument [Fis35], [Sal98], it is possible to rewrite Formula (D.16) in the following form:

$$\overline{y}^{(j)} = x_p^{(j)T}\cdot\hat{\beta}^{(j)} - T_j\cdot s_e^{(j)}\cdot\sqrt{x_p^{(j)T}\left(X_j^T X_j\right)^{-1}x_p^{(j)}}. \tag{D.17}$$

In this formula, $\overline{y}^{(j)}$ and T_j are treated as random variables, whereas the rest of parameters are constants. By summing $\overline{y}^{(j)}$ for all regression models, the following is obtained:

$$\overline{y} = \sum_{j=1}^{N}\overline{y}^{(j)} = \sum_{j=1}^{N}x_p^{(j)T}\cdot\hat{\beta}^{(j)} - \sum_{j=1}^{N}T_j\cdot s_e^{(j)}\cdot\sqrt{x_p^{(j)T}\left(X_j^T X_j\right)^{-1}x_p^{(j)}} \tag{D.18}$$

Formula (D.18) can be used to generate the confidence (fiducial) interval. For the confidence level α, this interval is calculated by the following formula:

$$\sum_{j=1}^{N}x_p^{(j)T}\cdot\hat{\beta}^{(j)} - u_{1-\alpha/2} < \overline{y} < \sum_{j=1}^{N}x_p^{(j)T}\cdot\hat{\beta}^{(j)} + u_{1-\alpha/2}, \tag{D.19}$$

where u_q denotes the q-th quantile of the distribution of the random variable $u = \sum_{j=1}^{N}T_j\cdot s_e^{(j)}\cdot\sqrt{x_p^{(j)T}\left(X_j^T X_j\right)^{-1}x_p^{(j)}}$, which is a linear combination random variables that have the t-distribution. Please notice that the prediction (fiducial) interval can also be obtained using the arguments above. The final formula for the prediction interval resembles Formula (D.19): $\overline{y}$ must be substituted with y, the prediction of sum of single future measurements of each linear regression model, and u_q is the q-th quantile of the distribution of the random variable $u' = \sum_{j=1}^{N}T_j\cdot s_e^{(j)}\cdot\sqrt{1 + x_p^{(j)T}\left(X_j^T X_j\right)^{-1}x_p^{(j)}}$.

The argumentation given above allowed us, using the Fisher's fiducial argument [Fis35], [Sal98], to reduce the problem of finding the confidence interval to the problem of

calculating quantiles of the distribution of a linear combination of t-distributed variables. The latter problem cannot be solved analytically, but numerical solutions are possible.

We consider two approaches to finding the quantiles of the distribution of a linear combination of *t*-distributed random variables: (1) using the characteristic function of a distribution, and (2) simulation. The two approaches are described in the subsequent sections.

A) Characteristic-function-based approach.

Witkovský described a numerical solution to the problem of finding the quntiles of the distribution of a linear combination of t-distributed random variables that involves one numerical integration in [WIT01]. This solution is based on employing a nice property of *characteristic functions* of independent random variables with respect to the linear combination of these random variables.

The characteristic function $\phi(t)$ of a random variable u with the cumulative distribution function $F_u(x)$ is defined as follows:

$$\phi_u(t) = \int_{-\infty}^{\infty} e^{-itx} dF_u(x). \tag{D.20}$$

According to [Gil51], the following inversion formula also holds:

$$F_u(x) = \frac{1}{2} - \frac{1}{\pi}\int_0^{\infty} \mathrm{Im}\left(\frac{e^{-ixt}\phi(t)}{t}\right) dt. \tag{D.21}$$

Witkovský showed that $\lim_{t\to 0} \mathrm{Im}\left(\frac{e^{-ixt}\phi_u(t)}{t}\right) = E(x) - x$ and $\lim_{t\to\infty} \mathrm{Im}\left(\frac{e^{-ixt}\phi(t)}{t}\right) = 0$, which allows the integral from Formula (D.21) be integrated numerically from 0 to a sufficiently large T, $0 < T \le \infty$.

Let us consider a linear combination of independent random variables $u = \sum_{j=1}^{N} \lambda_j u_j$, and let $\phi_{u_j}(t)$ denote the characteristic function of the random variable $u_j, i = 1..N$. The characteristic function $\phi_u(t)$ of the random variable u can be found as follows:

$$\phi_u(t) = \phi_{u_1}(\lambda_1 t)...\phi_{u_N}(\lambda_N t). \tag{D.22}$$

In [Ifr72], Ifram derived the characteristic function of a t-distributed random variable T_j with v_j degrees of freedom:

$$\phi_{T_j}(t) = \frac{1}{2^{v_j/2-1}\Gamma(v_j/2)} \left(v_j^{1/2}|t|\right)^{v_j/2-1} K_{v_j/2-1}\left\{v_j^{1/2}|t|\right\}. \tag{D.23}$$

In this formula, $\Gamma(v)$ denotes the *gamma* function, and $K_\alpha\{z\}$ denotes the modified Bessel function of second kind. Finally, to find the confidence or prediction interval with the significance level α, it is necessary to numerically solve for *x* the equation

$$F_u(x) - (1 - \alpha/2) = 0, \tag{D.24}$$

where $F_u(x)$ is obtained by using Formulas (D.21), (D.22), and (D.23). The constants λ_j and degrees of freedom v_j for Formula (D.23) are calculated as follows:

$$\lambda_j = s_e^{(j)} \cdot \sqrt{x_p^{(j)T} \left(X_j^T X_j \right)^{-1} x_p^{(j)}}$$
$$v_j = n_j - k_j - 1 \tag{D.25}$$

Let us denote the solution to Equation (D.24) as $u_{1-\alpha/2}$. This is exactly the quantile from Formula (D.19).

B) Simulation-based approach

The simulation-based approach is based on generating a large sample of the values of a random variable. This large sample is then sorted in an ascending manner, and the element of this sorted sample is chosen such that its index corresponds to the required probability. For instance, for the 0.975-quantile the 9750-th element of a sample consisting of 10000 elements can be chosen. This element is the taken as the estimate of the required quantile.

Repeating this procedure for a number of times and averaging over chosen elements will increase the accuracy of the estimate of the quantile.

This approach can be applied to the problem of finding quantiles of the distribution of a linear combination $u = \sum_{j=1}^{N} T_j \cdot s_e^{(j)} \cdot \sqrt{x_p^{(j)T} \left(X_j^T X_j \right)^{-1} x_p^{(j)}}$ of *t*-distributed random variables, as it is known how to generate random variables that equal these linear combinations.

Appendix E. Details of the prediction model for the Teletext software

This appendix summarizes the quality of the prediction model constructed (see Section 7.4). Table E.1 presents the values of the regression coefficients together with the corresponding *t*-statistics and their *p*-values. The table demonstrates that all regression coefficients are significant at the 0.05 significance level, as all *p*-values in the last column are less than this significance level.

Table E.2 provides the summary about the explanatory power of the model. The R^2-coefficient demonstrates the high explanatory power, and the *p*-value, being 0, of the *F*-statistic shows that the regression is significant at the 0.05 significance level.

Table E.1: Linear regression coefficients

Coefficient	Value	*t*-value	Pr(>|t|)
β_0	0.00004350811	8.7137	0.0000
β_1	0.0001624039	20.2392	0.0000
β_2	0.00003465766	3.3451	0.0008
β_3	-0.00009810098	-9.4839	0.0000
β_4	0.00002035258	314.5034	0.0000
β_5	0.00001901177	736.7930	0.0000
β_6	0.00001312129	37.4164	0.0000
β_7	0.00002954716	43.6820	0.0000
β_8	0.0001833823	180.0632	0.0000
β_9	0.00006551284	160.2898	0.0000
β_{10}	0.001698615	175.3527	0.0000
β_{11}	0.001206774	152.1587	0.0000
β_{12}	0.001130729	123.8286	0.0000
β_{13}	0.0005467161	210.1833	0.0000

Table E.2: Prediction model summary

Residual standard error	0.0004052 (on 33784 df)
Multiple R-Squared	0.9746
F-statistic	99900 (on 13 and 33784 df), the p-value = 0

Figure E.1 shows the most important residual diagnostic plots. The use of these plots is detailed in 0.

The standard deviation of the residual appears to be constant (see plot a) because of the following reasons:

- The magnitudes of residuals (the y axis) do not depend on the fitted values (the x axis),
- The points around the line “y=0” lie uniformly and there is no visible structure in the residuals.

The residual-fit spread (see plot e) demonstrates high explanatory power, as the spread of the residuals is less than the spread of the fitted values. Plot c) does not indicate the violation of linearity, as most of the point lie around the line “y=x” in a uniform manner.

The rest of the plots indicate some problems. *First*, plot d) demonstrates that the residual distribution is not normal (heavy-tailed), as points do not lie on the “y=x” line at

the ends of the distribution. *Second*, there are outliers, as demonstrated by plot b): a few points lie far apart from the rest of the points. Finally, influential observations are also present, because plot f) contains a number of points for which the Cook's distance is greater than 1.0.

Thus, the residuals need stabilizing. This can be done (1) by eliminating the outliers and influential observations or (2) by using other regression techniques (e.g., robust regression). However, we did not stabilize the residuals, as no use of inferences about the regression coefficients and predictions was planned.

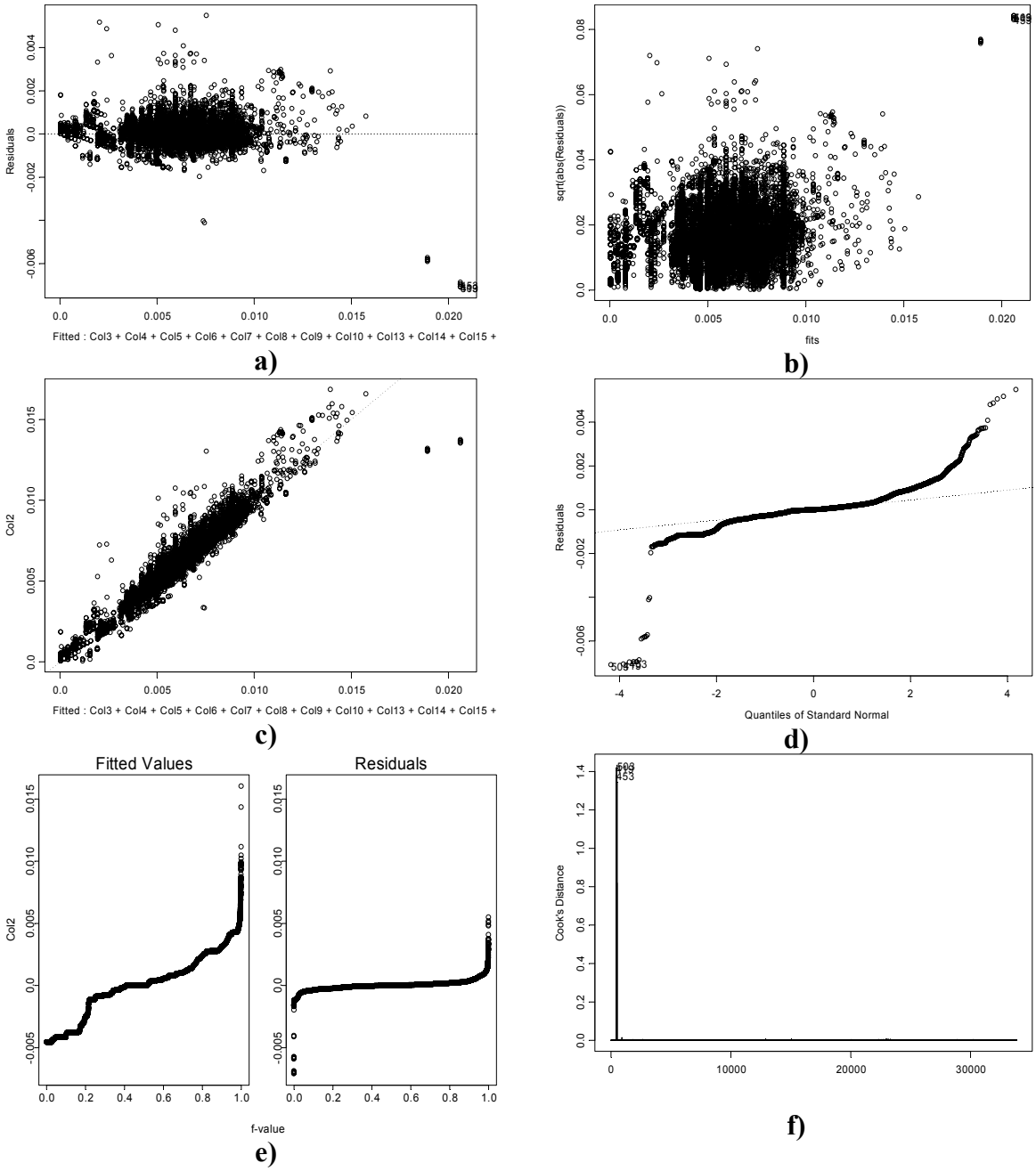


Figure E.1: Residual diagnostic plots: a) – Residual fit plot (constancy of standard deviation), b) The square root of absolute residuals versus fit (outlier diagnostic plot), c) Actual vs. fitted value, d) residual normal plot (the normality of residual distribution), e) residual-fit spread (explanatory power), and f) Cook's distance (diagnostic of influential observations)

Appendix F. Details of the prediction model for the MISS software

This appendix summarizes the quality of the prediction model constructed (see Section 8.3.6). Table F.1 and Table F.3 present the values of the regression coefficients together with the corresponding *t*-statistics and their *p*-values (last column). The table demonstrates that the majority of the regression coefficients for both cases are significant at the 0.05 significance level, as all *p*-values in the last column are less than this significance level.

Table F.2 and Table F.4 provide the summary about the explanatory power of the model. The R^2-coefficient demonstrates the high explanatory power for both cases, as it is close to 1.0. The p-value, being 0, of the *F*-statistic shows that the regression is significant at the 0.05 significance level.

Figure F.1 and Figure F.2 contain a number of diagnostic plots for both cases of the prediction model.

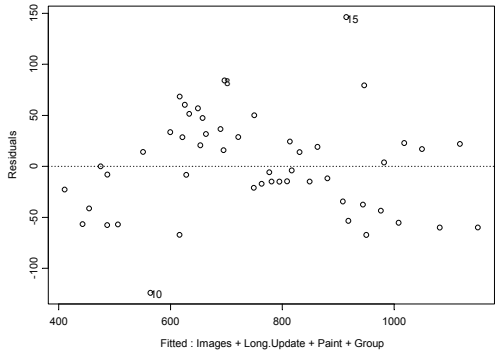
F.1 Prediction model case 1

Table F.1: Linear regression coefficients

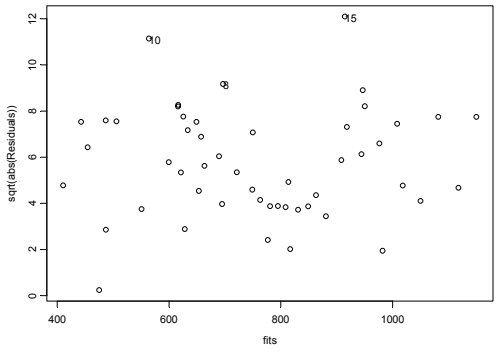
COEFFICIENT	VALUE	STD. ERROR	T-VALUE	PR(>|T|)
β_0 (Intercept)	409.8961	37.2172	11.0136	0.0000
β_1 (Images)	31.8872	1.9140	16.6603	0.0000
β_2 (LongUpdate)	-31.4294	11.9548	-2.6290	0.0118
β_3 (Paint)	77.8499	16.4290	4.7386	0.0000
β_4 (Group1)	-25.6175	14.4961	-1.7672	0.0843
β_5 (Group2)	-17.6571	14.7957	-1.1934	0.2393
β_6 (Group3)	39.6964	5.8666	6.7665	0.0000
β_7 (Group4)	18.5561	6.8267	2.7182	0.0094

Table F.2: Prediction model summary

Residual standard error	52.4 (on 43 df)
Multiple R-squared	0.9386
F-statistic	93.97 on 7 and 43 degrees of freedom, the p-value is 0



a)



b)

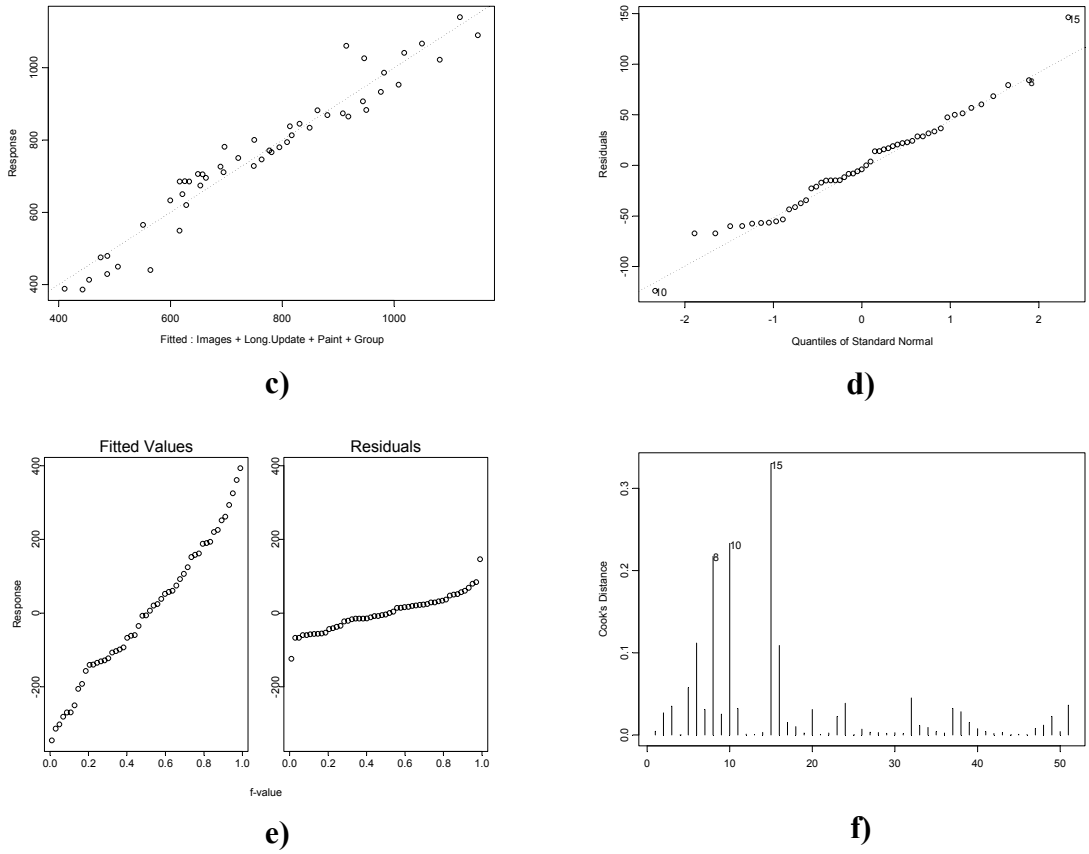


Figure F.1: Diagnostic plots: a) – Residual fit plot (constancy of standard deviation), b) Oultier diagnostic plot, c) Actual vs. fitted value, d) residual normal quantile-quantile (normality of residual distribution), e) residual-fit spread (explanatory power), and f) Cook's distance (diagnostic of influential observations).

Plot a) shows that residual is distributed normally and residual standard deviation is constant, as there is no structure or trends in the distribution "residual vs. fit". This enables the application of prediction intervals. Plot b) indicates a few outliers that do not have significant influence on the prediction quality. Plot c) confirms the linear dependency between response time and signature parameters, as the majority of the points are located within the "y=x" plot. Plot d) proves once again the normal distribution of the residuals, as 99% of the points are located close to "y=x" plot. Plot e) demonstrates high explanatory power of the prediction model, since the residual spread is considerably narrower than fitted values spread. Plot f) shows the absence of the influential observations (i.e. the regression coefficients are not sensitive to changes in any point from calibration dataset), since there is no point with Cook's distance greater than 1.0. More information about the interpretation of the diagnostic plots can be found in Appendix R.

F.2 Prediction model case 2

Table F.3: Linear regression coefficients

COEFFICIENT	VALUE	STD. ERROR	T-VALUE	PR(>|T|)
β_0 (Intercept)	360.1954	113.4709	3.1743	0.0033
β_1 (Runs)	99.8763	18.3074	5.4555	0.0000
β_2 (LongUpdate)	19.7293	5.0748	3.8877	0.0005
β_3 (Paint)	23.1766	60.9672	0.3801	0.7063

Table F.4: Prediction model summary

Residual standard error	78.75 (on 32 df)
Multiple R-squared	0.9907
F-statistic	1136 on 2 and 32 degrees of freedom, p-value is 0

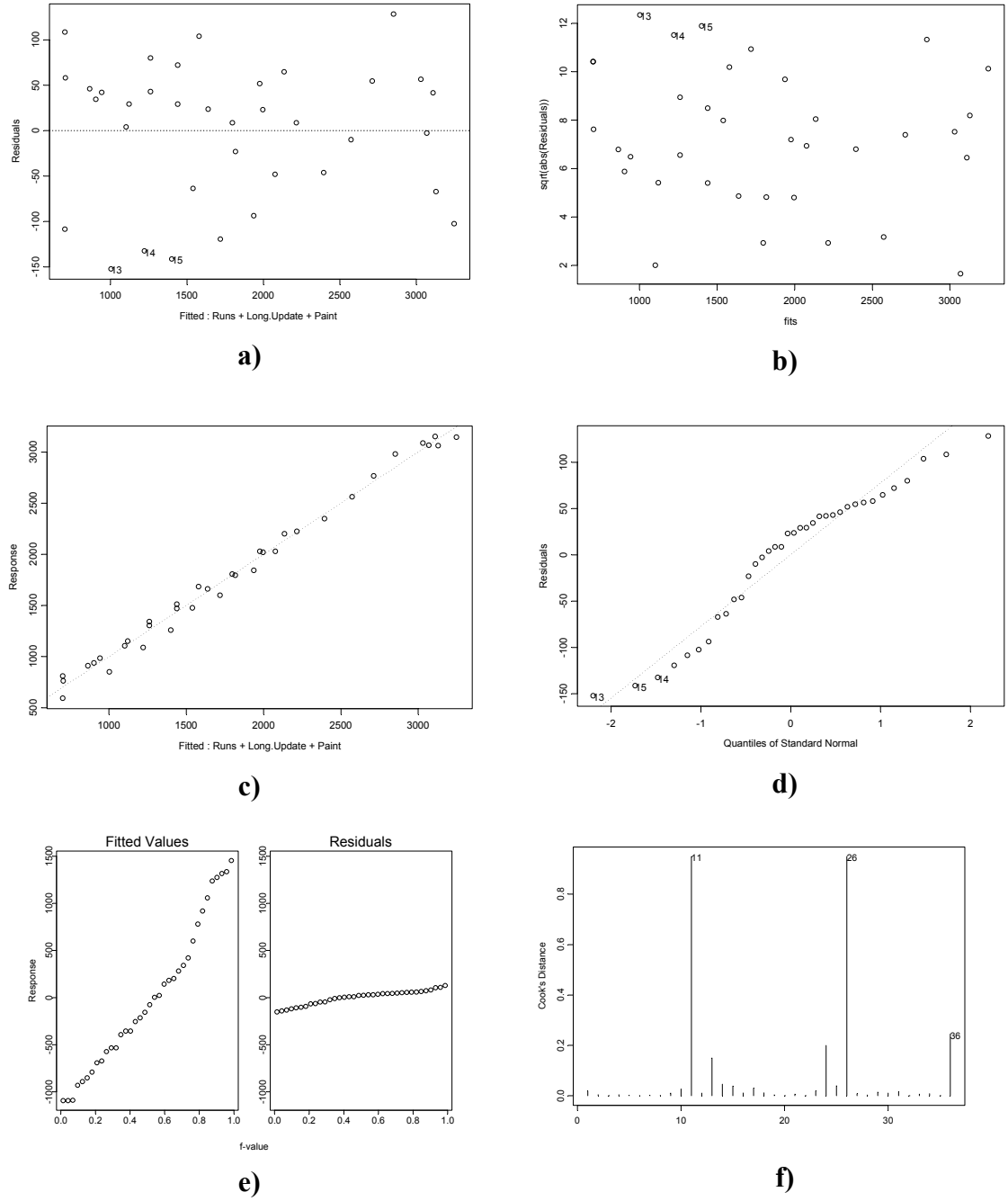


Figure F.2: Diagnostic plots: a) – Residual fit plot (constancy of standard deviation), b) Oultier diagnostic plot, c) Actual vs. fitted value, d) residual normal quantile-quantile (normality of residual distribution), e) residual-fit spread (explanatory power), and f) Cook's distance (diagnostic of influential observations).

Similarly to the plots in Figure F.1, the plots in Figure F.2 acknowledge

- Normal distribution of the residual (plots a and d),
- Constancy of residual standard deviation (plot a),
- Linear dependency between response time and signature parameters (plot c),
- High explanatory power of the prediction model (plot e), and

- The absence of the influential observations (plot f).

F.3 Prediction model validation

For the judgment on the quality of the predictions made for 10 excluded points, we use the following metrics (see [Jai91] and [WEI95]).

1. R-squared coefficient for the predictions:

$$R^2_{prediction} = 1 - \frac{\sum_{i=k}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=k}^{n}(y_i - \bar{y})^2} \tag{F.1}$$

2. Average squared prediction error:

$$ASPE = \frac{\sum_{i=k}^{n}(y_i - \hat{y}_i)^2}{n-k} \tag{F.2}$$

This error is compared to the mean squared error of the fitted prediction model:

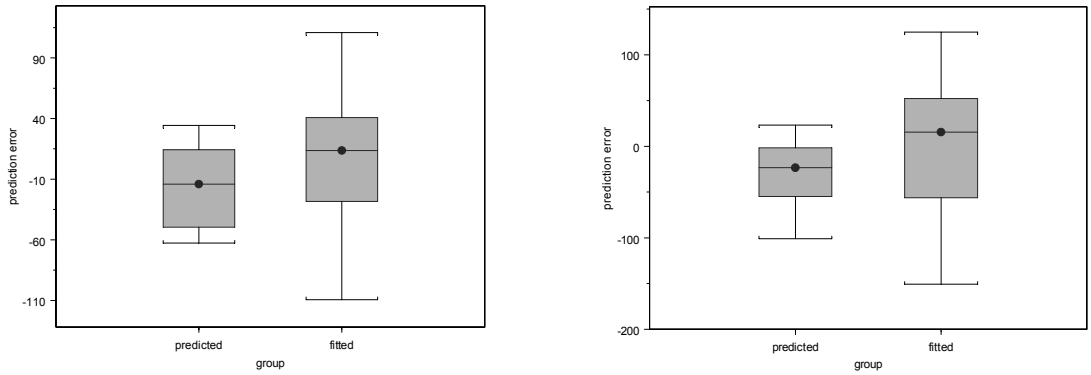
$$MSE = \frac{\sum_{i=1}^{S}(y_i - \hat{y}_i)^2}{S} \tag{F.3}$$

3. Average relative error:

$$E = \frac{\sum_{i=k}^{n}(y_i - \hat{y}_i)}{\sum_{i=k}^{n} y_i} \tag{F.4}$$

In formulas (F.1)-(F.4), k and n show the range of the excluded points, y_i are the measurements, $\bar{y}$ is the mean value of measurements, and $\hat{y}_i$ are the predicted values. In formula (F.3), $S = N - (n-k)$ is the number of points used for fitting the prediction model.

For the indication of the prediction quality, we use the box-whisker plots that demonstrate differences between residuals of the fitted model and prediction errors for the excluded points. The width of the box is equal to the *interquartile range*, or IQR, which is the difference between the third and first quartiles of the data. The IQR indicates the spread of the distribution for the data. Whiskers extend from the edges of the box to either the extreme values of the data, or to a 1.5 IQR distance of from the median, whichever is less. The plot in Figure F.3 shows that error distribution for the fitted model is wider than for the predictions for the excluded points. This confirms good prediction quality of the model.



case 1 **case 2**

Figure F.3: Distribution of prediction errors for the fitted model and for predictions for two cases.

F.4 Example of calibration data for the prediction model

Table F.5 contains an example of the signature instances used for calibration of the prediction model. The table is filled as follows. First, the name of the use case is given (with the state of the system). Then, the values of the signature instances are listed. Finally, the values of the response times are shown.

Table F.5: Calibration data.

StepRunReverse(Run)			
Images	Update	Paint	Response
2	8	6	388
3	8	6	386
4	8	6	449
5	7	5	475
6	8	7	549
7	7	8	685
8	8	6	685
9	8	6	781
10	7	2	650
12	8	2	735
StepRunForward(Run)			
Images	Update	Paint	Response
2	8	4	382
3	8	2	413
4	8	4	440
5	8	2	429
6	8	2	479
7	8	4	620
8	7	2	565
9	8	2	681
10	7	2	654

Appendix G. Construction of reduced CFGs of component operations

For modeling activities executing in isolation, it is necessary to be able to construct *reduced CFGs* (see Section 9.6.1) of all component operations that are invoked by the activities. The reduced CFGs can be constructed from the CFGs (see Appendix I) of these component operations. (The CFGs are constructed automatically by a compiler from the source code of the component). The following algorithm can be used for constructing the reduced CFGs (see Figure G.1).

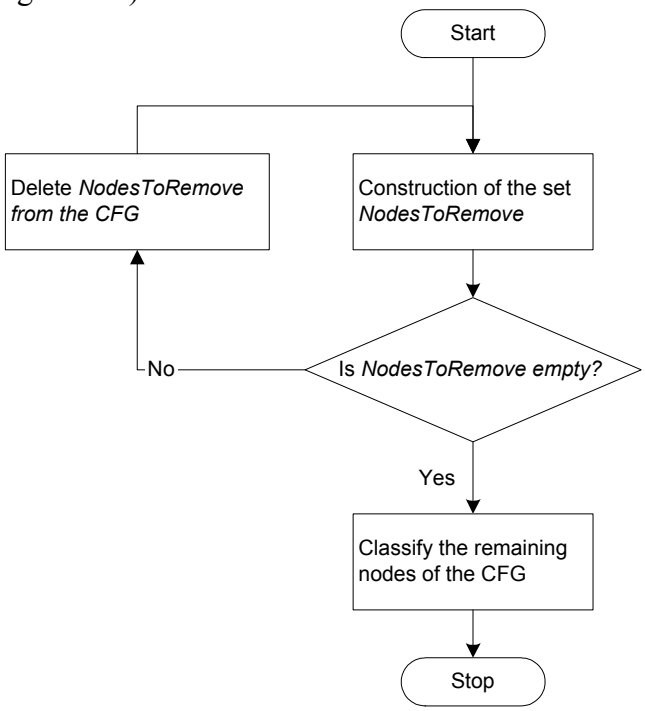


Figure G.1: Flowchart of the algorithm to construct a reduced CFG from the CFG of component operations

The algorithm is based on the sequential removal of the CFG nodes that are not needed for modeling the control flow of an activity. An example of such nodes are the basic blocks that perform some intermediate calculations, have only one ingoing and outgoing edge, and do not call any other operations.

Let us detail each block of the algorithm:

1. *Construction of the set of NodesToRemove*. This block assigns the following nodes of the CFG to the set *NodesToRemove*:
   - nodes that are not invocation nodes and have only one ingoing and outgoing edge,
   - nodes that are not invocation nodes and have a self back edge.
2. *Deletion of the nodes NodesToRemove from the CFG*. The deletion of the node from the CFG is performed in such a way that all successors of the deleted node are still reachable from all its predecessors. An example of the deletion of CFG nodes is shown in Figure G.2.

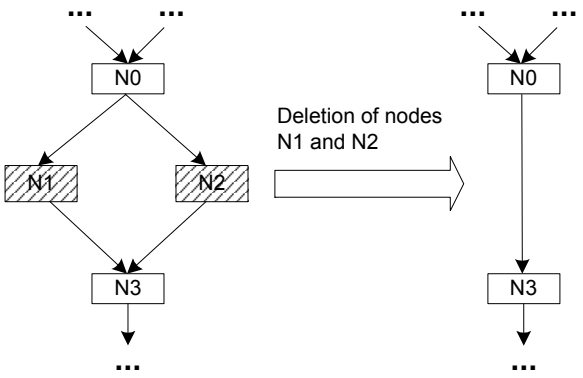


Figure G.2: An example of deletion of two nodes

3. Classification of the remaining nodes of the CFG. The classification if performed as follows:
   - *Entry* and *exit blocks* (see Appendix I) remain intact,
   - The CFG nodes that describe the invocation of component operations become *invocation nodes* (basic blocks),
   - The remaining nodes become *branching nodes* or *loop headers.*

Appendix H. Overview of the existing schedulers

Scheduling policies are usually classified as *online* or *offline*. Online schedulers decide what activity instance executes at run-time, whereas offline schedulers makes this decision at design time. Offline schedulers release activity instances at pre-calculated instants. This type of schedulers is often used in time-triggered architectures [EBK03]. A typical example of such scheduler is a *cyclic scheduler*, where each activity is allocated to a certain time frame.

Online schedulers usually determine, based on the event arrival patterns and the priorities of activity instances, which activity instance is the most eligible to be assigned the processor. This type of schedulers is used in event-triggered architectures. Two types of schedulers are usually distinguished: (1) fixed-priority-based and (2) dynamic-priority-based. For a fixed-priority scheduling policy, each activity is attached a *certain priority* (e.g., a natural number). All instances of the same activity share the same priority. The processor is assigned to the activity instance that has the highest priority. A dynamic-priority scheduling policy assigns the priority of activity instance according to certain criteria during run-time. These policies can be further classified in the two groups: *activity-instance-level fixed* and *activity-instance-level dynamic* policies. For the former, once assigned, the priority of an activity instance does not change with respect to the priorities of other activity instances. For the latter, the priority of an activity instance may change over time.

Finally, *preemptive* and *non-preemptive* schedulers are distinguished. Preemptive ones may allocate the processor to another activity instance, while the current activity is not yet completed; non-preemptive schedulers may not.

Figure H.1 shows a taxonomy of schedulers [Liu00].

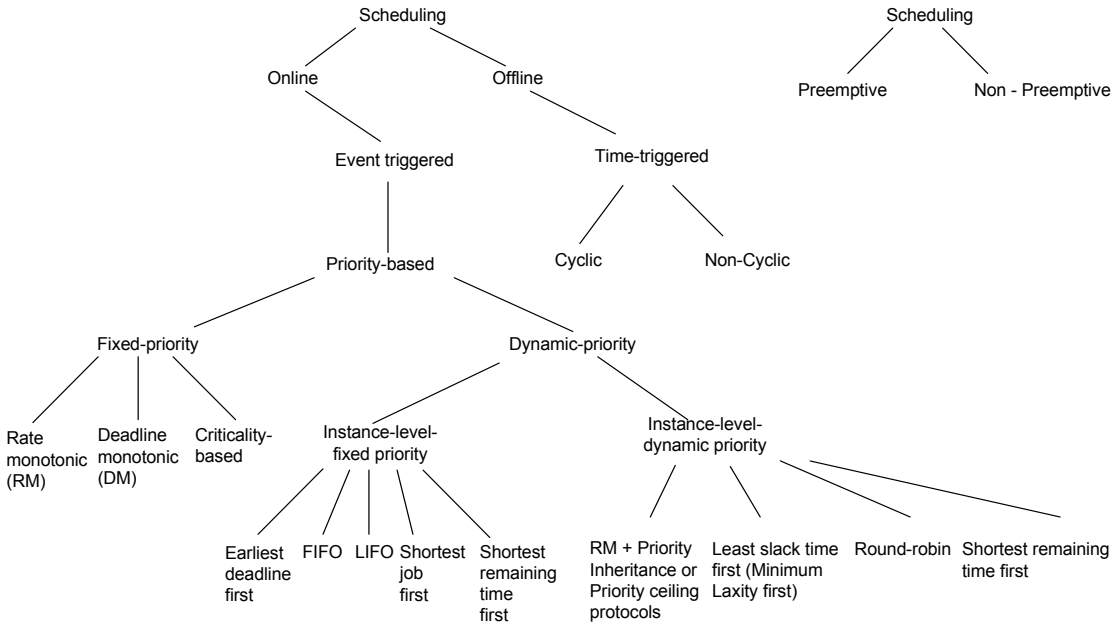


Figure H.1: Example of scheduling policy taxonomy

This taxonomy is not exhaustive; it rather indicates different aspects that are important in scheduling analysis. A detailed taxonomy of existing scheduling policies is presented in [Liu00].

When being executed, activities may need certain resources like buffers, locks, etc. We distinguish the following resource types:

1. Sharable resources that can be used by any activity instance at any time. A typical example of sharable resource is virtual memory;
2. Resources that may be used only once by a single activity instance: these resources are never replenished. An typical example is a message, which is destroyed once consumed.
3. Serially reusable non-sharable resources: only a limited number of units of these resources can be used by a single activity instance. After an activity instance does not need the resource anymore, the resource is replenished and can be used by other activity instances. Examples are shared-memory, a lock, a printer, etc.

Only resources of the third type need to be treated explicitly, as activity instances can contend for their access. When an activity instance cannot acquire a sufficient amount of such a resource, it gets blocked and cannot resume, until the resource is available.

In priority-based scheduling systems, the effect of blocking can cause scheduling anomalies like *priority inversion*. Priority inversion makes it difficult to reason about the timing properties of a system. To cope with this problem a number of resource-access protocols were suggested in the real-time literature [Liu00]. Among them are the following:

- Priority Inheritance Protocol,
- Highest Locker Protocol,
- Priority Ceiling Protocol,
- Non-preemptive critical sections.

Additionally, when multiple non-sharable resources like locks are used, *deadlock* can occur, that is, the involved activity instances cannot progress anymore. Some of the resource-access control protocols enumerated above not only prevent priority inversion, but also help in avoiding deadlocks.

Appendix I. The notion of Control Flow Graph

CFGs are often used to represent the control flow of a function or a program by compilers for performing different kinds of optimizations. Another area of the CFG application is worst-case execution time (WCET) analysis. The literature [PK89], [EE00], [EES01], [EES01a] describes a number of techniques for estimating the WCET of a program. These techniques use annotated CFGs. CFGs are similar to specification techniques such as execution graphs (see [Smi90], [Smi02]), UML activity diagrams, traditional flowcharts, SDL.

A CFG is a directed graph, which consists of nodes and edges. Traditionally, a node of a CFG is a basic block describing a sequence of instructions that does not contain branching instructions, except for the last instruction of this sequence. Edges describe jumps in the control flow. The source of the edge is a basic block that has a branching instruction at the end. The sink of the edge is the target of this branching instruction. Conditional branching instructions produce more than one outgoing edge: a different edge is generated for each branch. Figure I.1 shows an example of CFG for a hypothetical function "*Func*".

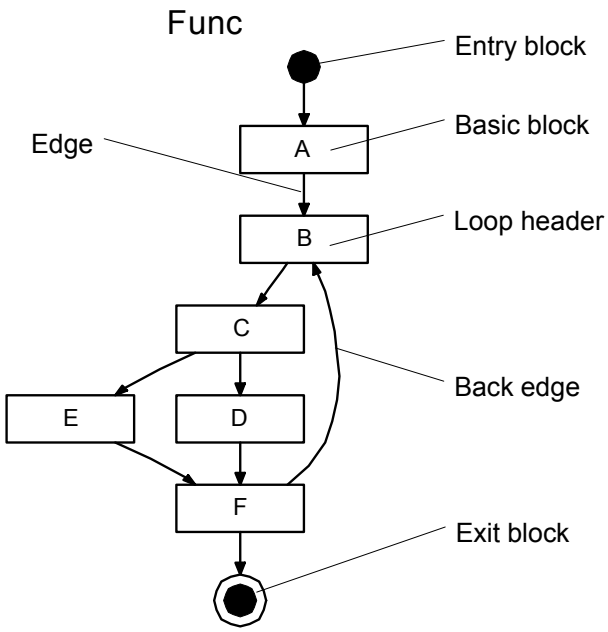


Figure I.1: An example of CFG for the "*Func*" function

The control flow enters the CFG through the entry node, which is denoted by a black circle. It leaves the CFG through the exit node depicted by a black circle with a border. The basic blocks (the CFG nodes) are denoted by rectangles with labels inside. The arrows denote the edges of the CFG. The "*Func*" function contains a loop. Each loop has a loop header, the basic block that executes at the beginning of each iteration. In addition, a dedicated edge (the back edge) indicates that the control flow may pass through the loop header many times.

Appendix J. Activity control flow graph

We explain how to build such a graph by means of an example of a hypothetical component composition (see Figure J.1).

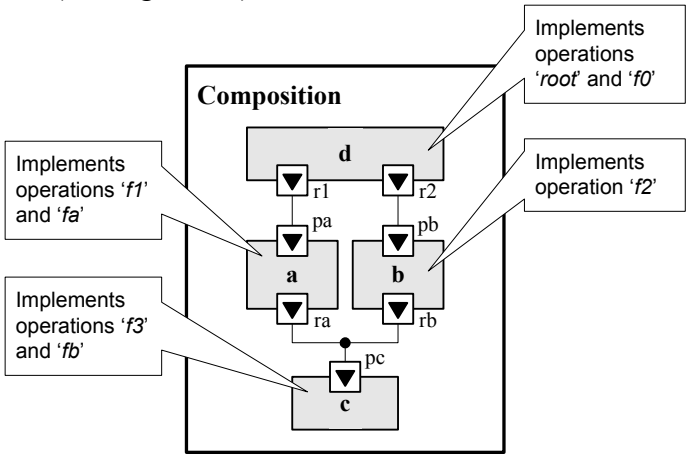


Figure J.1: A simple composition

Figure J.1 uses the same notations as in Figure 9.9. The "*d*" component is a component, which starts an activity with the internal "*root*" root operation. This activity calls the "*f1*" component operation, implemented by the "*a*" component, and the "*f2*" component operation, implemented by the "*b*" component. These two component operations invoke the "*f3*" operation of the "*c*" component. Some of the operations can also invoke particular internal component operations. The control flow of the activity is represented by the activity CFG shown in Figure J.2.

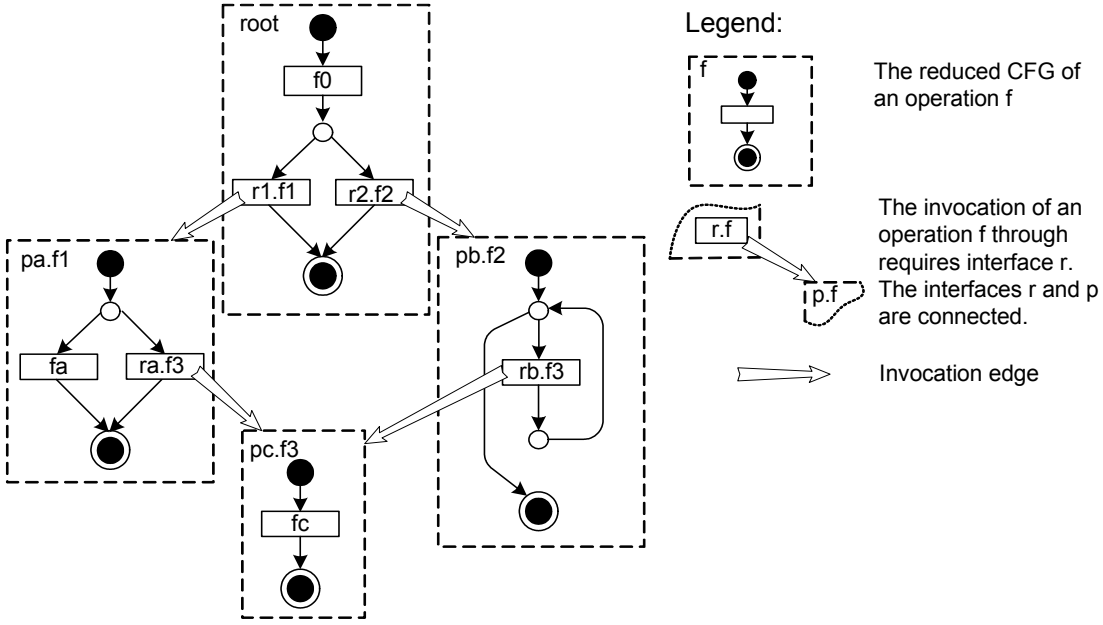


Figure J.2: An example of the activity CFG

The notations used in this figure are explained in the legend part (to the right). In addition, Figure J.2 also employs the notations for describing reduced CFGs of component operations (see Section 9.6.1 and 9.6.3). The activity CFG includes the reduced CFGs of all component operations that can be invoked by this activity. These reduced CFGs are *hyper-nodes* of the activity CFG. The hyper-nodes are labeled with the identifier of the corresponding operation. The prefix of this identifier is the name of the interface through

which this function is provided to other components. The suffix of the identifier is the name of the component operation.

The invocation of a component operation is described by an *invocation edge*. The edges of this type are denoted by a thick arrow. An invocation edge connects an invocation node of a particular hyper-node to the hyper-node that corresponds to the called operation. For example, an invocation edge connects the invocation node "*rb.f3*" and the hyper-node "*pc.f3*". This edge describes the invocation of the "*f3*" component operation by the "*f2*" component operation. In addition, the presence of this edge implies that the *requires* interface "*rb*" is connected to the *provides* interface "*pc*".

Appendix K. Validation of a simple formula

Table K.1 presents the comparison of the predictions and measurements of CPU utilization for various broadcast channels. The predictions are obtained by applying an analytical formula. The first column enumerates the channels. The second one shows the observation time in milliseconds. The third column enumerates the predicted utilization for a certain channel. The values in this column are calculated by Formula (10.7) from Section 10.3.1 and by using Table K.3. Finally, the fourth column gives the measured CPU utilization for each channel. The measurement of CPU utilization amounts to the measurement of CPU demand and the application of Formula (10.1). The average CPU utilization is calculated as explained in Appendix L on the basis of Table K.3 and the nominator of Formula (10.7).

Table K.1: The comparison of predicted and measured average CPU utilization for various broadcasts

Broadcast channel	Observed time (ms)	Predicted utilization	Measured utilization
Nederland 1	10680	0.17375817	0.160547
Nederland 2	11260	0.1742971	0.167087
Nederland 3	10560	0.17361661	0.169501
RTL 4	12340	0.17464702	0.158311
RTL 5	12520	0.17369108	0.163794
SBS 6	12380	0.17429737	0.162234
Yorin	12080	0.17350043	0.167421
V 8	15240	0.17392664	0.169413
Net 5/Kindernet5	12540	0.17363354	0.161285
UPC Infokanaal	13800	0.17446266	0.167335
Omroep Brabant TV	13220	0.17478783	0.163087
Lokale Omroep	11000	0.17407915	0.166411
Nieuws TV (kabelkrant)	12680	0.17390122	0.162684
VRT	10980	0.17305521	0.172133
KetNet / Canvas	11940	0.17398134	0.163586
National Geographic Channel	16760	0.17422175	0.175757
Discovery Channel	12440	0.17413831	0.171821
The Music Factory (TMF)	13600	0.1738733	0.161272
MTV Europe	18220	0.17471672	0.163665
ARD 1	10440	0.17413612	0.173966
ZDF 2	11380	0.17426229	0.170606
WDR 3	10680	0.17375817	0.165159
RTL Television	12140	0.17409548	0.168759
BBC 1	17120	0.17433938	0.162348
BBC 2	17360	0.17467769	0.161079
CNN International	13160	0.1739914	0.162944
TV 5 Europe	11260	0.1742971	0.165745
TRT International	12940	0.17406641	0.166525
Eurosport	13820	0.17470065	0.158719
Local VRT	15220	0.17415519	0.172229
Average	12992	0.17413355	0.165774

Table K.1 shows that the average CPU utilization is higher for predictions than for measured values. It is however necessary to check if the difference between these average CPU utilizations is not obtained by chance. To check this, Appendix M explains a procedure for comparing average CPU utilizations. We show that this comparison can be accomplished by applying the *t*-test [WEI95] for paired samples to the average CPU demands corresponding to the CPU utilization calculated from the measurements and CPU utilization predictions.

Table K.2 enumerates the measured and predicted CPU demands for each broadcast channel. Additionally, the difference between these CPU demands is given in the rightmost column.

Table K.2: Measured and predicted processor demands

Broadcast channel	Measured CPU demand (ms)	Predicted CPU demand (ms)	Difference (ms)
Nederland 1	1855.73728	1714.637	141.1007
Nederland 2	1962.58529	1881.394	81.1913
Nederland 3	1833.39138	1789.931	43.46082
RTL 4	2155.14425	1953.558	201.5865
RTL 5	2174.61226	2050.695	123.9176
SBS 6	2157.8015	2008.457	149.3446
Yorin	2095.88524	2022.446	73.43956
V 8	2650.64196	2581.854	68.78784
Net 5/Kindernet5	2177.36453	2022.508	154.8569
UPC Infokanaal	2407.58464	2309.223	98.36164
Omroep Brabant TV	2310.69508	2156.01	154.6849
Lokale Omroep	1914.87067	1830.521	84.34967
Nieuws TV (kabelkrant)	2205.06741	2062.827	142.2406
VRT	1900.14617	1890.015	10.13132
KetNet / Canvas	2077.33716	1953.217	124.1203
National Geographic Channel	2919.95646	2945.687	-25.7309
Discovery Channel	2166.28056	2137.453	28.82732
The Music Factory (TMF)	2364.67688	2193.292	171.3845
MTV Europe	3183.33858	2981.984	201.3541
ARD 1	1817.98111	1816.205	1.77607
ZDF 2	1983.10485	1941.491	41.61426
WDR 3	1855.73728	1763.898	91.83916
RTL Television	2113.5191	2048.734	64.78484
BBC 1	2984.69015	2779.389	205.301
BBC 2	3032.40477	2796.323	236.082
CNN International	2289.72681	2144.336	145.3904
TV 5 Europe	1962.58529	1866.289	96.29659
TRT International	2252.41935	2154.827	97.59232
Eurosport	2414.36298	2193.49	220.8733
Local VRT	2650.64196	2621.318	29.32419

The application of the paired *t*-test requires the differences between the CPU demands be normally distributed. Figure K.1 shows the normal quantile-quantile plot for these differences.

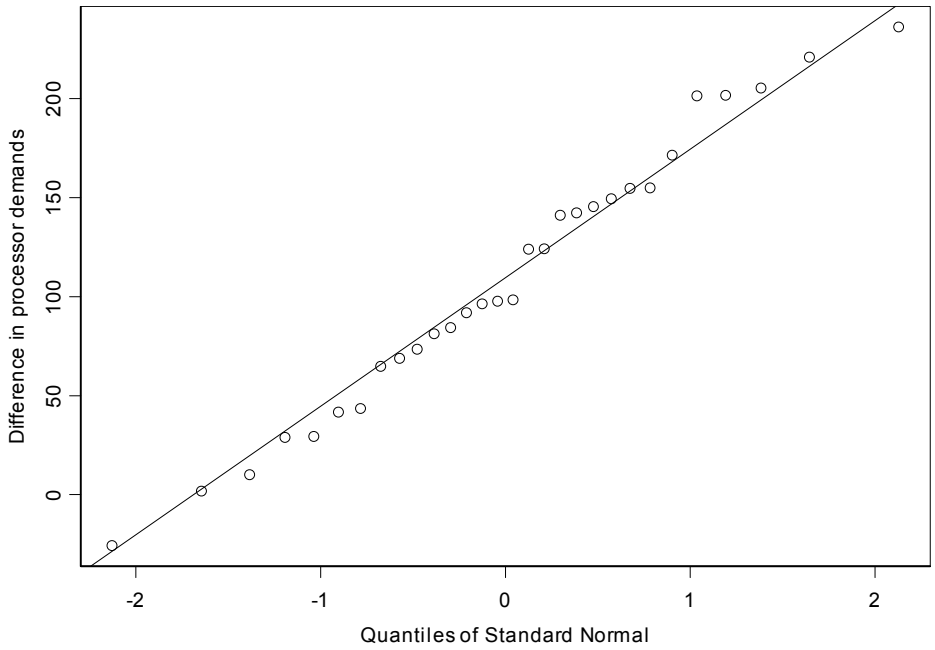


Figure K.1: The normal quantile-quntile plot for the differences in the CPU demands

As all points in this plot lie close to the "y=x" line, we conclude that these differences have a normal distribution (see also Appendix R). Therefore, the use of the *t*-test is justified.

We formulated the following null hypothesis for the one-sided *t*-test: there is no difference between the predicted and measured CPU demands. The alternative hypothesis was as follows: the predicted CPU demand is greater than the measured one. The *t*-test provided the following results: *t*-statistic = 8.6673 on 29 degrees of freedom, which gives *p*-value = 0. The p-value is less than the selected significance level of 0.05. Therefore we have to reject the null hypothesis and to accept the alternative one. The average difference between the predicted and measured CPU demand equals 108.61, and it is significant at the selected significance level. By dividing this average difference by the average observation period (see Table K.1), we can conclude that the simple analytical formula overestimates the actual CPU utilization by 0.0084 on the average, which means the average relative prediction error of 5.04%.

Table K.3: The average execution times and periods of the activities from the activity validation set

Activity	tvsndsys.pmp	tvhipdrv.mon	mgdpow.refrstep	tvfbxdrv.mon	mgdpow.cperiodic
Period T_i (ms)	100	300	217.43	300	100
Average execution time $\bar{e}_i$ (ms)	3.18011	2.54668	6.82064	6.77834	2.75227
Activity	mgatv.ctr	tvhopdrv.hop	tvsndqp.wait	mgtng.afc	
Period T_i (ms)	300	200	500	500	
Average execution time $\bar{e}_i$ (ms)	5.15159	2.65725	5.51427	8.35257	

Appendix L. Calculation of the average utilization

Let us consider that we performed a series of measurements, each measurement being the total CPU utilization of a number of the activities under consideration. Our goal is to define the average CPU utilization of these activities in the light of the measurements performed. In [Jai91], Jain shows that using the arithmetic mean of the CPU utilizations obtained in each measurement is inappropriate. The reason is that the arithmetic mean treats equally short and long observation intervals for calculating the CPU utilization. However, measurements obtained during long observation intervals contain more information about the average CPU utilization, which is supposed to provide a typical value for the utilization. Instead of using the arithmetic mean for modeling the average utilization, Jain shows the rule of finding the average of fractions must be used [Jai91]. In essence, this rule says that the average of fractions of which both the denominator and nominator has some physical sense must be calculated by dividing the arithmetical mean of the nominator by the arithmetical mean of the denominator.

For a series of measurements of the CPU utilizations the rule above instantiates as follows. Let $U^{(j)} = E^{(j)} \big/ T^{(j)}$ be the CPU utilization calculated from the *j*-th measurement from a series of *n* measurements in total. (Each measurement corresponds with a particular broadcast channel.) $E^{(j)}$ and $T^{(j)}$ are the processor demand (execution time) of the activities and the observation interval, respectively, that take place during the *j*-th measurement. The average CPU utilization of these activities obtained over all n measurements can be found by the following formula:

$$\bar{U} = \frac{\frac{1}{n}\sum_{j=1}^{n} E^{(j)}}{\frac{1}{n}\sum_{j=1}^{n} T^{(j)}} = \frac{\bar{E}}{\bar{T}}. \qquad \text{(L.1)}$$

In Formula (L.1), $\bar{E}$ and $\bar{T}$ denote the average processor demand of the activities and the average observation interval, respectively. The averaging is performed over the *n* measurements.

Appendix M. Comparison of average CPU utilizations

This appendix details the comparison procedure for average utilizations. We use this procedure to assess the quality of predictions made by simulation and by a simple analytical formula.

Let us consider two samples of values of the CPU utilization of particular activities. Each sample contains a single value of the CPU utilizations of the same activities for a particular broadcast channel. The number of broadcast channels equals *n*. For instance, one such sample may correspond to the measured CPU utilization, whereas the other may describe the CPU utilizations predicted by a simple analytical formula.

We compare the CPU utilizations from these two samples by looking at the difference in the average CPU utilization. For two samples, Formula (L.1) can be rewritten as follows:

$$\bar{U}^{(i,\cdot)} = \frac{\frac{1}{n}\sum_{j=1}^{n} E^{(i,j)}}{\frac{1}{n}\sum_{j=1}^{n} T^{(i,j)}} = \frac{\bar{E}^{(i,\cdot)}}{\bar{T}^{(i,\cdot)}}, \quad i = 1,2. \tag{M.1}$$

In Formula (M.1), the $\bar{U}^{(i,\cdot)}$ is the average CPU utilization for *i*-th sample. $\bar{E}^{(i,\cdot)}$ and $\bar{T}^{(i,\cdot)}$ denote the average processor demand (execution time) of the activities and average observation interval for for *i*-th sample. Finally, $E^{(i,j)}$ and $T^{(i,j)}$ are the processor demand (execution time) of the activities and the observation interval for *j*-th broadcast channel in the *i*-th sample.

Note that all experiments are performed such that the following formula holds for the observation intervals:

$$T^{(1,j)} = T^{(2,j)}, \quad \forall j = 1,...,n. \tag{M.2}$$

Therefore, $\bar{T}^{(1,\cdot)} = \bar{T}^{(2,\cdot)} = \bar{T}$.

The difference $\bar{D}$ in the CPU utilizations can be calculated by the following formula:

$$\bar{D} = \bar{U}^{(2,\cdot)} - \bar{U}^{(1,\cdot)} = \frac{\frac{1}{n}\sum_{j=1}^{n}\left(E^{(2,j)} - E^{(1,j)}\right)}{\bar{T}} = \frac{\bar{E}^{(2,\cdot)} - \bar{E}^{(1,\cdot)}}{\bar{T}}. \tag{M.3}$$

Observation intervals $T^{(i,j)}$ are not random variables, as we control their values. On the other hand, all processor demands (execution times) are in fact random variables. Therefore, $\bar{D}$ is also a random variable.

Consequently, statistical hypothesis testing must be applied to decide if the average CPU utilizations for the two samples are statistically different. We can decide on this difference, if $\bar{E}^{(2,\cdot)} - \bar{E}^{(1,\cdot)}$ significantly differs from zero according to the paired *t*- or *z*-test [WEI95]. We select the *t*-test only for the cases when $n<30$. Otherwise, the *z*-test may be selected.

Appendix N. Prediction models for the Consumer Electronics Case

This appendix describes the prediction models constructed for the activities from Section 10.1.3. The prediction models were constructed on the basis of measurements from the TV software in steady state. Please notice that we did not use full-fledged linear regression to fit these prediction models.

First, particular prediction models (e.g., models for the "*tvsndsys.pmp"* activity, "*tvsndqp.wait*" activity, and etc) were simple and did not require the application of linear regression. Second, for some of the remaining models, it was impossible to use linear regression, as the signature parameters were strongly correlated (e.g., the *"mgdpow.cperiodic"* activity). Third, we aimed at the prediction models that would be suitable both for predicting the CPU demand of a single activity instance and for estimating the CPU demand of a number of activities in a particular time interval. Such prediction models had to have interpretation in terms of average CPU demands. The construction of this kind of prediction models is often impossible if one uses linear regression. The reason that linear regression models often provide such coefficients that it is impossible to give any physical sense to them.

However, we used linear regression to explore the relation between the candidate signature parameters and the CPU demands of particular activities. We then used the obtained information to construct prediction models that can be related to the concrete designs and implementations of the activities. For example, the "*mgdpow.cperiodic*" activity invokes five *protection operations* (see Section 10.4.5) related to various components drawn in Figure 10.5. For example, Section N.3 shows that the prediction model for a sequence of activity instances concerns five average CPU demands of these five *protection operations.* However, due to the algorithm for constructing a schedule for the protection functionality, the signature parameters related to these protection operations turned out to have a restricted variation scope. Using linear regression, we were able to identify only a single signature parameter. The rest parameters were identified by observing the execution traces and analyzing the design specifications and code.

The only relevant signature parameter for many activities turned out to be the number of times it executes. The main reason is the excessively large observation period for calculating the average CPU utilization in comparison to the arrival periods of these activities. On the long run, each activity instance (or a short repetitive sequence of instances) exhibits the same behavior and execution time. Moreover, the same signature instance describes each activity instance. As a result, the signature instances of all activity instances become linear with the number of times the activity executes.

N.1 The "tvsndsys.pmp" activity

The CPU demand of the "*tvsndsys.pmp*" activity can be predicted by the following formula:

$$\begin{aligned} E_1 &= \overline{e}_1 \cdot N \\ \overline{e}_1 &= 3.18011 \end{aligned} \qquad \text{(N.1)}$$

In Formula (N.1), E_1 denotes the CPU demand consumed by N activity instances. The average CPU demand of a single activity instance equals $\overline{e}_1$.

N.2 The "tvsndqp.wait" activity

The CPU demand of the "*tvsndqp.wait*" activity can be predicted by the following formula:

$$E_2 = \overline{e}_2 \cdot N$$
$$\overline{e}_2 = 5.51427 \quad \text{(N.2)}$$

In Formula (N.2), E_2 denotes the CPU demand consumed by N activity instances. The average CPU demand of a single activity instance equals $\overline{e}_2$.

N.3 The "mgdpow.cperiodic" activity

The CPU demand of the "*mgdpow.cperiodic*" activity can be predicted by the following formula:

$$E_3 = \overline{e}_{31} \cdot N_1 + \overline{e}_{32} \cdot N_2 + \overline{e}_{33} \cdot N_3 + \overline{e}_{34} \cdot N_4 + \overline{e}_{35} \cdot N_5,$$
$$\overline{e}_{31} = \overline{e}_{32} = \overline{e}_{33} = 1.20367,\ \overline{e}_{34} = \overline{e}_{35} = 0.947. \quad \text{(N.3)}$$

In Formula (N.3), E_3 denotes the CPU demand consumed by a number of activity instances over a certain observation period. $\overline{e}_{3i}$ is the average CPU demand of the *i*-th protection operation. Table N.1 presents mapping of the CPU demand variables onto concrete protection operations. N_i is the number of times that *i*-th protection operation executes.

Note that $N_1 = N_2 = N_3$ and $N_4 = N_5$ due to the algorithm used for arranging the protection operations over various protection phases (see Section 10.4.5). Additionally, the measurements did not allow distinguishing the CPU demands of the protection operations that executed during the same protection phase. We therefore decided to spread the CPU demand equally for such protection operations.

Table N.1: The mapping of the CPU demand variables onto the protection operations

Protection name	Variable
HOP	$\overline{e}_{31}$
Tuner	$\overline{e}_{32}$
Protection 5V	$\overline{e}_{33}$
FBX	$\overline{e}_{34}$
Protection 3V	$\overline{e}_{35}$

N.4 The "mgdpow.refrstep" activity

The CPU demand of the "*mgdpow.refreshstep*" activity (see Sections 10.1.3 and 10.4.5) can be predicted by the following formula:

$$E_4 = \sum_{i=1}^{4} N_j^{MSP} \cdot \overline{e}_{4j} + N_1^{FBX} \cdot \overline{e}_{45} + N_2^{FBX} \cdot \overline{e}_{46} + N_1^{HIP} \cdot \overline{e}_{47} + N_1^{HOP} \cdot \overline{e}_{48} + N_1^{TUN} \cdot \overline{e}_{49}. \quad \text{(N.4)}$$

In Formula (N.4), E_4 denotes the CPU demand consumed by a number of activity instances over a certain observation period. $\overline{e}_{4i}$ is the average CPU demand of the *i*-th

refreshing operation. For a device D, N_j^D denotes the j-th signature parameter, being the number of times that the refreshing operation of the D device executes at the j-th step.

The CPU demands of the refreshing operation of various chips are given in Table N.2.

Table N.2: The CPU demands of the refreshing operations of various hardware chips

Sound chip	$\overline{e}_{41}$	2.359
	$\overline{e}_{42}$	24.1563
	$\overline{e}_{43}$	18.7397
	$\overline{e}_{44}$	22.3456
FBX chip	$\overline{e}_{45}$	8.7822
	$\overline{e}_{46}$	1.889
HIP chip	$\overline{e}_{47}$	5.2537
HOP chip	$\overline{e}_{48}$	2.9483
Tuner	$\overline{e}_{49}$	1.8064

The leftmost column enumerates the hardware chips. The middle column lists the variables that denote the average CPU demand of a refreshing operation at a particular step. The rightmost column enumerates the values of these variables obtained from measurements.

N.5 The "tvfbxdrv.mon" activity

The CPU demand of the "*tvfbxdrv.mon*" activity can be predicted by the following formula:

$$\begin{aligned} &\mathrm{E}_5 = \overline{\mathrm{e}}_5 \cdot N \\ &\overline{\mathrm{e}}_5 = 6.77834 \end{aligned} \qquad \text{(N.5)}$$

In Formula (N.5), E_5 denotes the CPU demand consumed by N activity instances. The average CPU demand of a single activity instance equals $\overline{e}_5$.

N.6 The "mgtng.afc" activity

The CPU demand of the "*mgtng.afc*" activity can be predicted by the following formula:

$$\begin{aligned} &\mathrm{E}_6 = \overline{e}_{61} \cdot N_1 + \overline{e}_{62} \cdot N_2, \\ &\overline{e}_{61} = 17.8964, \quad \overline{e}_{62} = 4.47096 \\ &N_1 = \min(k, N), \quad N_2 = \max(0, N-k). \end{aligned} \qquad \text{(N.6)}$$

In Formula (N.6), E_6 denotes the CPU demand consumed by N activity instances. The average CPU demand of a single activity instance equals $\overline{e}_{61}$ for the first k activity instances, whereas it equal $\overline{e}_{62}$ for the rest of the instances. For the majority of the observed channels, k was equal 6. These first k activity instances adjust the tuner to a particular frequency.

N.7 The "mgatv.ctr" activity

The CPU demand of the "*mgatv.ctr*" activity can be predicted by the following formula:

$$\begin{aligned} &E_7 = \overline{e}_{71} \cdot \mathrm{N} + \overline{e}_{81} \cdot \Omega_N, \\ &\Omega_N = \Omega_{N-1} + X, \quad \Omega_N = 0, \\ &\overline{e}_7 = 4.69345, \quad \overline{e}_8 = 0.1490665. \end{aligned} \tag{N.7}$$

In Formula (N.6), E_7 denotes the CPU demand consumed by N activity instances. Each activity instance consumes at least $\overline{e}_{71}$ units of CPU time. In addition, the activity instance may communicate with the SCAVEM device for a random number of times X. Measurements showed that the X variable has a distribution from the exponential family. We keep track of all these random numbers of the communications in the Ω_N variable.

N.8 The "tvhopdrv.hop" activity

The CPU demand of the "*tvhopdrv.hop*" activity can be predicted by the following formula:

$$\begin{aligned} &E_8 = \overline{e}_8 \cdot N, \\ &\overline{e}_8 = 2.65725. \end{aligned} \tag{N.8}$$

In Formula (N.8), E_8 denotes the CPU demand consumed by N activity instances. The average CPU demand of a single activity instance equals $\overline{e}_8$.

N.9 The "tvhipdrv.mon" activity

The CPU demand of the "*tvhipdrv.mon*" activity can be predicted by the following formula:

$$\begin{aligned} &E_9 = \overline{e}_9 \cdot N, \\ &\overline{e}_9 = 2.54668. \end{aligned} \tag{N.9}$$

In Formula (N.9), E_9 denotes the CPU demand consumed by N activity instances. The average CPU demand of a single activity instance equals $\overline{e}_9$.

Appendix O. Summary of the analysis of I^2C transactions

The TV hardware chassis under consideration uses the I^2C bus to control various hardware chips, which are responsible for hardware processing of the audio and video signal. The host CPU is always a "master", whereas the chips are always "slaves". This means that it is the CPU that defines how many bytes must be read and/or written in each transaction. The detailed explanation of the use of I^2C protocol can be found in [I2C03].

Table O.1 summarizes the measurements of the I^2C transactions collected from the TV software running in steady state.

Table O.1: The summary statistics of the demand of I^2C transactions

Chip	#Bytes Read	#Bytes Written	#Trans-actions	Min	Lower quartile	Median	Upper quartile	Max	Mean	Std. dev.
FBX	0	0	2161	0.5784	0.6735	0.676125	0.678425	1.13065	0.6767	0.055
FBX	0	2	3796	0.7751	0.785375	0.78815	0.793525	10.3042	0.8806	0.4537
FBX	1	2	2588	0.95605	0.96955	0.97345	1.04835	14.0562	1.063	0.3702
FBX	6	2	3843	1.33095	1.3896	8.878	13.0951	20.2121	8.1097	5.9247
HIP	0	16	64	1.66613	1.7183	1.7233	1.72645	2.17132	1.7654	0.1059
HIP	4	0	2855	0.939675	0.95375	0.956775	0.9593	1.92847	1.0029	0.1298
HOP	0	33	62	2.7835	2.79257	2.8037	3.02023	3.4729	2.9384	0.2069
HOP	3	0	4240	0.871925	0.88235	0.88635	0.890375	1.73933	0.9014	0.0733
MSP	0	5	3483	0.943925	0.9536	0.95675	0.9613	1.86392	1.0116	0.1367
MSP	2	3	4716	0.59055	1.0957	1.0999	1.1041	2.0167	1.1404	0.1226
SCAVEM	0	2	415	0.850975	0.904925	0.9064	0.9097	1.80025	0.9604	0.1441
SCAVEM	0	9	61	1.85425	1.85845	1.8615	1.87042	2.58078	1.9259	0.1397
Tuner	0	0	2155	0.597925	0.606575	0.60875	0.61095	1.25835	0.6186	0.0573
Tuner	0	2	218	0.857575	0.90475	0.906325	0.91015	1.63842	0.9626	0.1398
Tuner	0	4	64	1.1727	1.17608	1.17775	1.18005	1.59737	1.1953	0.0719

The first three columns of the table designate the type of an I^2C transaction and determine how many bytes are written to and then read from a particular chip. The remaining columns are as follows (from the left to the right):

1. the number of the transactions of a particular type observed in the measurements;
2. the minimum observed time for a transaction of a particular type;
3. the lower (first) quartile of the observed time for a transaction of a particular type;
4. the median of the observed time for a transaction of a particular type;
5. the upper (third) quartile of the observed time for a transaction of a particular type;
6. the mean observed time for a transaction of a particular type;
7. the standard deviation of the observed time for a transaction of a particular type.

The median values are greater than the mean values for each type of a transaction. This fact indicates that the distributions of the transaction times are skewed towards larger values for the transaction of all types. This skewness is attributed to the ability of a 'slave' device to stretch clocks, if the 'slave' device cannot cope with the dataflow and needs additional time to process or store the already written/read data. (More details about clock stretching can be found in [I2C03].) The hardware chip, being a 'slave' device, occasionally needs to stretch clocks, which takes more time then for a typical transaction.

Table O.1 shows that the time needed to communicate with the FBX chip varies significantly for transactions of particular types. This conclusion is supported by a large value of the standard deviation, the large difference between the upper and lower quartiles, and the large difference between the minimum and maximum times. This variation is due to the fact that the FBX chip can perform I^2C transactions only at certain intervals within a VBI (20 ms). Depending on the time when the CPU initiates an I^2C transaction, the delay before the FBX responds may lie within the interval from 0 to 20 ms. This delay explains the observed variation.

Table O.1 also lets us see that only a few transaction types are used to communicate with the hardware chips. Therefore, there is no much sense in constructing prediction models that relate the time needed for I^2C transactions to the number of bytes read or written. It is more convenient to consider the mean transaction times for individual transaction types. We chose the mean times instead of the median times, as the mean ones also accounts for possible clock stretching. This choice is valid for all chips, except for the FBX chip.

The FBX chip turned out to influence the timing behavior of the TV software significantly. This is why we decide to model the behavior of this chip in more detail. Particularly, the FBX chip allows reading data from it only once. The attempt to read data for the second time results in the stretching of clocks by the FBX chip until the next VBI begins. This behavior is illustrated by Figure O.1.

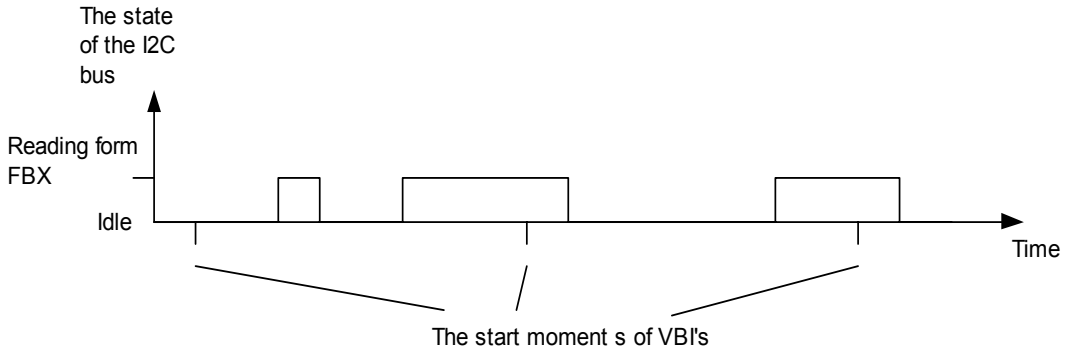


Figure O.1: The timing behavior of the FBX chip

Table O.2 shows which of the most CPU consuming activities (see Section 10.1.3) may perform transactions of a particular type.

Table O.2: The types of I^2C transactions performed by the most CPU consuming activities. The first three columns denote the type of an I^2C transaction. The remaining columns enumerate the activities. A symbol 'X' at a particular cell means that the activity from the column header performs a transaction of the type denoted by the first three columns of the row of the cell. A symbol '0' marks that the activity does not perform any transactions of the specified type.

Chip	R	W	mgtng.afc	tvsndqp.wait	tvhipdrv.mon	tvsndsys.pmp	mgdpow.refrstep
Fbx	0	0	0	0	0	0	0
Fbx	0	2	0	0	0	0	X
Fbx	1	2	0	0	0	0	0
Fbx	6	2	0	0	0	0	0
Hip	0	16	0	0	0	0	X
Hip	4	0	X	0	X	0	X
Hop	0	33	0	0	0	0	X
Hop	3	0	0	0	0	0	0
Msp	0	5	0	X	0	X	X
Msp	2	3	0	X	0	X	0
Scavem	0	2	0	0	0	0	0
Scavem	0	9	0	0	0	0	X
Tun	0	0	0	0	0	0	0
Tun	0	2	X	0	0	0	0
Tun	0	4	0	0	0	0	X

Chip	R	W	tvfbxdrv.mon	tvhopdrv.hop	mgatv.ctr	mgdpow.cperiodic
Fbx	0	0	0	0	0	X
Fbx	0	2	X	0	0	0
Fbx	1	2	X	0	X	0
Fbx	6	2	X	0	0	0
Hip	0	16	0	0	0	0
Hip	4	0	0	0	0	0
Hop	0	33	0	0	0	0
Hop	3	0	0	X	0	X
Msp	0	5	0	0	0	0
Msp	2	3	0	0	0	0
Scavem	0	2	0	0	X	0
Scavem	0	9	0	0	0	0
Tun	0	0	0	0	0	X
Tun	0	2	0	0	0	0
Tun	0	4	0	0	0	0

Appendix P. Details of the validation of the prediction of the CPU utilization by simulation

Table P.1 presents the comparison of the predictions and measurements of CPU utilization for various broadcast channels. The predictions are calculated by simulation. The first column enumerates the channels. The second one shows the measurement and observation time in milliseconds. The third column enumerates the predicted CPU utilization for a certain channel. The calculation is performed as explained in Section 1.5. Finally, the fourth column gives the measured CPU utilization for each channel. The measurement of CPU utilization amounts to the measurement of CPU demand and the application of Formula (10.1). The average CPU utilization is calculated as explained in Appendix L.

Table P.1: The comparison of predicted and measured average CPU utilization for various broadcasts

Broadcast channel	Observation time (ms)	Predicted utilization	Measured utilization
Nederland 1	10680	0.163982	0.160547
Nederland 2	11260	0.162283	0.167087
Nederland 3	10560	0.163161	0.169501
RTL 4	12340	0.141729	0.158311
RTL 5	12520	0.161141	0.163794
SBS 6	12380	0.179019	0.162234
Yorin	12080	0.161944	0.167421
V 8	15240	0.162052	0.169413
Net 5/Kindernet5	12540	0.161139	0.161285
UPC Infokanaal	13800	0.16371	0.167335
Omroep Brabant TV	13220	0.162721	0.163087
Lokale Omroep	11000	0.163837	0.166411
Nieuws TV (kabelkrant)	12680	0.16167	0.162684
VRT	10980	0.164749	0.172133
KetNet / Canvas	11940	0.162121	0.163586
National Geographic Channel	16760	0.168868	0.175757
Discovery Channel	12440	0.161949	0.171821
The Music Factory (TMF)	13600	0.163369	0.161272
MTV Europe	18220	0.165081	0.163665
ARD 1	10440	0.163659	0.173966
ZDF 2	11380	0.164081	0.170606
WDR 3	10680	0.163998	0.165159
RTL Television	12140	0.161729	0.168759
BBC 1	17120	0.168567	0.162348
BBC 2	17360	0.167135	0.161079
CNN International	13160	0.162879	0.162944
TV 5 Europe	11260	0.162451	0.165745
TRT International	12940	0.162164	0.166525
Eurosport	13820	0.163447	0.158719
Local VRT	15220	0.162133	0.172229
Average	12992	0.163404	0.165774

Table P.2 enumerates the measured and predicted (by simulation) CPU demands for each broadcast channel. Additionally, the difference between these CPU demands is given in the rightmost column.

Table P.2: Processor demands obtained by measurements and simulation

Broadcast channel	Predicted CPU demand (ms)	Measured CPU demand (ms)	Difference (ms)
Nederland 1	1751.322	1714.637	36.6858
Nederland 2	1827.301	1881.394	-54.093
Nederland 3	1722.98	1789.931	-66.9504
RTL 4	1748.936	1953.558	-204.622
RTL 5	2017.479	2050.695	-33.2156
SBS 6	2216.255	2008.457	207.7983
Yorin	1956.284	2022.446	-66.1622
V 8	2469.672	2581.854	-112.182
Net 5/Kindernet5	2020.677	2022.508	-1.83084
UPC Infokanaal	2259.198	2309.223	-50.025
Omroep Brabant TV	2151.172	2156.01	-4.83852
Lokale Omroep	1802.207	1830.521	-28.314
Nieuws TV (kabelkrant)	2049.969	2062.827	-12.8575
VRT	1808.939	1890.015	-81.0763
KetNet / Canvas	1935.725	1953.217	-17.4921
National Geographic Channel	2830.228	2945.687	-115.46
Discovery Channel	2014.646	2137.453	-122.808
The Music Factory (TMF)	2221.812	2193.292	28.5192
MTV Europe	3007.784	2981.984	25.79952
ARD 1	1708.6	1816.205	-107.605
ZDF 2	1867.236	1941.491	-74.2545
WDR 3	1751.499	1763.898	-12.3995
RTL Television	1963.39	2048.734	-85.3442
BBC 1	2885.858	2779.389	106.4693
BBC 2	2901.455	2796.323	105.1322
CNN International	2143.481	2144.336	-0.8554
TV 5 Europe	1829.198	1866.289	-37.0904
TRT International	2098.396	2154.827	-56.4313
Eurosport	2258.831	2193.49	65.34096
Local VRT	2467.657	2621.318	-153.661

Table P.2 shows that the average CPU utilization tends to be lower for predictions than for measured values. It is however necessary to check if the difference between these average CPU utilizations is not obtained by chance. To check this, Appendix M explains a procedure for comparing average CPU utilizations. We show that this comparison can be accomplished by applying the *t*-test [WEI95] for paired samples to the average CPU demands corresponding to the CPU utilization calculated from the measurements and CPU utilization predictions.

The application of the paired t-test requires the differences between the CPU demands be normally distributed for small size samples (<30). Figure K.1 shows the normal quantile-quantile plot for these differences.

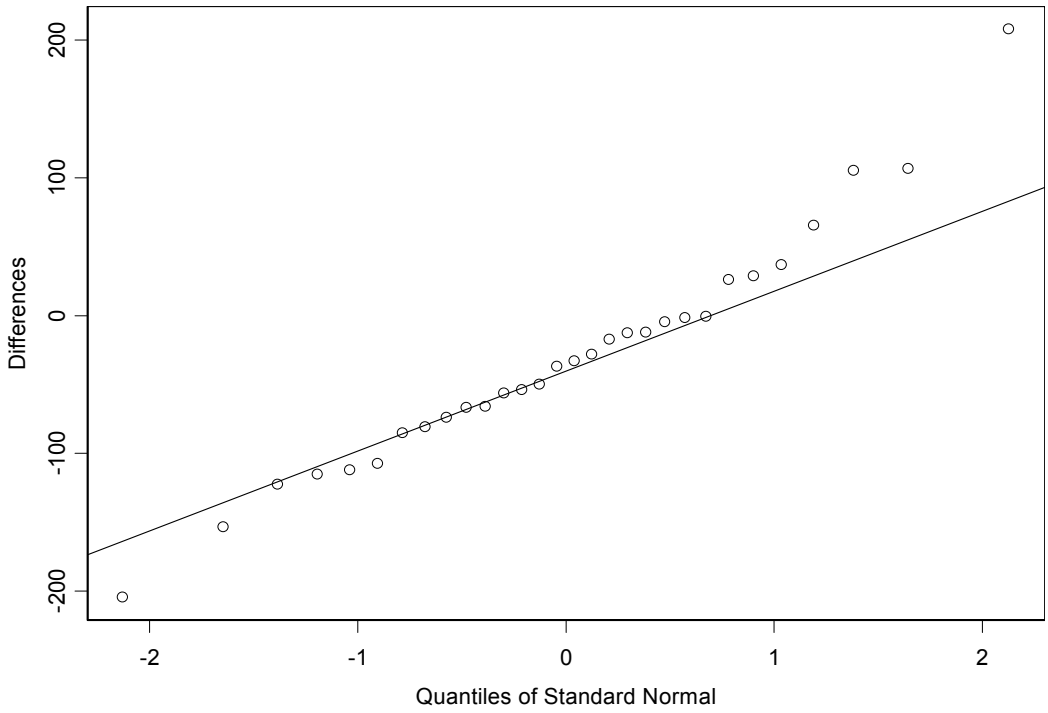


Figure P.1: The normal quantile-quntile plot for the differences in the CPU demands

The normal quantile-quantile plot demonstrates deviation from the normal distribution: there is a number of points that lie far away from the "y=x" line. Particularly, there are two oultiers, which have values of -204.6 ms and 207.8 ms. These outliers are also demonstrated by the box-and-whisker plot in Figure P.2.

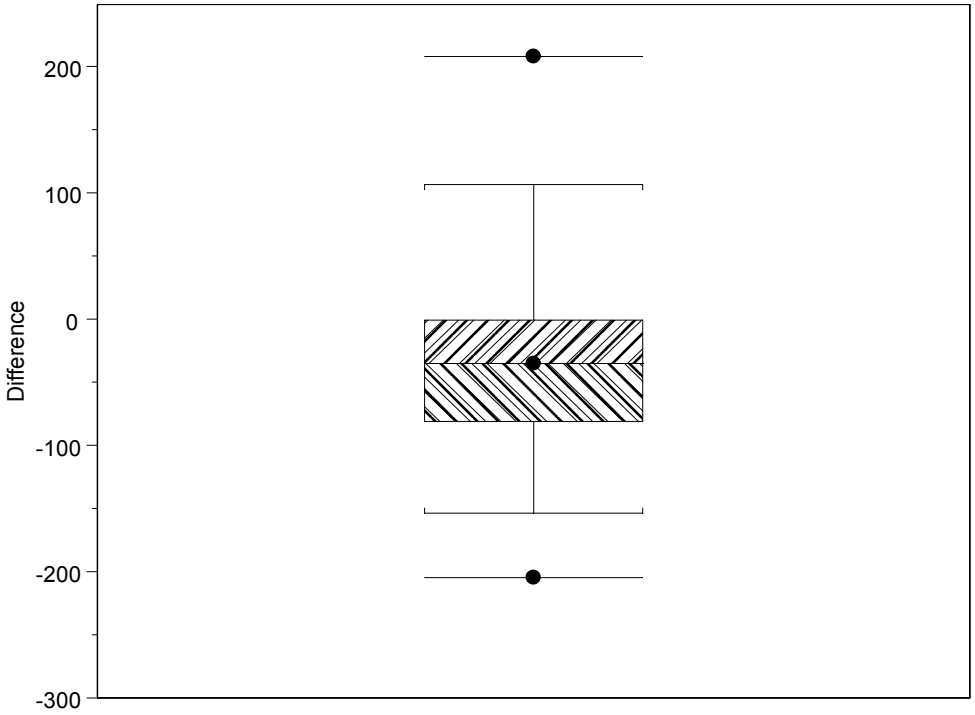


Figure P.2: The box-and-whiskers plot for the differences in the CPU demands

The width of the box is equal to the *interquartile range*, or IQR, which is the difference between the third and first quartiles of the data. The IQR indicates the spread of the distribution for the data. Whiskers extend from the edges of the box to either the extreme values of the data, or to a 1.5 IQR distance of from the median (217 ms), whichever is less. Data points that fall outside of the whiskers may be outliers, and are therefore indicated by additional lines.

Fortunately, as the number of observed channels equals 30, we can still rely on the results of the Central Limit Theorem [WEI95], saying the sum of a large number (30 and more) of independently identically distributed random variable is a random variable that has normal distribution. This means that the *t*-test is still applicable.

We formulated the following null hypothesis for the one-sided *t*-test: there is no difference between the measured CPU demand and the CPU demand predicted by simulation. The alternative hypothesis was as follows: the predicted CPU demand is less then the measured one. The *t*-test provided the following results: *t*-statistic = -2.0157 on 29 degrees of freedom, which gives *p*-value = 0.0266. The *p*-value is less than the selected significance level of 0.05. Therefore we have to reject the null hypothesis and to accept the alternative one. The average difference between the predicted and measured CPU demand equals -30.79, and it is significant at the selected significance level. By dividing this average difference by the average observation period (see Table K.1), we can conclude that the simple analytical formula underestimates the actual CPU utilization by 0.00237 on the average, which means the average relative prediction error of -1.43%.

Appendix Q. Performance prediction for the Professional Systems Case

Q.1 The quality of the prediction model for the response time of the "Archiving" component

In this section of the appendix, only conclusions from the plots are drawn. For more detailed description of the plots, the reader is referred to the Appendix R.

Figure Q.1 shows that residual is distributed normally and residual standard deviation is constant, as there is no structure or trends in the distribution "residual vs. fit".

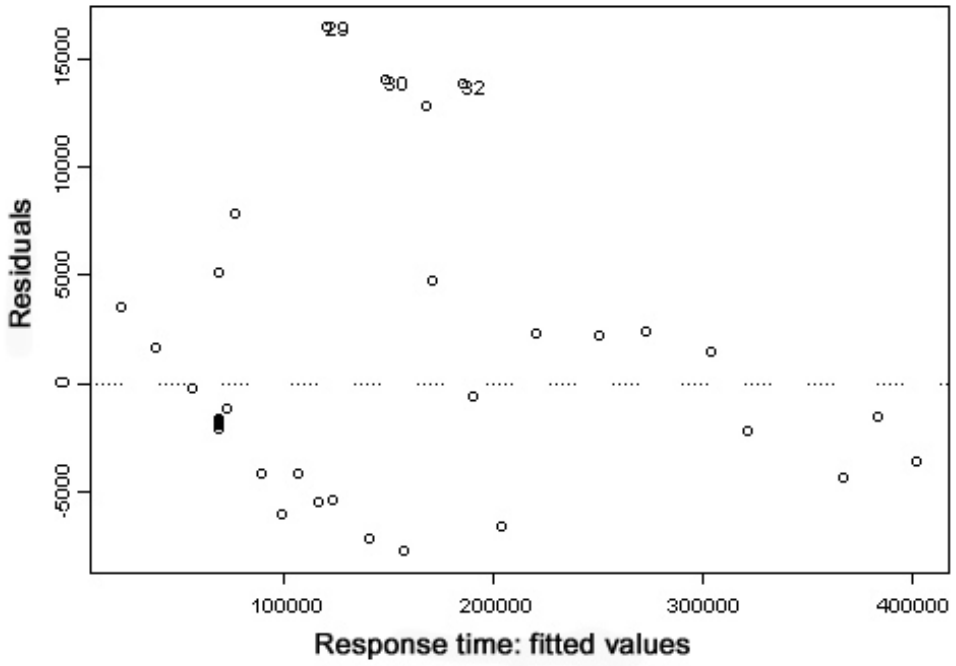


Figure Q.1: Residual distribution

Figure Q.2 confirms the linear dependency between response time and signature parameters, as the majority of the points are located close to the "y=x" line. The closer the fitted values to the line "y=x", the better the quality of the model is.

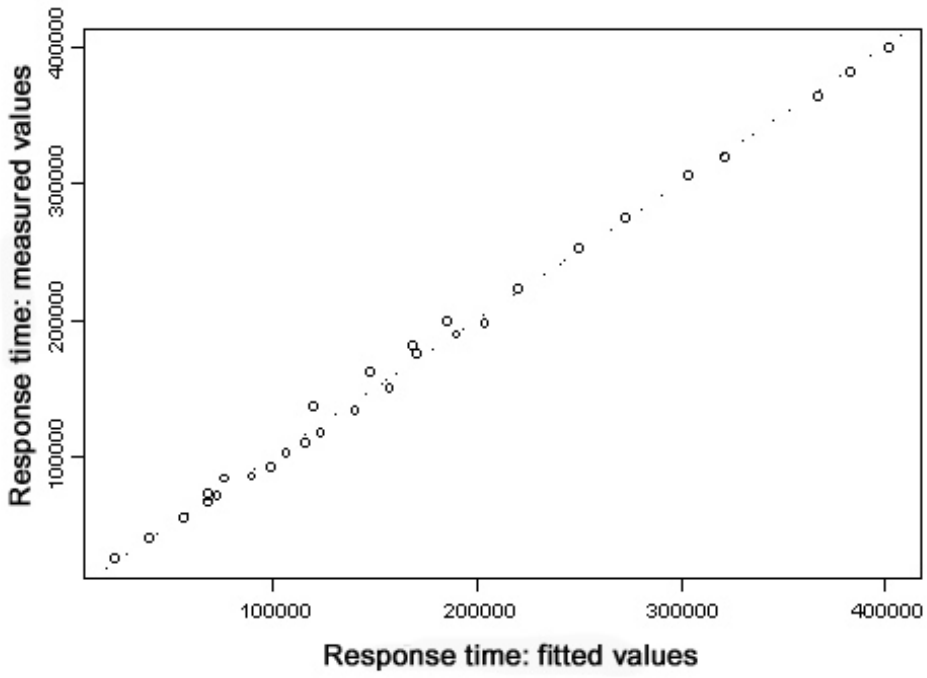


Figure Q.2: Response time: measured values versus fitted values

Figure Q.3 demonstrates high explanatory power of the prediction model, since the residual spread is considerably narrower than fitted values spread. The vertical spread of the residuals compared to the vertical spread of the fitted values gives an indication of how much of the variation is explained by the fit. The wider is the spread of the fitted values, in comparison to the residual spread, the better the quality of the model.

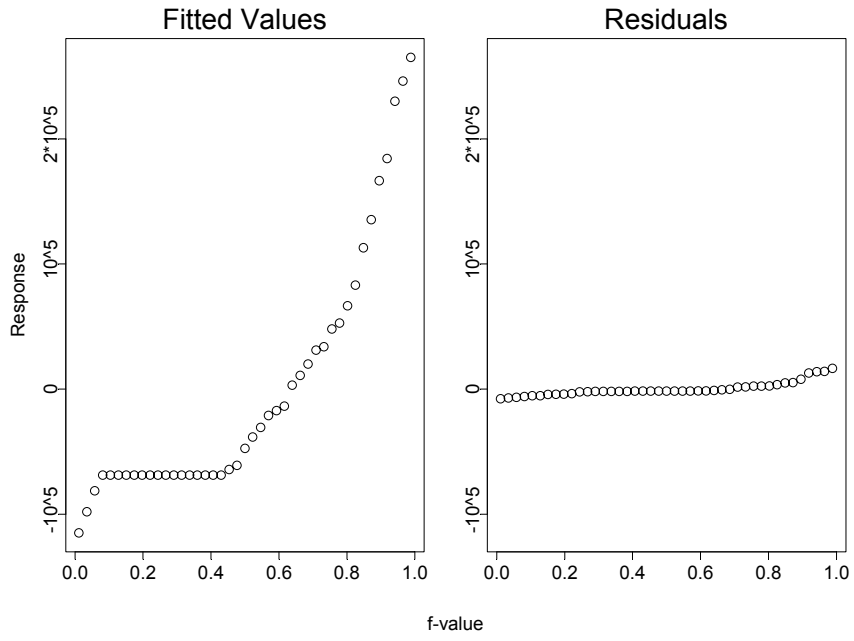


Figure Q.3: Fitted values and residuals of the response time

Q.2 Validation of the prediction formula

Table Q.1 presents the results of the measurements of the response time of the composition. The columns 1-4 contain a number of different invocation patterns for each use-case of the "Reviewing" component. "Pat1" has two timeouts (after each action), "Pat2" and "Pat3" have only one timeout (after first and second action, respectively), and "Pat4" does not have any timeouts. For each use case, the corresponding number of resource switches and timeouts were calculated. The response time, including the execution times of the components, all timeouts and resource acquisition overhead, is calculated by the Formula (11.4). This time is then compared to the measured time. The last two columns show the absolute and relative prediction errors, respectively.

Figure Q.4 describes the fitted (predicted) values of the $T_{unknown}$ time versus the measured values. The closer the fitted values to the plot "y=x", the better the quality of the model is. This plot confirms the linear dependency between $T_{unknown}$ and signature parameters, as the majority of the points are located within the "y=x" plot.

Figure Q.5 demonstrates high explanatory power of the prediction model, since the residual spread is considerably narrower than fitted values spread.

The vertical spread of the residuals compared to the vertical spread of the fitted values gives an indication of how much of the variation is explained by the fit. The wider the spread of the fitted values is, in comparison to the residual spread; the better the quality of the model is.

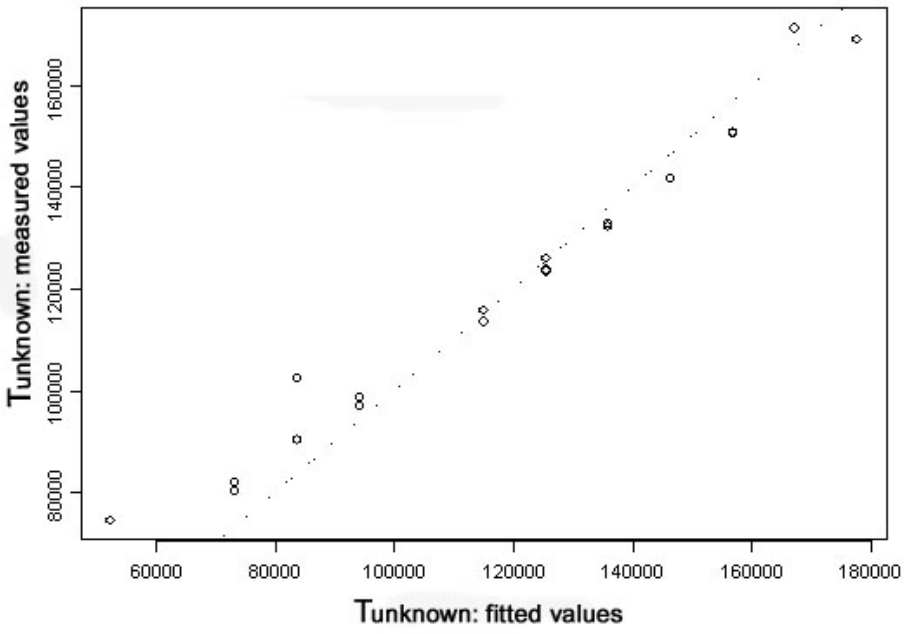


Figure Q.4: Fitted values

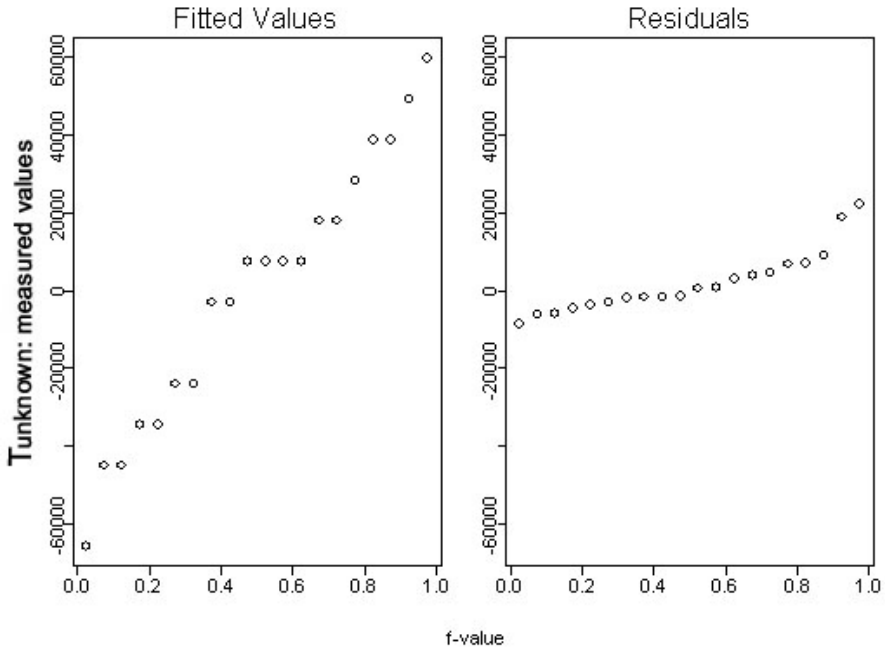


Figure Q.5: Fitted values versus the residual spread of $T_{unknown}$

Table Q.1: Results of measurements

PAT1	PAT2	PAT3	PAT4	#RESOURCE SWITCH	RESOURCE TIME	#TIME-OUTS	TIMEOUTS TIME	TOTAL TIME	MEASURED TIME	ABS. ERROR	REL. ERROR	TUNK-NOWN
1	1	1	7	10	3200	5	50000	119098	140371	21273	0.151548	74473
4	1	2	3	23	7360	12	120000	193258	191802	1456	0.007591	125904
5	1	3	1	28	8960	15	150000	224858	216419	8439	0.038994	150521
2	0	3	5	13	4160	8	80000	150058	168360	18302	0.108708	102462
3	5	1	1	30	9600	13	130000	205498	198156	7342	0.037052	132258
1	7	1	1	28	8960	11	110000	184858	179403	5455	0.030406	113505
1	5	1	3	22	7040	9	90000	162938	162861	77	0.000473	96963
2	2	1	5	17	5440	8	80000	151338	156182	4844	0.031015	90284
3	2	2	3	22	7040	11	110000	182938	181711	1227	0.006752	115813
4	2	2	2	26	8320	13	130000	204218	198770	5448	0.027409	132872
6	0	3	1	29	9280	16	160000	235178	237062	1884	0.007947	171164
2	6	1	1	29	9280	12	120000	195178	189527	5651	0.029816	123629
2	1	1	6	14	4480	7	70000	140378	147955	7577	0.051212	82057
2	6	1	1	29	9280	12	120000	195178	189587	5591	0.02949	123689
2	2	5	1	21	6720	12	120000	192618	189358	3260	0.017216	123460
7	1	1	1	34	10880	17	170000	246778	234961	11817	0.050293	169063
5	2	2	1	30	9600	15	150000	225498	216649	8849	0.040845	150751
5	1	2	2	27	8640	14	140000	214538	207649	6889	0.033176	141751
1	3	1	5	16	5120	7	70000	141018	146113	5095	0.03487	80215
2	1	3	4	16	5120	9	90000	161018	164564	3546	0.021548	98666

Appendix R. Residual diagnostic plots

This appendix describes a number of residual diagnostic plots typically used for checking the assumptions of linear regression [WEI95], [MON01], [KO02], and [MR03]. These are the following plots:

1. Residual versus fit
2. Square root of absolute residual versus fit
3. Response versus fit
4. Residual normal plot
5. Residual-fit spread
6. Cook's distance

The sections demonstrate the use of those plots by means of a simple example. We artificially constructed observation data based on the following formula:

$$y_i = 1.0 + 2.0 \cdot x_{i1} + 3.0 \cdot x_{i2} + e_i. \qquad \text{(R.1)}$$

In Formula (R.1), y_i denotes the independent variable that needs to be predicted. In chapter VI, this variable describes the static quality attribute; in the rest of the chapters, it corresponds to a particular performance measure. Independent variables are denoted as x_{i1} and x_{i2}. These variables correspond with diversity parameters from chapter IV and with signature parameters from the rest of the chapters. Finally, e_i is a normally distributed random variable with mean zero and a particular constant standard deviation. This random variable models measurement errors. In this experiment, we arbitrary chose the value of the standard deviation $\sigma = 0.25$.

We generated a sample of observation data consisting of 100 points as follows. For each point *i*, the values of x_{i1} and x_{i2} are obtained from a uniform random number generator in the range 0.0 to 1.0. The value of ε_i is calculated by generating a normally distributed variable with mean zero and standard deviation $\sigma = 0.25$. The values of the independent variable y_i are calculated by formula (R.1).

We fitted a (multiple) linear regression model to the observation data constructed as outlined above. Of course, we used the form of the model that resembles Formula (R.1):

$$y_i = \hat{\beta}_0 + \hat{\beta}_1 \cdot x_{i1} + \hat{\beta}_2 \cdot x_{i2} + \varepsilon_i. \qquad \text{(R.2)}$$

In this formula, $\hat{\beta}_j$ denote estimates of the true regression coefficients β_j, and ε_i are residuals. For the example from Formula (R.1), $\beta_0 = 1.0$, $\beta_1 = 2.0$, and $\beta_2 = 3.0$.

After applying the S-Plus linear regression tool [KO02] to the observation data, the following results were obtained. The multiple R^2-coefficient equals 0.95, which indicates that the obtained model explains the observed variation of the dependable variable y_i well. Moreover, the F-test for overall regression indicated that the regression is significant at the significance value 0.05 (*p*-value equals 0.000). The residual standard error σ_e equals 0.2532 (on 97 degrees of freedom). Noticeable is the fact that σ_e estimates quite well the chosen standard deviation $\sigma = 0.25$ of the measurement error. Finally, the estimates of regression coefficients are as follows: $\hat{\beta}_0 = 1.0023$, $\hat{\beta}_1 = 2.0236$, and $\beta_2 = 3.0589$. These estimates are significant at the significance level 0.05 (the corresponding *p*-values equal 0.000).

We can conclude that the obtained regression model has high quality, as (1) it has high explanatory power (the value of R^2-coefficient is close to one), and (2) the overall regression and all regression coefficients are significant. Notice that this conclusion is also supported by the fact that the estimates $\hat{\beta}_j$ are close the true regression coefficients β_j. The difference is in the order of few percents.

However, this conclusion still does not allows us to apply the "full-fledged" linear regression analysis (e.g., making inferences about the confidence intervals of regression coefficients and mean values of predictions), as a number of the vital assumptions of linear regression have not been checked yet. These assumptions are the following [WEI95], [MON01], [KO02], and [MR03]:

7. The normality of the residual distribution
8. The constancy (homoscedasticity) of the residual standard deviation
9. The absence of outlying observations (so called *outliers*)
10. The absence of influential observations

There are extra assumptions (e.g., the absence of serial correlation) that need to be checked. We do not consider these assumptions in this appendix, as they can be hardly checked by means of residual diagnostic plots. Within the context of Formula (R.1), the subsequent sections exemplify the use of the residual diagnostic plots enumerated above and the correct shape of those plots.

R.1 Residual versus fit

Figure R.1 demonstrates the "Residual versus fit" plot [KO02] for Formula (R.1). The x-axis is the fitted value, i.e., a predicted value of the independent variable. The y-axis denotes the corresponding residuals.

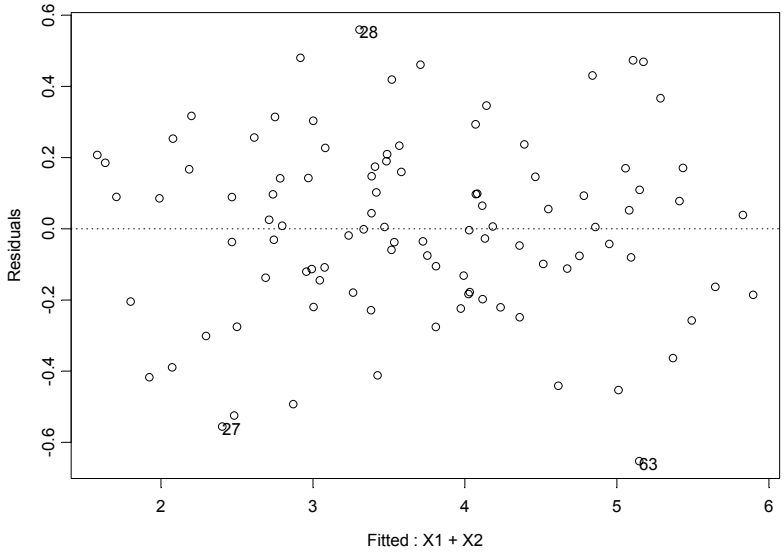


Figure R.1: "Residual versus fit" plot

Plots of these types are used to check assumptions 7 (the normality of the residual distribution) and 8 (the constancy of the residual standard deviation). The first assumption is likely to be violated, if points in this plot follow a certain pattern with a recognizable structure in it (e.g., distinguishable groups of points such as lines). The tendency of points to stay closer (or further) to (from) the x-axis with the increase of the fitted value indicates the violation of the second assumption.

Figure R.1 demonstrates a good example of the case when both the first and second assumptions are satisfied: there is no visible structure in the residuals and there is not tendency for decreasing or increasing of the residual, depending on the fitted value.

R.2 Square root of absolute residual versus fit

Figure R.2 demonstrates the "Square root of absolute residual versus fit" plot [KO02] for the example (R.1). The x-axis is the fitted value, i.e., a predicted value of the independent variable. The y-axis denotes the square root of the absolute value of the corresponding residuals.

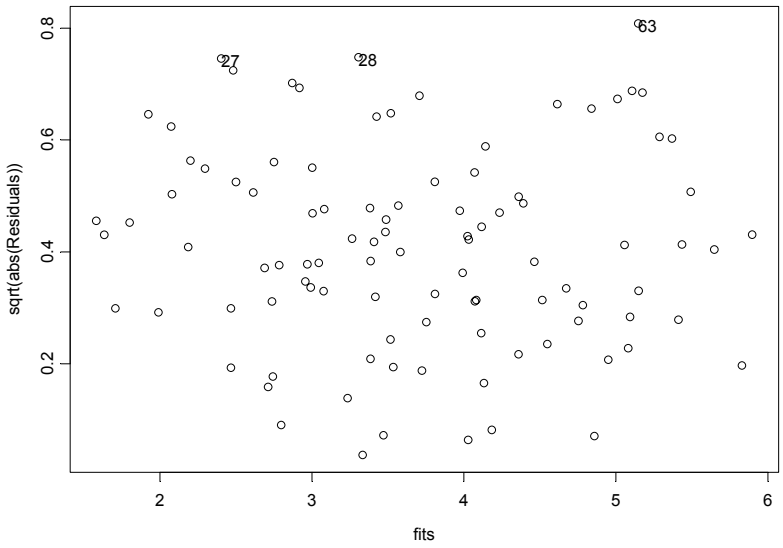


Figure R.2: "Square root of absolute residual versus fit" plot

Plots of this type are used to check for the violation of assumption 9, i.e., the absence of outlying observations. The existence of a few points from the upper part of the plot that lie far apart from the rest of the point indicate the presence of outlying observations.

Figure R.2 shows that example (R.1) does not have any *outliers*. The most outer points 27, 28 and 63 are close enough to the rest of the points. Those points would be outliers, if the square roots of the absolute values of the corresponding residuals were greater, e.g., than 1.0.

R.3 Response versus fit

Figure R.3 demonstrates the "Response versus fit" plot [KO02] for the example (R.1). The x-axis is the fitted value, i.e., a predicted value of the independent variable. The y-axis denotes the actual (measured) value of the independent variable.

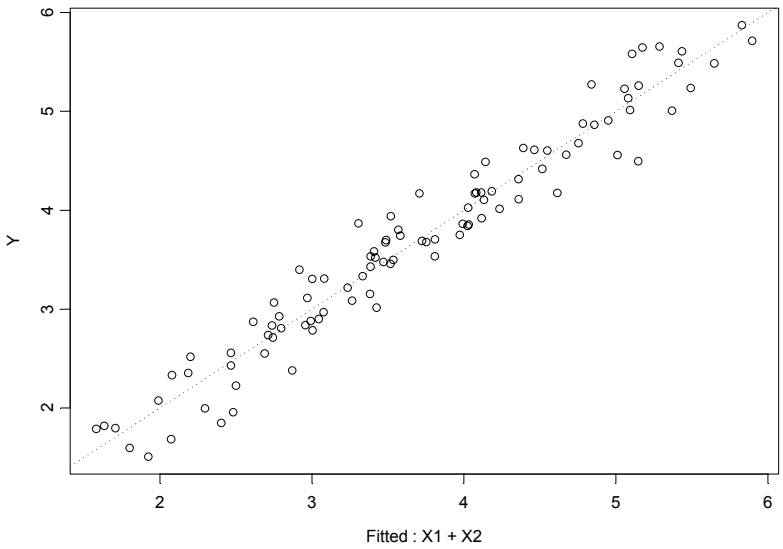


Figure R.3: "Response versus fit" plot

Plots of this type are primarily used for checking the assumption of linear dependency between the independent variable and dependent ones. All points must lie around the diagonal line "y=x" and there should be no visible structure in their location pattern. The location pattern that is different from a "blurred" line (e.g., U-shape) signifies the violation of the linearity assumption. A "parallel lines" location pattern may be caused by a missing independent variable.

Figure R.3 is a perfect example of the correct "Response versus fit" plot. All points are located around the "y=x" line, without any visible dependence on the fitted value.

R.4 Residual normal plot

Figure R.4 demonstrates the Residual normal plot [KO02] for the example (R.1). The x-axis enumerates quantiles of the standard normal distribution. The y-axis list the residuals.

Plots of this type are used to check assumption 7, i.e., the normality of the residuals. These plots are constructed as follows. Consider observation data that have *N* points. Consequently, the number of the residuals equals also *N*. These residuals are sorted in the ascending order and paired with the *N* quantiles of the standard normal distribution sorted in the same order. Each obtain pair is depicted in the plot.

We conclude that the residuals have the normal distribution, if they lie close to the "y=x" line in this plot. Violation of this pattern indicates the non-normality of the residual distribution.

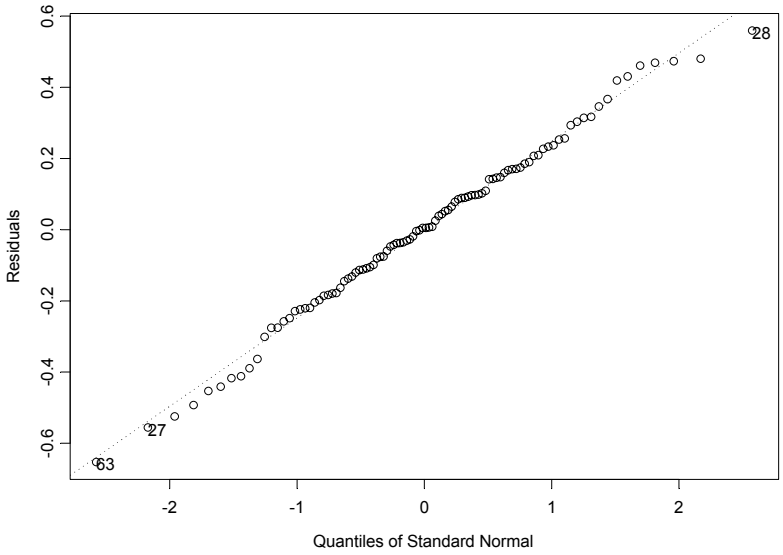


Figure R.4: Residual normal plot

Figure R.4 exemplifies the residuals that follow the normal distribution quite well. This fact allows us to conclude that assumption 7 is not violated in the example (R.1).

R.5 Residual-fit spread

Figure R.5 demonstrates the "Residual-fit spread" plot for the example (R.1). The y-axes denotes the fitted values and residuals in the left and the right subplot, respectively. The x-axes enumerate the corresponding *f*-values. The f-values are calculated as follows:

$$f(x) = \frac{x - x_{\min}}{x_{\max} - x_{\min}}. \qquad \text{(R.3)}$$

In Formula (R.3), $f(x)$ denotes the *f*-value for the point *x*. In the left subplot, *x* denotes the fitted values. In the right subplot, it denotes the residuals. Finally, $x_{\min}$ and $x_{\max}$ are the minimum and the maximum, respectively, value of *x*.

Plots of this type are used to judge the explanatory power of the overall regression. These plots are in fact graphical analogues of the R^2-coefficient. The explanatory power is checked as follows. The spread (the difference between $f^{-1}(1.0)$ and $f^{-1}(0.0)$) of the residuals is compared to the spread of the fitted values. The letter must be significantly larger then the former to ensure the high explanatory power of the model.

Figure R.5 demonstrates that the model fitted for the example (R.1) indeed has high explanatory power. This conclusion is also supported by the high value of the R^2-coefficient (R^2=0.95).

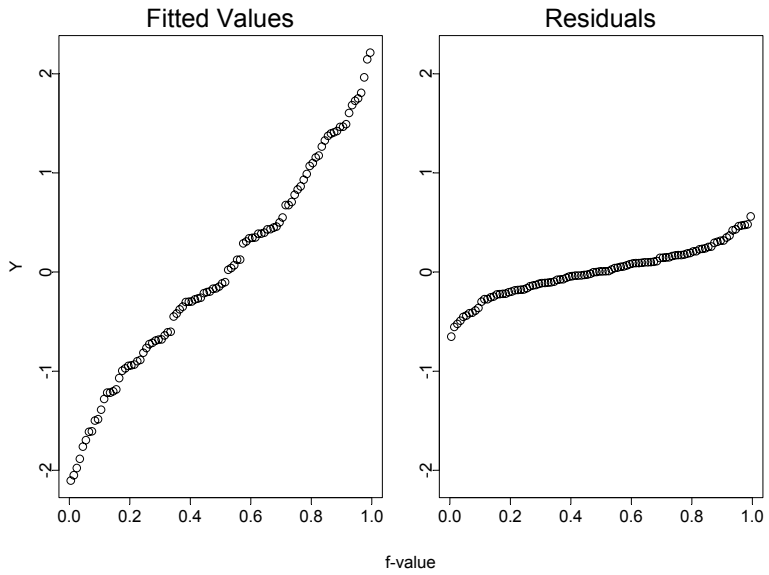


Figure R.5: "Residual-fit spread" plot

R.6 Cook's distance

Figure R.6 demonstrates the "Cook's distance" plot for the example of Formula (R.1). The x-axis enumerates the indexes of the points from the observation data. The y-axis denotes the corresponding Cook's distances.

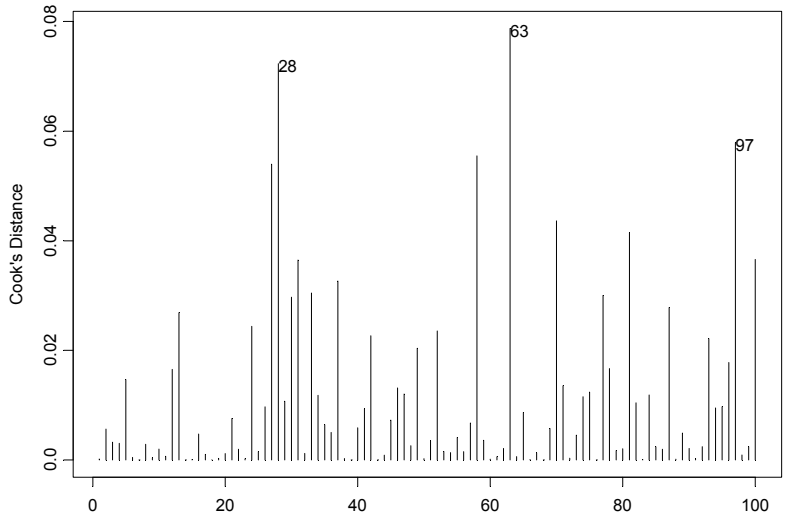


Figure R.6: "Cook's distance" plot

The Cook's distance measure [MR03] is a statistic calculated for a certain point on the basis of excluding this point from the observation data and fitting an auxiliary regression model without the excluded point. The estimates of the estimates of the regression coefficients of this model are used together with the corresponding estimates of the original regression model (one that includes all observation data) to calculate the Cook's distance measure. These calculations involve matrix operations and are not described in this appendix. For more details, the reader is referred to [MR03].

Plots of this type are used to check for the violation of assumption 10, i.e., the absence of influential observations. The influential observations are those observations that significantly influence the values of the regression coefficients. In other words, the regression coefficients are sensitive to small changes in the values of these influential observations. Points that have the Cook's distance greater then 1.0 are considered influential observations.

Obviously, Figure R.6 demonstrates that the model fitted for Formula (R.1) has no influential observations, as the Cook's distance does not exceed 0.08 for all points from the observation data.

Appendix S. Overview of Simulators

A table given below summarizes the results of a survey on available free and commercial event-driven simulation tools.

Table S.1: Overview of the existing simulator engines

Simulator / vendor	Functionality	Supported programming languages	User Interface	Support	License costs
SHESim / TUE	1. OO approach: • Process objects • Data objects • CSP-comm. semantics 2. Behavior is described in POOSL 3. Supports system decomposition 4. Supports model libraries	POOSL, also supports generating fast executables	GUI, user's manual	Restricted	Free academic license
Statemate MAGNUS/ I-Logix	1. Oriented on prototyping and executable specifications 2. Automatically generates design documentation 3. Supports "V" development process. 4. Supports system decomposition with activity charts 5. Based on Harel's statecharts 6. Supports hybrid models 7. Supports model libraries 8. Compatible with Rhapsody/I-Logix	1. C or Ada code generation for software developers 2. VHDL code for hardware developers	Menu driven GUI, Extensive documentation	Extensive support is offered, courses on the use of Statemate	Two types of licenses are available: academic and commercial. Both seem to be extremely expensive (thousands dollars)

Simulator / vendor	Functionality	Language	User Interface	Support	License costs
Stateflow (Coder) 4/ The MathWorks	1. Based on UML statecharts and flow diagram notation 2. Supports hierarchy, parallelism, junctions, and history 3. Allows visual programming of common code structures: loops, if-then-else statements, via flow diagrams 4. Supports directed event-broadcasting 5. Schedules events using temporal operators ("before", "after", "at", and "every") 6. Incorporates custom and legacy C code with input and output arguments	1. Uses a proprietary graphics language based on UML statecharts. 2. An additional product, Stateflow Coder, supports automatic C code generation	Menu driven GUI, oriented on visual programming	Support and courses are offered	The licensing is sophisticated. One distinguishes commercial and academic licenses and options like group, concurrent, individual use. To be able to use the package in a full scale, it is necessary to buy also MATLAB and Simulink. The total price for individual academic use is 2400 euro.
CSIM18 / Mesquite Software	1. Is a library of C/C++ routines 2. Used for process-oriented, discrete-event simulation models 3. A model consists of structures (to simulate resources) and processes (to simulate active entities) 4. Interactions between model processes correspond to ones within a typical OS: a process can be in some state, e.g. ready to run, it can acquire a resource, etc. 5. Supports statistical distributions to model timed activities	C, C++	No GUI is provided, a user model is linked to CSIM18 package	Installation and bug-fix support is available for 90 days through email and fax only on an as-available basis through a designated support contact. An Extended Support Agreement which covers unlimited fax or email, bug-fix updates, and feature enhancement upgrades is also available.	There are a number of licensing options, differentiating also by a number of seats: • Professional, • Educational, • Student, • Corporate Site, • Developer. The price depends on the type of the license and varies in a range 50-895$ for a one seat license.

Simulator / vendor	Functionality	Language	User Interface	Support	License costs
COVERS/ XjTek, SPbSTU, department of Distributed Computing Networks	1. Is a object-oriented framework for building simulators 2. Supports model observation and modification during run-time 3. Builds on the notion of communicating active objects 4. Communication mechanism is message passing 5. All interactions are performed through ports 6. Describes the behavior of an active object with a Harel's statechart 7. Provides an extended class library supporting statistics gathering and visualization 8. Allows building real-time models to be used with real environment	C++ Supports generation of GUI models with Borland C++ 4.5 and console models with both Borland C++ 4.5 and Microsoft Visual C++ 5.0	Menu-driven GUI, Visual programming, Extensive programmer's and user's manuals	Two types of support are offered: • Free support, including answering product related e-mails • Priority support for those who bought xjTek's products (for 90 days); the response is guaranteed within 24 hours	Two types of licenses are available: • Free academic license for non-commercial use • Commercial license. Prices are negotiable
AnyLogic 4.0/ XjTek, SPbSTU, department of Distributed Computing Networks	Extends the range of supported features by COVERS: 1. Supports hybrid models 2. More elaborated GUI 3. Modeling is entirely base on the UML-RT technology	Java, Java 2 SDK	Menu-driven GUI, Visual programming, Extensive programmer's and user's manuals	Two types of support are offered: • Free support, including answering product related e-mails • Priority support (for 90 days); the response is guaranteed within 24 hours	Evaluation version is available. Licensing is negotiable through e-mail

Glossary

Activity	A unit of concurrency in software systems (e.g., thread, process, etc.)
Activity instance	An instance of an activity, a unit of scheduling
Additive static quality attribute	A static QA is additive when the QA of a component composition can be calculated by summing up the corresponding QA's of the constituent components.
Alternative hypothesis	The alternative hypothesis, H_1, is a statement of what a statistical hypothesis test is set up to establish.
Analytical model	The model that describes the dependencies by an algebraic equation (e.g., formulas, Markov models, etc.)
APPEAR	Analysis and Prediction of Performance for Evolving Architectures
Application	An application is an independent software unit responsible for a specific part of functionality.
Arrival pattern	An arrival pattern describes the relationship between release times of different instances of the same activity.
Blocking	Causing a process to incur waiting time that is not attributable to preemption from higher priority processes (e.g., by occupying a required resource) [KRP93]
Calibration	A process of construction of the statistical prediction model by fitting it to the measurements
Categorical parameter	A parameter that can have only finite number of values
Component	A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only [Szy98].
Component binding	The act and/or result of connecting the interfaces of components
Component diversity	Options of a component to be tuned to its environment. A single aspect of this tuning is represented by a diversity parameter.
Component hierarchy	A ranked series of components. Usually, we mean the containment hierarchy of components, that is, a series of components that include one another.
Component model	A set of rules that specify how to develop, use, and connect software components
CBSE	Component-based software engineering; a software engineering discipline concerned with building software by assembling (composing) software from components
CFG	Control Flow Graph, a notation for modeling the control flow of artifacts such as functions, activities, etc.

Composition	A process and/or a result of combining of several independent entities (applications, components, etc.); often entities are combined to execute a common task. The result of this combination is also referred to as an assembly.
Confidence interval	A confidence interval gives an estimated range of values which is likely to include an unknown parameter. This range is calculated from a given set of sample data.
CPU	Central Processing Unit
Diversity parameter	A variable whose value allows tuning a particular aspect of a component. For instance, a diversity parameter may select the language in which a component reports messages to the user.
Dynamic quality attribute	Dynamic quality attribute is a run-time attribute, that is, it may change in run-time. For its estimation, it is important to consider not only the way that components are bound, but also the way that they interact with the environment and each other.
Effects of a factor	The difference between the observed values of the dependent variable at a number of factor levels
Factor	A factor of an experiment is a controlled independent variable; a variable whose levels are set by the experimenter.
Factor level	The value of a factor
File	A collection of runs for a single patient; there can be multiple files of one patient.
FLOF	Full Level One Facilities; the type of a Teletext navigation system
F-statistic	A statistic, which needs to be calculated to test certain hypotheses such as the significance of overall regression. This statistic has the *F*-distribution.
Hypothesis test	Setting up and testing hypotheses is an essential part of statistical inference. In order to formulate such a test, usually some theory has been put forward, either because it is believed to be true or because it is to be used as a basis for argument, but has not been proved.
Image	A single medical image retrieved during patient acquisition process
Image processing setting	An attribute of the image (e.g., brightness, contrast, etc.)
Influential observation	A single observation point (measurement) to small changes of which the values of regression coefficients are sensitive
Interaction of factors	The dependence of the effect of one factor on the level of another factor
Invocation pattern	An invocation pattern describes the order and the frequency of invocation of the operations of a particular component.
Koala	The component model used for the software development for resource-constrained systems (e.g., TV's)
Least-squared error approach	An approach for estimating the values of a parameter in regression. This approach is based on the minimization of the sum of squares of prediction errors calculated for the observed points.
MIP	Magazine Inventory Page

MISS	Medical Imaging Software System. This system is reponsible for medical image acquisition, viewing, archiving, etc.
MOT	Magazine Organization Table
MSD	Message Sequence Diagrams
Null hypothesis	The null hypothesis, H_0, represents a theory that has been put forward, either because it is believed to be true or because it is to be used as a basis for argument, but has not been proved.
Outlier or outlying observation	A single observation point (measurement) that is significantly distant from the majority of other observations.
Performance	A software quality attribute that shows to what degree the software meets its objectives for timeliness; timeliness includes two important aspects: responsiveness and scalability [SW02].
Preemption	The act of reallocating a resource to a higher priority process when it becomes ready to execute during the execution of a lower priority process [KRP93].
Prediction	Estimation of results of new experiments, given the inputs only, by means of statistical model based on the results and inputs of the previous experiments.
Prediction interval	An interval that quantifies the degree of uncertainty of a prediction
Prediction model	A model that is used for estimation of the properties of the future systems, usually based on the properties of the existing systems
Provides interface	A provides interface of a component describes the functionality provided by this component to other components.
p-value	The probability value (p-value) of a statistical hypothesis test is the probability of getting a value of the test statistic as extreme as or more extreme than that observed by chance alone, if the null hypothesis H0, is true.
Quality attribute	The attribute of a software system that does not relate to the functionality but to quality (e.g, reliability). This attribute usually cannot be pinpointed to a particular part of a system but it is a property of a system as a whole [Bos00].
Real parameter	A parameter that may have values from the set of Reals.
Regression	The mathematical analysis of numerical data to identify a formula that best fits the trends in the data - often with the aim of successful prediction of future data.
Requires interface	A requires interface specifies the functionality that component requires from other components.
Residual	Residual (or error) represents unexplained (or residual) variation after fitting a regression model. It is the difference (or left over) between the observed value of the variable and the value suggested by the regression model.
Response time	The time spent in handling an event by a software system; the time between reception of an event and generation of a response
Run	A collection of images of a patient obtained during a single acquisition session

Scheduling policy	A set of rules that define the order in which activities (processes, threads, etc) are allocated processors and other resources
Service call, signature call	Invocation of a service of a Virtual Service Platform
Signature instance	A value of the signature parameters forming a particular signature type
Signature instance extraction	A process of calculation of the values of signature parameters of a particular signature type
Signature parameter	A single parameter of a signature type
Signature type	A signature type of an application is a set of parameters that provide sufficient information for performance prediction
Signature type identification	A process of identification of signature parameters
Significance level	The significance level of a statistical hypothesis test is a fixed probability of wrongly rejecting the null hypothesis H0, if it is in fact true.
Simulation model	The model that represents the software behavior in a simplified, high-level form
Software architecture	The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them [BCK03].
Static quality attribute	Static QA describes the properties of a component composition that do not change at run-time. This QA is determined by the structure of software, but not by its behavior.
Statistic	A statistic is a quantity calculated from a sample of data. Its value is often used to decide whether or not the null hypothesis should be rejected in a hypothesis test.
Teletext	A European standard service for transmitting textual information to display it on a TV set
TOP	Table of Pages
Tracing	A process of printing the selected information into a file during program execution
t-statistic	A statistic, which needs to be calculated to test certain hypotheses such as the equality of the means of two populations. This statistic usually has the *t*-distribution.
Use case	One of the possible instances of user interaction with the software system
VBI	Vertical Blanking Interval

VSP	Virtual Service Platform, an abstract representation of the part of software application; it is considered as a platform that provides independent services to another part of the application and consists of stable parts only.
VPS	Video programming System
WSS	Wide Screen Signaling

References

[ABI00] 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.

[Alt96] P. Altenbernd, On the false path problem in hard real-time programs. In Proceedings of the eighth Euromicro Workshop on Real-Time Systems, pages 102--107, June 1996.

[BBB00] F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, R. Seacord, K. Wallnau, Technical Concepts of Component-Based Software Engineering, 2nd edition, Technical Report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, 2000

[BCK03] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison-Wesley, 2003

[BeB00] P. Bengtsson, J. Bosch, Assessing Optimal Software Architecture Maintainability, accepted for the fifth European Conference on Software Maintainability and Reengineering, September 2000

[BK02] 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, pp.971-976, March 2002, Paris, France.

[BKL95] M. Barbacci, M. H. Klein, T. A. Longstaff, C. B. Weinstock, Quality Attributes, Technical Report CMU/SEI-95-TR-021 ESC-TR-95-021, December 1995 Software Engineering Institute

[BKR97] A. V. Borshchev, Y. G. Karpov, V. V. Roudakov, COVERS 3.0 - An Object-Oriented Environment for Modeling, Simulation and Analysis of Real-Time Concurrent Systems, In Proceedings of 1st International Workshop on Distributed Interactive Simulation and Real-Time Applications (DIS-RT '97), January 09 - 10, 1997

[BM03] A. Bertolino, R. Mirandola, Towards Component-Based Software Performance Engineering, In Proceedings of 6th ICSE Workshop on Component-based software engineering, Portland, USA, May 2003.

[BMW04] E. Bondarev, J. Muskens, P. de With, M. Chaudron, J. Lukkien; Predicting Real-Time Properties of Component Assemblies: a Scenario-Simulation Approach; in Proceedings of 30th Euromicro Conference, Rennes, France, September 2004.

[Boe81] B. Boehm, Software Engineering Economics. Prentice-Hall, Inc., 1981.

[Bol01] F. Bolton, Pure Corba, SAMS, 2001

[Bon99] G. Bontempi, Local Learning Techniques for Modeling, Prediction and Control, PhD thesis, IRIDIA- Universite' Libre de Bruxelles, Belgium, 1999.

[Bos00] J. Bosch, Design & Use of Software Architectures: Adopting and evolving a product-line approach, Addison-Wesley, 2000

[Box97] D. Box, Essential COM, Addison-Wesley, 1997

[BP01] G. Bernat , P. Puschner, WCET Analysis of Reusable Portable Code. In Proceedings of 13th Euromicro Conference on Real-Time Systems, pages 45—58, June 2001.

[BWC04] E. Bondarev, P. de With, and M. Chaudron, Predicting Real-Time Properties of Component-Based Applications, In Proceedings of the 10th International RTCSA Conference, Gothenburg Sweden, August 2004.

[CKK02] P. Clements, R. Kazman, M. Klein, Evaluating Software Architectures, Addison-Wesley, 2002

[CL02] I. Crnkovic, M. Larsson, Building reliable component-based software systems, Artech House, 2002

[CP01] A. Colin, I. Puaut, A modular and Retargetable Framework for Tree-Based WCET Analysis. In Proceedings of 13th Euromicro Conference on Real-Time Systems, pages 37—44, June 2001.

[DMM03] N. Dumitrascu, S. Murphy, L. Murphy, A methodology for Predicting the Peformance of Component-Based Applications, In Proc. of WCOP 2003, the 8th International Workshop on Component-Oriented Programming, Darmstadt, Germany, 21-25 July, 2003.

[EBK03] W. Elmenreich, G. Bauer, H. Kopetz, The Time-Triggered Paradigm, In the Proceedings of the Workshop on Time-Triggered and Real-Time Communication, Manno, Switzerland, December 2003

[EE00] J. Engblom, A. Ermedahl, Modeling complex flows for worst-case execution time analysis. In Proc. 21 st IEEE Real-Time Systems Symposium (RTSS'00), November 2000.

[EES01] J. Engblom., A. Ermedahl, F. Stappert, Effcient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. Technical Report 12, Dept. of Information Technology, Uppsala University, 2001. http://www.it.uu.se/research/reports/2001-012/

[EES01a] J. Engblom, A. Ermedahl, M. Sjodin, J. Gustafsson, H. Hansson, Execution-Time Analysis for Embedded Real-Time Systems. Accepted for publication in the Software Tools for Technology transfer special issue on ASTEC, 2001.

[EFR82] B. Efron, The jackknife, the bootstrap and other resampling plans, Society for industrial and applied mathematics, 1982

[EFR93] B. Efron, An introduction to the bootstrap, Chapman and Hall, 1983

[EF02] J. Estublier, J. M. Favre, Building Reliable Component-Based Software Systems, Chapter Component Models and Technology, Artech House, 2002

[Fis35] Fisher, R. A. "The Fiducial Argument in Statistical Inference." Ann. Eugenics 6, 391-398, 1935

[FM03] P.J. Fortier, H.E. Michel, Computer Systems Performance Evaluation and Prediction, Digital Press, 2003.

[GE98] J. Gustafsson, A. Ermedahl, Automatic derivation of path and loop annotations in object-oriented real-time *programs*. To be published in Journal of Parallel and Distributed Computing Practices, vol. 1 no. 2, June 1998.

[Gil51] J. Gil-Pelaez, Note on the inversion theorem, Biometrika 38, pp. 481-482, 1976

[GL03] J. Gustafsson, B. Lisper, P. Puschner, "Input-Dependency Analysis for Hard Real-Time Software", 9-th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003F) (IEEE), 2003, Capri Island, Italy

[GMH01] 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, p.580-589, March 2001, Munich, Germany.

[GPT01] K. Goseva-Popstojanova and K.S. Trivedi, Architecture Based Approach to Reliability Assessment of Software Systems" Performance Evaluation, Vol.45/2-3, June 2001.

[Gus00] J. Gustafsson, Eliminating Annotations by Automatic Flow Analysis of Real-Time Programs. Proceedings of the 7th international conference on Real-Time Computing Systems and Applications (RTCSA'00), Cheju Island, South Korea, Dec 2000

[GZ01] T. Genβler and C. Zeidler, Rule-driven component composition for embedded system, In Proceedings of 4th ICSE Workshop on Component-based software engineering, Toronto, Canada, May 2001.

[Ha87] D. Harel, Statecharts: a visual formalism for complex system. Science of Computer Programming, vol.8, (no.3), p.231-74, June, 1987

[Han01] P. Hanna, JSP: The Complete Reference, McGraw-Hill Osborne Media, 2001

[HAT03] D. Hamlet, M. Andric, Z. Tu, Experiments with Composing Component Properties, In Proceedings of 6th ICSE Workshop on Component-based software engineering, Portland, USA, May 2003.

[HC01] D. K. Hammer and M.R.V. Chaudron, Component Models for Resource-Constraint Systems: What are the Needs?, Proc. 6th Int. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS), Rome, January 2001.

[HeC01] G. T. Heineman, W. T. Councill, Component-based software engineering: putting the pieces together, Addison-Wesley, 2001.

[HMS03] S. Hissam, G. Moreno, J. Stafford, K Wallnau, Enabling predictable assembly, Journal of Systems and Software, Special issue on: Component-based software engineering, Volume 65, Issue 3, March 2003.

[HNS00] C. Hofmeister, R. Nord, D. Soni, Applied software architecture, Addison-Wesley, 2000.

[HWR99] C.E. Hrischuk, C.M. Woodside, 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.

[I2C03] AN10216-01, I^2C Manual, Philips Semiconductors, 24 March 2003

[Ifr72] A.F. Ifram, On the characteristic functions of F and t distributions, Sankhya: Ser. A 32, pp 350-352, 1972

[ISW02] J. Ivers, N. Sinha, K. Wallnau, A Basis for Composition Language CL, Technical Note, CMU/SEI-2002-TN026, 2002

[Jai91] R. Jain, The art of computer systems performance analysis, Techniques for Experimental Design, Measurement, Simulation and Modeling, John Wiley & Sons, 1991.

[JMC03] M. de Jonge, J. Muskens, M. Chaudron, Scenario-Based Prediction of Run-time Resource Consumption in Component-Based Software Systems, In Proceedings of 6th ICSE Workshop on Component-based software engineering, Portland, USA, May 2003.

[KiP00] 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, Illinois, USA, 2000.

[Kle76] L. Kleinrock, Queuing Systems, Volume II: Computer Applications, John Wiley & Sons, 1976

[Kle96] L. Kleinrock, R. Gail, Queuing Systems: Problems and Solutions, John Wiley & Sons, 1996

[KO02] A. Krause, M. Olson, "The basics of S-Plus", 3rd Edition, Springer Verlag, 2002

[KP00] R. Kirner, P. Puschner, Supporting Control-Flow-Dependent Execution Times on WCET Calculation, url:citeseer.nj.nec.com/kirner00supporting.html.

[KP01] R. Kirner, P. Puschner, Transformation of Path Information for WCET Analysis during Compilation. In Proceedings of 13th Euromicro Conference on Real-Time Systems, pages 29—36, June 2001.

[KRP93] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, H. Gonzalez, A Practitioners Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. Boston, MA: Kluwer Academic Publishers, 1993.

[Lan02] R. Land, A Brief Survey of Software Architecture, MRTC Technical Report, ISSN 1404-3041 ISRN MDH-MRTC-57/2002-1-SE, Mälardalen Real-Time Research Centre, Department of Computer Engineering, Mälardalen University, February 2002.

[Lar04] M. Larsson, Predicting Quality Attributes in Component-based Software Systems, Ph.D. thesis, Department of Computer Science and Engineering, Malardalen University, 2004

[Liu00] J.W.S Liu, Real-time systems, Prentice-Hall, 2000

[LS00] G.T. Leavens, M. Sitaraman, Foundations of component-based systems, Cambridge University Press, 2000.

[LS99] T. Lundqvist, P. Stenström, An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution. In Real-Time Systems, 17 (2/3):183-207, November 1999.

[LWN02] M. Larsson, A. Wall, C. Norström, I. Crnkovic, Using Prediction Enabled Technologies for Embedded Product Line Architectures, In Proceedings of 5th ICSE Workshop on Component-Based Software Engineering, Orlando, Florida, USA May 2002.

[MAD04] D. A. Menascé, V. A. F. Almeida, L. W. Dowdy, Performance by Design: Computer Capacity Planning by Example, Prentice Hall, 2004.

[Mas02] D. Mason, Probabilistic Analysis for Component Reliability Composition, In Proceedings of 5th ICSE Workshop on Component-Based Software Engineering, Orlando, Florida, USA May 2002.

[Mcc76] T.J. McCabe, A complexity measure, In IEEE Transactions on Software Engineering, volume SE2 (4), pages 308-320, December 1976.

[MCD85] R.P. McDonald, Factor Analysis and Related Methods, Lawrence Erlbaum Associates, London, 1985

[MHW02] G.A. Moreno, S.A. Hissam, K.C. Wallnau, Statistical Models for Empirical Component Properties and Assembly-Level Property Predictions: Towards Standard Labeling, In Proceedings of 5th ICSE Workshop on Component-Based Software Engineering, Orlando, Florida, USA May 2002.

[MN97] McCullagh, J.A. Nelder, Generalized Linear Models, Second Edition, Chapman and Hall/CRC, 1997

[MoH01] R. Monson-Haefel, Enterprise JavaBeans, O' Reilly and Associates, 2001

[MON01] D.C. Montgomery, Design and Analysis of Experiments, 5th Edition, John Wiley & Sons, 2001

[MR03] D.C. Montgomery, G.C. Runger, Applied Statistics and Probability for Engineers, 3rd Edition, John Wiley & Sons, 2003

[OLK00] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, The Koala Component Model for Consumer Electronics Software. Computer 33, 3 (2000), pp 33-85, 2000.

[Omm02] R. van Ommering, Horizontal Communication: a Style to Compose Control Software, Philips Research Laboratories, (to appear in Software Practice & Experience, Wiley, 2003.

[Omm03] R. van Ommering, Building Product Populations with Software Components, International Conference on Software Engineering (ICSE 2002), Orlando, Florida, US, May 2002.

[Pel99] C. Peltz, A Hierarchical Technique for Composing COM based components, In Proceedings of 2nd International Workshop on Component-Based Software Engineering, Los Angeles, USA, 1999

[PK89] P. Puschner, C. Koza, Calculating the maximum execution times of realtime programs. In Journal of Real-Time Systems, vol. 1, pp. 159--176, 1989.

[PKK98] N. Pitts-Moultis, C.Kirk, W. Kelly, XML Black Book: The Most Comprehensive Resource for XML - The Next Hot Language for the World Wide Web!, The Coriolis Group, 1998

[Ren99] M.A. Reniers, Message Sequence Chart, Syntax and Semantics, Ph.D. thesis, TUE, 1999

[RZJ02] K. Richter, D. Ziegenbein, M. Jersak, R. Ernst, Model Composition for Scheduling Analysis, Model Composition for Scheduling Analysis in Platform Design, In Proc. 39th Design Automation Conference (DAC02), New Orleans, LA, USA, 2002.

[Rob76] G.K. Robinson, Properties of Student's t and of Behrens-Fisher solution of the two means problem, Annals of Statistics 4, 1976, pp. 963-971

[Rog97] D. Rogerson, Inside COM, Microsoft Press, 1997

[Sal98] D. Salome, Statistical Inference via Fiducial Methods, ph. D. Thesis, the University of Groningen, 1998

[Sch01] H. Schmidt, Trusted Components – Towards Automated with Predictable Properties, In Proceedings of the 4th ICSE Workshop on Component-based software engineering, Toronto, Canada, May 2001.

[SG98] 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.

[Sit01] M. Sitaraman, Compositional Performance Reasoning, In Proceedings of the 4th ICSE Workshop on Component-based software engineering, Toronto, Canada, May 2001.

[Smi00] C.U. Smith, Software Performance AntiPatterns, Proc. 2nd Int. Workshop on Software and Performance, Ottawa, September 2000 (with L.G. Williams).

[Smi02] C.U. Smith, New Software Performance Antipatterns, More Ways to Shoot Yourself in the Foot, Proc. CMG, Reno, Dec., 2002 (with L.G. Williams).

[Smi90] C. U. Smith, Performance Engineering of Software Systems, Addison-Wesley, 1990

[Sva03] M. Svahnberg, Supporting Software Architecture Evolution, Ph.D. thesis, Department of Software Engineering and Computer Science, Blekinge Institute of Technology, 2003

[SW02] C. U. Smith, L. G. Williams: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software, Addison-Wesley, 2002

[SW97] C. U. Smith, L. G. Williams: Performance Engineering Evaluation of Object-Oriented Systems with SPE-ED, Springer-Verlag, Berlin, 1997

[Szy98] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1998.

[USL00] L. Unnikrishnan, S. D. Stoller, Y. A. Liu. Automatic accurate stack space and heap space analysis for high-level languages. Technical Report TR 538, Computer Science Department, Indiana University, Feb. 2000

[WEI95] N.A. Weiss, Introductionary Statistics, Fourth edition, Addison-Wesley Publishing company, 1995

[WIT01] V. Witkovský, On the exact computation of the density and of the quantiles of linear combination of *t* and *F* random variables, Journal of Statistical Planning and Inference, Vol. 94, Issue 1, pp. 1-13, 2001

[WMW03] X. Wu, D. McMullan, M. Woodside, Component Based Performance Prediction, In Proceedings of 6th ICSE Workshop on Component-based software engineering, Portland, USA, May 2003.

[WOS03] B. W. Weide, W. F. Ogden, M. Sitaraman, Expressiveness Issues in Compositional Performance Reasoning, In Proceedings of 6th ICSE Workshop on Component-based software engineering, Portland, USA, May 2003.

[XML1] XML syntax specification: http://www.w3.org/TR/REC-xml

[XML2] Examples of XML parsers: http://www.xml-parser.com/examples.htm

[XML3] Google directory of XML parsers: http://directory.google.com/Top/Computers/Data_Formats/Markup_Languages/XML/Tools/Parsers/

[MXML] MathML homepage: http://www.w3.org/TR/REC-MathML/

[You01] M. Young, XML Step by Step, Second Edition, Microsoft Press, 2001

[ZG94] B. Zorn, D. Grunwald. Evaluating models of memory allocation. ACM Transactions on Modeling and Computer Simulation, 1(4):107--131, 1994.

Samenvatting

Dit proefschrift omvat de resultaten van het promotie-onderzoek van E. Eskenazi en A. Fyukov, uitgevoerd binnen het STW[1] AIMES (Architectural Modeling of Embedded Systems) project. Dit project wordt uitgevoerd door de Technische Universiteit Eindhoven in samenwerking met haar industriele partners Philips Research, Philips Semiconductors en Philips Medical Systems. Het proefschrift is gebaseerd op acht publicaties en een Philips-intern technisch verslag.

De focus van het onderzoek lag bij het tijdens de architectuur ontwerp fase bepalen van software kwaliteitsattributen (bijvoorbeeld aanpasbaarheid, onderhoudbaarheid en schaalbaarheid) in het algemeen en performance in het bijzonder. Er bestaan slechts enkele systematische processen voor het in een vroeg stadium bepalen van die kwaliteitsattributen die gerelateerd zijn aan de executie eigenschappen van component-gebaseerde software. Zulke processen beslaan ten hoogste een kleine subset van alle mogelijke kwaliteitsattributen en ze werken slechts in een zeer specifieke context. Bovendien is de ontwikkelingsfase gewoonlijk gericht op de correctheid van software, terwijl niet-functionele eigenschappen vaak pas een rol gaan spelen tijdens de integratiefase en de testfase. Dit resulteert maar al te vaak in grondig herontwerp van de software of hardware om aan niet-functionele eisen te kunnen voldoen of, erger nog, in ad-hoc aanpassingen die zorgvuldig ontworpen kwaliteitsaspecten onbedoeld om zeep helpen.

Kwalititeitsattributen komen vaak boven water op systeem-niveau, wat het onmogelijk maakt om ze in bepaalde delen van de software terug te vinden. Dit maakt het vroegtijdig bepalen van kwaliteitsattributen een complexe taak, en dan vooral in component-gebaseerde software architecturen: de software is doorgaans niet in componenten opgedeeld op basis van kwaliteits-, maar van functionele overwegingen.

Ons onderzoek is gebaseerd op case studies in twee verschillende industriele domeinen: Consumenten Electronica (CE) en Professionele Systemen (PS). E. Eskenazi onderzocht de software architectuur van moderne TV systemen, terwijl A. Fyukov zich richtte op de software voor een medisch onderzoekssysteem.

In beide domeinen is een aantal eisen met betrekking tot kwaliteitsattributen verzameld. Een van de meest relevante dynamische kwaliteitsattributen in beide domeinen is performance, waaraan voorheen te weinig aandacht werd besteed. Daarom werd besloten om een in de industrie toepasbare methode te ontwikkelen die het inschatten en voorspellen van performance mogelijk maakt. Daarnaast werd het benodigde geheugen van een TV geselecteerd als statisch kwaliteitsattribuut, aangezien TV's strikte beperkingen stellen aan de grootte van dit geheugen.

Het AIMES project heeft twee methoden opgeleverd: a) een methode om de waarde van statische kwaliteitsattributen van component-gebaseerde software architecturen in te schatten, en b) een methode om de performance van software architecturen in ontwikkeling te voorspellen (de APPEAR methode). Beide methoden leiden tot verbeteringen op het gebied van de kwantitatieve evaluatie van kwaliteitsattributen van component-gebaseerde software architecturen. De meest in het oog lopende gezamenlijke eigenschap van beide methoden is de flexibele selectie van het abstractieniveau, welke een

[1] Project EWI.4877

afweging mogelijk maakt tussen de specificatie en modelleer inspanning enerzijds, en de nauwkeurigheid van de evaluatie gedurende systeemontwerp anderzijds.

De eerste methode biedt een innovatieve manier om statische kwaliteitsattributen te bepalen voor complexe, component-gebaseerde architecturen met vele configureerbare componenten en verfijnde binding. Deze methode stelt een architect in staat om de complexiteit van het modelleren te beperken door middel van het rationeel selecteren van significante factoren zoals de configuratie parameters van componenten. De methode is echter niet toepasbaar op het bepalen van niet-additieve statische kwaliteitsattributen, noch op component modellen met endogenous binding (zoals bijvoorbeeld COM.)

De APPEAR methode biedt het beste van twee werelden: gedragssimulatie en statistische modellen. Simulatie geeft inzicht in voor performance relevante aspecten, terwijl de statistische modellen de methode minder bevattelijk maken voor een combinatorische explosie van details. Deze methode biedt flexibiliteit waar het gaat om het kiezen van de componenten welke beschreven worden door middel van een gedragsmodel en welke statistisch worden gemodelleerd. De APPEAR methode is echter niet toepasbaar op componenten die uit het niets ontwikkeld worden, en biedt slechts betrouwbare resultaten voor soortgelijke componenten. Bovendien zijn uitgebreide metingen nodig voor het construeren van de statistische modellen.

Beide methoden hebben hun nut bewezen tijdens case studies met betrekking op industriele software. De eerste methode werd toegepast in het Consumenten Electronica domein om het statische geheugengebruik van TV software te bepalen. Het resultaat had een relatieve inschattingsfout van 4%. Dit experiment werd in samenwerking tussen beide promovendi uitgevoerd.

De APPEAR methode is zowel in het CE als het PS domein geverifieerd. E. Eskenazi gebruikte deze methode om het gemiddelde CPU gebruik en de executietijd van Teletext software te voorspellen. A. Fyukov voorspelde de gemiddelde responstijd van software om medische beelden te bekijken. Hoewel deze domeinen verschillen, gaf de APPEAR methode een relatieve voorspellingsfout van minder dan 20% in beide, wat in de praktijk goed genoeg is.

Summary

This thesis summarizes the results of the work performed by two Ph. D. students– E. Eskenazi and A. Fyukov– in the context of the STW[2] AIMES (Architectural Modeling of Embedded Systems) project. This project was conducted by the Technical University of Eindhoven in close cooperation with its industrial partners Philips Research, Philips Semiconductors, and Philips Medical Systems. The thesis is based on eight publications and one internal Philips technical report.

The research focused on the assessment, during the architecting phase, of software quality attributes (e.g. adaptability, maintainability and scalability) in general and performance in particular. There are only a few systematic approaches for the early assessment of quality attributes related to the execution properties of component-based software. Those approaches cover only a small subset of quality attributes and operate in a very narrow context. Moreover, during development, the focus is usually on software correctness, whereas quality considerations are often postponed until integration or testing phases. This has often resulted in major redesign effort spent on tuning the software or hardware in order to meet quality requirements, or even worse, in ad-hoc changes that ruin all other carefully tuned quality attributes.

Quality attributes often emerge at the system level, making it impossible to localize them in particular parts of software. This makes early assessment of quality attributes complicated, especially in component-based software architecting: software is usually componentized not with respect to quality attributes, but on the basis of functionality.

Our research was based on case studies in two different industrial domains: Consumer Electronics (CE) and Professional Systems (PS). E. Eskenazi studied the software architecture of modern TV sets in the former domain, while A. Fyukov considered the software for a medical imaging system in the latter domain.

For each domain, a set of requirements for quality attributes was collected. One of the most relevant dynamic quality attributes for both domains is performance, to which insufficient attention had hitherto always been paid. It was therefore decided to develop an industrial-strength method for performance estimation and prediction. In addition, the memory demand of a TV, being a static quality attribute, was selected for investigation, as TV sets have stringent memory constraints.

The AIMES project resulted in two methods: a) a method for estimation of the static quality attributes of component-based software architectures, and b) a method for predicting the performance of evolving software architectures (APPEAR method). Both methods lead to improvements in the field of quantitative evaluation of quality attributes of component-based software architectures. Their most remarkable common feature is the flexible selection of the level of abstraction, which allows tradeoffs between specification and modeling effort and evaluation accuracy during system design.

The first method is an innovative solution to the problem of the assessment of static quality attributes of complex component-based architectures with many configurable components and sophisticated binding. The method enables an architect to decrease the modeling complexity through rational selection of significant factors such as component

[2] Project EWI.4877

configuration parameters. This method can however not be used for assessing non-additive static quality attributes or for component models with endogenous binding (e.g., COM).

The APPEAR method combines the best of two worlds: behavior simulation and statistical modeling. The former provides insight into performance-relevant aspects, whereas the latter makes the method less susceptible to combinatorial explosion of details. The method allows flexibility in choosing which components are to be described by a behavioral model and which are to be modeled by statistical means. The APPEAR method does however not work for components developed from scratch, and it provides reliable estimates for similar components only. In addition, construction of the statistical models involves extensive measurements.

Both methods were validated by case studies on industrial software. The first method was applied in the Consumer Electronics domain to assess the static memory consumption of TV software. The results had a relative estimation error of 4%. This experiment was performed by both Ph. D. students together.

The APPEAR method was verified in both CE and PS domains. E. Eskenazi used the method to predict the average CPU utilization and execution time of the Teletext software, while A. Fyukov predicted the average response time of the viewing software used for medical images. Although these domains differ, the APPEAR method exhibited a relative prediction error of less than 20% in both of them, which is good enough for practical purposes.

Curriculum vitae of Evgeny Eskenazi

Evgeny Eskenazi (1976) was born in Leningrad in USSR. In 1999, he graduated from the Technical University of Saint-Petersburg with M.Sc. degree in Computer Science. During his study and one year after graduation, he worked as a software engineer in the Radio and Research Development Institute in the field of satellite communications systems. . He worked on his Ph.D. thesis in 2000-2004 within the AIMES project at the Mathematics and Computing Science department of the Eindhoven University of Technology. His research concerned component-based software architectures, software performance, and early prediction of the quality attributes of software architectures. The research project was conducted in the close cooperation with Philips Research and Philips Semiconductors, which provided premises for exercising the methods developed. He published several papers in the field of component-based software architecting and contributed to international software conferences such as OOPSLA (2001) and Euromicro (Best Paper Award, 2002) and ICSE (2004). Since 2004, he is employed as a software designer at Philips TASS in Eindhoven.

Curriculum vitae of Alexander Fyukov

Alexander Fyukov (1976) was born in Saint-Petersburg, Russia. In 1999 he graduated from the Technical University of Saint-Petersburg with M.Sc. degree in Computer Science. After graduation, he worked for three years as a software engineer in the fields of satellite communications and aviation simulators. He worked on his Ph.D. thesis in 2000-2004 within the AIMES project at the Mathematics and Computing Science department of the Eindhoven University of Technology. His research project was coupled with two industrial partners: Philips Research and Philips Medical Systems. Both partners provided experimental settings for the validation of theoretical approaches presented in this thesis. As a result of his research activities, Alexander Fyukov has published several papers in the field of component-based software architecting and contributed to international software conferences: OOPSLA (2001), Euromicro (Best Paper Award, 2002), and ICSE (2004). Since 2004, he is employed as a research scientist at Software Architectures group of Philips Research Laboratories in Eindhoven.

Titles in the IPA dissertation series

J.O. Blanco, The State Operator in Process Algebra, Faculty of Mathematics and Computing Science, TUE, 1996-01

A.M. Geerling, Transformational Development of Data-Parallel Algorithms, Faculty of Mathematics and Computer Science, KUN, 1996-02

P.M. Achten, Interactive Functional Programs: Models, Methods, and Implementation, Faculty of Mathematics and Computer Science, KUN, 1996-03

M.G.A. Verhoeven, Parallel Local Search, Faculty of Mathematics and Computing Science, TUE, 1996-04

M.H.G.K. Kesseler, The Implementation of Functional Languages on Parallel Machines with Distrib.\ Memory, Faculty of Mathematics and Computer Science, KUN, 1996-05

D. Alstein, Distributed Algorithms for Hard Real-Time Systems, Faculty of Mathematics and Computing Science, TUE, 1996-06

J.H. Hoepman, Communication, Synchronization, and Fault-Tolerance, Faculty of Mathematics and Computer Science, UvA, 1996-07

H. Doornbos, Reductivity Arguments and Program Construction, Faculty of Mathematics and Computing Science, TUE, 1996-08

D. Turi, Functorial Operational Semantics and its Denotational Dual, Faculty of Mathematics and Computer Science, VUA, 1996-09

A.M.G. Peeters, Single-Rail Handshake Circuits, Faculty of Mathematics and Computing Science, TUE, 1996-10

N.W.A. Arends, A Systems Engineering Specification Formalism, Faculty of Mechanical Engineering, TUE, 1996-11

P. Severi de Santiago, Normalisation in Lambda Calculus and its Relation to Type

Inference, Faculty of Mathematics and Computing Science, TUE, 1996-12

D.R. Dams, Abstract Interpretation and Partition Refinement for Model Checking, Faculty of Mathematics and Computing Science, TUE, 1996-13

M.M. Bonsangue, Topological Dualities in Semantics, Faculty of Mathematics and Computer Science, VUA, 1996-14

B.L.E. de Fluiter, Algorithms for Graphs of Small Treewidth, Faculty of Mathematics and Computer Science, UU, 1997-01

W.T.M. Kars, Process-algebraic Transformations in Context, Faculty of Computer Science, UT, 1997-02

P.F. Hoogendijk, A Generic Theory of Data Types, Faculty of Mathematics and Computing Science, TUE, 1997-03

T.D.L. Laan, The Evolution of Type Theory in Logic and Mathematics, Faculty of Mathematics and Computing Science, TUE, 1997-04

C.J. Bloo, Preservation of Termination for Explicit Substitution, Faculty of Mathematics and Computing Science, TUE, 1997-05

J.J. Vereijken, Discrete-Time Process Algebra, Faculty of Mathematics and Computing Science, TUE, 1997-06

F.A.M. van den Beuken, A Functional Approach to Syntax and Typing, Faculty of Mathematics and Informatics, KUN, 1997-07

A.W. Heerink, Ins and Outs in Refusal Testing, Faculty of Computer Science, UT, 1998-01

G. Naumoski and W. Alberts, A Discrete-Event Simulator for Systems Engineering, Faculty of Mechanical Engineering, TUE, 1998-02

J. Verriet, Scheduling with Communication for Multiprocessor Computation, Faculty of Mathematics and Computer Science, UU, 1998-03

J.S.H. van Gageldonk, An Asynchronous Low-Power 80C51 Microcontroller, Faculty of Mathematics and Computing Science, TUE, 1998-04

A.A. Basten, In Terms of Nets: System Design with Petri Nets and Process Algebra, Faculty of Mathematics and Computing Science, TUE, 1998-05

E. Voermans, Inductive Datatypes with Laws and Subtyping -- A Relational Model, Faculty of Mathematics and Computing Science, TUE, 1999-01

H. ter Doest, Towards Probabilistic Unification-based Parsing, Faculty of Computer Science, UT, 1999-02

J.P.L. Segers, Algorithms for the Simulation of Surface Processes, Faculty of Mathematics and Computing Science, TUE, 1999-03

C.H.M. van Kemenade, Recombinative Evolutionary Search, Faculty of Mathematics and Natural Sciences, UL, 1999-04

E.I. Barakova, Learning Reliability: a Study on Indecisiveness in Sample Selection, Faculty of Mathematics and Natural Sciences, RUG, 1999-05

M.P. Bodlaender, Scheduler Optimization in Real-Time Distributed Databases, Faculty of Mathematics and Computing Science, TUE, 1999-06

M.A. Reniers, Message Sequence Chart: Syntax and Semantics, Faculty of Mathematics and Computing Science, TUE, 1999-07

J.P. Warners, Nonlinear approaches to satisfiability problems, Faculty of Mathematics and Computing Science, TUE, 1999-08

J.M.T. Romijn, Analysing Industrial Protocols with Formal Methods, Faculty of Computer Science, UT, 1999-09

P.R. D'Argenio, Algebras and Automata for Timed and Stochastic Systems, Faculty of Computer Science, UT, 1999-10

G. F'abi'an, A Language and Simulator for Hybrid Systems, Faculty of Mechanical Engineering, TUE, 1999-11

J. Zwanenburg, Object-Oriented Concepts and Proof Rules, Faculty of Mathematics and Computing Science, TUE, 1999-12

R.S. Venema, Aspects of an Integrated Neural Prediction System, Faculty of Mathematics and Natural Sciences, RUG, 1999-13

J. Saraiva, A Purely Functional Implementation of Attribute Grammars, Faculty of Mathematics and Computer Science, UU, 1999-14

R. Schiefer, Viper, A Visualisation Tool for Parallel Program Construction, Faculty of Mathematics and Computing Science, TUE, 1999-15

K.M.M. de Leeuw, Cryptology and Statecraft in the Dutch Republic, Faculty of Mathematics and Computer Science, UvA, 2000-01

T.E.J. Vos, UNITY in Diversity. A stratified approach to the verification of distributed algorithms, Faculty of Mathematics and Computer Science, UU, 2000-02

W. Mallon, Theories and Tools for the Design of Delay-Insensitive Communicating Processes, Faculty of Mathematics and Natural Sciences, RUG, 2000-03

W.O.D. Griffioen, Studies in Computer Aided Verification of Protocols, Faculty of Science, KUN, 2000-04

P.H.F.M. Verhoeven, The Design of the MathSpad Editor, Faculty of Mathematics and Computing Science, TUE, 2000-05

J. Fey, Design of a Fruit Juice Blending and Packaging Plant, Faculty of Mechanical Engineering, TUE, 2000-06

M. Franssen, Cocktail: A Tool for Deriving Correct Programs, Faculty of Mathematics and Computing Science, TUE, 2000-07

P.A. Olivier, A Framework for Debugging Heterogeneous Applications, Faculty of Natural Sciences, Mathematics and Computer Science, UvA, 2000-08

E. Saaman, Another Formal Specification Language, Faculty of Mathematics and Natural Sciences, RUG, 2000-10

M. Jelasity, The Shape of Evolutionary Search Discovering and Representing Search Space Structure, Faculty of Mathematics and Natural Sciences, UL, 2001-01

R. Ahn, Agents, Objects and Events a computational approach to knowledge, observation and communication, Faculty of Mathematics and Computing Science, TU/e, 2001-02

M. Huisman, Reasoning about Java programs in higher order logic using PVS and Isabelle, Faculty of Science, KUN, 2001-03

I.M.M.J. Reymen, Improving Design Processes through Structured Reflection, Faculty of Mathematics and Computing Science, TU/e, 2001-04

S.C.C. Blom, Term Graph Rewriting: syntax and semantics, Faculty of Sciences, Division of Mathematics and Computer Science, VUA, 2001-05

R. van Liere, Studies in Interactive Visualization, Faculty of Natural Sciences, Mathematics and Computer Science, UvA, 2001-06

A.G. Engels, Languages for Analysis and Testing of Event Sequences, Faculty of Mathematics and Computing Science, TU/e, 2001-07

J. Hage, Structural Aspects of Switching Classes, Faculty of Mathematics and Natural Sciences, UL, 2001-08

M.H. Lamers, Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes, Faculty of Mathematics and Natural Sciences, UL, 2001-09

T.C. Ruys, Towards Effective Model Checking, Faculty of Computer Science, UT, 2001-10,

D. Chkliaev, Mechanical verification of concurrency control and recovery protocols, Faculty of Mathematics and Computing Science, TU/e, 2001-11

M.D. Oostdijk, Generation and presentation of formal mathematical documents, Faculty of Mathematics and Computing Science, TU/e, 2001-12

A.T. Hofkamp, Reactive machine control: A simulation approach using chi, Faculty of Mechanical Engineering, TU/e, 2001-13

D. Bosnacki, Enhancing state space reduction techniques for model checking, Faculty of Mathematics and Computing Science, TU/e, 2001-14

M.C. van Wezel, Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects, Faculty of Mathematics and Natural Sciences, UL, 2002-01

V. Bos and J.J.T. Kleijn, Formal Specification and Analysis of Industrial Systems, Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e, 2002-02

T. Kuipers, Techniques for Understanding Legacy Software Systems, Faculty of Natural Sciences, Mathematics and Computer Science, UvA, 2002-03

S.P. Luttik, Choice Quantification in Process Algebra, Faculty of Natural Sciences, Mathematics, and Computer Science, UvA, 2002-04

R.J. Willemen, School Timetable Construction: Algorithms and Complexity, Faculty of Mathematics and Computer Science, TU/e, 2002-05

M.I.A. Stoelinga, Alea Jacta Est: Verification of Probabilistic, Real-time

and Parametric Systems, Faculty of Science, Mathematics and Computer Science, KUN, 2002-06

N. van Vugt, Models of Molecular Computing, Faculty of Mathematics and Natural Sciences, UL, 2002-07

A. Fehnker, Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems, Faculty of Science, Mathematics and Computer Science, KUN, 2002-08

R. van Stee, On-line Scheduling and Bin Packing, Faculty of Mathematics and Natural Sciences, UL, 2002-09

D. Tauritz, Adaptive Information Filtering: Concepts and Algorithms, Faculty of Mathematics and Natural Sciences, UL, 2002-10

M.B. van der Zwaag, Models and Logics for Process Algebra, Faculty of Natural Sciences, Mathematics, and Computer Science, UvA, 2002-11

J.I. den Hartog, Probabilistic Extensions of Semantical Models, Faculty of Sciences, Division of Mathematics and Computer Science, VUA, 2002-12

L. Moonen, Exploring Software Systems, Faculty of Natural Sciences, Mathematics, and Computer Science, UvA, 2002-13

J.I. van Hemert, Applying Evolutionary Computation to Constraint Satisfaction and Data Mining, Faculty of Mathematics and Natural Sciences, UL, 2002-14

S. Andova, Probabilistic Process Algebra, Faculty of Mathematics and Computer Science, TU/e, 2002-15

Y.S. Usenko, Linearization in μCRL, Faculty of Mathematics and Computer Science, TU/e, 2002-16

J.J.D. Aerts, Random Redundant Storage for Video on Demand, Faculty of Mathematics and Computer Science, TU/e, 2003-01

M. de Jonge, To Reuse or To Be Reused: Techniques for component composition and construction, Faculty of Natural Sciences, Mathematics, and Computer Science, UvA, 2003-02

J.M.W. Visser, Generic Traversal over Typed Source Code Representations, Faculty of Natural Sciences, Mathematics, and Computer Science, UvA, 2003-03

S.M. Bohte, Spiking Neural Networks, Faculty of Mathematics and Natural Sciences, UL, 2003-04

T.A.C. Willemse, Semantics and Verification in Process Algebras with Data and Timing, Faculty of Mathematics and Computer Science, TU/e, 2003-05

S.V. Nedea, Analysis and Simulations of Catalytic Reactions, Faculty of Mathematics and Computer Science, TU/e, 2003-06

M.E.M. Lijding, Real-time Scheduling of Tertiary Storage, Faculty of Electrical Engineering, Mathematics \& Computer Science, UT, 2003-07

H.P. Benz, Casual Multimedia Process Annotation -- CoMPAs, Faculty of Electrical Engineering, Mathematics \& Computer Science, UT, 2003-08

D. Distefano, On Modelchecking the Dynamics of Object-based Software: a Foundational Approach, Faculty of Electrical Engineering, Mathematics \& Computer Science, UT, 2003-09

M.H. ter Beek, Team Automata -- A Formal Approach to the Modeling of Collaboration Between System Components, Faculty of Mathematics and Natural Sciences, UL, 2003-10

D.J.P. Leijen, The λ Abroad -- A Functional Approach to Software Components, Faculty of Mathematics and Computer Science, UU, 2003-11

W.P.A.J. Michiels, Performance Ratios for the Differencing Method, Faculty of Mathematics and Computer Science, TU/e, 2004-01

G.I. Jojgov, Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving, Faculty of Mathematics and Computer Science, TU/e, 2004-02

P. Frisco, Theory of Molecular Computing -- Splicing and Membrane systems, Faculty of Mathematics and Natural Sciences, UL, 2004-03

S. Maneth, Models of Tree Translation, Faculty of Mathematics and Natural Sciences, UL, 2004-04

Y. Qian, Data Synchronization and Browsing for Home Environments, Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e, 2004-05

F. Bartels, On Generalised Coinduction and Probabilistic Specification Formats, Faculty of Sciences, Division of Mathematics and Computer Science, VUA, 2004-06

L. Cruz-Filipe, Constructive Real Analysis: a Type-Theoretical Formalization and Applications, Faculty of Science, Mathematics and Computer Science, KUN, 2004-07

E.H. Gerding, Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications, Faculty of Technology Management, TU/e, 2004-08

N. Goga, Control and Selection Techniques for the Automated Testing of Reactive Systems, Faculty of Mathematics and Computer Science, TU/e, 2004-09

M. Niqui, Formalising Exact Arithmetic: Representations, Algorithms and Proofs, Faculty of Science, Mathematics and Computer Science, RU, 2004-10

A. Loh, Exploring Generic Haskell, Faculty of Mathematics and Computer Science, UU, 2004-11

I.C.M. Flinsenberg, Route Planning Algorithms for Car Navigation, Faculty of Mathematics and Computer Science, TU/e, 2004-12

R.J. Bril, Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets, Faculty of Mathematics and Computer Science, TU/e, 2004-13

J. Pang, Formal Verification of Distributed Systems, Faculty of Sciences, Division of Mathematics and Computer Science, VUA, 2004-14

F. Alkemade, Evolutionary Agent-Based Economics, Faculty of Technology Management, TU/e, 2004-15

E.O. Dijk, Indoor Ultrasonic Position Estimation Using a Single Base Station, Faculty of Mathematics and Computer Science, TU/e, 2004-16

S.M. Orzan, On Distributed Verification and Verified Distribution, Faculty of Sciences, Division of Mathematics and Computer Science, VUA, 2004-17

M.M. Schrage, Proxima - A Presentation-oriented Editor for Structured Documents, Faculty of Mathematics and Computer Science, UU, 2004-18

E. Eskenazi and A. Fyukov, Quantitative Prediction of Quality Attributes for Component-Based Software Architectures, Faculty of Mathematics and Computer Science, TU/e, 2004-19